In this chapter, we’re going
to study how to transform data with Hive.
So beeline is the shell
that you should use
if you’re using a terminal
to connect and issue
Hive queries.
You can use in a script, you can use
Beeline -e to execute a query
or Beeline
-f to execute query from a file.
If you’re connected, then
you can issue your queries
after the prompt.
So those are the data types and
different semantics that you can use.
It’s the it’s supposed to match
what you already know about SQL.
And so,
so what are the similarities?
Well the language
for one, the connection with the ODBC and JDBC,
the security.
The main difference is lies
in the way you approach your-
the design of your database.
RDBMS, In RDBMSs,
the approach was to
model your databases for any use case
and the results would be so you would have-
if you followed those standards,
you would have a mathematically
perfect model but a slow one.
If you make make sure
that each piece of information
would be located in one table,
if you did not do any denormalization,
then it would result in a slow, slow
queries.
In my consulting days,
it was not uncommon to see queries
that would take a whole page
with multiple joins and nested selects.
That’s-
that is supposed to be normal, but
it leads to slow performance.
But with Hive
and with big data,
the idea is to design your tables
for queries, specific queries,
not just any queries but specific queries.
So what you’re going to do is
maybe if the same data will be
laid out in two different tables
differently

to accommodate for two different queries.
But you, you’re free from this
constraint of having the information
only in one table.
And because performance
is a very important with big data,
the need for performance is more acute
because of the size of the data.
Then we
free ourselves from this constraint
of having a normalized
database design.
We try to focus on the performance
of the queries
and do whatever it takes
on the design of the tables to to have
efficient queries.
Another difference is that
RDBMS must use schema on
Write.
Okay. So
data has to comply with the contract
before being accepted in the database.
On the other hand, Hive uses
schema on read so all the data
is welcome.
It’s when you try to access the data
that the schema is applied
and can eventually fail.
But the data is
accepted already.
Having used RDBMS for decades,
we- we are aware of the constant
and sometimes this was
a too much constraint
for some types of activities.
Like when we- we used to do
data migration,
we would disable all constraints
and then migrate the data
and try to fix things afterwards.
But that was the price to pay
to have queries
that return things and,
and hopefully have clean data.
Of course, because we have believed that
if use these systems for the case,
there is,
there is a problem with data quality.
It doesn’t prevent problems
with data quality; maybe
it reduces them, but there
still are still problems
with data quality.
So that’s the schema on write

verus the schema on read.
So for the syntax itself,
it’s very similar
to what you use with a SQL database.
So to create a database,
you use CREATE database; to drop database,
you use DROP database; to use a database,
you use USE; to alter the database, use ALTER.
So this is the syntax
for to create a table.
So the
the difference in the create table in
Hive is there are two main
reasons in the statement is the
logical section where you
provide the schema of the table with the
the column name and the data type.
Okay.
But there’s below the fold.
Then there’s the
physical layout of the table.
The way it’s partitioned, clustered,
or skewed, and the directories, the format.
All this information
specifies on the table-
on the table will be stored.
And this is a very flexible.
This allows Hive to store
data of any given format provided
you have the
the storage handler,
java classes,
then you can
store data of any format.
And so
with Hive you can perform SQL
operations on data that is stored
physically in different formats.
That is quite powerful.
It’s not something that you it’s
not a goal to have all of your tables
in different formats,
but it’s a, it’s a nice
to have feature.
So you can rename tables
or change the properties of your,
of your table.
What you have to bear in mind
when dealing with Hive and big data
in general
is that when you change something
in the metadata, it doesn’t apply
immediately to the data.
The reason for this
is that with a simple command

you could create a storm in the cluster
by changing everything.
So the changes apply to the data
that will be inserted in
the table after the change.
So if you
change something, you have to be careful
and bear in mind
that you have to keep the data
and metadata in sync.
So the data
types are the usual suspects.
It’s the same that we saw with Spark,
and on top of that
you have the complex types also
which are not mentioned here,
but the same complex
type that we saw with Spark
and we study them later on.
So this is all you can create
a managed table.
This tells Hive that
this will be a
a CSV formatted file
in this customer table.
Now this is an external table,
so this is the keyword "external"
and again is going to be
a CSV file and the location is specified,
it will be in this folder.
Those are the differences
between the external and managed tables.
The main difference is the behavior.
When you DROP a table, the- because the
the data in HIVE
tables is stored in folders,
it can be tempting to
just copy
or move the files into those folders.
It becomes increasingly difficult with
the latter versions of Hive because of the
security.
And the proper way to do this
is to use the Hive syntax with LOAD DATA
or to INSERT INTO table statements
which can accept either values
like this one or a query like this one.
By using those statements, the
the metastore is kept in sync
with the data, whereas if you just
copy files into the folder
then the Metastore is not aware
of the changes in the table.
So when you’ve done all this work,
then you can enjoy a SELECT *

FROM customers
or any kind of SQL queries
with where clauses, joins.
This will all be translated to the
whatever
the execution engine is set for Hive.
And now, you
going to perform
the exercise of using Hive SQL.
WEBVTT
Now we are going to look at the solution
of the introduction
to Hive SQL notebook. So
we need to restart the session.
We interpret with the JDBC interpreter
because otherwise we’re going to get
an invalid session,
and I’m going to execute all the steps
before the solution.
Okay, so you see here that
we are using a new interpreter
and the JDBC interpreter.
So we can use a show table,
and for the time being
we have zero tables.
We do a drop table If it exists,
and then we are going to create a table,
the marvel_comics_row external table.
And if we show the tables again, then
we see our table.
And we can do a describe() on this table.
Next we can
load the data, which
is a CSV formatted file.
And since we have data
we should do a SELECT
and then we can see our data
in the table.
Next, we’re going to
drop the marvel_comics table we created.
But this one, this time
we’re going to create
an ORC stored table
and insert the data from the raw table
that we created
previously into this new one.
So that’s the way that you can-
that’s the usual workflow of
ingesting
row formatted files and then creating
ORC formatted or Paruqet
formatted tables.
You could see that these took some time
because that’s the time taken
to create the ORC files

and yes, the data is in our table
that’s cool.
You can look at
where the table is
stored.
So we can
we confirm that it’s using the ORC format
and it’s a managed table
that is stored in the warehouse table space
managed.
Since now we have the data in
ORC format.
We can drop the raw table
and we can check that
it has been deleted.
And since it was an external table
we can check that the row data
is still there.
And if we drop the managed table
then we no longer
have the table in the metastore
and we can check
that the files are going to HDFS.
So there is no such write object to me,
but there is marvel_comics.
I prefer
when the sales do not trigger an evo.
So I would prefer this version of this.
So interacting with S3 is always available.
but you see there is no file
in this folder.
I agree it’s quite difficult to see.
Okay, that’s it for this solution to this notebook.
WEBVTT
Now let’s talk about Hive ACID transactions.
So if you are familiar with RDBMS
transactions with
multiple tables, commit and, roll backs,
this is not one of them.
It’s a single table/
single-statement transaction
so that can
support multiple partitions
but not multiple tables.
The transactional
behavior of Hive is implemented
by adding a layer of transaction
oriented folders
between the root folder of the table
and the files that contain the data.
Those are the delta folders here,
okay.
So if you have this
and first we look at insert only table.
If you have these three transactions,

this creates three delta folders
and if we assume this,
the second transaction fails,
it will be identified as failed in the metastore.
The metastore also keeps track of the transactions
that are running
so that if users
ask to see the content of the table,
Hive will skip the
transactions that failed
and the transactions that are running; thus
offering a consistent view of the content
of the table to other users.
This is how isolation is achieved.
To implement updates
and delete,
Hive needed
additional information, so the
transactional table contains a row ID
metadata column.
Let’s look at how it works in an example.
Let’s use this example,
where we have three rows
with oranges, apples, and bananas.
They get this row_id column.
If we want to delete
the apples row,
then this creates a delete delta folder.
And in this folder you’d have this
information where the corresponding row
is the values are set to Null.
So for updates,
it’s a simple
sequence of delete
the corresponding row and update.
So this
creates two folders a delete delta folder
and a delta folder.
Before we move on to the exercise,
you might think and you would be right
that this system creates
a lot of small files.
That is true, but
Hive also does compaction
on those small folders on a regular basis.
It compacts the delete
and delta folders into a base folder.
And this is the- use-
and all those folders contain ORC files.
So this is
quite efficient.
Now it’s time for you to play with
this using the corresponding notebook.
WEBVTT
Let’s look at the solution

for the Hive transactions notebook.
I’m going to run
all the steps above.
Okay.
First, we need to drop the table
if it exists.
Then we create this table
with the insert-only property.
Okay.
So we have the, okay,
and then we’re going to insert
three rows into this table.
So this will lead to three transactions.
Okay.
And now we can
verify that the data has been inserted.
We have our three rows.
Now we create another table
that is transactional
and not only insert only.
And we insert those values
in a single statement.
And we can verify that the table contains
our data.
Here we
perform some
delete and update. So
we perform several transactions
and look at the results.
So we have deleted the
row with Superman and change the
the name of
Batman to be Robin.
There we go.
Let’s see.
We had successful credit operations.
Okay so now we have potentially
some folders with
small files.
We can perform a minor compact.
Being done already.
And we can look at the compactions
with the show compactions and
okay, the last command did not work out for me.
Maybe it will for you.
And that’s it for this exercise on
Hive transactions.
WEBVTT
Now let’s talk about
data retrieval, views,
and materialized views.
You can use a groupBy statement to group
rows that have the same values
and to use the alias
in the groupBy

you need to set this property to true
and be aware that the default is false.
You can use orderBy.
Okay.
So you need to be also aware that
ordering is a challenge
for distributed processing.
That’s why
orderBy without limit in subqueries
and views will be removed by the optimizer.
To compare
values, they need to be in the same JVM,
because we use a shell nothing model,
and there’s a limit
to what can fit into one single JVM.
That’s why there is a mechanism
in Hive to avoid problems.
So the sortBy statement is
something that an
inheritance from the Java MapReduce
framework.
It’s the sort-
it sorts the row before
sending the rows to the reducer.
So in MapReduce you add the-
after the shuffle you add the sort
and this is the same thing.
Distribute is a hint to the shuffle, to use
one column in particular to distribute the
the rows.
It’s often used with a sortBy
the idea is that you use
in the last statement you use the age
to assign
the each row to the to reducer,
the way this can be done
is a simple mechanism
using an ASC of the value of the age
modulo the number of reducer.
This way
you can guarantee that
each row with the same age
will go to the same reducer
and that
this reducer exists.
So this does not imply
that one reducer
will get only one value of age.
It will get several values of age,
all the values
that are of the same result
when you
compute the ASC of this value, modulo
the number of reducers..
So you can still do a sort afterwards.

So about Hive views,
they used to be
not materialized.
So it was just a definition
in the Hive metastore, so
you would create views
like you would create tables.
So it’s a convenience where to
encapsulate some complex queries and
allow users to use
simpler, simpler
tables of use.
So they used to be not materialized.
Okay.
So they would leave a need in the Hive metastore until
they were materialized.
So this was a view example.
This is a view example.
And with Hive 3
they introduced materialized views.
And the idea behind this is to trade off
storage for performance.
So the difference
is that you add the keyword materialized
and when you do this, then
the data will be materialized.
Not sure we support Druid anymore.
So I don’t know about the Druid
part but since the Druid
and Hive share the same Apache Calcite
engine, Hive could decide that,
okay,
this query would be better handled by druid.
And since Hive
with the proper
Java classes you could
store in any format, you could say
for a view or table store
as a Druid format.
So what you can do with materialized views is DROP,
SHOW, DESCRIBE
and you can also ALTER.
So this is an example of a materialized view,
so you create a view
by joining customer,
order and order item tables
provide a simpler access
to the underlying data.
So this is an example
where you create the materialized view
and you can
use Explain
also to see what Hive will execute
and then you use
a SELECT,

formatted as Hue.
So it makes you select simpler.
And now you can
play with the next exercise,
which is about materialized views.
WEBVTT
Let’s look at the solution for
the materialized views exercise.
I’m going to perform all the steps above
the solution.
We are using an IMDB dataset
and we are going to create several tables.
So there’s one movie title, one movie
crew, the movie
crew info, and ratings.
Okay, so we have several tables
to play with
to create views,
to simplify access to the data.
Those are our tables.
So we can look at the content,
for instance, of the movie crew.
Okay, now we are going to create views so,
we’re going to create two views:
movie year,
where we
just select
two columns from
for movie title
and title releases, which
takes two columns from movie title
which startYear
equals 2015.
So now we can
select
with a limit type of views.
And so we have the primary title
in both cases
and the start year and the release
start year.
We can create a transactional table
for a movie title,
from the movie_title table.
So as you can see,
it takes a little more time because
the data is converted to ORC format.
So there is some processing involved.
Okay, now this is done.
Now we are going to create a materialized view
as the select from movie_title
collection, where titletype equals
movie.
Again, you see that it’s more
than a definition in the metastore.
The view is materialized.

There is some processing.
Okay and then we get the results.
So when we do a DESCRIBE
formatted with it we have more information
than just the schema of the table.
So we have the location
and we see the table
type is materialized view
and it’s stored in the managed
part of the warehouse.
So we can
use this,
and do a query and the query is fast.
The reason being that
since it is materialized,
when we do a "SELECT *,"
it’s just a dump.
There’s no distributed processing involved.
So can we delete movie titles
from the year 2000
and then rebuild the materialized view?
So this should trigger an error
and we can check that
by copying this. Okay,
you get an error that says you cannot
update or delete records in a view.
Makes sense.
So we can
show our view
and this one,
this doesn’t seem to work.
Okay, so we try to fix this.
If I do show tables. Okay,
it works.
And my
my materialized views is here.
It’s mv_movie_title.
Now I’m going to drop the materialized view.
It’s finished.
And if I replay this one it should
in mv_movie_title
should disappear
and it did.
Okay that’s it for this exercise.
WEBVTT
In these sections we are going to study
Hive Joins.
So Hive can do three
kinds of joins: the shuffle join
which is the worst one
where all the key value pairs are shuffled
across the network;
the broadcast join,
which is a map-side join so it’s the
the trick that all those frameworks

used to reduce the shuffle, so the smallest
table is broadcasted to the memory of the
the workers
and the join takes place in memory
so there’s no need for shuffle;
and the sort-merge-bucket join is the
the thing you can do
when you have to join to large tables,
so theres no broadcast join possible.
And for this join to take place
you need to lay out
the tables in symmetric manner.
So this is the shuffle join.
So this is the one you want to avoid
where everything gets sent
across the network to perform the join.
That’s the same thing,
but in a different visualization, okay.
So we look at u4,
there are two icons with u4, they are sent
to the same reducer.
The broadcast join is
is the optimized join
when one of the tables is small enough
to fit into memory.
So in this example
it should be the customers.
So each worker
gets its copy of the customer’s data,
and the join with orders happens in memory
without the need to shuffle the data.
So that would be the case for star schemas
with a dimensional table
and more generally for when the table
is small enough to fit in RAM.
So this happens automatically.
If you set this property to true,
Hive would auto convert the join to true,
and you can control the threshold below
which this takes place
by setting the corresponding
property.
Last, so this is the last resort
for when you want to join
two large tables, you have to lay out
those tables in a symmetric manner,
and this is back to our discussion about
how you design your databases and tables
with big data.
So it’s a query driven design
knowing in advance
that you’re going to join the customers
and orders
and that those tables are huge.
You might want to create

those tables
using a symmetric layout.
So cluster by CID, sorted by CID
into the same number of buckets.
And what happens is during the join
instead of having all the
workers exchanging data,
then they do-
so we have a one too many relationships,
then you have a 1 to 1 relationship.
And hopefully the
the data that needs to be joined
will be on the same workers.
And so there will
be no need for shuffling.
But if there is a need for shuffling,
that will be only to one machine.
So it reduces the data that is
sent across the network.
So in this example,
you can see that orders
and order items are co-located,
so that the joins will happen
on the map side.
So it’s possible to do cross joins
or Cartesian joins.
You have to be careful with this of course,
and I don’t know what the use case for
this would be except to create you know
a combination of all the
possibilities for testing purposes.
So the optimization is
mainly using map-side joins.
So using the broadcasting
in the smallest table in the memory.
So this happens for star schemas.
Hive can do unions also
and unions,
the default behavior
is that union does not remove duplicates.
Remember that in distributed processing,
In order to remove duplicates
you have to add an extra stage
of processing
in which you send all the equivalent hosts
to the
the same machine
in order to only save one.
So that’s why it’s optional
and you can do subqueries also.
So the subquery has to be given a name
because every table in a From clause
must have a name.
But the columns in the Subquery select
list will be available

in the auto
query like columns of a table.
And you can see
this in the example below:
there’s a
nested query where we SELECT a+b
and we give it a name,
col,
which is not the best name,
and this nested
query is given a name quality t2.
And then you can in the outer
query, you can call the col column.
Hive contains functions that
enable you to do very basic
natural language
processing with ngram’s.
For instance,
you can do the first query,
retrieve the first hundred
most frequent sequences of two words
on my table.
If you want to provide context to that
retrieval, you can set the context
with the context_ngrams
and say, I want to retrieve the
the words in just a second,
the words that come after error code
and you have to end the array with null.
About vectorizations, so
it allows Hive to process
blocks of 1,024 rows at a time.
So it seems to be off by default
I say seems because
things change in different versions
but at least
it used to be false, off by default.
So if you’re using ORC
managed tables you can switch it to true
to enjoy better performance.
And now it’s your turn to perform
an exercise with Hive joins.
WEBVTT
Let’s look at the solution
from the Hive joins lab.
As I said, I’m going to perform
the steps above the solution.
Okay.
So first we need to join
movie_title and rating title
using the tconst key. So
we do a SELECT
and we use the dot notation for the
for the columns
so it’s not ambiguous.

And then we do the join on the tconst column.
So this is creating a test job
that is taking time to perform.
It’s not just a dump.
It’s not just a simple SELECT,
and of course, we have this shuffle
because we are doing a join.
Okay, now
we have the desired result.
Okay and let’s do another join.
So we will join the movie_crew_info table
to the movie_crew table.
So we do the join on-
oh, so this one is
the key
we are joining on a substring
of the
tconst and nconst columns.
But likewise we use the dot notation
to remove any ambiguity
in the statement.
And also it gives us nice column names.
Remember that
when you join two tables, if the
columns have the same name,
then it becomes an unmanageable form.
It did for Spark,
and I guess it’s the same thing for Hive,
so it’s-
it’s better to
remove the ambiguity from the column names.
So we can ask
Hive to give us the
the execution plan on the joined statements.
So the way you read
this is from the bottom up.
So you see that it scans this table,
and we do have a shuffle.
Both tables are shuffled.
So the movie_title.
and movie_ratings
and then the join occurs.
For the second one
let me scroll down.
Okay.
Now same thing there’s a-
we perform the substring
and then on both sides
and then there’s a shuffle
and a join.
Okay so those joins were not optimized.
So now we are going to create managed
versions of the movie
title and ratings table.
So this will take time because

it will read the tables and
convert the file studio ORC format.
Okay, this is done
and we can create
a materialized view
based on those table
with a drawing on the movie_title tconst
and the int ratings sequence,
again this is going to
perform some processing so it takes time
but we should leverage this materialized view
in next cell and experience a
snappier response.
This is done
and now,
and a snappy response we got.
Okay.
So that- it’s an investment,
it’s- you can see it’s a tradeoff.
You spend time building this and then
you have a-
you’re rewarded with performance.
So, of course, if you do this only once,
then there’s no point but
to use the data mesh vocabulary,
if this is your data product,
then you can deliver this
to outside of the organization
and they will enjoy its
fast performance.
That’s it for this notebook.
WEBVTT
In this
section, we’ll take a look at Hive
features
on windowing and grouping.
So like Spark, Hive
has provisioned for
windowing using an OVER clause,
often associated with a partition by statement. So
in this example,
the first one we select the max price
from orders,
group by CID.
So there’s no,
we are not using the windowing so we get
the result that you see on the right
and we don’t
and the number of rows is reduced.
If instead we use the max(price)
OVER (PARTITION BY cid),
then we get the
the same number of rows, but with the max
price added to each row.
So if we look at the sum(price)

and we do a ORDER BY price ROWS
BETWEEN 2 PRECEDING AND CURRENT ROW.
So for
cid 4150, the
first price is 5.99,
then we add 10.50,
and it gives us 16.49
and then we add
another 20,
and it gives us

and the last one, and for the last one
for the row is 39.99.
We add up
this plus

and that gives us 70.49.
So that's the logic of this.
Also you can use
ROWS PRECEDING AND FOLLOWING
or UNBOUNDED PRECEDING AND UNBOUNDED
FOLLOWING AND CURRENT ROW.
So about the analytics function,
so we have rank,
and rank gives you
the results on the right hand side
and that's okay.
There are no
ex aequos (equally)
and it gets more complex
when you have ex aequos (equally),
there other functions that you can use.
Then like for Spark,
we have rollups and cubes and groupings.
WEBVTT
In this section, we take a look
at Hive User Defined Functions.
So Hive defined user
functions are written in Java
and can be of three types:
the regular UDF, UDAFs, and UDTFs.
So we'll study each of them.
A UDF works on a single row
and produces a single row output.
So for instance, a lower of a string.
So if you want to write your own UDF,
you need to
overload the evaluate function
to write your business logic
and then
register your JAR
in your Hive script.
Your job must be exported to HDFS.
A UDAF takes more than one row
and gives a single row as output.

For instance, account is a UDAF
and this is a little more involved
for if you want to write your
own, you need to overwrite,
init, iterate, terminate partial, merge,
and terminate with your custom logic.
And then to the rest of the
workflow is the same,
you register your JAR
in your Hive script
and before you need to put it in HDFS.
The most complex kind of UDF
is a UDTF.
It works on
one row as input and returns multiple rows
So for this
you need to specify your custom logic
in initialize, process, and close
and the rest of the workflow is the same.
An example of a UDTF would be explode.
WEBVTT
In this chapter, we
are going to study
the specific features that are interesting
for data engineering purposes with Hive.
This is an overview of the data
abstractions that you can find in Hive.
So tables are members
of databases for a given table.
You can choose to partition
according to one or
the several columns, and partitions
work at the folder level,
and at the file level
you can choose to bucket the files.
And for the second example on the right,
it shows you that
if there is skewed data in your
in your tables,
you can use the skew feature of Hive
to create something
that is akin to a partition,
but only for the popular keys.
And then there will be one
big partition for the rest of the keys.
So you can view skew as
something that is halfway
between no partitions
and every partitions.
The idea behind
partitioning is a simple one.
It’s the most efficient strategy
for performance
enhancement, it’s "Do not read
what you do not need."

And this allows queries
to focus on the partitions
that are interesting for the queries.
So you do not scan the whole table,
you scan a subset of the subfolders
that contain the partitions. So
this is a non-partitioned table.
Okay.
So- and this is,
the way tables were laid out before
they became managed
and they had this
transaction oriented level of folders.
But an external table
would be like this,
laid out like this.
So, in the customers folder,
you would find files
that contain customers.
And since this one is non-partitioned
for a given query on the customers table,
then all the files would be scanned.
So even if we are looking at only
the state of New York,
all the files would be scanned.
So what we can do is
partition by state
and that’s a business decision.
It implies that
your queries drill down on states
often. It’s a good
investment to partition
by state. So if you partition by state,
then the query, previous query
that looked at the
customers from New York
will only read the state=NY folder.
And that’s a huge improvement
because that’s one folder
amongst 51 or 52.
So that’s a huge improvement.
To do this, you create the table
with the PARTITIONED BY statement.
Note that the state
this time is outside of the schema,
but it’s still there
when you say DESCRIBE.
It’s a virtual column
and the physical layout is the following:
You have state equals,
I guess, this is Alaska and the zipcode
equals this, and zipcode equals this.
And as you can see,
you can nest partitions. So there,
again, like always,

there is a tradeoff between
how much you nest your partitions,
and the gain you
you get from performance. So
you have to be careful
not to do too many nested partitions;
that would be counterproductive,
but this seems to be reasonable.
So loading data
into a partition table used to be manual
and painful.
Now it can be automated. So the-
the problem was to load
each data in to each partition.
For instance,
if we had a partition with New York,
you had to write this statement
to insert data
into the New York partition,
and you would have
to write the same statement
for all the other states.
Let's look at an example.
A common use case for partitioning
is to use the-
the dates.
And so if you have daily logs like this,
you can partition by date.
So if you use static partitioning,
then you will
every day create a new partition
and load the data into the partition,
using one of those statements.
There is a shortcut that Hive allows this,
this command to do the older steps.
So it's still an improvement
and if you're using a partitioning
by date, it's
something you need to do every day anyways.
And probably this is done in a script,
so that's not a huge overhead.
There is something now
available which is Dynamic Partitioning,
and it just takes care of everything.
You just insert everything.
The problem with this,
but it's the problem
that is common to partitioning
is that you should be
thoughtful about the choice of the column
on which you partition.
It should have a limited number
of unique values.
So in order to
control this, to

to avoid too many partitions,
you can set those properties.
You have the
hive.exec.max.dynamic.partitions.pernode,
so this limits the dynamic behavior
by limiting the number of partitions
created on any given node.
So you have also
the max.dynamic.partitions,
that's for all the partitions created by one
HiveQL statement,
so the default value is larger
than the previous one.
And you have also the hive.exec.max.created.files
which controls the
the number of files created by a query,
because
too many partitioning leads
to small files.
Once you have partitions
you can show the partitions for a table,
add or drop partitions.
So to summarize the
caveats on partitioning,
it's reasonable when you're reading
the entire data set it takes too long,
when you use columns on a regular basis
to do filtering, when that column
has a reasonable number of unique values,
when it's compatible with your ETL process
and when the
current values
on which you wish to partition
are not part of the data itself.
So you want to avoid
too many small files,
and this can happen
with dynamic partitioning.
That's why I presented those
three properties to control this behavior.
And now it's going to be your turn
to do the lab on Hive Partitions.
WEBVTT
In this exercise,
we are going to study Hive partitions,
so I will run through the set up first.
In the setup we create our movie's
data environment.
I'm going to run
all the code above.
Okay, that's done.
Okay, so let's look at the solution.
So first we
explore our database,
so what do we have?

We have a movie_title,
a movie_crew, a
movie_crew_info, and a ratings table,
and here they are.
Okay, so we can
query them.
For instance, look at movie_title.
Okay.
Those are the
columns in that table.
Okay.
And we see that
the movie has several genres,
and how is all this stored.
Let’s look.
Okay.
It’s an array.
So that’s a good representation
for this data.
Okay, so
let’s look at the execution plan.
If we wanted to SELECT count(*)
for movie_title
where startYear=’1999’, so,
so, as usual, this execution plan
reads from the bottom up.
So this-
this table is
not partitioned.
So we have to scan the whole-
the whole table.
Who performed the count?
We look at the directory here,
and the path is the full table.
Now we are going to create
the same
table partition by here
with the same information, so-
Okay, that was quick.
It’s just a definition,
and now we need to
load that table with the data
from the previous one. So,
so this takes time because it requires
execution.
Not just a definition
this time we’re moving data around.
But we are doing this
with a single statement.
We’re leveraging the two recent features of
Hive where we can load multiple partitions
in one single statement
and we do the same,
we try to explain the same
query

and what we want to look for is that
you see the path,
now is this path.
So it’s just a folder
that contains the movies from 1999.
So this is the
benefit from partitioning.
It’s the-
the one of the most efficient
performance enhancing strategies
which I refer to as, "Do not read
what you do not need."
So we don’t read any movies
from other years than 1999
and that’s a huge,
huge improvement.
Next, we are going to
create a new partition table
for the movie_crew information.
And it will be partitioned
by the birth year.
So as you could see
by the previous example,
the point of partitioning is-
is you use the column
that is frequently used for doing
aggregations.
So we need to load that table again.
So like previously
this is going to take some time
because data is moving around.
This is done by a Tez job using YARN
and but with a single-
a single segment as previously,
we are leveraging the dynamic
loading of partitions.
Okay.
So this is done
and if we want to see the movie_crew
info partition for birth year equals 1989.
It’s very quick.
Okay.
So why is this quick? It’s because,
if we look,
if I try to see
the execution plan of this.
You see the execution plan is short
and that’s always a good sign.
It’s just a dump
of the- it’s just a fetch,
there’s no map and reduce.
Very simple execution plan which gives
you this very quick execution.
So we can look at the partitions
of this table.

So the birth year is a good candidate
for partitioning,
because there is an infinite number
of birth years.
So next we are asked to execute a query
on the partition movie table.
So we-
okay, so we want
the job to be short.
So we are using an array_contains
because the genres are stored in an array,
a runtime of

Let’s see what this is.
Even though the table is partitioned,
you can still perform queries that
do not
use the virtual column
that is used for the partitions.
So it behaves as usual.
We didn’t filter for non-adults,
but you get the idea.
So if we want to- next if we want to delete
all the movie titles
that have a starting year of 1959.
So if we try to do this on the-
on the existing table, then we will-
you cannot do it--it will fail
because its not
a transactional table.
So in order to have this feature,
you need to have a transactional table.
So what we do in the solution, we create
a new table for movie_crew,
based on the previous one, but
also transactional.
And then we delete from this table
where birth year
equals 1959 and we check
that there are no movies
from that year after the deletion.
So again, the data is being moved around,
so this takes time.
It takes a Tez job to move data around.
Okay,
so it took
a couple of minutes.
But the result is what we expect, no-
no records from
where the birth year equals 1959.
Last we can
perform an aggregation on runtimeminutes
from a table that is partitioned.
Okay.
And that is it.

And that is the end of
this lab on Hive partitions.
WEBVTT
Now let’s talk about Buckets.
So what is Bucketing?
Bucketing is a way to co-locate data
that has the same value
for a given column.
So partitioning works on
on folders, subdirectories.
And it’s good
for the workloads of your queries
because it filters
on leveraging the subdirectories.
Bucketing is good for the,
the groupby and the join of your queries
because it co-locates in the same files,
records that have the
same value for one column.
So it uses an
old trick that has been used
since MapReduce.
Its to use a hash of the current values.
This becomes the key
by which you distribute
the records in different files.
And this is-
so this helps for the aggregation part of your,
the reduce part of your queries,
because it
diminishes the network traffic,
because your records are not spread
over all the files,
but are co-located on specific files.
So for instance,
when you do a join
and this is something that
Hive can do
if you arrange your tables accordingly;
so there is some
preparation involved.
You have to lay out your tables
using the same key, the same
order by, and the same
number of buckets that are
equal or a multiple of the smaller ones.
But by doing this then so,
instead of having one to many
relationships, you got a 1 to 1
relationship between the bucket
that contains your value
for the key you
are joining on, on both sides.
So it diminishes the network
and as you should know by now,

the network should be a concern
and that's a good thing, and that is a
one of the unique
features Hive, to be able to do these
sort, merge bucket joins
because for instance, Spark
doesn't know how to deal with buckets
so there's no, it can do a sort merge join
but not a sort, merge bucket join.
So how do you create those Buckets?
You add the statement CLUSTERED BY,
the name of the column
on which you want to do the Bucketing
and the number of Buckets.
Yeah.
So you want to have Buckets
of similar sizes.
So to avoid problems with
skew, so you want the data
to be well distributed according to these,
the values of this column.
And for the rest of the
data processing it, there's no changes.
Okay, you just insert and select
the way you always do.
So one thing you can use
with bucketed tables is a table sample.
So that you can select,
for instance, one of every 10 records
using the syntax that you see
in the second select statement,
and that's it for Buckets.
WEBVTT
Let's talk about Skew,
so Skew is one of the major
challenges for distributed processing.
It's when data is not evenly distributed
and that creates an unbalance in the-
in your cluster, meaning that one worker
is doing more work than the others.
So it makes your distributed
processing less efficient.
It turns distributed processing into
single processing;
it's something you need to try to mitigate.
So Hive has limited support for mitigate.
Skew, the best tool so far
for mitigating skew,
it comes with Spark3.
But okay, let's see what Hive proposes,
so what you can do is-
so you have two strategies:
you have either runtime of compile time.
So if you know that
your data is skewed for a given value,

then you can create your tables
accordingly with a create table Sales
skewed by storeID
on 500, and that means that you expect your,
you know, it’s not an expectation.
It’s knowledge, you know, in advance
that there will be more records
for this storeID,
because maybe it’s a very popular store.
So you design your table accordingly, so
this is the draw back of this strategy.
You have to know in advance your data
and it’s hardwired
in the definition of your table.
What you can do
if that’s not
something that sounds good to you.
Then there’s the runtime strategy,
where you set a threshold
for being popular.
And this, again,
is not completely satisfying
because this also is hardwired,
so in your scripts,
you put a number
and it’s in your scripts
and this is something that, to make sense,
you would have to tune,
for each of the, nearly each,
of your queries, okay, to avoid-
because you cannot set a threshold
for all your queries.
So this is something
that you need to tune for your
your queries and that would be hardwired
in your scripts, meaning that
if your data changes, evolves, grows,
then this value needs to be changed.
So it’s a high maintenance code.
So what Hive will do is
do a mapside join for the popular values
and then do a common
join for the the remaining values,
in both cases.
And that’s it for working with Skew.
Next, I will do a demo of
what it does to your tables.
WEBVTT
In this demo, I’m going to demonstrate
how Skew works with Hive.
So I will be using the static strategy.
So I need to do some setup,
I need a Kerberos ticket.
I’m going to
create a database,

set some property.
I think this needs to be removed.
So I’m setting up the
data context for this demo.
Going to use salarydata.
So I’m creating a table
first called salarydata,
on which I,
I load the data and then I create a
second table called skew_demo.
And the second table is skewed on two
zip codes,
which you can see here:

Okay, so now we have our two tables.
We can insert data into this,
the skewed table.
Okay.
So again, this involves moving data.
So it took some time.
Fortunately, there’s not a lot of data
and now...
So I answered that question, it’s-
it’s in the table definition
and this is one of the drawback
of this strategy.
It’s static, but look what happens.
So the table is in this skewed folder
and what you have is a three subfolders
one for the-
for this-
this zip code
and another one for this other zip code
for which there is skew,
and the rest of the record are in a common
folder, a default folder.
So you can by looking at this, you can
look at skewing
as a halfway measure between no-
no partitioning and partitioning on-
on the popular partitions.
That’s one way to look at it, also.
Also something to note it’s-
that I used the external table, if I had
used a managed table, then the
this would not have worked
because the managed tables have
a layout of subdirectories
that is a transaction oriented
and this orientation
would have prevailed over this skew,
the skewed orientation.
So yes.
Next we check that
we can still get the data

even though it’s spread around
amongst several folders.
But that’s not a new feature.
That’s not specific to skewing,
but still nice
to know that it works
and that it really for
for this demonstration of skewed table.
WEBVTT
Yes in this
section we are going to study SerDes
and how to use them for ingesting text data.
The text format is a very important format
for big data.
It’s the native format for a lot of data
sources, and it’s important
to ingest those data sources in a database
like Hive to be able to query
and handle those information,
to extract value from them.
So one of the
key parts of data engineering
is to take unstructured data
and make it structured. That involves
usually regular expressions, you can do
this programmatically using spark,
but you can do this with less code
with Hive using
SerDes. So whether
and so the sources of text
information, of text data are well known.
You have log files,
tweets with JSON, emails,
we’ve got the product previews, text
messages, all kinds of
very common and big data,
sources of data.
So far we’ve only used the row format delimited
system for creating tables.
But this is a shortcut
for more involved syntax
which shows that Hive is able to
read and write data
of any format, provided you
give it the Java class to serialize
and deserialize on this format.
The serializer/deserializer is commonly known
as a SerDes.
And Hive has several built in SerDes to,
to work with text files.
So the most common
one is the LazySimpleSerDe,
but you also have a RegexSerDe,
then you have also an OpenCSVSerDe and a JsonSerDe.
And this covers

most of the text formats that we use
with the exception of XML.
It’s not built in, but it’s possible
also, there is a SerDe available.
It’s not shown here,
but there’s one available.
So instead of writing what we’ve done so far
with this convenient syntax:
ROW FORMAT DELIMITED
FIELDS TERMINATED BY.
We could say, ROW FORMAT SERDE
and provide the Java class name
that does the work.
Okay, so let’s look at an example.
This is a log file
and it’s unstructured data.
Well, there is some structure.
We can see the structure,
but it’s implicit--it’s
not explicit.
So we can use Hive to
extract the information
from this unstructured data
or project a structure
using a regular expression.
So we are using the RegexSerDe,
we provide the pattern
for the regular expression
and this will give us
the separate
columns from these the records.
So all you have to do is what I say,
all you have to do, but
well you have to be prepared to acquire
some skills in the regular expression
when you work with text data.
But this is a very elegant way to
ingest such data,
and this is the result.
So we have clean, structured data,
and this illustrates my motto:
the difference between structured data
and unstructured
data is a set of well-crafted
regular expressions.
In this instance, there’s only one
regular expression, so
it’s worth the effort.
Another popular format
would be fixed-width format.
Okay.
And for this
we can also use RegexSerDe.
So the regex pattern is in,
it’s in blue here.

And you see
at the top of the slide
you have
something that is difficult to handle.
And then by the magic of Hive,
you have something
that is very convenient to handle,
or at least that we are very
used to handling using SQL.
So the CSV format can be tricky because of the
there can be embedded commas,
or quoted fields or missing values.
So there’s has been a CSVSerDe
for processing CSV data
and that should be the one
you should be using. So
you see here in this example
for the third record,
there’s a nested comma
in the- after Bitmonkey.
And so if you’re using
row format delimited, then
this will give you a bad record.
But using the correct SerDe
this will give you the correct
record. And that’s it
for ingesting text data with Hive.
And next we are going to do
an exercise about this feature.
WEBVTT
In this notebook
we are going to study how to leverage the
text analyzing features of Hive. So
I’m going to
execute the set up.
So the context of this exercise
is a ratings table,
so ratings information that is,
that contains a message
that would be
text format.
And then we have products,
so have the products-
products table and ratings table that
that are linked by
a foreign key.
And then there’s the customer
also. I go to the solution,
so we create the web logs table first
and you see this is the regular expression
that is used
to extract from the each weblog
the separate informations.
Now we load the data into the table
and perform a query on this.

So we check that
the data is correctly loaded.
So we have a nested SELECT
and we are using regular expressions.
So what we’re finding- so
the most popular
search phrases are "tablet," "ram," and "wifi."
So let’s look at ratings:
so we have the rating is made by
a customer, points to a product,
and there’s a number and then a message
that is a string.
So to find the product that customers
like most, okay,
so we get the product ID
and the average rating.
And so okay, this is the product.
And so now we’re
looking at the consistently least
liked product.
So we set the threshold to have fifty ratings.
So for the time being,
we are not working on text,
but we’re
first finding the least liked product.
Okay,
so this is the product we have found.
Okay. So it’s a classic
SQL problem/challenge to get the least-
the least
well the navigation, but to retain
the product ID at the same time.
So that’s why we have a nested select.
Okay, so what’s the deal with this
product?
So what we can use
to get a feel for what’s wrong with this
product, is to look at the messages
and maybe
look for the bigrams using the ngrams
built in function.
So on this cross analysis we,
we infer that the product is
deemed to be too expensive,
and so we can confirm this.
That’s an intuition.
So let’s look at the trigrams.
So we just
changed the two by three
in the previous query.
Ten times
more than the other is too expensive.
This is true. This is way
too expensive
and if I-

okay let’s- let me try to do this.
If I change by five:
Do they charge so much?
Why do the charge so over? Must be a mistake.
This overcharging is crazy, other stores sell it for...
Okay, so something about the price of this
item is off.
So what we can do also instead of using
bigrams or trigrams or pentagrams,
I just made up this one.
I hope it’s correct.
We could do a- use LIKE,
so we get full sentences this time.
Why does the red one cost ten times
more than the others?
So, so it seems that
there is a red,
the color red creates a problem.
So why don’t we filter the messages
by using this product ID and also
when they contain the word red?
Okay.
So we have those messages
that confirm that it seems that the red-
the red version of the product
is ten times more expensive.
So let’s look at the
products here.
So it’s a flash drive,
the 16 gigabyte hard drive.
Look at the price.
So is that dollars?
I guess it is.
Well, it must be cents.
Okay.
But still it’s a,
so you see this the blue and green
of a price of 4299
and the red one has a price of 42999.
So there must be a problem with the,
the price of the red one
and that’s it for this exercise.
WEBVTT
In this chapter, we are going to study
the complex types that Hive provides
and how you can use them to
denormalize data.
So those types are, there are three types:
you have the array, the map, and the struct.
And if this sounds
familiar or similar to what Spark
proposes, it’s because
they are aligned.
It’s on purpose.
So they are the same ones,

so they have the same use cases.
And why would you use them?
For the same reasons,
you want to avoid doing expensive joins
by folding up one too many relationships
using the appropriate
representation: can be an array, can be
a map, can be a struct, or can be
a nested structure with a for instance
an array of maps.
What’s the difference between them?
The array
is for storing similar things.
So for instance,
we had in a previous exercise we
we stored
for one given movie.
We had an array of genres.
So that would be a good representation
of this information.
A struct is for very structured data.
So if you expect,
always a first name and a last name
for a customer,
then you can create a struct
to hold that information.
And the most flexible one is the map which,
expect any number of key values.
And it’s a
it’s your gateway to know
SQL, if you have
data that needs to be represented
like this, then that’s
your only solution is a map in Hive.
So here’s an example using phone numbers.
The phone number is a good example.
So usually the SQL way of
doing- of storing this information
would be to have a
a phones table and a customers table
and then you would have in
every query that request
phones from customers,
you would have to do the join.
That’s the way
we have been taught to do things.
But this is a very expensive
joins for not a lot of value
because we know when we selecting a list
that we are going
to retrieve those three records,
so we know what’s going to happen.
It’s not, just a convenient way to,
well, convenient,
I think the idea at the time was

there were two motivations
one was data integrity and one,
and the other one was the cost of storage.
So you've heard me say
before that
the cost of storage was not the
really a motivation for
the design of Hadoop,
and about data integrity,
it should be the role of other
software layers before reaching Hadoop,
or maybe it can be done
in data engineering before,
before serving the data to end users.
But it's not something that should be
a concern for the storage engine.
When you use this kind of representation,
using satellite tables
and doing a lot of joins,
it becomes really expensive in
in big data
because of the cost of the shuffle.
So one way to do this with Hive
would be to use arrays
and it makes a lot of sense.
And then you can access the members
of the arrays using a square bracket
and a zero index notation.
And there's no join involved in this query.
So how do you create an array
in your table? You do a,
phones is an array of string,
and you have to provide
COLLECTION ITEMS TERMINATED BY '|',
and physically, you see
the way it is stored,
and it's a trick
that people have been using
also for a long time:
it's to store
one too many relationships in strings
with a separator.
Okay.
But this time it's not done manually,
but it's done by the engine itself.
You can use map, and you see the
benefit of map of arrays.
Then you can
provide a semantic value to the key.
You can say this phone number
is the phone from the home,
for the work, for the mobile.
So that's
added value at the expense maybe,
of some performance--I'm not sure.

And the syntax is readable also.
So that’s a good solution.
Okay.
It provides added value.
So how do you create a map in your tables?
By using the keyword map
and as you see it’s the <STRING,STRING>.
So it’s it could be anything.
We used home, work, mobile,
but we could have used color
or make or model.
Anything goes into those maps.
So that’s why
I say it’s the gateway to know SQL
and it’s using the same-
well, there’s an added statement
for the map key,
so you see it’s using the pipe to
separate the records and the colon
to separate the key from the value.
And you can use also structs.
So that’s the last one.
So it’s the less flexible one.
But if your data is very
consistent, with has always the same keys
and the same number of values,
then why not use this?
You can use the dot notation with this,
so it’s quite nice.
And again, you avoid a join.
Yeah.
So you can include complex
columns in the select list in Hive queries.
We’ve seen that because we use that
also, yeah.
You can use size on your complex types
for arrays and map.
And if you want to retrieve
individual rows
per member of arrays or maps
you can do explode.
So what can you not do?
Always the same challenge.
Get a value and also one column.
So the way you overcome
this problem is by using a lateral view
where you explode the phones and then you-
you get the name from the customer phones
and the phones from the lateral view.
This is a specific to Hive,
Impala has a better way
to handle this problem.
So to load data containing
complex types, what you use is insert.
Or use create table as select so

it’s business as usual.
The reason for which we study
Hive and not Impala
in this class about data engineering
is because Impala cannot
insert data containing complex types.
So it’s okay for querying.
But when doing data engineering
you want to insert and Impala
cannot insert data
containing complex types
at the time of this recording;
things change.
The main reason you should use
those complex types is to denormalize
the one too many relationships
and thus avoiding
costly joins with a limited value.
You can see you can nest those
those complex data.
So here we have a rated products table
which combines the
the products and it’s ratings.
So the ratings is an array of structs.
So that’s quite powerful.
And we talked about transactions,
and we said the scope of this transaction
can be only one table.
But when one table equals several tables
in a normalized
equivalent,
then you can look back at transactions
and say maybe that’s quite,
it’s quite good.
It’s not something to be dismissed.
It’s if you are using all those features, then
you do transactions on
maybe a you can do star (*) schema
for instance, if you denormalize your star schema
and then you can do transactions on the,
this big table.
So to populate a denormalized table,
you have to-
so you use named_struct
to cast a row of the detail table
into the proper structure
and then do a collect list
to get an array. So,
previously we had a rating
table and a product table,
and this query gives us a
rated_products table,
the definition of which we just
saw in the previous slides
and now we’re going to

study this
in the next exercise called Complex Data.
WEBVTT
In this exercise, we are going to look
at some examples of using complex data.
As usual, I’m going to run the setup.
So we will be using some
data from the dualcore example.
Okay, so what was asked was to create
a table with the- for the loyalty
program. And this table contains
complex- three complex types.
There’s a map for the phones
and then there’s an array of
other values which are structs.
So it combines all the complex
types into one single table.
And if we look- examine the data, it
should map the structure of the table.
We load the table- we load the data
into the table, and let’s run a query
to check that everything went well.
So we are selecting the phone from
home from one customer and works.
We can also, since there’s an
array of structs for the order IDs,
we can select one from a customer.
We select order ID 2, and
it’s the third order ID because
the array is a zero index.
Our data is
correctly loaded into
that complex table.
So for the same customer, we’re
going to look at the total
attribute from the other value field.
So we want to return the
order ids for customer, for
one given customer, one per row.
So we are using explode of order IDs.
And we see the order IDs for
this particular customer.
Now, if we want to retain
the customer ID it’s the
classic challenge of SQL,
you have to do a lateral view.
So we have the order IDs on
one side, and we join this
with the customer ID, and we
just filter for the other value
average being greater than 90,000.
And that’s it for this exercise on complex data.
So you see that in one
real life example,
we combine all the three

complex types in one single table,
and we were able to query the table
and minimize the shuffles involved.
WEBVTT
In this chapter, we are
going to have a look at the integration
between Hive and Spark.
Hive and Spark are an old couple;
they have been together for as long
as Spark existed, I think,
and there have been
several chapters in that story.
Initially you had,
so if you initially means
when the Spark version was
less than the Spark 2, you need-
needed to have a SQL context to access Hive.
But you could do pretty much
everything with Hive.
There was no barriers.
Everything was accessible, so
I will move on swiftly because
I expect nobody to be
in that version of Spark:
smaller than Spark 2.
Then after Spark 2 came
the session object
which merged the SQL context and the Hive context, so
previously you had two flavors
of SQL context,
the Hive one and the SQL context.
So they merged into the Spark session.
So you had to
create your own Spark
object, Spark session object
and you could use the Spark object to do
Spark SQL.
You all saw in 2.3, Spark added
the support for ORC so you could
use ORC formatted tables.
And then came the time for the
Hive warehouse connector.
So when Hive introduced
ACID tables
then to access managed tables
you had to have a Hive warehouse connector
and provided you had this
object, so you had- then you could
work with your managed tables.
WEBVTT
In this video, we're going to
experiment with Spark and Hive.
I'm going to
execute the set up.
So we are using some data generated

by TPCDS
benchmark kit.
Okay.
Okay.
So we first created a database
using Spark.
So this is the first cell that uses Spark.
So there's a initial cost of establishing
the Livy session.
Okay, so we are done.
So we create this external table
for the customers,
and we load the data.
Now we have this external table.
We want to create a managed table
using a CTAS statement
based on the previous
external table.
So this is how we do it.
So data is being moved,
so there is some processing done
that it's not just a definition,
it's actually processing.
We can check
that the customer external is not empty.
So this is the data
that has been generated for
by this TCPDS benchmark kit.
So it's
random data.
And we can also check
that the customer managed table
also has data.
So everything looks fine.
Let's check
the create statement
from the external table.
It says create external table.
Okay.
And the location is the one we expect.
And that's all good.
Let's check the
managed table.
And what we see is that it's
an external table.
We see it here.
Create external table,
and we see this property
translated to external equals true.
So although we asked Spark to create
a managed table
because it could not,
it created
without any message,
without any notification,

it created an external table instead.
Okay.
So this is the problem
that of the lack of a Hive warehouse
connector that is being solved
currently but not available
for all the environments.
Let’s do an experiment.
So if we really want to
create a managed table, we can use JDBC
to tell Hive to create the managed table
and we can
check various aspects of this.
And we have a- you can see the table here.
And we can have a look
at the create statement
and to create table, it’s
not to create external table
and there’s no property saying,
like previously
the property
that said translated to external,
there’s no such property.
But instead we find the property
transaction equals true so that-
that’s really a transactional table
stored as ORC.
So that’s all- that’s all good.
So now
if we try to access this table
via SPARK, we get this error message
and it’s quite clear
we are lacking a Hive warehouse connecter for Spark 3.
In this environment
again, it’s being developed
or released for certain environments,
but not all.
Can Hive have access a table it
created using Spark? Yes.
Okay.
So that’s why there is a maybe there’s
still some use for
Hive scripts to create managed tables or
things like that that you cannot do
unless you have the latest version
of Spark and Hive.
So depending on your version
of Spark, it’s-
you can always use external tables,
but maybe not managed.
tables and that’s it for this lesson
part and now you’re going
to do the next lab underneath.
And I will
demonstrate the solution afterwards.

WEBVTT
Let’s go through
the solution of the previous exercise.
So we are going to create a
staging database first.
And in which we are going to create
a staging table for customers.
So let’s see if we have data
in that table.
Let’s check; it’s okay.
Next using JDBC this time, we create
a prod database
in which we create a prod table.
With a CTAS statement
based on the previous staging table.
So again, this is not just a definition;
it’s an execution of a
this time Tez jobs.
Okay.
And let’s confirm
the successful creation of this
managed table.
Yes, we have data inside.
So this gives you
an example of the workflow
that you should use,
if you don’t have the Hive warehouse connector
to create and load a managed table.
WEBVTT
So distributed
processing comes
with its own set of challenges
which are different from non
distributed processing.
Let’s talk about shuffle
and using a very common type of query
where we do a group by count.
Let’s see all this distributed processing
that tackles these kind of queries.
It applies
rules best practices
of distributed processing.
The first goal is don’t create
what you don’t need. So
it’s the first because
it’s the one that comes the most upstream,
so it’s the most efficient.
And Spark uses push down predicates
to avoid reading parts of the files
that one won’t be necessary to the
the completion of the query.
That is done, next
if there are still things that need to be -
that can be filtered,
the rule number two is filter early

so the work load is applied, if you will.
Next you project early.
So you apply the SELECT
and this part of the processing
of the query is the most efficient
one: it’s the one that uses
parallel processing when the file is,
the files are split before the
where it’s using as distribution.
So there’s no skew, and there’s
no shuffle, there’s no network involved,
there’s no need to regroup the records.
So everything happens in memory
and it’s really fast.
So that’s where distributed
processing in Spark shines.
Next comes the GROUP BY.
Okay, so that’s the part
that is messy for distributed
processing because it puts
a stop to the pipeline that occurs before
and the data that comes
out of- what is called the first stage
needs to be materialized in memory.
Previously it was not.
It was materialized but it went through
the pipeline of processing.
But now it needs to be stopped
and materialized in memory.
So it can lead to out of memory error
messages,
because there you put pressure on
the memory of your workers.
But to be shuffled
to other machines
it needs to go through the network.
So before going through the network,
it needs to be spilled to disk.
And there you may have problems
with your local file system.
It may run out of resources.
So now you put pressure on your disks
and eventually,
then you need to use the network
to shuffle the records to their destination
after the shuffle.
So you put pressure on your network
and if you’re using
YARN you can become the noisy neighbor
by grabbing all the network bandwidth.
Yes, the shuffle is messy.
It’s one of the main pain
points of distributed processing,
and it’s also something that you cannot avoid.
Well,

you can avoid to a certain extent by,
for instance,
distributing small tables in memory.
That’s the very old trick in the book that
Big did
and Hive does and Spark does. But
when you cannot do that then you have
the problem of the shuffle.
WEBVTT
Next comes another problem and it’s
Skew. So when you distribute your data,
when you’re doing distributed processing,
what you have in mind
is the way a car engine works.
You distribute the gas to all the cylinders
and everything
flows through the engine
and you get powerful-
you get power from the engine.
The problem with data is
when it’s has some business
logic inside,
it’s going to be skewed more or less,
but it’s going to be skewed.
So it’s like your cylinders will not get
the same share of gas.
So you’re running on a fewer cylinders
than you would expect,
and that’s never a good thing.
Another problem that is for
Spark is after the shuffle,
how many partitions should we use?
If you’re using data frames,
the catalyst optimizer is clever enough
to identify the number-
the ideal number of partitions
at the beginning of the query.
When all the information is known,
you have the,
you know, the size of the data,
the distribution of the data,
the hardware that you are using;
but after the first stage, you no longer know
the distribution and the size of the data
because you would have to apply
the processing to know that. In an older
version of Spark, when the static-
when the execution plan was static,
then you would have to use a magic number
and the default value was 200.
But like any magic number,
it’s never a good number.
So with Spark 3 there’s a new engine
that makes the execution plan dynamic.
So this problem is solved.

But you still have this,
if you’re using old Spark,
then you still have the problem
of how many partitions should be used.
And what if the data is skewed
for this GROUP BY key?
Since you’re doing a GROUP BY
and using data
that is not randomly generated.
It reflects your business
and you don’t have even business.
For instance, in all the states of the US
or if you doing group by hour of the day
then if you’re using,
if your business is like Uber,
you don’t have the same number of rides
by hour of the day or day of the week.
Any business data has some skew.
Really important it can lead to
problems or crash in your cluster.
For for instance, here
you see at the bottom
you have the 200 number.
So the 200 comes from the default value
for the partitions
after the shuffle.
But one partition is so large
that the tasks
for this partition never completes.
So you see you get an insane duration
and it’s
because you’re doing the same thing.
So you should know by now
that collecting data from your workers to
your driver can be dangerous
because you overwhelm your machine by
retrieving data
that should be larger than what
one machine-
one single machine can store
or handle.
But you can do the same thing
by using skew.
If you have lots of customers
in California and you do a group by state,
then you’re going to
send all the data to one executor
that will tackle
California state partition.
But this worker will be overwhelmed
by the size of the data
that is going to be sent to it
and that is why it can store your,
you know, the whole job.
WEBVTT

Last, and it’s a less acute problem for distributed processing,
but it’s still worth mentioning.
The problem is Order,
since we are using a share-
nothing model between executors,
we cannot compare records
that are in different JVMs.
So applying Order is
difficult; it’s a challenge. You need,
you need to have the right set,
you want to compare in the same JVM.
So you cannot compare a very large set of records,
a set
that would not fit into a single JVM.
So that’s one problem,
but it’s not a very acute problem
as I said, because most of the time
your business
stakeholders
don’t want to order billions of
records they want to know
the top ten or the bottom ten.
So that’s not a very acute problem.
Another
manifestation of the challenges of
order in distributed processing is
if you’re used
to a non distributed processing,
you can rely on the implicit order
of the records in your files.
But when you’re processing-
you’re doing a distributed processing,
first, your data will be
shuffled at the beginning by the cache
that is done by the split.
So your order is already
at the very beginning of the processing.
Your order is lost, and then
eventually at the end of
your processing
when you’re doing a write,
each executor will write its
part of the result
and you cannot control the order of this.
Whichever executor finishes
first will write the first file.
Okay, so this first file is written by the
first executor that completes its part,
and there’s
no way that you can control this.
So if Order is implicit-
important for you,
you should make it explicit.
So that you can manage it
even in distributed processing.

WEBVTT
In this
chapter, we’re going to study our Spark,
implements Distributed Processing.
So partitioning,
happens
at the beginning of processing and splits
the files
on which you want to apply processing
in partitions that your,
that will be spread
across your executors.
It used to be something that you would
control, that would be of a concern
when you were doing RDDs, using RDDs. Nowadays
if you are using data frames,
then partitioning is done
automatically,
and the number of partitions is also
something that
Catalyst determines.
One thing to note is the
the number of partitions control
the parallelism of your execution.
So in this instance,
if we have three partitions,
then we are going to use three executors
and maybe that’s the
right number of partitions,
maybe not.
There’s such a thing
as the right number of partitions,
and that’s something that Catalyst
can identify
at the beginning of the processing
for the first stage
of the Spark execution.
If you’re using RDDs, and really you,
you shouldn’t,
there’s no optimization.
So your files are in,
let’s say in HDFS
and they are spread across blocks.
The blocks are uploaded into memory
and theres a 1
to 1 relationship
between a block and a partition.
You can wind up with a lot of partitions
because if you’re using small files,
if you want to know
the number of partitions, you have to
query the RDD.
So there’s a method that is provided
for RDDs called getNumPartitions.
And if you’re using a dataframe,

you have to get to the underlying RDD
and do the getNumPartitions.
So if we look at the sum processing,
so it’s going to use RDDs,
we want to compute the average word
length by letter,
we read from a file that creates an RDD,
and then we apply a first processing
which splits the lines
using the whitespace.
And for the result of this,
we apply another lambda function that
creates a key value pair
with the first letter of the word
and the length of the letter.
So we have the input
to compute the average word length,
but so you need to group by the key:
the first letter.
So in our alphabet
you have 26 first letters
and talking about skew
depending on the language
you are using, some letters will be less
represented than others.
So there you go.
Once you apply some business logic,
then you have skew.
Here we decide that we want two RDDs,
two partitions after the shuffle.
So each partition will tackle 13
letters and maybe that’s the right number.
I don’t know.
But the fact that you are grouping- instead
we could have said group by 26, but then
the letter Z would have been less busy
than the letter A for instance, in French.
So grouping
is one way to mitigate skew
because you can expect maybe
popular first letters to be
to be grouped on
the same executor
with less popular first letters.
So it’s one way you can
mitigate Skew
and then we can continue the processing
to compute.
The average word length by letter.
So by using this example,
we can see several data
abstraction that are used by Spark.
This does the task,
which is a series of operation
that work on the same partition

in our pipeline together.
Those tasks can be grouped
by stages and stages occur
between shuffles,
and the job consists
of all the stages that make up a query.
So there’s limited optimization
when you are using
RDDs and again, really
nowadays you should not
and you should consider if you have legacy code
with RDDs consider this
to be technical data.
So this is what we can track in the Spark UI.
We can stages task and jobs.
This is the first stage.
It’s what occurs before
the one
and only shuffle in this query.
So it’s composed of the flat map.
The map.
Okay.
And this occurs in memory
and it’s pipeline together,
and the second shares the groupByKey
and then theres the map and maybe after.
If we want to say yes, there’s the table.
So we have two
stages in that query
and a task(?) pipeline
when a line is
read from a file. It goes from
in the stage zero, it travels
to- from the flat map to the map.
And then eventually, after the flat-
after the map,
it needs to be held in memory.
But otherwise each record
is pipeline so the result of this
is not held in memory.
Okay.
You have this in the-
in the file and then after the first stage,
you have those key value pairs in memory.
And this is ephemeral in memory.
It is only materialized
to become this eventually.
Okay, so we have two stages,
and in those stages
we have tasks that are pipelined.
So this is the representation
that you can see through the Spark UI.
So to summarize, you have a job made of
several stages separated by shuffles
and inside each stage you have tasks

that are pipelined.
You can- Spark creates execution plans
that you can
ask to see.
So for Catalyst,
this execution plan is optimized,
so it goes through the
Catalyst optimizer
that applies the best practices
of distributed processing recursively
until it can no longer optimize
for RDDs.
There’s no-
there’s no optimization
or there is limited optimization,
there’s some caching that occurs,
but by and large there’s no optimization.
So the result of this is that data
frame processing is
always faster than the RDD counterpart.
When tasks
do not require
data to be regrouped
and can be pipelined.
So it’s narrow dependency,
so this is narrow dependency.
And when there’s a shuffle,
then there’s a wide dependency.
And so those are the expensive, expensive
task, reduceByKey, join, and groupByKey.
This is expensive.
I told you that Catalyst
can identify
the right number of partitions,
and you see for the same task,
the picture in the middle,
is when you are using RDDs on 181
small files, they are loaded into 181
partitions,
whereas the right number of partitions
for this given job
according to Catalyst is 7.
So you have 181 against 7.
And so the number of partition
is critical
for the performance of your jobs.
Okay.
And the way you can
assess this graphically,
you want to see green in the diagrams.
Green is good.
Green is computing;
the rest is not computing.
You see that at the bottom--
so you have two partitions,

and that’s very efficient.
Most of the time is spent on computing.
The first screenshot,
you’re not using distributed processing.
So you’re not leveraging your cluster
and there’s a lot of
deserialization going on
and scheduling delay also.
For the one in the middle,
there’s just too much overhead
because of the number of partitions.
So you should be using
data frame because of Catalyst.
And until Spark 3 Catalyst, the execution plan would be final.
But now with a Spark 3
and adaptive query engine,
the execution plan can be dynamic.
So what happens is that
at the boundary of each stage,
Spark will pause the execution
and see if there are opportunities
to apply new optimization and also
compute the right number of partitions
on the other side of the shuffle.
So you don’t have to say
groupByKey(2) for instance, which is
that’s the magic number.
It’s hardwired in your code.
So it’s a good thing
that the number of letters
do not change in time.
So it could be
not ideal, but
you could forgive to put a number here.
But if you’re in a use case
where your data grows,
then you would have to change your code
when the data becomes too large for this number.
Okay so Catalyst works on- relies on the transformation names.
So it knows, okay, this is a select;
this is a filter.
So it’s a filter.
Okay.
I can push it back and can apply this
at the beginning of the query,
but this logic is broken
if you introduce lambda function
because Catalyst doesn’t know
what’s going to happen
in another function, there’s no queue
as to what
is going on inside the lambda function.
So Catalyst will refrain
from optimizing your
your processing, so use the data frame

API with the other methods.
But do not put inside of them
lambda functions.
It’s possible
it will be valid
syntax, but it will
jinx the Catalyst optimizer.
So the Catalyst
optimizer goes through several plans.
First its parsed, then it’s analyzed, then it’s optimized.
And then there’s the physical plan.
Afterwards,
after the physical plan,
there’s another optimizer
called Tungsten that
finds the best code
for your physical plan
according to the hardware
that is available.
So you can view
the execution plans using explain
or in the Spark UI,
there’s a, if you look at the SQL tab,
there’s a details link
that you can click,
and it will show you the execution plan.
If you’re using Spark Adaptive Query Engine,
if you’re doing an explain,
it will show you the initial
execution plan.
So it may not be the real execution plan
because when you do an explain,
you’re not executing.
So the real execution plan
in the context of AQE,
that will be known after the execution.
So it will be available
in the details of the SQL tab.
So let’s look at one execution plan.
So what we’re doing here is a join
between two
data sets, people and pcodes.
Okay.
So the first level is
the parsed logical plan.
It’s parsed so there’s nothing to note.
Then it’s
using an analyzed logical plan, just one step.
There’s no optimization yet.
Okay,
then the optimization happens.
And yeah, so
we are using data frames
and look at what happens.
Did we specify anywhere in

the code that we didn't want the,
the join key to be
not null?
No, we did not.
But Catalyst knows that
if we're going to do an inner join,
we don't want Nulls.
So it adds this to your code.
And then there's the physical plan
and here we can see several things.
Each file is read,
and there are push down predicates
that are applied to avoid reading Null
pcodes in both files.
Okay,
so you read from the bottom up on
those plans. So it avoids reading
Null pcodes on both files,
and then it does a broadcast exchange.
So what is this?
It's a way to minimize the shuffle.
It sends the pcodes--
which is the smaller
table of the two--it sends the
the pcodes to the executors
so that the
the join that occurs at this-
can I say stage?
At this step of the
processing happens in memory.
So it doesn't completely
remove the
use of the network bandwidth,
but it makes it more rational
because we are still using broadcast,
so the data is being sent
across the network but then the
the join happens in memory.
In the Spark UI, you have the
SQL tab that you can use to
look at your queries.
And you can visualize
the D-A-G or DAG of your query.
And this is the same thing
visually.
So if you have an optimization like
the one we just talked about with
a broadcast exchange and a broadcast hash join;
you have this asymmetrical figure
where the- the small file
is higher than the big file.
This if you see this, then you
you can be assured
that you have an optimized
transformation.

This is as good as it gets.
You just need to look at the details.
You see this asymmetrical shape, say, "Okay,
my join is optimized."
If it’s not optimized,
then you have a symmetrical representation
of the two tables,
and you don’t have the broadcast exchange;
you have an exchange. So
after Catalyst,
there’s another optimizer that takes place
and looks at the hardware
that you’re using and tries
to leverage the best-
to make the best of your hardware.
It also serializes the data in the
into the Tungsten
binary format, which is a
a cool format that is
much smaller than the Java native format.
Bear in mind
that Java was a design at the time
where big data was not a concern.
So it’s very verbose.
Tungsten binary format is much smaller.
It’s also for Scala and Java you can,
it support operations
without the deserialization
so this is a very cool feature also.
It’s not available
for piSpark because for piSpark
the data needs to be deserialized
into its
Java format to be processed
and that problem is
for when you’re doing UDFs.
And it can be
that format can be stored off-heap,
and thus avoiding
the garbage collector problems.
But the default is on-heap, so
if you want to turn there’s
a configuration that you can use to turn
this off-heap,
but it’s at your own risks.
WEBVTT
In this notebook
we are going to study the query execution
and use the Spark UI.
And as usual I am going to execute
the setup steps.
So one thing to note here is that
with this version of Spark,
the Explain was broken,
so I had to add this cell.

So we are using the loudacre data.
We have our Kerberos ticket.
So what I’m going to do
when this is going on, I’m
going to open in a new window
the Spark UI.
Okay, so I’m going to my cluster.
Okay, I need to go to the
Resource Manager
UI and look at the applications.
This is the one that is running;
I’m going to drill down on this one
and click on Application Master
in a new window.
Okay, so this is my Spark UI.
I’m going to make it
smaller so that both can fit
on the screen.
What I like to do is to have the,
the Spark UI on one side
and the Zeppelin Notebook
on the other side.
So my Zeppelin Notebook should be somewhere here.
We’re going to
resize the window, and I’m good to go.
So, let’s
look at the solution.
So for the time being,
I have no jobs in my Spark UI
and if I look at the event timeline,
well I started a driver,
and then I added two executors.
Okay, I’m using Spark 2.4 for this exercise,
but it doesn’t really matter.
Okay, let’s do some processing. So,
I’m going to do read from a table
and create a new dataframe.
So let’s see
what happens.
So this is finished,
I’m going to refresh my Spark UI.
And for
the time being, I haven’t triggered
any execution on the cluster.
I just created in the driver
two new definitions: one for AccountsDF
and one for activeAccountsDF.
I can say for the activeAccountDF show me
the plans,
and I can see all analyzed(?)
past the argument to be true
so I can see all the plans.
Okay.
And when I do this, yes,
I still haven’t triggered any jobs.

Okay.
Now, I am going to ask to see
the content of this dataframe.
So this dataframe
will need to be materialized,
and so there will be some execution.
Okay.
So I see the results
and let’s- okay, now
you see now some processing appeared
on the timeline.
Okay. And
that’s
back to me-
oh, you’re looking at this job
and we can look at the job.
So it’s a show.
Okay.
Can look at the DAG, so
we see all the steps that are done.
There’s only one stage,
so there’s no shuffle involved
and let’s look at the-
So here we are on the SQL tab.
Okay, so there’s a filter and a project.
So what’s the- yeah- we have a- we do
the filter on the accountsDF.acct_close_dt.isNull(),
and when we do a project on "acct_num,"
that reflects here
and if I click on details
I see the same thing
as the,
when I do it an Explain.
Okay, let’s try a more complex
query.
Okay. I had a little query
is defined, but
query has been defined, okay,
and executed.
So...
there’s a read and
there’s are two- yeah.
So what happens here?
The query hasn’t really been executed, but
the read with the two
options have triggered some
some jobs on the cluster.
So one-
because I said header equals true,
then the executors
need to read the header
to retrieve the header,
and inferSchema
equals true.
Then Spark

needs to scan the file
to infer the schema.
But what is missing here
is a show.
So I'm going to add a show
to execute a more complex
query. So,
and when I'm going to execute, it's
going to execute again.
The two-
the two previous jobs.
So I'm going to add- oh
no, I'm not going
to do that because otherwise
I would need to do it
for all the new cells.
So let's execute this.
Okay, so I'm refreshing my page now.
So we had the two previous CSV;
now we have two CSV again,
the same ones.
But we have two jobs
to complete this query.
And let's look at the
more complex, okay,
we have a more complex DAG which shows us
a broadcast exchange.
So one file must be smaller
than the other,
and below
the accounts
and we're-
so there's a broadcast exchange
and we see the filters happening first,
then the project,
and then the broadcast exchange.
And well, I could do this here
with explain equals true.
Okay so now I mess things up.
So I need to do this.
I can
use this.
Okay, the execution was there,
okay, afterwards.
But yeah I can see the details here.
So let's go to the physical, so
the execution is parsed first,
then analyzed and, let's see, optimized.
So optimized introduces the not nulls.
Okay and then
if we look at the physical plan,
we have the push down filters.
Okay.
And then the file- the- sorry,
the dataframe is

broadcast exchanged
and then the join appears in memory.
So it’s what I wanted to show to you;
I can do that right now.
In order to make sense of those
Spark UIs-
so you see the small jobs we were doing.
So sorry need to go back.
Since we are using small data sets,
the widths of these execution
is not sufficient
to be able to click on them.
But if you don’t take any steps,
I am going to show you
this is what you get, and this is
and we have a small number of jobs but
this is
what Spark generates for those groups
and description is not very informative. So,
what I like to do is use this:
the set job group
which allows you to customize those labels,
so you can say
"Explore Query Execution,
A more complex way."
Okay, let me execute this.
Okay.
So I have not defined this
because I did a-
okay let me re-execute this.
Okay.
So now this active
thing is defined and it should work.
Okay. So now it works.
And if I refresh
my Spark UI,
I have now more informative
groups and description.
So and you can use this as you want.
So I know that this belongs to this notebook.
I’m using the job group to
indicate the notebook
to which these jobs belong.
And then for the description, it’s
what you would use
as a comment or as a title.
So but I know that there were two things,
two jobs
that were taking place
after I’ve done this.
So I encourage you
to use this technique to document
your Spark code.
The downside of this
is that you have to be consistent. So

you have to document all your actions.
On the plus side, it makes you
become an expert on what triggers an action.
And it’s not always obvious, such as the
when it reads a CSV
while there are the header and the info column.
But when it reads the Parquet file,
it’s going to
to retrieve
the schema from the Parquet file,
when it reads a JSON also.
So there are times where it’s good to
have this technique to understand
what goes on your cluster.
And that’s it for this exercise.
WEBVTT
In this chapter, we are going to study how
Spark implements distributed persistence.
So persisting is a tradeoff between
spending memory for efficiency.
So in order to be worth it,
the data frames
or the RDDs that you are persisting
should be reused several times.
The persistence is using a cache that is
specific to the Spark session.
It’s not a shared cache between applications.
That would be something that Apache Arrow would be good for,
but that is not what the Spark cache is about.
Let’s study this code where we once again
join people with their zipcode.
Okay so assume that after joining,
we start drilling down
on the pycode to look at who’s
living in those specific pycode.
There’s a good chance that this drill down
will happen again and again using
different values for pycode.
As it stands now, every time you do a where pycode,
it equals something new,
you load both data frames
in the memory and do the join, and so on and so forth.
This can be expensive, so a good idea would be
to persist after the join.
And this is what we do now.
We do a persist.
What does persist do?
At this stage, it tells the driver,
the next time you materialize
this data frame, do not evict it from memory,
which means that at the time where you write persist,
it’s not in memory.
The take away from this, and this is the
main take away from this chapter,
is that persistence is lazy.

After the persist, I repeat, the data frame is not in memory,
but the driver knows not to evict it
from memory next time it’s materialized.
So after the persist, you do a show
on one value, where value.
Now, the data will stay in memory
and the next time you drill down
on another pycode, then it will
leverage the the data in memory.
This is how it works.
You can also persist in memory
tables using a cache table.
So it works lazily also,
same as the persist we just saw.
WEBVTT
What kind of persistence storage levels
do we have?
So we have several choices.
You have
memory and/or disk,
and in some cases
you can choose to serialize the data or not,
and you can choose
also a level of replication.
So when you’re
persisting tables and views
they’re always persisted in memory,
but for data frames and data sets
and RDDs you have the choice.
So the data is persisted in memory
based on the partitions
of the underlying RDDs, meaning that
wherever the data the files are,
then those files will be uploaded
in memory
of the same executors.
And this information
is attracted by the driver.
So for the storage location
you have MEMORY_ONLY, which gives you
the fastest result, but also is the most
expensive in terms of memory.
DISK_ONLY doesn’t give you
as quick as a result,
but it doesn’t use memory
and MEMORY_AND_DISK,
which is the kind of best of both
worlds. It starts with memory
and then if everything does not fit
in memory, it spills on to disk.
So this is the best option
because it combines
the advantages of MEMORY_ONLY
and DISK_ONLY based on what’s available
and that’s the default, also.

So in Python, data in memory is always serialized.
So in Scala you can choose to use either
a serialized or not serialized.
So that's also another level of tradeoff.
The raw data takes more memory than serialized data.
Again, for data frames and data sets,
remember that a data frame is a data
set of type
row, so it's the same thing.
It's using the tungsten format,
which is very efficient.
If you're using RDDs
the recommendation for serializing
is to use the Kryo
serializer
instead of the default one.
So you can add to this
a level of replication,
you would have to have a lot of memory
to use that.
I have never seen people
using that.
Usually memory is a scarce resource
so people don't waste it
with several levels
of replication.
So you can say the storage default
for data frame and data set
is memory and disk, which makes sense.
For RDDs the default is memory only
and the way you can persist,
so if you don't provide the
the storage level
parameter,
it will use those default values.
Then you can say persist or cache.
It's the same thing.
As a reminder for table and views
the level is always memory only.
So again, when would you use one level or the other?
To specify,
memory must mean that somehow
your processing
won't work
if you're using memory and disks.
So you have a in important constraint
for time response and you have no-
it's not possible for you to use the disk.
So you must have enough memory to use that.
You can use DISK_ONLY
as a tradeoff
also, when executing the queries
more expensive than reading from disk.
And that can be a good choice.
And if you want to use replication,

you will use it when, the execution
would be more expensive than bandwidth,
because replication means
you’re going to retrieve from a
another executor,
the replica that is in memory.
It’s going
to travel across the network, thus
taking some network bandwidth.
So you cannot dynamically
change the storage level.
You have to unpersist first
and persist again to a new level.
WEBVTT
How can you track your persisted RDDs and data frames?
It’s in the Spark UI.
For a dataframe,
the name of the dataframe, the how
it is identified by Spark internally
is the
execution plan.
For RDDs,
you can provide a label to make
this page more readable.
So for each record
you will get the storage level,
the number of partitioned cache,
the fraction that is cache,
and the size in memory and on disk.
And you can drill down further
and look at the distribution
of the partitions
across the different nodes
on your cluster.
And in this example, most-
so there are five partitions,
four of which are on worker-2
and one is on worker-1,
and the partitions are all in memory
and well distributed, there’s no skew in the partitions.
WEBVTT
In this new notebook
we are going to study persistence
with the Spark cache.
I am going to perform
the set up steps as usual.
And while this is running,
I will open the Spark UI.
I need to go to the YARN Application Manager.
This should be my...
What is the story?
So, wait.
I need to start the Spark session,
and it’s starting now.
Okay.

I should have the Spark session now.
I open this in a new window.
I resize it,
so that I have my notebook,
and the Spark UI side by side.
That’s the way I like to work,
and that’s the way I recommend
you do the same.
Of course, at home or at work
you can’t have too many tools.
Okay, so
I’m ready for the
lab.
So I will go through the solution.
Okay, so first we create
three data frames.
So what’s
going on here? There are three definitions.
But there are two jobs so let’s-
let’s verifiy this.
Yes, two jobs:
one is to retrieve
the header from the files,
and one is to infer the schema.
Now, we are going to
perform a SELECT and a SHOW.
So this will trigger
distributed execution on the cluster,
so we’ll have a new job
coming here.
And it took- it created two jobs.
And if I look at the query,
it’s- two
jobs are here to create this.
So it was a join and the join was optimized
using broadcast hash joins.
So that’s good.
Okay, so
now I’m going to persist
this accountsDevDF.
So I’m saying persist.
Okay.
So if I look at the storage tab
for the time being, it is empty.
Remember from the lesson,
persistence is lazy.
So if I redo this
query that I did previously.
Okay, so
first, I’ve created new jobs.
And if I look at the storage
tab now, because it’s-
while doing this transformation
it materialized the
dataframe I was persisting,

I should have now,
some something in my cache,
and it is there.
Okay.
And it has the very long name
based on its execution plan.
So what else can I see?
So it’s been serialized in memory,
because it fit into memory.
There’s one partition.
Okay.
And I can drill down on this. And
the long name is a bit unwieldy, but
I see that it’s stored in the memory
of this
executor.
Okay.
So now
if I leverage this again,
to do another select, I must have
created a new job
and you see what happens.
Something has been skipped
because this
something has been skipped, so-
and we see here.
So I’m going to make this larger.
What has been skipped
is all those two previous steps
that we are doing the join.
And the data was read from memory
and this is materialized
by this green dot.
So the execution of this job
started from here
from this point.
And the execution must have been faster.
So it took 44 milliseconds
and nothing is worth that much or that little before.
Okay.
So previously it was

so a huge improvement.
And if we do it one more time.
Okay.
We leverage the cache again.
But it doesn’t yield
any further improvement.
Get in the same ballpark
in around 40 milliseconds.
And if we instead of a show,
we perform a write.
Let’s see what happens, so-
Okay, so now we have the dreaded, dreaded
And if we drill down, we still

have something that was skipped.
Okay.
We did....
Then it was.
So it’s -
open this up.
Yeah.
So we leverage the cache again.
Okay.
But it takes more time because
we write to disk
and we write the full
data frame instead of just five rows.
So let me go to the storage.
It’s still there.
Okay.
And watch what happens when I unpersist.
I execute this
and immediately
my cache is empty, so unpersist is eager.
It’s one of those
exceptions in Spark.
And unpersisting is eager
which makes sense, by the way.
Okay now we
we repersist
using disk and two replicas.
So again
persisting is lazy.
Okay.
So if we
materialize
the data frame by trying to save it.
Okay.
Now, it should be
in the cache
and it is anything-
Okay.
Let me-
It’s on disk.
Okay.
There are 200 cache partitions.
Everything is on disk,
and there’s a

And those two on a partition
are spread on those
workers
evenly
and twice.
So there’s one on one machine and one
on another machine.
Okay. So what I’m going to do now
is unpersist this.
And again,

if I refresh my cache then there’s nothing.
That’s a-
that’s all for this
exercise on
Distributed Persistence.
WEBVTT
In this chapter, I will
introduce the Cloudera Data
Engineering Service.
CDE is a data service from Cloudera,
so it’s part of the data services
available, which means it’s targeted to
a persona that would be a data engineer.
It contains the services that a data
engineer expects to perform his job.
So in the case of CDE, then
it’s Spark and Airflow
and it’s plugged into SDX so-
so it’s a it does not bypass
the security or the data governance.
It’s powered by Kubernetes.
So it’s this data service.
So you see in the public cloud,
all the data services available; well,
there’s one more
with the DataFlow
service, but
you see the Data Engineering,
the Data Warehouse,
Operational Database, and Machine Learning;
And there’s one more,
which is the DataFlow service.
Maybe I can show this to you
on my machine.
Yeah, it’s there.
You have Data Flow,
Data Engineering, Data Warehouse,
Operational Database, and Machine Learning.
Okay.
So to leverage this, you need to install
the CDE service like I did.
So you have the service and
to use the service, you can have one
or many clusters.
So in my case here, I have one.
I could create a new one.
And when I create a new one, you specify
the number of CPU’s that you want to use,
the memory. So
the auto scale max capacity
for both of these things,
their version of Spark,
so I could use Spark 3.2.
I can enable Iceberg
and do several things too.

And so that’s all you need to create a new
virtual cluster.
So if I’m a data engineer
and I have a new workload to tackle
because I can size very simply-
very simply
my new virtual cluster and
and I will be set
after a period of time;
that depends.
I will be set with
a new cluster and it will
enjoy the scaling
with Kubernetes and isolation
provided also
by this architecture.
So it’s
Spark on Kubernetes on top of SDX
with orchestration by Airflow.
So it looks like this.
So you have the service
and the virtual clusters.
So here we have two virtual clusters
and you can monitor
the activity on your clusters
and see when there’s auto scaling.
And you can also monitor
the capacity and resource
usage through Grafana.
This one is a quick and easy way to-
to see the scaling.
But then if you want to see
more information you can use Grafana.
This service is not meant for you
to write code.
It’s meant to deploy code.
So your code must be written or
and/or package elsewhere; packaged
if you’re using Scala and Java,
packaged in a JAR file.
So the first steps you do in this
environment is to upload
already made code.
Okay.
And you have several,
several parameters
that you can use to specify
the typically the,
the number of executors, the course
for the driver and executors, memory
and for the driver,
and the memory for the executors.
So typical
performance parameters
for your Spark workloads,

and then you can chain
on more generally orchestrate
your jobs using Apache Airflow.
So Airflow is a
workflow tool that replaces
Oozie. It does a small model
and it does a nicer feature
and it allows you to express or specify
your workflows in Python instead of XML.
So nowadays
it’s the preferred tool for
workflows.
So you can, instead of using a resource
that is Spark, you can
upload an Airflow file
and schedule the workflow.
And there’s also
the Airflow UI is available.
You can see a graphical representation
of your workflow
but also track everything that goes on
in your workflow.
This is when the information from
your job
execution is sent to another tool
that we will study last,
which is the workload manager.
So you can troubleshoot your
your workloads and compare the runs
and apply
statistics to your runs and visualizations
to historical data
about your runs that helps
you troubleshoot your current jobs.
Also, it’s plugged in SDX,
so it provides
information to Atlas.
And so you can see
what your processing does to your data.
It’s all captured into Atlas.
It has a rich Rest API that you can use
to automate and interact with the tool,
and that’s
it for the Data Engineering Service.
Now we are going to use several notebooks
to experiment and
play with those features.
WEBVTT
In this notebook
we are going to work through the CDE data
service.
For this, you need an environment
in the in your cloud provider.
You need the CDE data service.
So it’s a long running Kubernetes

cluster and services.
And you need a virtual cluster
to perform your virtual workloads.
There will be jobs that will run resources
that you upload and job run,
which would be an individual job run.
So to get to the service,
you click on the CDP
console on the data engineering icon.
So I will do the same thing here.
Okay.
So you see the service itself
and the virtual clusters
that are using
the service.
So I will click on the pencil icon
for the service,
here,
to the service details.
So this service
is of type small.
It does not use Spark instances,
and there's an auto scale range from 1 to

So I can have from 1 to 50 nodes,
from 0 to 400 CPU, from 0 to 1600 gigabytes.
Okay, so that was something
like this.
Let's go through all the tabs.
So I'm not sure here- so we have,
the cluster is idle,
so it doesn't run for the time being.
I have six nodes
and I have four nodes
from the infrastructure
and nothing has been used.
Logs should be almost empty.
We don't have access and diagnostics
for the time being.
No events have been generated.
I can look at Grafana Charts also.
So this opens up a new tab
over by Grafana
and we see kind of the same thing.
But for the time
being, the cluster is new,
but nothing has been
run, so there's nothing to see.
What else,
can we look at the resource scheduler?
And that opens a YuniKorn dashboard.
Okay.
And for the same thing, for the time
being, there hasn't been any
run of the work, of any workload,

so nothing happens here.
So YuniKorn
is like the Resource Manager for YARN
but for Kubernetes.
So you can look at applications, queues
and nodes very much like what happens
in the YARN Resource Manager UI.
Okay, what’s next?
Let’s go back to the main page
and click on the pencil
for the virtual cluster.
Go back to Overview.
Let’s look at the virtual cluster.
This is something
I showed you in the previous
lesson.
So
what’s there to see so
we can scale in memory from 0 to 20-
sorry, on CPU on 0
to 20, and memory from 0 to 80.
We’re using this old version of Spark,
so chart’s are empty,
logs are almost empty, access-
no access restrictions are in place.
Then we have links to
this thing:
downloads
a client
for CDE, which we will not use in this
class.
But there’s a client
that can run on your desktop.
You have the link to the API
docs, a link to the jobs API URL.
Okay.
This one copies to the clipboard.
Grafana again,
but still empty
and the Airflow UI
should be empty also.
Okay.
Let’s go back to the Overview
and click on view jobs. So,
go back to overview
and look at view jobs.
Okay.
So for the time being,
we need to create jobs,
so I think we’ll
do that in the next notebook.
WEBVTT

In this notebook,
I’m going to create and trigger Ad Hoc Spark jobs.

First, I need to download
four files that
corresponds to Python jobs.
So this file is a process from data,
from log data.
So let’s look at the jobs.
I will open view jobs
on my environment.
And as we saw previously,
it should be empty.
Oh, yes, view jobs is empty.
Let’s make this bigger.
So the next- the first thing I need to do
is to import-
to import the resources
and give them proper names,
because those names
will be used to chain them
in a Airflow workflow.
I need to upload all of those four
and then create the jobs.
Let’s do this.
So I click on resources.
That’s the name I should give,
"username-resources".
So my username,
here, is
this thing.
And the syntax, "-resources".
Okay.
And there are files.
Okay, now I can upload the files,
or I can drag and drop them.
Let’s see if I can do this.
Success!
Okay, let’s upload those files.
Okay.
Now I have the four files in my resources.
So now I think I can create the jobs.
Let’s check.
I have done this now I can create the job.
So I’m going to create a job.
I need to create the four jobs.
So let’s do this.
Go to jobs,
create job.
I have still this in my buffer,
and the first job will be the name
will be "Lab3A_access_logs_ETL".
It’s a file
and upload from the resource.
It’s this one.
I will be using Python3.
No arguments.
Let’s check before I click on save.

Resource, okay.
Ignore...
Okay.
It’s using Spark 2.4.
Python3...
And click on create
without doing run.
Okay.
One of them.
I need to do all four, so.
Okay,
what’s next?
Okay.
I’ve done this.
Go to the jobs tab.
Okay.
Okay.
I’m going to trigger some jobs. So,
I’m here.
Which what makes sense is to trigger
the first one, for instance.
So if I click on this menu, I can see that
I can run now, add schedule, clone
configuration, and delete.
So for timing I’m going to use run now.
And let’s check
where this goes on.
Okay.
We’re going to look at the job logs.
So let’s do it.
Click on this one job runs,
and this is my run
that is starting.
I can dial down,
can see the run history for the time being
nothing happens, the configuration,
all the major settings for a Spark job.
And there’s no schedule; it’s
running Ad Hoc for the time being.
So if I go back once step,
I want Job Runs.
Still starting.
So the job has run
successfully.
So I have one point in my diagram:
it’s run the other jobs.
So the next one would be either
this one or this one, so I’m going to run this one
and run the other one.
Maybe that can be run in parallel
since they are doing things on
different segments.
Let’s look at Job Runs.
Okay, so there’s a quota
that’s limiting me,

so that’s a good thing also.
So, I’m going to wait for the
execution of these two jobs.
My jobs have run,
so I can drill down,
both of them being in the history;
yeah, I just of one point.
And then I can run the last one,
it would be this one:
create the reports.
Okay.
I’m being limited by the quota again,
I should have a word with my Admin.
He got scared.
The job is starting.
So if I click on the run ID
then I can look at the configuration,
the logs; so I should see more things.
Yeah,
so-
Go back to Job Runs.
Okay.
So four jobs I’ve run successfully;
let’s see in the next notebook
how we can schedule one of those jobs.
WEBVTT
In this exercise, I will add
a schedule to an existing job.
So I’m going to
add a schedule to the first job
using a common expression.
Let’s do that.
The first job is this one.
Okay, you can click here,
but I’m looking at Job Runs now.
I need to look at jobs
and this is the first one.
I’m going to add a schedule.
I’m going to use a Cron Expression,
so that it runs every 5 minutes
or maybe every 3 minutes,
for times sake.
And I need to be careful with the timing,
so I’m going to change
that to
yesterday
and the end date to tomorrow,
so that these parameters don’t
get in the way.
Okay.
Adding a schedule.
Now, if I go back to the jobs
page, this one:
the first one has a schedule,
so it should run soon.

So let’s look at Job Runs,
and it has started.
Okay,
so I have a new run ID
and it should run for 2 minutes,
so I can follow the execution
by drilling down again and looking at
the logs
and that things are moving.
Okay.
It is still running,
and it succeeded.
And I have another point.
Okay.
For another run
that took less than the previous one.
Okay, bear in mind
that we are working in the public cloud,
so there’s going to be some discrepancies
between the run,
because we do not control the network.
And so what I’m going to do now
is remove the schedule for this
so that it doesn’t run again
in a few seconds, remove schedule.
Okay.
And let’s check
the Job Runs. Okay.
So there’s an additional step
in the exercise.
I’m not going to do the same thing
for all the jobs. You get the idea.
And we removed the schedule from this job,
and now it’s Ad Hoc.
Let’s check that it is the case.
So those are the Job Runs, so it’s Ad Hoc.
Okay, that’s all for this notebook.
WEBVTT
In this video,
I’m going to show you how Atlas
is connected to the CDE environment.
So I have to click on jobs,
look at the registry, select
a successful run,
click on Atlas;
and then it gets me to Atlas
and I can show you
things in Atlas.
Okay, so-
So those are my jobs.
I can look at this one for instance.
There’s only one run.
Here’s the linked to Atlas.
It opens up Atlas and selects the
the Spark process.

So let me make this bigger for you.
So this is the result of a query
on the Atlas database.
Our type
equals Spark process and qualified
name is this one.
So I can click on this
and look at the details.
So the details gives me the logical plan.
Okay.
So first let me
talk about Atlas for a short introduction.
So Atlas is
a tool for data governance,
and it’s built on top of a graph database,
and here you see the graph.
So it’s a graph database
on top of which has been
defined a layer
for data governance semantics.
So we have entities.
So if you know
a bit about graph databases,
you have two types of things:
you have data abstractions,
you have entities, and a relationship
between those entities.
So this is an entity,
and this is a relationship.
Both of these data abstraction can carry
dictionaries of key/values.
So here, among the-
on the page for this execution-7 (spark_process),
and I have several tabs.
So this tabs shows me
the dictionary of key/values
for this entity.
This tab shows me the
neighborhood of this entity in the graph,
so I can see comes from
it’s a Spark process, takes CDE data sets
and then creates a load data table,
which is then
by other Spark processes, transform into other
Hive tables.
And I can use the graph to navigate
on the entities.
I can also click on the relationship
between those entities
and see some details about them,
like whether I want or not
to propagate classification.
So besides this
collection of
entities from your metadata,

what you can think that you can add
within Atlas, you have classifications
and glossary and
they kind of work the same.
So classifications
I don’t have any,
but they work as-
well
they can be leveraged in Ranger to
to create target based policies
for access or masking.
So you can define- so-
so these
information like classification and glossary,
you directly add in Atlas.
As you
saw, the rest comes from the execution
of programs using Spark, Hive, and
all the tools that are plugged in
Atlas and that
manipulate data.
The link between those tools and Atlas
is implemented with Kafka.
So those tools a Atlas hook
that sends a message on the Kafka topic
that Atlas is listening to,
and then Atlas decodes the message
and updates its graph.
So this part is automated and that’s
a huge benefit
because if you were to do this manually,
it’s a never ending process
and it’s a- it’s-
that-
and the idea of maintaining
metadata
like this is doomed from the start.
So automating
this path is a life saver,
but then you can add
manually, classifications who say, okay,
this column, for instance, is a PII,
and your classification can now
be nested in a tree.
So you can have a Finance PII or
an Insurance PII as leaves of
the Root PII
and likewise you can add business terms
to make your
Atlas database
more readable by business users.
The Atlas tool
should be the main tool of data stewards
in your organizations.
So they can clarify the terms

that are used in the columns
of your databases
and provide some definitions.
And likewise, you have two levels you have-
you can have a category,
so you can have a
if your organization has several
businesses, it’s
maybe it’s doing insurance,
maybe it’s doing banking also
and in both categories,
the same term would have a different
meaning.
Then you can leverage this feature,
create an insurance category
and a banking category,
and define whatever terms,
term has a different meaning
in those two categories.
So those are the things you add,
and this is what you get
from the automated
population of the graph database.
And that’s it’s really, from
Atlas and the integration
of CDE within Atlas.
So you see that
what we did with
the execution was tracked
in Atlas so that we know
the load data was created from execution_7 of
using the CDE datasets.
WEBVTT
In this
chapter, I will introduce Airflow.
So Airflow is a new addition to the CDP.
It’s only available in this data
engineering service.
It’s meant for the DE and ML practitioners,
so that they can
conveniently create data
pipelines, leveraging Python for Airflow,
and using a mix of Spark and Hive.
So why Airflow?
It’s the previous tool that people
were using
in every organization was Oozie.
But Airflow was born at
Airbnb;
from the same team that created
Druid and Superset,
and it has
won the hearts and minds of the community.
You can define pipelines.
It’s open source, of course.

It’s an Apache Software Foundation project
and there’s a large community. So,
actually let’s look at the community.
One way to gauge an open source
project it is to look at
the GitHub repository. So
if I do "Apache
Airflow GitHub"
and increase
the resolution.
What can we see?
Wow, this is
this is bigger than Spark.
I did not expect that,
this is the largest community I have ever seen.
It’s written in Python mainly.
Okay.
And if we look at the-
okay, that’s Mr. Crunch.
That’s the guy from
from Airbnb.
Wow-
It’s-
That’s a good look for a project.
You see it’s
gaining momentum.
Yeah.
So clearly
you can see the excitement
of the community about this project.
It’s going up and up,
and it has a very large community, so
it’s a safe bet-
it’s a safe bet for the future.
It’s a good replacement for Oozie.
So in the virtual cluster
for the time being,
we imported Spark jobs.
But what you can do, also there’s a switch.
You can import Airflow workflows
and then
they run like the jobs we previously ran.
Okay.
And you get access to the Airflow server
and UI
like you would normally.
So for
the CDE service, it’s
just a new type of job.
And so you can
define your Spark job in an Airflow file
using the CDE operator.
So you have to import the CDE-
JobRunOperator in your Python code
and then define, for instance,

ingest step as being the-
defined by this call
to this object with those parameters.
And this is a complete airflow DAG, so
you see it’s Python code--
that’s why it’s one of the reasons
for which it’s popular.
And we have the ingest as a step one,
but we have also a step two,
a step three.
Okay?
And the DAG is made of all those steps.
So you have step-
step one goes into step two
and into step three.
So here you see our Airflow
job is presented,
so we for the time being we are
we only add Spark jobs,
but now we have an Airflow type of job
which of course is scheduled.
And in the next exercise
we’ll have a look at how we can
use Airflow with the CDE Data Service.
WEBVTT
In this new lab
we are going to study how to
use Airflow in the context of CDE.
So I need to
upload some
Airflow files;
of course for that I mean, all I have
are Spark
jobs.
So I need to use this and
like we did previously,
I need to add those resources.
But before I do that, I need to check
the schedule on those files.
So I’m going to open this
in text edit,
and I need to change those values
so that we can see something.
So the start_date is okay,
but end_date,
let’s make this something in the future.
And 5 minutes can be a bit long.
Okay.
Let’s try

I’m going to save this.
So there’s another file
which is meant for stopping
these.
And does it need to be

also changed?
Yep. So I'm
going to change this also.
And I can save this.
Okay, so now I'm going to upload them,
and I'm going to create a job
for the first one
and the second one also.
So we're going through the same steps
as previously.
But this time I'm using the Airflow
radio(?) button.
I am going- so it's going to be a file.
I'm going to choose the file.
It's going to be this one
and I've got to use the proper
name on this.
So I need to get my user
and the rest should be
this.
Okay, looking good.
I'm going to
copy these.
Just create.
Oh- so,
I need to do some-
something before
because I created the job before
and it was not-
I didn't change the date,
so I need to delete it first.
Then I can
create it again.
Okay.
And create.
Okay.
So I have this new
job of type Airflow
which will run within 3 minutes.
Let's check on the
instructions.
Okay.
Okay.
So we saw that,
and let's
see if this job runs.
So let's look at the jobs.
Okay,
things are running.
So next what I can do,
I can click on the Job Runs.
Let's check what I need to do:
cluster details.
Click on virtual cluster, cluster details.
Okay.

You see something is happening here.
Okay.
And next,
I guess I’m going to go to the Airflow UI,
and something is running
and we’ll drill down.
And then you are in Airflow.
So you have and see
that several are already starting the second run,
and everything is
working as expected.
So you see the very simple graph
I’m executing:
the calendar, test durations,
there’s a Gantt,
really simple.
I can look at the code also.
Let’s see what we have to do.
I go to the Airflow UI, yes, review, we saw that.
Click
on the configuration.
Okay,
so now we’re going to remove,
remove the DAG file.
So let’s do that.
So this is my Airflow job.
I go through the configuration.
I don’t see how I can
remove the DAG File.
Let’s check again.
(unintelligible)
Okay, so let’s check again.
Where’s my DAG File?
I don’t think that I can do
anything here.
So let’s go back to jobs.
So what I can do,
Maybe I can
click on this,
yes, this way I
can schedule.
Okay.
Airflow UI.
Okay, so
what I can do from this
and maybe there’s a change in the UI
because this is a new product,
so its bound to evolve.
I’m - let me just
delete the thing
and voila!
I don’t need to stop it or
use the other-
the other job.
I think I have managed to stop it

using the UI
and that’s it for this short
introduction to Airflow in CDE.
WEBVTT
This chapter is a short
introduction to Workload Manager or WXM.
Okay, so
the problem that WXM addresses
is a problem of lack of visibility
when trying to troubleshoot
your workloads.
You have so much information, it’s difficult to
find the root cause of any given failure,
so that maybe because of this
you make a bad decision and
make the situation even worse
than it was before.
In those environments,
with so many moving parts,
you spend a lot of time
sifting through all this information,
and eventually then you can have
the right information
that allows you to find the root cause.
But a lot of time is wasted
and looking at different things.
So what Workload Manager tries to do
is to save a lot of first steps
of troubleshooting by presenting you with
improved information.
WEBVTT
What WXM does is-
it collects all your, all
the historical data from your workloads
and that allows two things:
it allows
to visualize trends.
Okay.
And that is powerful already.
And another powerful, powerful thing
it does it allows to do statistics.
So you can have a scientific way to define,
"this is a normal run," because it’s in
within the standard deviation
of the population of those same runs.
But you can say scientifically,
this is a slow run
by the same way, this is a slow run.
This is an outlier.
It’s not within the standard deviation
of my runs
for this job.
So this is very powerful.
It allows you to compare runs
from the average of runs

for different properties.
And you can quickly see where the problem
starts.
WEBVTT
It’s collecting
data by one way or another.
So it’s available for Impala, Oozie, Hive,
YARN, and Spark.
It collects data one way or another,
so it’s either pulling or pushing,
but anyway,
it collects the data from those runs
and then send it to Workload Manager.
And your Workload Manager
can be on premises
or it can be hosted by Cloudera
and in that case it’s a service.
WEBVTT
Why? What is this tool good for?
It’s for a lot of the
of CDE cluster
stakeholders should be interested
by what this tool provides.
For instance, for the
the system admin or the database admin,
it shows you the trends of the-
of ingesting- the data you ingest.
So it can act as a indicator
to, for instance, procure new worker
nodes.
You can say, okay,
if this trend continues,
I’m going to need new worker
nodes in six weeks.
And knowing that procuring workers
takes three weeks,
I need to start the order in three weeks.
That could be something, okay.
For the data architects
it could mean okay, this
because of the growth of the data,
the layout of this table is
no longer relevant,
and the queries that use this table,
are getting slower and slower.
I need to refactor this table
and use some new
partitioning, for instance.
So that’s for the data architect.
For the data engineer, they can say,
"Oh, my queries are slow" or "I have-
I have problems with them with my Spark
workloads, they are failing,"
or they exhibit this characteristic.
The tool is not just a shell

for historical data
that shows you graphics and statistics.
It also characterizes
the workloads according to--
for in the case of Spark--
to 18 different characteristics
and this allows you to
organize your work
according to the technical depth
that your Spark jobs have.
WEBVTT
Let’s look at an example, so
we collect from the YARN Job
History Server, the data
from the MapReduce jobs.
And for Spark we collect
also the from the
the Spark History Server
the same information
from your own Spark workloads.
WEBVTT
Let’s look at this example.
So first, you can specify a time range.
And these are the characteristics
that can be tagged to your Spark Job.
These characteristics,
they come from your experience
in the field of using on-
our own experience as a Cloudera user,
because as you may know,
Cloudera uses Cloudera for its back office.
But also the experience we gain from
being in the field for various customers
and helping them
fixing their Spark workloads.
So you have these characteristics
and of course,
a single job
can of several of those characteristics.
But this allows you, as I was saying,
to organize your jobs, say,
this week
I need to fix the fail to finish jobs.
I think that would be
a reasonable priority.
But once this is done, maybe I can look
at the abnormal duration jobs.
You see the jobs view and
you can
see that for a given job,
they have several-
they can have several health issues.
And now we look- we drill down on a-
on one specific job,
and you can see the historical trend

of the duration
and the fact that the tool is
able to say
this is slower than the median duration,
and the tool points
you to where the problem is.
So the problem is in stage two of job one.
And when you drill down on stage
two of job one, it tells you
there’s poor parallelization.
So it’s likely a skew problem.
That’s the skew in the data
that leads to poor parellelization.
And there’s one task out of 1980
that takes close to 6 minutes.
We asked the others 2 seconds.
So this is typical of a skew problem.
One of the partitions
gets a sizable chunk of all the data.
Something needs to be done
and you cannot change your data.
So something needs to be done on the code
to mitigate this heavy skew.
It’s not extreme; it doesn’t
make the job fail, but it’s still heavy.
So what you could use, for instance,
if you’re using Spark 3,
you could define what skew means for you
in terms of a ratio
between the offending partition
and the median of the partitions.
So it’s more than five times
the median.
Then for you it will be skewed and a Spark AQE will
automate the mitigation of this
skew problem.
And you can compare also
with the average for the same job
to see where the problem
lies.
Okay, so here
we have a problem of garbage collection.
It’s taking
much more time than usually,
and you can view the same information
that you can see on the Spark
UI on the Spark History Server,
but this time
it’s in the context of historical data
and look at properties also
and compare the properties
with between ones if they have changed.
Maybe this is the reason for the change
in the behavior
of your workload.

Next, we are going to look at a short
video of WXM for Spark.
WEBVTT
We’re going to work on WXM,
and I’m going to pick up
my Spark demo cluster.
So let’s say that I know
a Spark job,
so a certain Spark job of mine has failed.
And I want to directly analyze that job
just to quickly troubleshoot, right.
I have the ability
of going back to actions
and clicking on view job or query.
So this is a search functionality
where if I paste my application ID,
it will directly
bring me to that specific Spark job.
So I don’t really need to traverse through
the summaries and figure things out.
So if I have a specific
query/job, right,
and I wanted to deep dive
as far as I have the query
ID or the job execution ID, I can go in
and search directly from the cluster list
page itself.
So this could be an Impala query, Spark job,
Hive job, Oozie job--doesn’t matter.
As far as I have the right ID, I can go in
and directly jump into that query.
So here’s what’s happened, right?
So I’m taking a look at the,
what WXM tells me is, of course
I already know that the job has failed,
and it immediately tells me that
I have this failure
that is coming in from job
three-stage three, and it also parses out
just that specific part of the log
so that I don’t have to really thumb
through the entire log
to go in and find out what’s really happened.
So before I
dive a little more deeper,
there are three things that
or three buckets that
WXM focuses on
when it comes to Spark, Hive,
Oozie--any of the typical YARN jobs.
It shows me Baselining issues
which focuses and tells me
if, are there any issues
with this run of my job
compared to any of the previous runs

of the job.
It then highlights also any skew issues
which talks about issues
within this specific run itself.
And then the third area is
the resource issues where it talks about
do I have any resource contention problem,
garbage collection problem, things like
waiting for a container to be allocated,
like task wait times, etc..
So based on where I see
these issues, I can go in
and then start doing
deep dives around that as well.
So if I click on this specific stage,
it takes me to the execution details.
It directly goes in, sets up,
you know, the job it is,
where we see the failure
and it gives me a list
of all the tasks that have failed.
And so the details,
if I go back on top to the summary
of this job itself, it tells me
what the reason of the failure was like
we already saw on the other page.
So here you can see it was a Java
runtime exception, and it was caused by
a reset, right?
So the connection was reset by peer.
So it kind of lost a connection
to that local host.
Now, let’s take a different example.
I can go back to
the Spark summary page
and say I just pick this specific job.
Here we can see that this job is completed,
right?
It’s not like the failure situation
that we saw earlier.
Here the job is completed,
but the performance could be improved.
So there are optimization opportunities.
Of course, that are Baselining problems.
There are some skew issues as well,
that our WXM has identified
and I can either click on any of these
health issues or
I could take on health checks,
and it pretty much
will take me to the same page.
So you can see here there
definitely has been a difference
in the Baselining of this job itself.
And of course there is the skew, right? So

WXM again sets
context for me to highlight
where exactly I’m seeing this queue problem.
So I can see job 5,
stage 18, seems to have a skew problem
and has taken about 10 minutes.
And this is because it suffered
from some poor parallelization.
So out of the 2,600 of our tasks
that were spun out for this stage,
one task took,
you know, abnormally too long.
So the typical range was about 3 seconds
to a minute.
And this specific task
took about 6 minutes and 14 seconds.
So you can always click on this task
and find out what was going on
and compare
the metrics of this particular task
with the average of all the other
twenty something, hundred tasks that we saw.
So if I wanted to do some deeper analysis,
if I wanted to do some historical analysis
because users always want to figure out
what happened with my previous runs,
did I see any differences, etc.,
especially now that we have seen
some baselining problems,
you can go to the trends tab
and it gives you a nice view
of what was happening across these jobs.
And I also have the ability
of looking at all
the previous runs of mine
in a specific job.
So you can see here
the execution IDs are different
because they are different runs
of the same job and similar
to what we saw earlier,
every single run has been attributed
with a specific health issue
or no health issues,
right?
And I have the ability of time
traveling here as well.
When I just click on this
and it’ll reset the timeline for me.
But more importantly,
I can go in and do quick comparisons.
So if I looked at my current run,
let’s just assume this is my current run,
and if I looked at one of the golden runs,
I’m going to assume this one is a golden run

because there was no health issue,
there were no health issues here.
So I can click on these two
and hit the compare button.
And WXM immediately
gives me a side by side comparison.
How cool is that? Right?
So here I can easily go in and see
what was the different
specifics of my job,
what kind of structure did we see.
So this looks like
a completely different job.
Probably there are certain conditions
that are actually getting triggered.
Look at this structure.
It’s very, very different.
If we have looked
at the configurations, it highlights
any of the differences.
So it becomes easier for me to go in
and see was it,
were there any changes in the board
or were there any memory
configuration differences and so on.
And I also can look at
if there are any SQL execution.
So it looks like my current
run has certain
SQL executions,
which were not there in the previous runs.
And then finally, you know,
I have the ability of looking at metrics,
which gives me a good idea in terms of
where do I see the swing
in terms of the metrics.
Again, the idea here was to highlight
as a spot programmer,
how can I very quickly
go in and do a few things?
One is troubleshoot a failed job,
and the second one is
try to look at optimization opportunities.
You also have the ability
of looking at,
you know, the suboptimal jobs.
If you wanted to start with something
very specific in terms of health issue,
you have the ability of going in there
and start using
that as a starting point.
So with that,
I hope I was able to give you a good idea
of the various things you could do
in Workload XM, that the two personas

like the administrator persona
or an end user persona.
And how do, you know, you can also go in
and look at the different engines and run
analytics,
troubleshooting, optimization opportunities
in an easy self-service manner.
WEBVTT
In this chapter, we are going to
look at why Data Engineering?
In this statement, we see that data
engineering is a requirement
for machine learning, data
warehousing and data cataloging.
So Data Engineering
comes before those activities
and is a prerequisite and its in the critical
path of these activities.
So what you have is data being captured
upstream
on machines, sensors or other
devices, weblogs, things like that,
and before
being consumed by Data Scientist,
business analysis,
it needs to be engineered.
It needs to be curated, fixed
and shaped into the right format,
in the right order.
The to the need to be this interface
between the birth of the data
and the consumption of the data.
Because as it turns out,
as fate would have it,
data never occurs in the right format
for consumption
by machine learning algorithms
or SQL processing.
Well, that’s the area of Data Engineering,
and there’s a lot of
terms that fly around
and some are short lived and some stick.
Data Engineering is one that sticks
because it really
responds to reality.
The fact that data needs to be engineered
to be consumed by downstream activity.
So it’s a very much in-demand skill.
Okay.
Even more than senior Data Scientist.
So it seems
now that they are too much Data Scientist.
I remember a time when Gartner predicted
they would need,
there would be a lack of Data Scientists.

Well their still in demand,
but less so than data engineers,
because whatever you do,
it can be data science or just SQL
queries,
you need your data to be engineered.
But what are the challenges
with this activity?
The main tool for this is Spark.
Though there’s one
competing framework which is Flink,
but the most common one is Spark.
So you need to understand how Spark works
and since you’re dealing
with large datasets, you need to manage
your Spark resources accordingly.
Usually Spark jobs are
combined into complex pipelines,
which you need to orchestrate,
and that’s more
the realm of a tool like airflow
and when everything works,
then everything’s fine.
But as fate would have it
sometimes you have to troubleshoot
and troubleshooting in those environments
is complex.
Okay, Because those pipelines
are difficult to test
because of the size of the data.
It’s not like you, you can test a
a user defined function, it
has input, output.
So okay, that’s great.
But testing a complex pipeline
is like similar to what you have today.
Like when you want to test a SQL query
with the added difficulty
that you have very large data.
So it gets even more complex.
The complexity is amplified
by the size of the data.
And of
course you want all of this
to happen in a secure fashion
and your business wants this
probably yesterday.
So this is a diagram that we use commonly
in our marketing slides,
and it’s one of those that stick
because it’s really reflects
the reality of the data journey.
Of course, it maps also to our tools,
but it’s meaningful. So Data Engineering
is the tool on this

journey, is where you curate your data
to make it for the downstream activities
to be able
to consume it
So it is our strategy to provide
data services aligned to key
personas in the big data world.
Since there’s an agreement
on the existence of data engineers,
we have such a data service
tool called Cloudera Data Engineering
CDE for short,
sits on top of CDP,
public cloud or private cloud.
It is leveraging Kubernetes
for resource negotiation
and Spark
for processing, airflow for orchestration,
and it allows you to create complete data
pipeline.
Okay, so by using Kubernetes, you get the auto
scaling features of Kubernetes,
but also you, because it’s part
of CDP, it’s plugged in on SDX layer.
So you get security
and secure access to your data
and also your data governance
repository in Atlas will be updated.
You also get airflow for orchestrating
your jobs,
and with the addition
of a workload manager, you get visual
troubleshooting and complete monitoring
and alerting of your jobs.
And all of this use Rest APIs
so that you can automate
all of your activities.
WEBVTT
In this chapter, I
will cover a brief overview of Zeppelin.
The idea of notebooks
comes from the early seventies.
At the time
it was called literary programing
and the idea was to have executable code,
which would be interspersed
with paragraphs of text and visualizations
so that the author could document
his train of thoughts.
In the 1990s,
this was implemented first in RStudio.
Where you would have code
and marked down and you would need
a notebook into a PDF
and more recently it has been implemented

in both Jupyter and Zeppelin
and there are other notebooks.
The notebooks has become the de facto
standard for data science
because they allow users to perform
reproducible data analysis.
In this reproducible data analysis,
you want the user to be able
to share his notebook,
and for the people who received
a notebook, they would want to be able
to download the data
in the notebook, apply
all the transformation
and look at the visualization and agree
or not
with the conclusion of the analysis.
And this is very different
from the experience with the most popular
tool for data analysis,
which is Excel, where the
the data
you don't know where it comes from.
The transformations are spread
across multiple sources,
which are easy to break
and it's difficult to agree or
Not with an Excel spreadsheet.
You can play a notebook, execute a notebook
and decide to agree on what, quickly.
WEBVTT
Zeppelin is an Apache software project.
It's the only notebook system,
I think, in the Apache Software
Foundation.
It has some specific features
that are nice for educational notebooks,
such as multi kernels.
It's popular amongst big developers.
It's a better tool than the Spark Shell
because it allows visualizations,
it's more intuitive.
So as I said, one of the
best features of Zeppelin
is the fact that it's multi-kernels.
So as opposed to Jupyter, for instance.
Each cell
or paragraph in a notebook
can be bound to a different interpreter.
So you can have a cell
that is about markdown,
a cell that is about Shell
a cell that is about Spark
A cell that is doing JDBC.
So that's why it's very convenient

for us in education,
because it allows us to dematerialize
a full exercise book.
The community edition of Zeppelin
has over 30 interpreters.
But inside the CDPDC, we are using only a
limited set of interpreters
and we are communicating
with the Spark cluster with Livy.
So Livy acts as a middleware
that enables multi-tenant and secure
communication with the spark cluster.
And for us it is transparent.
So there are several options available
to control all the paragraphs
at the note or notebook level.
In the documentation
Zeppelin refers to notebooks as notes,
but also in the UI,
there’s a mix of notebooks and notes.
So, I use notebooks instead of notes.
So what you can do
for all the cells you can hide or...
Hide or show the code.
Hide or show all the output
and clear on the output.
There are several also views.
We use the default one.
But if you use simple,
you remove all note level controls
and for report
you remove
all note level controls and all code.
So it becomes a very static report.
For each set or paragraph
you have the following menu with the
self-explanatory options.
You can control the width of your cells
or paragraph,
and thus create a very simple dashboard.
But it’s very quick also to do so.
You can add a title
to your cells and this we use consistently
and on notebooks.
It’s a good feature.
You can show the line numbers,
so that is less useful
because the programing style in notebooks,
usually
you don’t put a lot of lines in one cell.
If you need to show the line numbers,
I think it’s
maybe time to reconsider
the what you’re doing in the cells.
You can prevent a cell from running.

A use case for this would be if you have a
a paragraph that trains a model
and that particular cell
takes 4 hours to execute.
You don’t want to execute this cell
by mistake or by distraction
when you click on at the top
of the notebook or in other cells.
So sometimes it’s useful to disable runs
and that’s it.
So by default, Zeppelin stores
the notebooks in its own folder.
So that’s the best
practice is to change that
in the configuration of the Zeppelin.
Also good practice is to clear the output
before you export your notebooks.
So it stays on this space
and also it it reduces the risk of being unable to import your notebook back in.
One thing that you should
strive for when you are creating
notebooks is to minimize the dependencies
with your local environment
to increase reproducibility.
One also important feature
and one that we implemented on the books
is to make those notebooks
independent so that it can run
multiple times without error
and that involves an extra effort.
Usually with either an HDFS, so that for instance,
you drop a table
before you create the table
or you delete a directory
before you create directory again
and so on and so forth.
Use the paragraph titles also, it’s a good feature
and use markdown
and visualizations
to tell your story with your notebooks.
WEBVTT
So you should have
credentials
that enable you to log on to CDP Public Cloud.
I have a mine.
and so this is the landing page
of CDP Public Cloud,
and I’m going to click on Management Console
and this will display
the list of environments
that are available for me.
There’s only one,
I’m going to click on this link,
and this environment contains
one data hub.

Okay.
Also a data lake.
So, which contains the
SDX layer of CDP.
I’m going back to the data hub
and I’m clicking on this data hub cluster
and this data hub just there
contains services, one of which is Zeppelin.
So it is linked to the data lake.
The data lake will
and all the SDX functionalities.
And so we have those
selected services
as part of this data hub.
But I’m going to click on Zeppelin.
Okay.
I’m going to switch
to a friendlier resolution
and here you will find in the DE folder
all the notebooks
that are the exercises for this class.
So I’m going to demo
and I have the choice for this class.
I will use the PySpark versions of the notebooks,
but they are also available in Scala.
I click on this link
and I get access to the notebook.
So another book is a set of paragraphs.
Each paragraph is linked
to an interpreter, which can be different.
It’s one of the distinctive features
of Zeppelin
to be what we refer to
as a multi kernel.
So, I can have a look at the
interpreters that are available for this notebook.
We’ll mainly use
Livy, Markdown
and maybe SH.
So when I have this and I can execute
so that I can execute each
cell individually by clicking on this icon.
Or can execute all of them
by clicking on this
icon at the top of the page.
Quick point on vocabulary.
I use cells and paragraphs,
those blocks of content, the correct wording
according to Zeppelin simply is paragraph
and for notebooks
I use notebooks like it’s written here.
Notebook,
but Zeppelin itself is not consistent.
There’s sometimes in notebook,
sometimes note,

but I will use notebook.
The paragraphs are marked as finished
because I already executed
the code,
but I'm going to execute it again.
So those marked down.
There's a paragraph quickly executed.
You see here in the previous execution,
the version of Spark was 2.4.8.
I changed them
something in the configuration to point
to the Spark version 3.2.
So I'm going to click on this
and as it says in the cell above, it's
going to take time
because it's the initialization of Spark,
which includes a negotiation with YARN,
it takes sometimes a couple of minutes,
but it's the it's the price
you pay with the first paragraph,
the first Spark paragraph.
Okay, so now I'm using Spark 3.2
and to interact with
HDFS, I need to have a Kerberos ticket.
So this paragraph takes care of this,
you see here.
So what we had previously,
we had Shell, we had
Markdown and Spark.
There is now, we have a Shell there
and here's another one which allows me to
download from GitHub
the file on which I'm going to work.
So this file contains data
about the Silicon Valley TV show
I use the Wget.
So it's on my local file system.
I'm going to upload it in HDFS.
Okay,
and I can check that it is in HDFS
and it is there.
Okay.
Now I can do some Spark processing
on this data,
so I'm going to read the data
from HDFS into a dataframe
which is called SVEpisodes.
We'll study dataframes later on.
But for the time being
it says it's just
a Spark object
that represents data in a tabular manner.
So it has a schema
which I can ask to see.
Data is in JSON format.

It reads from the JSON file,
the name of the
columns, and it infers the types.
So when you get data
like this from Internet,
it’s nice to have a data cookbook,
something that tells you what is what.
And so we see that
we have episodes of the show
and for each episode
there’s a brief summary of the episode
we can ask to see the first by default, 25
rows of a dataframe.
And from what we see,
it’s a very clean data.
There’s no missing field,
no funny characters.
So it’s we don’t know
if it has been created before,
but it’s clean.
So what we can do with a dataframe
is create a temporary view
and this will allow us to use
an SQL statement
so from this dataframe,
which is a data abstraction of Spark
we create in
Hive, a temporary view with this name.
Okay, so
once we have this
data in a temporary view,
what we can do is use another
magic like %SQL
and use a SQL statement
that will be processed by Spark
and see the output
in a nice tabular display.
And we have the option
of a couple of visualizations.
So it’s not something
that is going to replace
a fully fledged BI tool, but
it allows you to explore
data conveniently without a 3rd party
tool.
For instance, here we can
look at the number of episodes by seasons
in a bar chart.
And so that was SQL processing,
but you can do also functional programing
So here with
this line of code,
creates a dataframe
containing the words
which we process again to see,

to count the frequency of words.
So this is something that can be used
in natural language
processing to understand
what one piece of text is about.
So let's assume
we have no prior knowledge of the TV show.
We want to know what this TV show is
about.
So, our first attempt at this is disappointing
because we haven't removed
what what we call stop words.
So we need to remove the stop words.
So there are better ways to do this,
this is a manual way.
I'm just adding this because
I know it could be useful.
Okay.
So I add
these stop words
and create this list of stop words
and punctuations and then
I can filter the previous list.
Okay and if I look
at my results again now, I see that
very likely those are the
characters that are in this show.
Okay.
So without any outside knowledge,
by just processing the
the summaries of the episodes, I know
the names of the characters of the show.
I think those are the characters.
I'm not a fan of this TV show,
but I think they are. Yes.
So that's it for this quick introduction
to Spark and Zeppelin.
The goal was to show you how Zeppelin
works.
It's very intuitive and for you
the most common feature of Spark,
it allows you to do SQL processing
and also programing for
ETL purposes.
WEBVTT
In this chapter
we're going to take a look at HDFS.
HDFS is a solution
that enables users to store files
that are larger than what a single hard drive can store.
So that's one of the motivations
for distributed storage.
The other one, of course, is to be able
to process those files.
In order to have distributed processing,

you need distributed storage.
There’s always some apprehension
when you split the file into file blocks,
but Hadoop and HDFS have been designed
to be fault tolerant.
You have to bear in mind
that Hadoop was designed at Yahoo
by a guy named Doug Cutting to run on commodity hardware.
So I was not part of that team.
But legend has it, that it would take 16 gigs of RAM, motherboard,
put 400/500 gigs,
disk drive on the board and that would be
another node in the cluster.
They did not
buy the hardware from reputable vendors.
They made their hardwares themselves
and this is where the idea of big data is
cheap, and storage is cheap, comes from.
It certainly isn’t as true in your context
where
in your organization hardware is procured
from reputable vendors, there is no,
no one in the dark corner of the office
building nodes for your cluster.
So it’s not as true as it was in at Yahoo.
But the idea of storage is cheap,
is built in this software.
So, what’s the idea? It’s very simple.
So you split the big files into big blocks
and you replicate the blocks,
a number of times
across the cluster and the standard
replication factor is three.
And, so that gives you a good compromise
between the cost of the storage
and the resilience and performance
of your cluster,
of your cluster, of your storage.
Okay.
So you can see in the diagram here
that if any given node goes down,
you still have one
copy of the file blocks.
So it’s expensive.
It’s expensive,
it’s simple but expensive
and this is why it has been the request
from customers
to bring the cost of storage down
and this is the motivation for projects
like erasure coding and ozone also.
HDFS looks like a file system
and there are many ways
that you can interact with HDFS
you can use Cloudera manager

and use Hue,
you can use the Name NodeUI for web UI interaction.
Or you can use the command line with HDFS space DFS
.
So it looks like Unix,
but there are 30 commands in the Shell,
most of which are straightforward.
If you come from
a Unix background, of course.
WEBVTT
The whole storage, uhm,
distributed storage, is implemented
by two components.
The NameNode,
so it's a master work architecture.
The master is the NameNode
and the DataNodes.
The DataNodes are daemons
that report to the NameNode
that are stored
on the workers of your cluster
and the workers implement
the real storage.
The NameNode stores in memory
the metadata about your files.
This is a very frequent pattern
in big data.
You have master daemons
dealing with metadata and workload
daemons, dealing with the real data
that only you will find in
in HDFS, in Hive.
It's very frequent.
So because metadata is small, well smaller.
In the case of HDFS, it can fit in memory
until it cannot eventually,
but usually it fits into memory.
And you can think of the NameNode
as a in-memory lookup
service with one main table.
In this main table, there will be one
whole profile, each file
there will be a one
too many relationships with the blocks.
And this is called the block map.
I told you that the HDFS was designed
to failover
to be fault tolerant. So the way it works,
and again, this is a pattern you will see
again and again in big data projects.
The failure of a component is materialized
by the absence of communication
during a configured period of time.
This implies that the components
of the architecture

permanently communicate with each other.
So this is implemented with heart beat.
So all the DataNodes send heart beats
to the NameNodes on a regular basis.
By default, it’s every 3 seconds
they send heartbeat to the NameNode.
So after a while, if the NameNode
does not receive heartbeats,
it will consider the corresponding
DataNode dead and we’ll proceed to copy
the blocks that are on the dead DataNode.
Copy them from another replica
to another DataNode
so as to have always its replication
factor equal to 3.
And this is the way it heals
itself.
So
writing to HDFS involves several steps,
and the first steps involve the client
and the NameNode.
So the client asks to store a file.
The NameNode creates a lease to the file
path.
Then there’s an exchange about the blocks.
Eventually the NameNode provides
an array of DataNodes
on which to store every block,
so it’s an array of array.
For every block there’s an array of nodes.
And this is the end of the exchange
between the client and the NameNode.
After that,
the client send for each block,
it sends each block to its first DataNode
in the corresponding array,
and then each DataNode
creates the pipeline of replications
to all the DataNodes in the array.
So by design
there’s no large data exchange
between the client and the NameNode.
All the large data is sent directly
from the client to the DataNodes.
And by design
also, there’s only one exchange
between the client and the cluster.
The rest of the copies are made
inside the cluster.
Of course,
this is a simplification of the process.
There’s also acknowledgments
every time a block is accessed,
created or read,
there’s a checksum is made

and an acknowledgment is
sent back to the NameNode via
the client and those checksums
are material to alert the NameNode in case
there's a corruption in one of the blocks

so the NameNode is accessed
by having 3 strong replicas
of each block, not two, not four, three,
and they need to be not corrupted.
Also, bear in mind that this architecture
was designed
at Yahoo on a physical cluster.
In a physical cluster
you can make Hadoop rack aware
by providing a rack,
to all your nodes,
assigning your rack to all your nodes.
So it's a very simple system.
Hadoop, HDFS will consider
that your nodes are in the same rack
if they have the same string
assigned to the rack property.
And then it will apply
this as a simple topology to say, okay,
if it's in the same rack then it's
less costly to to copy a block.
If it's not in the same rack,
it's a little investment.
But I will make the investment
for one of the copies
so as to be sure that not all the copies
of a given block are in the same rack.
So that this way Hadoop and HDFS
are immune
to a top of the rack switch failure.
WEBVTT
If you know
anything about architecture,
you have understood already that by design
the NameNode is a single point of failure
and that was the case until Hadoop 2.
If you lose your NameNode,
if somehow your NameNode
becomes unreachable, then you have lost
just lost a very, very big hard drive.
Since Hadoop 2 the NameNode,
It's possible to make a HDFS highly available.
It's something that you can do
with Cloudera manager.
It takes 15 minutes
and it's driven by a result,
So there is really no reason
to not make your HDFS highly available.
If you have a couple of master

nodes,
it doesn’t require additional hardware.
If you have a single masternode,
then it’s difficult.
It’s impossible.
But if you have a minimum
of two then you can make your
HDFS highly available.
So theres built in security in HDFS.
The built in security is is not enough.
It can be worked around quite easily.
So the real answer for authorization
is Ranger.
Which is beyond the scope of this class.
HDFS
provides is the POSIX permissioning
and also Access Control List,
ACLs, but for a proper
authorization and also audit,
because if you have
just the first two bullets,
then you don’t have an audit.
And that’s also a problem for security.
That’s one reason why you need Ranger.
And while you are at it
then Ranger for authorizations.
Also,
you can assign quotas, and for quotas
you have always these dual
aspects of objects
in big data, you have the the size
of objects and the number of objects.
So the size impacts
the, in HDFS, the storage.
Okay?
The workers. And the number impacts
the NameNode, so it impacts the metadata.
So you can assign quotas for users,
by saying, this user is allowed this
size of data and this number of files.
WEBVTT
HDFS has been around now for something
like 15 years,
so it has been proven okay, it works,
but also its shortcomings have been
well documented
and known for a long time now.
And the new Ozone
Project builds on HDFS
and also fixes the bottlenecks of HDFS.
It’s the next system for doing big data.
It’s available in CDP, okay.
One of the problems with HDFS was the
the problem with small files.
Small files and many files,

I will explain those too.
This came from the limit of the NameNode.
Each file in the NameNode memory takes
between 150 and 250 k,
and that’s regardless of its size.
So instead of having one big file,
if you have a million
small files,
you take 1 million more time and space.
The limit of files that can be stored
in the memory of a NameNode
is around because it’s difficult
to get there,
But it’s around 350
millions of files of objects and
those are the two main problems of HDFS.
So, the way that Ozone fixes
those problems is by introducing
another layer of storage
called Containers.
And this is better
explained in this diagram.
Because you have these
new managers, storage containers,
you can, in one container,
you can put many files.
So this diminishes
the number of objects to handle.
Okay,
so it fixes the problem of small files
by introducing this new layer of storage,
those containers,
and then the too many files
is also fixed by this
layer
which allows for server Ozone manager.
So you can have like adding
several NameNodes
and that fixes
the problem with too many files.
So too many files is fixed here,
more files fixed by containers.
As you can see, the
problems are with additional complexity
and that’s always the Ozone pattern.
Projects and systems become more complex.
They solve more problems,
but they also become more complex.
The blocks are stored now in containers,
that’s the container layout.
WEBVTT
In this notebook we’ll study
the common lines within HDFS.
So you have to go into
the DE folder, look at the Labs,

and the notebook is working with HDFS.
So the notebooks
have all a similar structure.
There’s an
about there which identifies the notebook.
There are set up steps that are required
for bringing the environment
to the level that is expected
for the lab to start.
Then you have the lab part where you are
asked to execute some command,
sometimes with some help
from the markdown instructions above,
sometimes without.
And if you get stuck,
then if you scroll down
you have the solution.
Okay, so let me execute this
step.
So what I will do is
use this feature of Zeppelin.
I will scroll down to the solution
and execute
all above.
So this will execute the set up steps
and go quickly through the
empty cells.
Okay. So we are ready.
The HDFS
directory is a forward slash.
So,
the command
to lease the content of the HDFS root directories
is HDFS (space) DFS
and dash LS
then space, forward slash
So we see three folders.
There’s a TMP, /user and /YARN.
So what is in /user.
So we add /user
and we see
this is where the home
of all the users of the cluster
as told.
Okay, the one I am, dev 16.
So my folder is dev 16,
my own folder is dev16.
So I can if I omit
the directory by default, LS will give me the
the content of my home directory
My home directory in HDFS has already been populated by
some data that I will use
for the exercises.
I’m going to create a new directory called,
"loudacre’

with a dash mkdir command
and copy
files from my local file
system located in /var/temp
/dev/data/activations in loudacre.
Okay, that is done
so I can check the the successful
copying of the files
with the HDFS DFS -LS command.
Okay.
I have copied 132 more .XML files.
If I want to have a
a peek at what the files contain,
I can use
a -cat.
But remember,
in our case we are putting small
files in HDFS, which is
a bad habit.
Okay.
But when you're doing a -cat in HDFS,
you should be careful
because files should be big. So,
it's a good idea to do a | head,
and it confirms the
activations, okay.
What you could do also
is do a -tail of the same file, okay?
It gives you the bottom of the file
and because,
let's test if head is implemented.
It used to not be,
but now it's available.
So -head is available.
So the reason for doing the | was that
head was not available previously.
So I can retrieve files from each HDFS
on my local file system with the get.
So be careful when you're doing that.
You want to retrieve on your local file system small files.
If you try to retrieve a file
that is larger than,
your local drive,
you will overwhelm your client.
Okay, I can check
locally what I have
downloaded.
Okay.
And I can also delete
the directory in HDFS
using a -R are for recursive
and -skipTrash by
not putting the files in the HDFS trash.
So the HDFS also has a trash system
which works

like a folder, its .trash folder
that is usually hidden
but that contains items that are removed
and the trash also has a
lifecycle. So
it’s not permanent.
You have to,
the trash is emptied on a regular basis.
This is configured by your admin, so,
you still need to be careful
when you delete files.
from HDFS.
That’s it for this HDFS note book.
WEBVTT
So YARN stands for Yet Another Resource
Negotiator.
It’s the second core component of Hadoop.
It’s the one that implements
distributed processing.
You can think of it as the operational system, the OS, of your data center,
of your cluster.
So it provides centralized
resource management and job scheduling
for multi types of workloads.
So the idea is that you share
all those resources
across multiple types
of workloads and across multiple users.
That’s the idea of a cluster.
This is the architectural component.
So you have the the master daemon,
which is the Resource Manager.
You have for each node,
you have a Node Manager
and for each application
you have an Application Master.
Those components of different scope of,
scopes of responsibility.
The scope of the resource
manager is the cluster.
The scope of the node manager is the node.
And the application master
is at at 90 degrees angle with the rest.
Its scope is the application
So I think of the application master
as the project manager
for your application.
He has no hierarchical link
with the node manager
and the resource manager.
Its job is to carry out the application
and oversee
the good execution of the application.
So and for this it will need
the resource manager, so it will

ask for resources to the resource manager
and the resources
in the resource manager
will grant or not.
And that’s the negotiation
part of YARN.
The resources to the application master
for the execution of the application.
The execution of the application
will be carried out in containers
managed by the node managers.
Those containers, we use the same word
containers, those are JVMs.
So with a specification for CPU and RAM.
WEBVTT
So together the
ResourceManager and the NodeManager
implement distributed processing
with the different scope
of responsibilities.
And so you will find
two types of containers in a YARN cluster.
You have the containers
that execute tasks on the application,
and there’s a specific
one for the Application Master.
There’s one per application.
So for a Spark job,
then you will have an application
master and several executors the
the task containers are called executors
in the context of Spark.
So this is where the executors,
this is where the
Spark distributed processing will happen.
Okay.
So heres an empty cluster
and you see visually
the symmetry between the HDFS and YARN.
Okay.
So this cluster has four nodes.
So application are typically submitted
from the gateway, a gateway node
to from outside
the cluster on a machine that can connect
to the cluster.
So when you do a spark-submit MyApp,
the ResourceManager immediately
creates an Application Master container.
Okay.
And what the Application
Master does immediately
is ask for more resources
because the Application
Master, like a good project manager,

cannot execute anything.
It just distributes work.
And this is where the negotiation happens
between
the Application Master
and the ResourceManager.
So the Application Master says, Please,
can I get three executors
with one gigabyte of memory
and one core.
And depending on the
the state of the cluster,
if it’s busy or not, the ResourceManager,
will grant
the total request
or less than the request.
And from this
this can be configured by the settings
in YARN, that tell YARN
what is the minimum container size.
What is the maximum container size
and what are the increments in between?
And these three properties in terms of CPU
and memory.
So in our case, we are lucky
we have three executors,
the cluster is idle.
No contention on the
resources since the cluster is idle,
executors are located on these data
nodes.
It’s a safe bet to bet
that the data that needs to be
processed
is also located on Node A and B.
Okay.
When the application terminates,
the containers are terminated.
WEBVTT
For a developer,
What you need to know about YARN is
you need to be able to submit your jobs,
monitor your jobs, and manage your jobs.
For this, you have several tools.
You have the YARN web
UI, and you have a command line interface;
in case you prefer the command line
or you want to automate
and script your tasks.
So the
ResourceManager
UI gives you a lot of information
about the state of your cluster,
so you have the overview
that gives you the cluster resource usage

globally or by queues,
or the finished app from all users.
Okay, so this is a good starting point
for troubleshooting this page.
Then you have the queues.
So in order to control
multi-tenancy and manage
multi-tenancy,
a cluster admin can define several queues
with several properties,
and in our case we have only one queue
called default which inherits from all the
root
queue, which is a virtual queue.
So everybody has access to
all the resources of the cluster.
To click on applications,
you see all the applications
that were started
on the cluster.
And if you look at nodes you can
look at the list of nodes available.
It seems that
when this screenshot was taken we had only
it was a one cluster node.
From the command line,
you can do many things.
So, the command lines will start with YARN
and then if you want to interact
with application,
it is YARN base application.
One common usage
is to do a list
to get all the applications ID,
because this is
the identifier you need
for instance to kill an application
or to get the logs from an application.
If you
want to look at all the options
you have the YARN -help command.
So like HDFS, YARN is getting old,
and there are new kids on the block.
So, the one,
the latest one is Kubernetes.
This means Pilot in Greek.
It comes from Google,
and it uses containers.
So this time with containers,
not JVM but with containers.
And it’s scales faster and better.
So it’s a better mousetrap.
And we use Kubernetes,
in our data services
like Cloudera Data Engineering,

for instance, or CML.
Yes, you have several concepts,
new concepts, so you have the Kubelet,
which is a container agent,
and you have the pods,
so that’s the smallest and simplest
Kubernetes object.
And the service is made out of Pods.
So you have a hierarchy of concepts
again, like with Ozone.
WEBVTT
In this note book we are going to study
how to work with YARN.
We go into the DE folder, open
the lab folder, and
there is working with YARN notebook.
I’m going to execute
all the setup.
Okay, that’s it.
So,
we are going to have a look at the-
First,
we need to have a look
at the ResourceManager.
Okay.
Let’s try to go to this URL.
Okay.
Now I have the ResourceManager
landing page.
There’s one application
that is running.
On to step number two.
We have only one application running.
Okay.
Let’s look at the nodes.
So we have four nodes.
And only one has one container
running.
So the application
we saw previously is running
on this node.
And let’s look at the
content of the,
let’s drill down on this node.
Okay,
and we can see the resources that are used
and we can look at the applications
on this node.
So this application that we saw
that was running is here,
and it’s running in one container.
So you have quite a comprehensive set
of information
in the ResourceManager UI.
So now we are going to run

some application,
so we need some data
and in the setup step, we created
a short word count program
that we’re going to run on this data.
So if we go back now,
application is running.
If we go back to the applications;
okay, this is our application
that is running.
Well, it’s not running; it’s accepted.
So, it means it’s waiting for
an application master to be allocated
by the ResourceManager,
and now it is running.
Okay, so
let’s look at the node level.
Where is it running? So
it’s already
finished,
the applications already completed.
Okay,
so we can still have a
drill down.
Okay.
So but we cannot look at the,
the resources used by the applications.
So this is not kept
as part of the historical data
about this information, this application.
Let’s run it again.
But this time, let’s go
let’s be quicker about this.
Okay,
now it is accepted.
If I go to Node,
okay, there things are running everywhere.
and list
And list of applications on this node.
Not sure this is mine-
Yes, this is me, the user.
That’s me. It’s running.
And if I look at the containers,
okay, it’s already finished.
But you get the idea:
you can drill down and see exactly
what the application is running on
which node
and in which containers.
Okay, so
we could have used this command line
to do the same thing.
So by default,
it shows you the running
application.

So, as there are no running applications
then the list is empty.
So if you want to see all the applications
in all the states,
you have to provide this flag.
-appstates ALL,
and then I had to use some kind of Shell
scripting to filter
on my specific application.
So,
yeah, I had two applications
because I ran the program
twice.
Okay.
And this is all the information
that you see in the Web UI,
but this time
you see it on the command line.
And with
this small example, you see that
the more you
you can do with the Shell script,
the more you will get out of this command
line system.
That’s it for this notebook
about working with YARN.
WEBVTT
In this chapter
I’m going to talk about a little
of the history of distributed processing.
I split the time in three buckets, so
the first bucket is from the year
And I call this bucket the "Disk Years"
because at the time,
frameworks, mainly the MapReduce
framework, relied on disks.
Then from 2010 to 2020,
I call this decade the "Memory Years"
because distributed processing frameworks
started to leverage the memory
starting with the Spark.
And for the next decade,
I predict with my Google hat on
that it will be the "GPU Years."
MapReduce is a method to
distribute processing, okay.
If you have a large problem,
you can- you need to,
if you can split the problem
and distribute the chunks of the problem
to several machines, then
you can have those machines processing
parallel, subset of the problem,
provide you with the intermediate results.
And this intermediate result you need

to aggregate to get the global results.
So this is the
the MapReduce
paradigm, but it’s a bit
more complex than MapReduce.
While I was presenting it,
I already mentioned splitting
which occurs before the map,
because if you don’t split your files,
your data,
then you cannot process in balance.
There needs to be a split
before the map.
And between the map and reduce,
there needs to be
several things in this diagram
we see shuffle
and sort, but for the initial MapReduce
framework
it would be map, it would be shuffle,
sort, and group sort
before reduce happened.
So, all of the this distributed
processing framework
that are common nowadays,
they use a massive simplification
of distributed processing,
the user model called the "Shared Nothing Model,"
which means that
components of
the processing
do not communicate with each other.
So a mapper does not communicate with other mappers
or reducers do not communicate with other
reducers and in Spark
where executors do not communicate
with other executors.
All these processing units process
data in isolation.
They get,
they get the data from the
the drives.
Hopefully, the local hard drive.
And they get the processing
from the Spark driver.
And that’s all they need
to do to perform their processing.
This is a huge simplification,
but it works
well for what we’ve been doing
with classical IT for years.
It’s counting things, you know,
in the bank you’re counting money;
if you use telco, you’re counting calls;
For counting things, it works.

For things like solving a puzzle
it would not work.
If I had ten students
in a classroom and would distribute
pieces of a puzzle to those students
and say to the students,
"Solve the puzzle for me now,
but do not talk to your neighbor."
The process will not end because
necessarily at one point in time.
One student
needs to communicate
with another student to say, "Hey,
do you have a piece of the puzzle with
this kind of shape and this kind of colors?"
And in MapReduce or Spark,
you cannot do that.
So it works for
counting things like coins.
Okay.
And yeah,
we have this natural tendency to resist
to doing this with processing
until we cannot no longer escape
this rigid processing window.
The problem at hand
is embarrassingly parallel.
And then
we do this with processing.
The Mapper
reads the data in the form of key/value pairs.
So what happens
is that before the mapper,
the split happens and
in the case of text files, outputs
the text in the format
with key as the
being the offset of the line and value
the content of the line.
And what the mapper does is
apply
whatever processing you want to happen
to this key/value pair.
And I'll put a new set
of key/value pairs.
So in the case of a counting words,
it will output
key/value pairs, with the key
being the token
that has been isolated from the line
and the constant one.
You can already make some comments
about this framework.
The size of the output of the mapper
is of the same order of magnitude

of the size of the input.
And the input was lifted from HDFS,
the size of the output
is going to the local drive.
It’s always a potential problem
and also the processing
done by the mapper is not optimized.
So in the second line there is twice
the word "I" and the simple processing
from the mapper outputs twice
the key/value pair (’I’,1), (’I’, 1).
So this could be optimized
by adding another stage in the third
pipeline called the Combiner,
which is a map side reducer.
And then we combine
(’I’,1), (’I’, 1) to output
(’I’, 2) before the shuffle.
So the shuffle is the next stage
and it’s the one
that creates problems
for you in distributed processing
because it relies on a scarce resource,
on a shared resource,
which is the network bandwidth.
Remember that in this
architecture, CPU is scalable,
RAM is scalable,
storage is scalable, but network isn’t.
Your job as a data engineer
is to try to minimize the impact
of your processing on the network.
You can try, but
you cannot avoid altogether
using the network
because the processing
we like to do always involves
joins and group-bys and by design,
those operations require
the data to be recombined,
so that the joins can happen or
the group-by can happen so
it’s something that you have to be aware of.
So between the
the mapper and the reducer,
the shuffle took place but also the sort
and the group sort took place.
And so the result of this
is this what you see here,
I, the value now is a structured value.
It’s as an array of values.
Okay?
And the job of the reducer is to sum up
all the members
of the array of the value.

So that’s the job of the reducer.
So to summarize for MapReduce:
doing a word count
with MapReduce, you would have to read
from HDFS, split the files
then feed
the key/value pairs to the mapper
that would perform the
the map processing and output
new key/value pairs with the token
and one as a constant.
These are to be shuffled
across the network to reach the reducer
that would eventually sum
all the values in the value
of the input pair
and then store the results in HDFS.
And this is the price to pay
to have a scalable processing.
Okay.
A non-scalable
processing would be what a Java
developer would write naturally
for every document in every file in the
in the folder, for each line,
for each word.
Add one to a counter and then exit.
So there would be several
loops nested for loops and then
the program would end,
but it would take forever if you were to
do to perform a word count
on the large set of files;
or even worse, a set of files
that would increase
faster than you would count.
Like all the data from the internet.
The process would never end.
So to do this simple task,
you would instead of doing
the three nested full loops,
you would have to write one class
on the mapper
that would inherit from the virtual class,
another class of the reducer
that would inherit from the virtual class,
and stitch everything together
in a main class,
which would set everything
and then launch the
execution with the last line.
WEBVTT
So this
was the only framework available
until Spark arrived into 2010.

This created a lot of frustration
by the early adopters
like Facebook and Yahoo.
That’s why they created
frameworks like Hive and Pig to overcome the
the low productivity of the Java MapReduce
framework.
But also it gave ideas to people
starting in 2010 when the,
they could leverage better hardware
with more memory,
more disk, and more network bandwidth.
In 2009
a guy in a Berkeley lab
started to create a general purpose
data processing
engine leveraging memory instead of disk
for storing intermediate results.
The idea of storing everything to disk was
the consequence
of not having a lot of memory,
but also from using hardware
that was not reliable enough.
Remember
at Yahoo, they used commodity hardware.
So any given day,
commodity
hardware is a fancy expression for
glorified PCs.
So on any given day something would break.
The motto of the CTO of AWS
is everything fails all the time,
so that’s why because they are
not a lot of memory and hardware was not
reliable enough,
they would save everything to disk.
So that was a huge
impediment for performance.
Another one was the MapReduce
graph was the only graph
that these frame work would tackle
was limited to having a single shuffle.
So you could do map
and then shuffle and reduce.
But if you had to do something that
involved several shuffles, then you had to
change several MapReduce jobs.
The creator of Spark
was aware of these shortcomings
and created an elegant solution
written in Scala
that would eventually be available
in several languages
Scala, Python, Java, and R
that would be

with a dedicated libraries
for this SQL processing, Machine Learning,
Streaming, and graph algorithms.
So that was also a very cool feature
because prior to Spark,
if you add a very complex use case
involving streaming and machine learning,
you would have to tackle several projects.
Okay, so you would do maybe some Pig,
some Hive, some MapReduce with Mahout,
maybe some Storm, so its
five or six projects to
to come up with a solution
to the use case.
Whereas with Spark
you could do only Spark.
So that’s one of the cool features of Spark:
the fact that it uses memory,
well, it makes it much more efficient
for processing and also effective
for its iterative computation,
such as machine learning algorithms.
Okay, well,
most many
of them are based on
iterating on a gradient.
And so there was a machine
learning library available
with the Java MapReduce framework, but,
well, it took a long time to,
to process things with this library. So those,
early adopters had to find
workarounds to perform machine
learning at night in a batch
and then store the results
and leverage the results during the day.
So, it’s also a Spark is a good open
source tool
that is open and works
with a lot of systems.
That’s the, that used to be the baseline
of Spark
that was on the landing page.
Now this diagram is
not on the landing page.
It’s on one level down,
but that’s mainly the,
the result of the using memory
instead of storing things with HDFS
between processing.
So you get a 100
factor performance improvement.
Also compared to the
Java MapReduce, it was much more
compact.

Okay, so the less scrolls
you write, the less mistakes you
you are likely to make.
So that was also an advantage
of the solution.
It relied on functional programing.
So why
this interesting functional programming?
Because it's
it's very suitable for this
with processing,
you want all those qualities.
So because we are doing a Shell Nothing Model,
we want data to be immutable.
Okay?
We don't want a one executor
to change the value of something
that is already used by another one. So
because there's no way
to communicate the change.
Okay, so
data has to be immutable.
We won't know state or side effects.
Okay?
That's something that functional programing does,
and we want to always
get the same result
from the same input.
And because we're doing parallel execution,
we want to be sure that
processing the same inputs
in two different executors
will provide the same outputs
so that we can aggregate the same outputs
and get accurate results.
It's also using lazy evaluation.
Okay.
So that's something that a modern language,
programing languages do
is to wait until the last minute
to perform evaluation,
and thus it gives them
the opportunity to optimize the graph
that they are going to execute.
So yes, that sums up the thing.
So if you have immutability
plus functional programing,
you get predictable output from
the same input
and then
this allows for safe aggregation.
And that's the comparison between
Hive with Tez and Hive with MapReduce.
So Tez is part of this second generation
of distributed processing

engine that rely on memory
and can tackle complex graphs.
So distributed -
sorry, so directed acyclic graphs,
so DAGs.
And you see the difference with MapReduce,
you had to chain four jobs
and in between those jobs
you had to write to HDFS and read
from HDFS.
And then what you really want to do
was reduce, reduce.
But then you had this dummy mapper here.
So the mapper would be the identity
function to perform the reduce again
and then store the results in
HDFS and to do another
dummy mapper to
do the
reduce and same on the other branch.
So there was a lot of waste of processing
in this initial MapReduce framework.
And with Tez and Spark,
they are able to do these,
these DAGs
and the do not need to write
with HDFS between
between steps.
Its a huge improvement
and you see the
query is a simple one.
So imagine if
you want to do a more complex query.

This query -
to do this query,
you would need to write 12 Java classes.
Imagine the
look on the face of the business,
analyst at Facebook
that you are to use SQL
to query
the logs of Facebook to infer the price
of advertisements
in various places of the website.
These guys
did not want to learn to program Java
to do something as simple as that.
So that’s
the motivation
for the creation of the Hive project.
WEBVTT
So from 2010 to 2020,
we had several processing engines
that would leverage memory.

Okay.
But nowadays,
GPUs are being more and more
easy and frequent
to use.
And the change also the,
the workflow of people
who are able to use them.
Okay.
You see in this comparison that
when a data scientist can
use a GPU enabled machine,
then you can perform a lot of iteration
in a given day
and optimize this work, whereas
it can only do so much if he has to wait
using CPUs to train his model.
So starting with Spark 3.0,
there was this project with NVIDIA
that allows Spark
to leverage GPUs.
So what the project does is to
switch the dataframe library
that is based on CPU
with another library that is using GPUs.
So it's transparent from the user.
As long as you have a GPUs
in your cluster, then Spark will switch
to the appropriate library
to perform the dataframes operations.
So it's good for the processing.
It's also you see the importance
of the shuffle.
It's also good for the data transfer,
because if you have several GPUs,
then Spark can leverage -
optimize that work
between GPUs and avoid
sending everything to the PCI bus.
So this library works for data frames.
Okay. So
it doesn't work for data sets apparently,
and certainly not for RDDs.
Okay.
One that's,
I think, that's a compelling reason
to stay away from those data
representation and use data frames.
And so for the time being,
it's for Spark SQL only, not for Spark
Streaming and Spark Machine Learning
Library or Spark Graphics.
So it's an ongoing project.
Okay. So maybe those libraries
will be updated in the future.

But the vision is the following, is to
instead of what we were doing with
Spark 2.0 was to do data preparation
with only CPUs and then the save
the GPU processing for model training.
The idea is to power the whole
data pipeline with
GPUs from start to finish.
WEBVTT
In this chapter, we are going
to look at the core concept of Spark,
which is are RDDs. So RDDs
stands for Resilient Distributed Dataset.
It’s resilient
because if data is lost in memory,
it can be recreated. It’s distributed because
it’s processed across the cluster
on the different executors
and the less distance from dataset.
This was the original data
abstraction in RDD.
With Spark 1.3,
the Spark team introduced DataFrames,
which is now the preferred
data abstraction
for Spark programming.
DataFrames
have a lot for them going on,
and there’s no
compelling reasons to use RDDs anymore,
but you still might find them somewhere
in legacy Spark code and so
and it’s good to understand
our Spark work at its core
because the code that you write with
DataFrames will eventually
be translated to RDD code,
that is going to be optimized RDD code.
So that’s what’s good about DataFrames.
So RDDs are created by
loading data,
uploading data in memory from files.
Okay.
Then RDDs undergo
sequence of transformation,
and that sequence of transformation
is triggered by an action.
So it’s lazy evaluation.
So it’s when Spark,
reaches an action that the
RDDs materialized in memory.
So uploading the data
from the files into executors,
processing the initial RDD
with all the transformations

and eventually materializing to the RDD
that you want it to
either display or save
and then the RDD will be evicted
from memory.
What I described
as the chain of transformation
is really a graph from a base RDD.
So inthe end
this graph is stored
in the driver. So each in the driver,
the driver holds a record of all the
RDDs that you define and
all the
operations that can materialize
those RDD.
That graph is executed
when an action occurs.
So this is where you really need
to see the data.
And this allows Spark
to perform optimizations.
And it’s also the key
to the resilience of Spark, because
if by accident a node fails, then
to recreate the lost partition of the RDD,
then it only needs to perform the
the graph, the corresponding graph
to get the new partition.
So the lazy evaluation is something that
a lot of modern programing languages do.
Let’s say you want to
see the first error message
from a very large log file.
So you first you create an RDD
that reads the log file
and then you filter on the line,
starting with error.
And then you say,
I want to see the first one.
It’s only when you say
when you say, I want to see the first one
that the file is read from,
from HDFS.
Filter and the execution stops
after the first line is read.
Spark does not read
the whole file and that’s
because of a lazy evaluation.
Okay.
Like other programing languages
would upload
all the file in memory,
then filter on error
and then and then show you the first one.

But with lazy evaluation,
you don’t need to do that.
You just
read the first line of the file,
process it in the directed acyclic graph
and that that gives you the first one.
So when an RDD
is created, it is partitioned
in the executors of the cluster.
And what happens is Spark
creates RDD partitions
based on the HDFS blocks.
So there’s a 1 to 1 relationship
between partitions
and the partition
distribution in memory.
This is a very simple scheme.
It’s also not great
and this is something that DataFrames fix
because the number of partition is
a key
element
in the performance of your processing.
So and there is such a thing as not
enough partition and too many partitions
and the right number of partitions
is something
that the DataFrames
can have, thanks to
one of the catalyst of the team at Cloudera.
So with RDDs there’s no optimization
of the number of partition.
It’s a 1 to 1 relationship
between a HDFS block and a partition
and something that could, for instance, be
processed into partitions
will be processed with 181 partitions
because it corresponds to 181
file blocks full of small files.
So that’s a lot of waste
of a lot of resources.
But then
what happens is that a task
can be executed in parallel on the partitions.
So this is where
all the processing happens.
And here we look at the example
where we read
data from HDFS,
then we filter to keep the error lines,
then we filter to keep
the MySQL lines. So
the idea here is to get the
the MySQL error lines
and filtering happens

in the memory of the executors.
It does not require the data
to be recombined.
In other words, it
does not trigger a shuffle.
So this is very efficient.
This is where
distributed processing shines:
performing those narrow operations.
This is called the narrow operations.
So if we look at the different partitions
here and so the initial file is split
using partitioning
into three executors.
So one of the consequences of
this is that if you relied
on the physical order of your file
for your processing,
this has been already lost by
the split
that happens. And another caveat
about distributed
processing, it's
by nature, by design,
although it's challenging for distributed
processing, it can be dealt with if you
specify explicitly
order, but if it's implicit,
then it's going to be destroyed
as partitioning.
So, we say when we do the first filter, okay,
it creates new partitions
in the executors memory.
Okay. With
only by keeping only the errors
and when we filter again on MySQL,
it keeps only the error on
MySQL.
So if you want to collect this, let's
assume this is the
the goal of all your processing.
You want to collect the MySQL errors.
Be careful
because as you can see in this example,
if your size of your log
file is one terabyte in HDFS
and we assume that the MySQL lines
are 10% of the data,
if you're going to collect this,
you're going to try to collect
a hundred gigabytes
on your laptop.
And that is very,
probably more
than what your laptop can take.

So be careful with collecting data.
When you’re using distributed processing,
it should be because the size of the data
that you’re processing
cannot fit on one single machine.
So you should collect only aggregated data
like maybe a count of your MySQL lines;
that would be something that you could do.
But the MySQL lines themselves,
if they amount to 100 gigabytes,
you shouldn’t collect those.
So always filter around achieved
its because Spark
has a record of the
what we call the lineage of an RDD, so
if for instance the fourth
executor goes down
or partition four is lost,
then Spark can replay the
the lineage and
retrieve the
the missing partition.
So what we say is that a
Spark is optimistic, so
Spark
was created
around 2010 when the hardware
was more reliable than the hardware
that was in the landscape
when Hadoop was created. So Hadoop
is pessimistic about the hardware.
Hadoop thinks
everything is going to break all the time,
so it saves two disks, Spark bets
that the hardware is going to be reliable.
So it’s optimistic, so
it thinks that the trade off
is a good one.
The trade off of
re-computing a missing
partition is a good bet
because this would be a rare occurrence.
So by default
RDDs once they have been materialized,
displayed
and/or saved, they are evicted from memory.
So if you want to keep your
RDDs materialized in memory,
you have to explicitly specify this.
Okay,
and so this for instance, in the
use case for
this is when you want to drill down
from one RDD
that contains all the errors

and you want to drill down on error
and MySQL and evolve and Spark
and so on and so forth.
RDDs needs to travel across the network.
So they need to be serializable.
Yeah, they are specialized RRDs like Pair RDDs,
but we are not going to talk about them.
The rest of the course will be about DataFrames.
We don’t want to include too much
of RDD stuff because it’s not updated.
RDDs can be created from files,
from data in memory, from other RDDs
and you can also get
to the underlying RDD of a
Dataset or DataFrame.
So if you want to create RDDs,
you can use the SparkContext object.
So it’s something
I should have mentioned here
also when you see DataSets.
Between the Spark 1
and Spark 2, there have been a lot of
concepts that were created
and some of them,
well, they are still around.
They’re no longer
explicitly used or they have been merged.
So initially there was one data
abstraction RDD then in 1.3
there was DataFrames, and in 1.6
they introduced DataSets
and in 2.0, they merged the two concepts
DataSets and DataFrames.
DataFrames represent
your data in a tabular
manner, and DataSets represents
your data in an object oriented manner.
It has a stronger, type checking
and with this strength comes
at the price of additional complexity.
And I don’t see a lot of customers
using DataSets
and DataSets are not available by Spark anyway.
So it’s something that
you can find, you can use in Java
or Scala version of spark.
Yes, I don’t see a lot of datasets,
but now DataFrames
are officially datasets
of that type row.
And likewise there was an object
in Spark 1 called the SparkContext.
Now, it’s something that is inside
the Spark Session object,
which is the main object for Spark.

And to access this, then you can use SparkSession.SparkContext.
And this is
and it’s by convention, it is called SC,
and this is what you use to create
RDDs.
So you can read
from a file to create an RDD
or from multiple files
or from HDFS.
And here we are using the Scala language
for which there’s a the Val/Key word that
there is a Scala that
this object is a immutable.
So when you do this on the text file,
you get an RDD
that contains rows that corresponds to lines.
If you want
a different behavior because you have
small files
and you want the content of the
the whole file to be with
what is in one row,
you can use the whole text
file method on this SparkContext
and this is only valid if the,
you’re dealing with small files
like in this example,
you can also create RDDs
from collections.
This is more anecdotal,
but it’s useful for testing
and generating data programmatically
so use the parallelize
method on the SparkContext.
To save an RDD,
you would use a save “saveAsTextFile."
The annoying fact about this method
is that there’s no overwrite mode.
If theres already
some data behind in the target folder
you get an error,
so you have to manually
clean your folders before
you use the save as text file.
This is something that is common with RDDs and DataFrames.
So you have the two kinds of operations,
there’s transformations and actions,
and so, actions will trigger the execution
of the corresponding graphs.
And there’s no exception to these
RDD operations are performed lazily.
For DataFrames there are exceptions:
there’s a
the rule is to execute lazily, but
there are some exceptions

for which operations
will be performed eagerly.
So what are those actions?
So you have count(), first(), take(), collect().
And here are some
examples. So if you
assume that the RDD is
made of [1,2,3,3].
You do a count it will report 4.
If you do first, it will give you back ‘.
To take three, it takes the first three. If you do collect,
it gets all the four members
of the RDD.
Transformations create
a new RDD from an existing one,
so you cannot mutate an RDD.
It’s immutable,
so some transformation
will have their own transformation logics.
And for many you will need to add
a function to perform the transformation.
Still, with the same
example RDDs,
you can use map
and you have to provide the function so
you can use filter and then you can
you will also provide the function.
Distinct does not take a function
as an argument,
it removes the duplicates.
FlatMap is like Map
but can output more than one result per element,
and MapPartition is like Map,
but it runs on the partition,
not on each element.
What are the transformations you can do
with two RDDs?
So you can do a union,
intersection, substract and sample.
WEBVTT
In this notebook, we are going to look at
simple operations with RDDs.
So the notebook is in DE, Labs by Spark,
working with RDDs.
So I’m going to execute the setup steps.
In those steps we clean
a HDFS folder
that we will use
in the exercise.
I’m going to execute the solution.
So, first, we
read some data
from a local file system,
then using
HDFS command line,

we put the corresponding file
into HDFS the folder.
Then we define an RDD based on this file
using this Spark Context, which is
SC by convention,
and we can perform a count
to view the number of lines,
so there are 23 lines.
And it’s okay to collect this RDD
because it’s a small one.
It came from the local list, so
it should be-
it should fit in our local disk
when we get it back. We can use
four loops to print the lines.
What I can also do is
use some Spark to
display the lines.
So I’m going to
say, take 23.
Okay, I get the content of the RDD,
but it’s not as nice
as the output of the
previous paragraph.
Next,
I’m going to upload two text files
to work on
different operations.
So I have a mixed one,
and this file contains
this fictitious mix of phones
and next to contains this.
So the goal of this
part of the note book
is to illustrate the set operations
with RDDs, and so I can do union.
And you can see that
the union does not remove the duplicates.
And there’s a reason for this.
Removing duplicates
implies adding another shuffle,
so it’s always comes at an additional cost.
So that’s why it’s
not built in the
the function, the method.
If you want to
remove the duplicates,
you have to specify a distinct().
Okay and that’s it for this notebook.
WEBVTT
In this chapter, we are going
to look at how DataFramework in Spark.
We talked about RDDs
previously in the previous chapter,
RDDs are low level API.

You specify the how and not the intent.
Queries implemented with RDDs
cannot be optimized
by Spark.
It makes- you can write
very, quite obscure code using RDDs,
and Spark cannot optimize them because
of the lack of semantics.
Spark can optimize a
DataFrame-based code because
when you say/when you write "Select,"
it knows what to expect;
when you write "Filter,"
it knows what to expect; or "Groupby,"
it knows what to expect.
When you have a map
on RDDs using any function, then-
but cannot know, cannot guess what is the intent of the function
included in the map
and therefore cannot optimize the code.
Also SQL is a very popular
programing language,
so its- for the adoption of this technology,
it’s better to have a SQL like syntax.
I talked about that
in the previous chapter.
So DataFrames and Datasets have become
the primary representation of data
in Spark.
The difference between the two
is that DataFrames
represent structured data
in the tabular form, whereas Datasets represent the data
as a collection of objects
of a specified type.
It is my belief that
when dealing with big, big data,
you should use the tabular form
of representation.
Don’t think
object programing belongs
to big data processing.
Object programing
relied on the concept
that the code and the
and the properties would be co-located
in one single entity.
That doesn’t make sense in big data.
The data needs to be
stored on large files and the processing
needs to be sent
to the where the data is, so
I don’t think that the object programing
is valid
for big data.

And Datasets are only defined in Scala and Java,
and they provide you with a strongly-typed security,
but also one that is
that should be short lived.
Those problems should not live
past the first unit test.
If you made a mistake in the name
of a column, then at the first execution
your program will crash,
and then you will fix the problem.
And that is
the whole benefit of using
strongly-typed Datasets.
On the other hand,
they require more maintenance
to maintain the types.
Okay, if you drop a column,
then that’s another schema
that you need to provide.
So, it’s a lot of work
to keep strongly-typed Datasets.
These assets were introduced in Spark 1.6,
and so between 1.6 and 2.0,
things were confusing.
Okay.
But both have been unified
in 2.0 in the DataFrame is a
Dataset of type row.
So you have untyped operations
that work for DataFrames
and you have some type operation
that work for Datasets.
DataFrames-based code
is translated to RDD code
and this
process involves two optimizers:
the first one is Catalyst,
which is akin to a Database Optimizer.
It creates a tree
representation of your query
and then recursively applies
optimizations
until it reaches the best plan.
And then this plan is dispersed on
to Tungsten,
which creates the best execution plan
given your hardware.
And the result is
the best RDD code possible. So,
the code is
easier to write and read,
and it’s also more efficient.
So it’s a win-win.
So it relies, the Catalyst relies
on the semantics of the operations,

as I mentioned.
So select, group by, where
it knows what to expect and can
perform optimizations.
The Tungsten storage format
is a serialized format that is four times
more efficient than the Java format.
Bear in mind that Java was a design
added at the time
where big data was not a concern.
So it’s very verbose
and it’s not serialized.
Tungsten format is serialized,
but at the same time, Scala
and Java can perform operations
without deserializing
the data,
so it’s kind of magic.
This is the same code
written in three different ways.
Okay, so you see why
Scala got a bad reputation?
The first one is difficult to read, it’s
difficult
to write, it’s difficult to maintain,
it’s just painful.
Then the second one
is more pleasant to read.
You get the gist of the code very quickly.
And the third one
leverages the temporary views
and is plainSQL statement.
So between the second
and the third,
there’s no difference in performance.
Both of these statements
will be processed
by Catalyst which will create the same
tree representation
and the same execution plan.
So what you choose to use
is your choice.
Okay?
There’s no,
if you come from a SQL background,
you might prefer the third representation.
If you come from a programing background,
maybe you prefer the second one.
So DataFrames does contain a collection
of objects, a row being an
ordered collection of values of basic types
that can be serialized.
And DataFrames need to have a schema,
and the schema contains
the column and names and the types.

And this requirement from DataFrames
leads to the exceptions
I was talking about
in the previous chapter.
This is where Spark will perform
eager execution
in order to determine the schema
of a DataFrame
needs to be performed at once
because the DataFrame cannot be
without a schema.
There’s a default schema eventually,
but there needs to be a schema.
In Spark 2 they introduce the SparkSession,
so it contains the SparkContext,
and it allows
to execute SQL queries
and read and write DataFrames, access the
the configuration.
We are going to process this
simple
Dataset in this first example,
we read the content of a previous file
into a user’s DataFrame.
This line of code triggers
an execution on the Spark cluster.
So as I said before,
the users DataFrame needs to have a schema,
and the schema will be inferred
from the JSON file.
So this line of code
eagerly triggers a scan of the JSON file,
which can be a problem
because the JSON file, our JSON
files, are very small
but could be very large or so.
So this is a point that we will study
in the dedicated notebook later,
but for the time being, our users
DataFrame now has a schema
which we can ask to see
and when we do this,
then for this line of code, theres no
executors are contacted.
This information is stored in the driver.
Okay, so there is no distributed
processing for printing the schema.
On the contrary,
when we ask to see the actual content
of the DataFrame,
then the executors are contacted,
and they
send back to the driver
some of the contents
because by default show

only displays

But in our example,
there’s only five of them,
so it’s okay to collect.
So since DataFrames are based on the RDDs, they inherit
from the same types of operations.
So you have transformations and actions.
You know, some of the actions are similar
to what we saw
from the RDDs.
You have count(), first(), take(),
show(), collect(), and write();
and this time write has an overwrite mode.
So you don’t need to clean the folder
before you write the folder.
Here’s an example in Python in Scala,
and you see in this example that
what changes between the two languages
is the use of the Val-keyword in Scala.
Otherwise the code is
very similar.
So transformation
we’ll create a new DataFrame
based on an existing one.
Okay, it’s this similar mechanism
as for RDDs.
DataFrames are also immutable,
like RDDs. So some transformation,
well,
the transformation for the DataFrames
mimic the SQL statements
we have select, where
orderBy, join, limit,
collect, write.
If you want to see the name and age,
we do a select
and we can say where age is
greater than 20.
The code is pretty explicit
compared to the equivalent in RDDs.
So a query is a sequence of transformation
followed by an action.
So the show here,
in the third line
is a query, okay;
and you don’t have to write things
like this.
You can also
write them using functional programing
where it pretty
much matches the SQL equivalent.
You have the select, where, show.
So what Catalyst will do for the last time,
for this example,

it will apply to the Where first
and then apply the Select.
Okay.
It’s one of the golden rules
of distributed processing
is filter early and project early.
It will perform the Where
and then the Select.
But it doesn’t matter.
You write it like you want
and then Catalyst will
create this free
representation of your code
and apply the best practices for you.
So if you write it like this,
it will not be executed
in that order.
And this is the same for Scala.
So in Scala
you have to add
the Val-keyword, but you can
do without the parentheses.
WEBVTT
In this notebook, we are
going to introduce DataFrames.
So we look at how to work with columns
and then work with rows.
So here are all the things
we will look at for columns
and the things we look at for rows.
It gives you
a solid foundation on working with
with DataFrames,
and it’s a good reference notebook
for all these operations.
So we need to perform the set up,
and get our Kerberos ticket.
And we will be working with some
the Datasets from the Duocar fictitious use-case.
Duocar is a company like Uber,
who has data sets about rides, drivers,
and riders.
So we are going to
read the data
into dataframes and print the schemas.
So this is the first Spark paragraph. So
as you should know by now, we have to pay
for the initialization of a
Spark, Livy, YARN.
Okay, now it’s running.
Okay.
And I got the new Livy session
and those are my data frames.
So I specified in the read
method that I wanted to

use the header
to infer the names of the columns
and then scan the file
to infer the schema.
So this is the results.
Okay.
Now let’s start
the lesson part of this notebook.
DataFrames have been introduced
first by R
and then in the Python pandas Library.
So it’s a common data abstraction.
It’s a collection of
of row objects
and so the properties of the DataFrames
are that they are immutable like the
RDDs, they
evaluate lazily unless
you need
to infer the schema or you need to-
when the DataFrame is first
created, then it needs to have a schema.
So this part would be evaluated eagerly.
They’re "ephemeral"--that means that they’ll be evicted
from memory
once they have been materialized.
Unless you explicitly say
I want to persist
this DataFrame.
So the lifecycle of a DataFrame
is to be uploaded
from files,
under go all the chain of transformation
to be eventually materialized
and then be evicted from memory unless
specifically told to not evict from memory,
so thats persist.
You have transformations
and actions like with RDDs.
What causes the
the materialization of a DataFrame in memory,
is the
an action and an action
then triggers the old lineage
of the DataFrame.
So you can
interact with
DataFrames using the
the SQL API or the DataFrames API.
So, it’s not a-
it’s different from RDD code
because it’s declarative like SQL
and it’s not imperative,
like a RDD code
where you explicitly specify

what you want to do, operations
you want to perform.
In Spark SQL,
you describe a
a problem, a set of equations,
and you let the system solve the
the system of equations.
Okay,
so let’s start with working with columns.
So the first thing we can do is a Select
so it’s called a Projection.
Okay, and
we can drop also, use Drop, to Drop columns.
So here we drop the first name
and the last name in the initiative.
So for those two first example,
we use the column name
to access the columns.
So this is admittedly a fragile
way of doing things,
and it can be also ambiguous, so
ambiguous from the Spark
passive point of view.
So sometimes you need to wrap
the column name
into a column or col function.
So this is a-
and also what you have in PiSpark
is the ability to use the dot notation.
So this one is never ambiguous.
Okay.
Doesn’t require as much typing as this one,
or this one. So
this is where it’s a matter of style,
but this is the one I prefer
and with this
you can with dot notation
you can chain methods
to your columns,
which you cannot do
when you use the column names.
Okay.
So these are all the methods
you can use to access
columns.
Now, if you want to add a column,
you will use the withColumn method.
So, here we want to add a column
that reflects the status of the writer.
It can be
and we want it to to be a boolean
according to its
status as a student or not.
So what you can use
is this which is called the column

expression.
Okay.
And this will be false
when the writer is not a student.
But you could use also an alias.
You go to an alias
and you could do an alias. Then
with a Select,
then you get also the same result.
And you can use also
the SelectExpr,
which accepts
partial SQL expressions.
Okay, so
if you
create a temporary view with Hive,
then you can use
a SQL statement
to perform exactly the same thing.
Okay.
So you have all these
methods
available to perform the same operation
and
there are no differences in performance.
Okay.
Because of the Catalyst optimizer,
everything will be processed the same.
And so it’s just a matter
of programing style.
You can also change the column name using
withColumnRenamed.
And if you
provide a column name
that already exists to Scala,
you can change the column type.
So here
we change the type of one block.
So this is something that is used in
the census
in the United States
and it’s supposed to be a string.
So the inference from Spark
was initially wrong.
So the inference was wrong
but it would be a string.
Okay. So if you needed to do a-
if you need to do a
lot of changes to the names and the types,
maybe you should consider
specifying the schema
instead of having Spark infer
the schema and then fixing all the column
names and types.
So that it with working with columns.

Now we can look at working with rows.
So the things you can do
with rows is sorting.
So in this,
so you can use the alternatively
sort or order by; they map to the same thing
and you can say
ascending equals true or false.
And if you need to order by
you can also use
the asc() and desc() methods.
But for this
you need to apply these to a column.
So you need to wrap the column
name into a col function and then
call the asc() or the desc() method.
Alternatively, what you could do is
use the asc() and desc() method,
which take as input a column name.
So you see
all the possibilities
that are available to you.
And again,
it’s a matter of programing style
which one you will want to use.
And note that
ascending is the default order.
So what you can do also
is limit the number of rows.
So that’s using limit,
and it’s
something
that you should become reflex for you
because you should be dealing
with big data.
And so you don’t want to
retrieve or show a lot of rows.
You want to see
a limited number of rows.
And the question here is,
what is the difference between those two
statements?
The second one creates a new DataFrame
definition in the driver
to which the show is applied.
So its,
it takes one more step to create
something in the driver:
a new DataFrame
that is called df.limit(5).
The interm names of DataFrames is actually their lineage.
Okay so this one will be called
df.limit(5).
So sometimes you want to
remove duplicates in your DataFrames

and for this you have two options
which are equivalent.
You have distinct and drop duplicates.
Bear in mind that this is an expensive
process in distributed processing
because it involves
an additional shuffle
to send all the
duplicates to the same machines
and take only one.
But what you can do with rows,
is filter on them, and you have several
options.
So you have filter and where, they do
they do the same thing.
Yeah, and
you can use a column expression
and you can mix and match it.
It doesn’t really matter.
What you can do with rows too is sample
the rows
and a good practice of sampling
is to use a seed
so that the random sample you get
will be always the same
because of
you are using the same seed.
This allows your experiments
and analysis to be reproducible
while being random.
It will,
will create the same randomness
and what you can do also
is provide a fractions.
And this will create a stratified sample.
So in this instance you want

and 80% of females to provide this
this information to the
sample by method.
So another important process with
rows is to deal with missing values.
So in our writers population,
in the first 25
we have
one male for which
the ethnicity is null,
one record with a null
for the gender or sex,
and one record group with both columns null.
So what you can do is use a dropna,
or na.drop,
they are equivalent.
And you can specify the strategy
you want to use.

So the first one we use
is How=any,
it will remove all those
three records that contain nulls.
Second goal is using How=all
and we leave out
the one with only one null.
And remove the last one which can contains
only nulls.
But what you can do so that removing
nulls can be a valid strategy
if the number of null records
is not statistically significant
in your population.
What you can do also instead of deleting
information is tagging
those null records with explicit strings.
So for instance, you can say other/unknown.
That’s what we do
and for this we use a "fill na" method.
And there’s also an NA field that exists,
and you can
specify different labels for your column.
So in the second example,
we are using missing
for ethnicity and other/unknown for sex.
And if you want to replace those,
you can with a Replace.
And in this example,
we replace everything with NA
and we can also specify the
the different strings per column.
So we came back to adding NA everywhere.
And now we say we want
no response for ethnicity
and NA for sex.
So that’s it for missing values.
So you saw that we had several equivalence
to do those processing:
you have dropna, na.drop; fillna and na.fill;
replace and na.replace.
Now it’s going to be your turn to
do the lab.
WEBVTT
In this notebook, we’re going to study
reading and writing DataFrames.
The notebook is here.
So Spark can read
a lot of different data sources,
including text, delimited text.
JSON, Parquet, ORC, Hive,
or a JDBC connection
and also to add third party packages
from other data sources.
So if you have a

given format, file format
and it’s not in the bullet list,
I would look at the third party packages,
you might find what you need,
very probably because the community is very large.
Okay, so I’m going to execute the set up,
get my Kerberos ticket,
set up the data context,
and for the database,
I will use some data
generated for a TPCDS benchmark
and we’ll use only the customer table.
So read the raw data
and then proceed to create
an optimized table stored as Parquet.
So we read first
we create a customer_raw table
and then create a customer table,
based on the same data,
but stored as Parquet for
improved performance.
But once this is done,
we can delete the raw files.
You can see that
this says its run before, so
that’s why they are tagged as finished.
So a very common format for files
is delimited text files
and we’ve done this already.
We know that we can retrieve the header
and infer the schema of the file.
So when we do spark.read.csv,
it’s a convenient method
for a longer syntax, which is a showed here.
So if you use this, so I can perform this also,
but it’s going to be the same thing.
So be careful:
if you say header equals true
then every file in the directory
should have a header.
So that’s
the tricky part
with the headers;
you need to be sure that every file
in your directories
has a header.
Now what I’m going to do is open
this Spark History Server next to my Zeppelin,
so that we can see what goes on
when we perform those operations.
To open the Spark History Server
you need to go to your data
cluster and click on the Spark History Server link.
So I have done this already
and made it so that

Zeppelin and the history server is side by side,
so, as
its name implies, the history server's
initial purpose
was to display historical data
about the Spark applications,
but it does also allow you to see
the data about incomplete applications.
I'm going to click on this link.
And my Livy session
should be this one.
Okay.
And for the time being, I have
launched four jobs,
four jobs,
and that corresponds to two jobs for this
cell and two jobs for this cell. I can-
so if you see here, we are the
job ID number 3; if I replay this
and refresh this,
okay, I have two more jobs,
and one job corresponds
to reading the header
and getting the name of the columns.
And the second job corresponds
to scanning the
the CSV file to infer the schema.
So this is
an instance where Spark uses eager
execution because it needs to, because after
this line, the writers DataFrame
should have a schema.
And so this is the schema
that has been inferred.
And that schema
inference is a fancy word for guesswork
and in this instance, Spark guessed wrong.
There are things that are not correct,
like the dates that are inferred
as strings
and the home_block is inferred as long
and should be a string.
So what you could do is specify the
the schema manually
and provide the schema
to the read function.
So, if you do that,
look at watch what happened,
what happens--what happens was nothing.
Okay?
Because all the information
was provided to Spark.
There was no need to perform
any distributed
processing on the cluster.

So you might think this is better.
And it probably is.
The downside of this
is that you have to manually
provide this schema
and it’s not uncommon for DataFrames
to have a several
tens of columns.
So it might be a bit tedious.
Also another downside
is that if your schema changes, then
you code will break,
but that might be something
that might be desirable.
So may not be a bad thing,
but it’s more tedious.
If you’re using this approach,
then your code will adapt
to changes in the structure.
But I’m not sure
this is a very desirable feature.
Maybe it’s safer to have your code
break on the schema
of your incoming data changes.
And the downside of this
is that you trigger one short
job to read the headers.
But one job that can be as long
as the file that you’re reading
to infer the schema
because it’s going to scan the whole file,
so be aware of this.
Okay.
So we had to
include the header option
so that the header was not read as another record.
And so
the correct schema is the following.
So we see that home_block is a string
and the dates
have been typed as dates,
so that’s good.
You can write also your DataFrames
and this time you can use a overwrite mode
to need to do some housecleaning
before writing
and you can specify a different separator.
Okay,
so this will trigger some distributed
processing.
Okay.
And if we check the results,
we have one file that is a TSV
file, but with CSV
extensions. So do not trust the extension

and only one file,
which means that only one executors was
was involved
in the process of writing the file.
So with the Spark
UI, you can drill down.
So this was our
latest job,
and we can look at the timeline
and we can drill down on this.
Okay?
And we see that only one executor--
executor number 2--was involved.
Right.
Also, you can change the
the codec.
So we saw how to use the node,
but you can also use compression
and specify the codec that you want to use.
And if you check the results,
And we see that
the file has been saved as a bz2 file
and is smaller than the previous one.
So it's okay to have
text files
as of first stage of the format when
data arrives, it lands in the HDFS.
Then when you do some work on it,
then you should save those files
in a better format, formats
like ORC, Parquet, or Avro that are compressed.
So, what you can do with a
text file is
with the read.text,
so this time not
not CSV or TSV,
but plain text files like
Apache Logs-
We are job ID 6;
if I trigger this
it did some things
or did it so let me do it again.
We had number 8,
so it did something.
What did it do? Oh there's a head.
Okay that's the head alright,
so and the point I wanted to make
was the following.
So if I remove the
the head
and execute again.
Okay, then it
the first part,
the first two lines do not trigger any
distributed processing.

So this is just a definition in the
in the driver memory and it
because it’s text
it gets the default schema
which is only one column
called value of type string.
And of course then if I add to the head,
then it triggers some execution
to retrieve the data from the executors.
Okay, so
this is a very common use case.
Okay.
And so this is
a frequent use case
where you have some text data.
Okay.
And what you can do with the text data
is use regular expression.
So I often say that the difference
between unstructured data and structured
data is a set of well-crafted
regular expressions.
So in this case, we want to retrieve the
the request from the logs.
And by using
this method
called regexp_extract,
then we are able to precisely do that.
There are regular expressions button
on the net that allows you
to completely pass
an Apache weblog and get all the,
the components of the log
in a structured manner.
And once you do that, once you invest
in your regular expression, then you can
leverage all the DataFrame API
and Spark good stuff. So,
we can use the text method to write
requests
and check the results.
Okay, so I told you that a better format
would be Parquet, so what we can
and it’s a very common use case
to read some
text based format and then do some work
on the data
and then save using Parquet.
So here,
we save to Parquet,
and we can see that
also this involved only one executor,
and the
codec that was used was snappy
which is the default

codec for Parquet,
and the codec can be configured.
And so we can also
and you see that there’s a convenience
method like CSV called Parquet.
So we can read the
read back the Parquet file
and let’s do this,
let’s do a little experiment.
So, so we are at index 14.
What happens when I do this?
So there was
some distributed processing
and this distributed processing
was to retrieve the schema information
that is embedded in the bucket file
and that’s-
so it’s-
not a full scan of the file, it’s
just the extraction of the
of the schema
that is embedded in the file.
So that is okay
and the end result is correct.
Okay.
So there’s no guesswork
or inference involved, and there’s a
short performance overhead.
You can also
read data from Hive tables.
So I have my table
in my database.
I could also, there is some
problem with a S3 permissions
that prevent the data
from being there, but
it should work with the correct
set of permission on the bucket.
I could have done so in Zeppelin,
you can use the %SQL magic
to perform
the same processing. Again my table,
I cannot access the time in my table.
But it’s
not to worry.
So to read from the table.
You do .read.table
and so let’s see-
let’s see if there’s any
processing involved with this.
So we are at index 19.
Okay, so there’s a show()
so the
show() will of course trigger
some distributed processing.

But before that, that was only the driver
getting the schema from the Hive metastore.
And of course the schema is correct.
There’s no inference
or guesswork involved.
This schema is precisely
what was in the metastore
and I can write
the results to Hive table
and check that the table has been returned
and I can do a describe().
When we
study the Spark integration with Hive, we look at the
the real nature of the table
that was created,
because we didn’t specify whether it’s
it’s a managed or external table
and we will talk about this
when once we have presented Hive.
So I can drop the table,
I can also work with object stores.
So here we are going to
use S3.
So to access S3
you need to provide somehow
an access key and a secret key
so that you don’t put
the access key and the secret key
inside your code.
But somewhere.
And once you have provided
this information,
you should be able to read file
from S3.
So it takes a little longer, but it’s okay, manageable.
But we don’t have the permissions
to write to this bucket,
so we are not going to see that.
But it will be the same syntax.
And now you’re going to do the next exercise.
WEBVTT
In this new notebook, we’re
going to look at working with columns.
And this notebook is divided into four
sections: working with numerical columns,
working with string columns,
working with date time columns,
and working with Boolean columns.
So Spark supports
the usual suspects for
SQL types
so, and also
supports complex types which
will be the subject
of the next notebook.

I’m going to execute the set up.
And I’m going to use the
rides, drivers, and riders
DataFrames.
So this cell is starting my Livy and Spark session
and now I have my Spark session so
I am to execute
the reads.
I’m going to go to my new applications.
Yeah.
This should be the latest one.
And I have
six jobs already,
and they were triggered by these three
operations.
So one
to retrieve the header for each file
and one to scan the file
to infer the schema for each file.
Okay.
Now on with the lesson.
So first we study numerical columns.
So, what can we do with numerical columns?
Well, we can do arithmetic operations,
so we can convert from meters to miles
for instance, the distance.
And so for this
we need to import
the col and the round functions.
I need to
make it clear to Spark that
this is a column.
So I cannot use the column name,
so I have to wrap the column then
inside a col function; I could have used
also write.distance
make it clear its a column
and if I want to save that result
in my DataFrame, I could use a withColumn,
and I
should have a distance in my last column.
If you don’t want to add the column,
but just override the column,
I provide the existing column name
and now
I do have the distance, but it’s now
its in miles instead of kilometers, oh, meters,
sorry.
If I need to convert
the format of something
like the ID, I can use
a format string function
and that will
give me a left zero
balance string of ten positions.

For this I use the print format string.
If I want the student flag
to be a boolean,
I can use a column expression
like this one.
Okay.
Or I could use the ’cast’ method.
That’s it
for working with numerical columns.
Now we are going to look at
working with string columns.
So you got the usual
set of string
handling functions
such as a upper() and trim().
If you want to normalize
a column
you can do this,
you can extract a substring.
So for the US census,
you should know that
the first 12 characters of the home_block
make the home_block group.
So I select with a
one base index,
I select the first 12 characters
of the home block
to get the home_block group.
Okay, but I could make this more
challenging
using a regular expression.
So maybe that’s not the best use
of regular expressions because using
a substring is less confusing,
but it’s still a good idea to,
to be,
to develop your skills
in regular expressions.
Okay, So
maybe not the best use case, but still
don’t be shy to learn
about regular expressions
because that’s the key to, as I often say,
it’s the key to make unstructured data structured.
That’s it for working with string columns,
now working with date and timestamp columns,
its always a bit tricky to
use in this. So
we have
for the riders,
we have a birthdate and start date
and they are read as timestamps,
so maybe that’s a bit too precise.
There should be dates so we can
either cast the,

the corresponding column with the cast
or use a dedicated function
called to_date().
And this will give me the same results.
So if I want on the opposite,
if I want to convert
a string to a timestamp,
I still have two methods
that I can use.
I can cast to a timestamp
or I can use a to_timestamp()
if I have imported the function before
in the case where
I’m using to_timestamp(), I can provide
and I should provide the format,
but this gives me the same results
and this allows me to
to do meaningful
operations like computing
the age of each rider.
So this gives me
the age by using this
computation,
by using several functions
from the SQL function.
And you can see here that the
SQL functions library
has already a lot of the functions
that you need. So before
creating your own functions,
check the library;
it’s quite exhaustive.
So by using all those three,
you can compute the age of the
driver, oh the rider.
Sorry.
And that’s a more convenient
variable
to use in a machine learning context.
than the birth date, it’s
it’s easier to manipulate.
Now we’ve looked at working
with dates and times
stamps, we can look at working
with Boolean columns.
So you can create
column expressions,
So you can define this
as a column expression
and use it
instead of the column expression,
to create a column or to filter.
And you can create as many as you want,
like here
we created a student filter and a male filter

and we can use them to combine
the different filters.
Okay, so we can with the
commercial and/or if you use the pipe feature.
So be careful
with those kinds of expressions, make it-
they use
parentheses liberally so that
you’re not surprised
by the result of Spark.
And also it’s one of those instances
where you need to be
aware of nulls in your data
because the nulls will have this
behavior.
So one, for me,
it’s one more reason to deal with nulls
before you do filtering.
But that’s me.
I’m going to execute this
because I haven’t.
So in this last example
we use,
so it doesn’t matter,
how you write your filters,
whether you write with a ’filtered filter’
or a ’filter and,’
it’s going to be analyzed by
Catalyst in the same way and it will create
the same kind of execution plan.
So it doesn’t matter how you write it.
In fact, the internal representation
would be filter,
male filter and student filter.
And also you need to be careful with,
you need to be careful with the nulls.
In the last example,
when I say
the column ’sex’ is different than male,
it removes
the nulls also, it shows me only female
and there are null values for the sex.
So it’s tricky again.
I think that’s it for this,
that’s it for this lesson
on working with column.
Now you should do the the exercise
and this exercise.
WEBVTT
1
Now
that we have studied the simple types
we are going to study complex types
in this
dedicated notebook.

There are three complex types, there’s the arrays,
the maps, and the structs.
They are similar to the
complex types
you will find in Hive, because initially
Spark had to be compatible with Hive.
And there are different use cases
for them.
So I’m going to do the setup.
For the set up using the run all above.
Okay.
This time my Livy session was still live.
So this is where I am now.
Okay, so let’s so the lesson part of this.
And first, we start with arrays.
So you can create an array
using the array function,
provided you import it first.
And this will create,
so here we are using the two columns
about the vehicle,
the vehicle_make and vehicle_model,
and we make an array of these.
So, unfortunately,
this is not the best use of arrays.
The choice was made
to use the same column
to illustrate arrays,
maps, and structs so that you can see the differences.
But they should not be used
for the same use cases.
Arrays should be used to contain
similar things like an array of user
ratings,
an array of phone numbers,
an array of vehicles.
If we could imagine that
a driver could have several vehicles,
but not to have something as different
as vehicle make and vehicle model.
So this is
would be better suited for a struct
or a map.
But anyway,
so here we have a vehicle array now,
and we
can access the elements of the array
using zero index notation.
And we can compute the size of the array
using size().
And you see here, the
the size of the arrays is very consistent
so ultimately the best representation
for this
information would be a struct,

and the struct is the least flexible
representation.
But in our case it would be the best
because for each record there will be
a vehicle model and a vehicular array.
So its structs are meant for clean data
and our data is very clean for this.
You can use the sort_array(), and
this makes no business sense
to sort the can make it a vehicle model.
Okay.
Now everything is shuffled
and you can use the array_contains()
method.
So we are looking for Subaru
and we get a Boolean
and see the nice column name
that Spark created automatically,
and you can
generate rows
for each member of the array
using Eeplode().
And if you want to keep the information
of the position
of the member in the array,
you should use posexplode().
If you’re not happy with the default column names
you can provide them
with using an alias() method.
So that’s it for arrays.
Again, the use case for arrays
is more for things that are similar
than things that are as different
as the vehicle make and vehicle model.
So let’s talk about maps.
So to create a map,
you use the create_map() function
and a map is
a collection
of key value pairs, so for the keys
you need to
you use the lit() function
to create a fictitious column
of the same string.
Okay.
And now you have
created a map.
And what is interesting in maps
is that
it’s very flexible.
You see here
in the arrays you had, what-
is an array of string
of elements here.
What we have is

key/value pairs.
So anything goes.
You could have a any number in a given
row, you could have
five key/value pairs and the next one
you could have two or zero.
So this is very flexible.
This is your gateway to no
SQL data.
You want to have a flexible representation
and you need
this flexibility because of the
unstructured nature
of the data you are trying to process,
then you should use maps.
For instance here we could have the power of the engine
on one row and not on the other,
the number of those on
and so on and so forth.
So that’s really flexible.
So you can use the dot notation
to access
the values of the map.
You can use size
to get the length of the map,
execute those.
Okay.
And you can use, explode() and posexplode()
also, like
like with arrays.
Last complex type is structs.
So the use case for structs
is this--what we have here.
Okay.
So when you have data
that is very structured
and consistent,
you can say, okay, I’m going to create
a vehicle struct,
with make and model
and what you will have
is this.
So you have the vehicle struct which is
embedded into the,
into your data frame
with the make and model.
And this is less flexible than the others.
Remember for array
you have an array of strings.
For maps you had keys and values.
Here you were expecting something
that is hardwired to be a make
and something
that is hardwired to be a model.
So the struct is a row object

fitted in another row
object. So,
you can use the dot notation for this,
and there’s a
a function that allows you
to convert the struct to a JSON string.
So you can use a to_json.
That’s it for this group.
Now, it’s your turn
to do the corresponding lab.
WEBVTT
In this next notebook,
we are going to study how to combine and split
data frames.
So combining involves joins
and also union and
more generally set operations.
So I’m going to perform
the setup steps. This
lesson we are going to use
two new data
sets, the data_scientists and the offices.
And the reasons for that
is that it’s much more convenient
to work on small data to see
the behavior
of the different type of joins.
So we have four data scientists
and five offices and
data scientists are assigned,or not,
an office.
So first up, we have the crossJoin.
This gives you the combination of all the
all the data scientists
and all the offices.
So for me, the only use case
for cross joins is to create
all the to create these data
with all the combinations.
Otherwise, you’ve been
told to stay away from Cartesian products.
And I think
it’s a-
it’s a good thing or you wouldn’t use
cross joins.
So the next one is the
very popular, inner join.
So the first syntax we use
is the full syntax
where we specify the join condition
with two equal signs
in PiSpark and Scala, it’s three equal signs
and we say we want to inner join.
The result of this is this
what is shown here and

it’s problematic.
The problem is that you have two
identical columns called office_id,
and this is not manageable.
You can not drop one,
if you say drop office_id it will drop both.
So your result becomes unmanageable.
So either you have to rename
if you want to keep both,
you have to rename them or what you can do
is just use the second notation here.
Okay.
And Spark knows that because the
the column names in
both data sets are the same
that it can use these two to do the join
and this time
you get a more manageable result.
Yeah.
And since inner
is the default type of join,
you can just
forget about the type
of join and say join on office_id.
And this is the most compact notation.
Next we have the left semi join, so
if you want to have the list of data
scientists associated with an office,
you can do left semi and the poor guy, Emmanuelle Auzenne,
the one without an office,
is left out of this join.
So the opposite of the left semi
join is the left anti join
and it gives us Emmanuelle.
So these two joins behave like filters
and to me
it’s a convoluted way to do filters,
but to some people
it might be the natural way.
Next we have the outer family,
starting with the left outer,
which gives us the list of data scientists
with or without an office.
So Emmanuelle is back in the list,
but he has no office.
The opposite of that
is the right outer join.
So we have all the offices,
whether or not they have people in it.
So we have offices
in Marseille, Lyon, and Toulouse
that are empty
and Paris has two
data scientists.
Last we have the full outer join

with all the data scientists
in all the offices,
regardless of whether they match.
Okay, so what we can do now
is use this knowledge for DuoCar.
And this is a very common
workflow in big data.
To denormalize your data, to create
what was referred to
in Facebook and Google
as an everything table.
Remember that
joins
create shuffles and shuffles
are your enemies.
So one very powerful way of avoiding
shuffles is to denormalize.
So in our case,
we have at the center we have the rides.
This is the main object and the rides
the point to a rider,
a driver, and eventually a review.
So we are going to read those
data.
So this time we are using the
the Parquet format of this data.
So that’s why it was quick.
And now we can do a
build a new data frame with everything
using left outer
for drivers, riders, and reviews.
So there is a catch
and it’s the following.
So if you don’t do any
pre-processing on your column names,
you might end up with confusing
columns, so
drivers and riders,
both have a birthdate
and this is
and in the data frame we create-
we created
they are not distinguishable. So
they also have a first name
and a last name.
So this is a confusing
data frame and it’s unusable.
One of the things you need to do
,and it’s a good practice
also, to keep track of the origin
of your data
is to perform this to prefix
the columns with the name of the table.
And so we have done this for you
and it’s stored in Parquet format

and no, it’s no longer confusing,
you have the driver birthdate, the rider
birthdate.
This is clean, and
this is the data product
that is the result of our data engineering
that we could provide to data scientist
or business analyst
to analyze.
So let’s study set operations, also.
So for this, we are going to work
on the first names
of our riders and drivers.
So we can do a union
if we want to.
There’s a slight bug in Zeppelin
that prevents us
from seeing the information.
So I need to-
that’s the last count.
So drivers names count is,
doesn’t show.
Okay you have to trust me that
the sum of all the rider name
and driver name
is equal to this number.
If you want to
have only unique names,
so if you want to remove the duplicates,
you can add a distinct().
If I want to see the number I have to
command the last line.
So now we have only 911 different names.
If we want to see the
the names
that are the first names that are common
between drivers and riders,
we can use intersect.
So those are the names of
the drivers and riders share.
If you want to see the opposite
of the intersect(), you do a subtract()
and in Scala it would be an except().
So these names are unique to drivers.
There’s no riders with the first name Bobby.
So, if you want to split a dataframe
and this is very helpful for machine learning purposes,
You use the randomSplit() method
to which you provide
weights.
So if you want to
your samples-
to be, or
your split to be reproducible,
you need to provide a seed.

And if you provide a list of weights
that is not
normalized, Spark
will normalize to the weights and
give you the right results.
Now it’s time for you to do the
short exercise in the lab.
WEBVTT
In this new notebook
we are going to study how to summarize
and group DataFrames.
I’m going to perform
all the setup steps.
And you see that now
we have our data product that joined
the DataFrame stored as parquet.
So we read this
DataFrame,
and we persist it.
So we say to Spark
next time
this DataFrame will be materialized,
please do not evict it from memory.
So what can we do
to summarize numerical data?
For instance, in our case
we have the distance
in our DataFrames
so we can have the count of distances,
the means of distances,
the standard deviation,
the mean, and the max.
That’s why this describe
function works
well with numerical columns.
We have also functions
like count(), count_distinct(),
and approx_count_distinct()
that we can use on a numerical column--
on any column, for that matter.
Why we use a-
where would we use an approx_count_distinct()?
It’s because,
as I said earlier, distinct() is expensive
distributed processing.
So it’s a tradeoff
between latency and accuracy.
You see, it’s not completely accurate, but
it gives us the right order of magnitude.
Okay, so
we could have wrapped everything
into an agg() method.
also. It gives us
the same results.
There’s a sumDistinct()

function that exists.
Doesn’t make a lot of sense
in our case, so it does what it says
it’s summing the distinct
distances.
Doesn’t mean
anything for us, but it’s-
it’s there.
And you would use it
if you had-
maybe when you want to-
when you have distances that are-
you have
columns with the same values
and you want to do
a sum of the distinct values.
I guess this sum distinct
is optimized,
rather than doing a distinct
and then a sum, you do a sum distinct.
Then you can do
some partial work before the shuffle.
You can sum distinct values before
shuffling them.
And it saves on the network bandwidth.
You also have statistic functions
like the mean, the standard deviation,
the variance, the skewness,
and the kurtosis.
So you may not be familiar
with the skewness and kurtosis.
So skewness,
there’s you, whether your
the distribution of your column
is skewed to the right
and to the left and how skewed it is.
And the kurtosis measures
the pointy-ness of the peak,
so the more pointed it is, the more kurtosis you have.
Of course, for
regular human beings,
this information is better understood
with a visualization.
You also have min and max.
So here we see that
the shortest ride
was 300 meters,
and the longest one was 92 kilometers.
You have first and last,
but be aware
that first will give you a null,
if your distances
if you have a null in your column.
And we have null in our distances
because we have constant rides.

You have also the correlation,
covariance
sampling, and covariance population
functions.
For those of you that use those metrics
for statistics.
So here we
we use this on distance and duration and
this value
is interesting.
It appears that there is a strong
correlation between distance and duration,
which is to be expected
when you’re talking about rides.
So given
the distance,
we could predict the duration
using a regression technique.
If you want to
create a column of array type,
you can use a collect set
or collect list, so
here we use that to
create an array of the categories
and we use the collect set because
the collect list would create a very large
array.
That’s for summarizing data.
Next, we’re going to look at how
we can group data so this can be done by
groupBy().
And in this example,
we are focusing on the
student riders,
and we can see that
how many rides they do,
what is the average distance
and the standard deviation for
the distance of the rides of the student
and of the non-students.
We can groupBy more than
one column so we can do a
final analysis by
adding the service.
If we want to have
intermitted subtotals,
then we can do a rollup.
And to keep track of the groupings, we can
provide groupings information.
And if we want to get all the subtotals,
then we- instead of rollup,
we can use cube.
And the same goes with groupings.
If we want the cube and the groupings,
we have the grouping information.

Next we can pivot data.
So with the very usual groupBy(), count(), orderBy(),
so we look at the
the study, the distribution of services
for rider students and non-students.
In Python
you have this crosstab function
that is very convenient;
they do the same thing.
The crosstab function is not available
in Scala or used to be.
Let’s be cautious.
So you can also use a pivot method
to do a producer cross tabulation.
So gives you the same information
and when you,
we can do a mean instead of a count,
and you can wrap your aggregation
into an agg method
and you can select
the column- the values
for which you want to pivot so,
we’re interested only in car and brand
in this example,
and we can also
have several aggregations for
a given column.
So here we focus on the distance
and we have the count of distance
and the average of distance.
Notice that Spark always provides
meaningful column names
for those aggregations.
Next, you’re going to do the corresponding lab.
WEBVTT
In this chapter, we are going to study
how to work with UDFs.
So UDF stands for "User-defined Function,"
and it’s easy
to create UDFs
and the process is to define
the Python function
that operates on a row of data,
then register the python function
as a UDF and specify the return type,
and then apply the UDF.
So bear in mind
that built in functions are more efficient
than user defined functions.
So you should always check that
the processing that you want to create
does not exist already.
Also there’s a penalty with Python
unfortunately for user defined function
because the

the data that needs to be
in its Java format.
So it needs to be deserialized
from Tungsten to the Java format
passed on to the Python process
and then transferred back
and serialized, so
there are workarounds for these, like
using the PiArrow library
and using vectorized
UDF to process
a folder of 24 rows at a time instead of
one row the time.
And also
what is even simpler is to write your UDFs
in Scala or Java, which
can process the data
without deserializing the data.
So I’m going to
perform the setup.
The first example is the hour of day,
so you can create a simple
hour of day function
that takes the timestamp
as input and returns
the hour of the timestamp.
This is actually a bad example
because the hour of the function
already exists in the
SQL functions library.
But since we are using a function,
then you can pull back on
your good software development practices
and test
your function.
That’s the correct answer.
So it looks good.
The next step is to wrap it into a UDF
and provide the return type
so that the
UDF function can work on a column
instead of a single timestamp.
And then you can enjoy your UDF
and compute the hour of day
on the rides
and perform some
analysis on the distribution of rides per
hour of day, to see whether
there is an influence of the hour of day
and there is one.
So this is a case
where your data is skewed, which is
to be expected for business data.
The fact is that there are less rides
during the small hours of the day

compared to the rush hours.
That’s,
that’s natural.
Next, we can also
create
a function that computes
the relative distance between two points.
It’s a classical problem
to compute the distance
between two points on a
sphere, which applies to UDF,
and here we have an
approximation of the function
and we can test our function.
Of course, now this becomes a bit
more tricky, but we have- the answer is correct.
Okay, we register the function
as a UDF and specify the return type
and then we can apply it.
And look at these distances. So
we expect that the Haversine Approximation to be shorter
than the actual distance of the ride.
And that is true except for 53 rows
and if we look at those rows,
the problem might be from
might be coming from the approximation.
If you really want accurate results,
we should
provide a better
implementation of the Haversine
Computation.
That’s it for your for this lesson.
Now it’s your turn to do the lab,
where you have to answer three questions.
WEBVTT
In this notebook we are going to study
how to work with Windows for analytic purposes.
Those Spark SQL supports,
Window functions.
You can do aggregates of a Window
specification.
And a Window specification
consists of at least one of the following
partitioning column,
ordering column or row specification.
I’m going to execute
the set up steps
and we are going to use all joined datasets.
So this is the first cell that starts
the Spark process.
So it takes a little time
to get a new Livy session.
So let’s see how this works. So
to illustrate the concept, we can create
a simple data frame

with values from 0 to 9.
And we can create
a Window specification.
We need to inport the Window object
and we create an instance of this object
with the rows
from the beginning, which
that comes from unbounded
preceding to the current row.
And now,
if we use this Window specification in a
in a transformation,
so we say
we want the count of our WS
and the sum of our WS, you see that
the count
grows to
from 1 to 10
and the sum, the sum of IDs,
start with 5, then 5
plus 6, 11 and so on and so forth.
So that's what? 41 plus 3
plus 4 equals 45.
So that's the behavior we expected.
As usual Spark provides
automatic name for the column
and this time it's
one of the longest ones
so that's
why we used aliases in the first place. But
if you forget to use aliases,
you get those very long yet
explicit names.
Okay, so let's use these concepts for
our dataset, on the duocar dataset.
Let's assume we want to compute
the average days between the rides
for each rider.
We define a Window specification,
but we partition by rider,
and order by the date_time
and we apply this
to get the previous ride home,
on riders using a
a lag
of the date_time of our WS.
So now we have
for each row we have the ID of the rider
and the date of the ride
and the date of the previous ride.
So when, for you see
each time there's a new rider, then the previous ride is null.
Okay,
so this is-
for this rider.

First ride is null and then this one
would be equal to this one.
And it is.
Okay,
This usually is an intermediate step
for further processing
in our case.
Then we can
compute the number of days
between consecutive rides.
So we use a datediff, okay
and we get the number of days
between the rides.
And to summarize this into one
meaningful aggregation,
we can compute the average days
between rides
for each rider.
And now that we have this new column,
we can look at the top and bottom
ten riders.
Okay, so what we see is that for the busy
riders, this
metric is not good enough.
It’s not.
It saturates all the values are
zeros for the busy rider. So,
we should have used more granular
information like instead of days
between rides, minutes
between rides
would have been more appropriate.
So maybe we should review this,
but for
not so busy riders, this is good enough.
Okay.
So that’s the how can we make this better
by choosing minutes
instead of days.
And now, this is going to be your turn
to apply this new concept.
WEBVTT
Hive is an SQL semantic layer on top of Hadoop.
It was developed originally by Facebook,
and it provides schema on read.
It translates SQL queries to
a distributed processing
initially to MapReduce, then to Spark
and also Tez.
So it splits the
the data into metadata and data.
The metadata is stored in a
meta store,
which is a relational database,
and the data is stored as folders

in HDFS.
So it projects the schema
described in the metastore
on the files stored in HDFS,
thus giving you the illusion
that the data is structured
and allowing you to process this data
with a SQL like operations.
So those are the components
that create the Hive service.
So there’s the Hive Server 2,
to which you can connect
using both ODBC and JDBC.
And this is one of the
unique features of Hive in the big data
ecosystem.
This ability to
to receive queries
from ODBC and JDBC is kind of unique.
So if you have this requirement in your
in your system, you need
you need Hive Server 2.
So the metastore is this
can be described at the system tables.
So it’s the metadata about your tables,
and it’s stored in a relational database
that can be either Postgre,
mySQL, MariaDB, or Oracle.
So the Hive Server 2 receive the query
via JDBC or ODBC
then creates
either a MapReduce, a Spark, or a Tez job
depending on the execution engine
that is plugged in Hive
and that creates a job that is executed
on the Hadoop cluster
using YARN.
Additionally, there is
a caching system called LLAP,
which stands for Low Latency Analytical Processing.
And this system
allows for low latency,
always on, shared cache services and can be used for
BI purposes.
Historically,
Hive was more for batch processing,
but with all the improvement
that have been made
and the latest one was LLAP,
it can be used for BI purposes.
Although now a days after the merger,
the solution that is preferred
for BI is Impala.
Yes, about those improvements.
So you can see this is the commits

graph from GitHub,
and you see that it’s an old project.
It has more than ten years now.
So it was born at Facebook,
and so people at Facebook
were business
analysts that had to, knew how to
query the
logs of the website with SQL
to derive the price of advertisements.
But because of the catastrophic success
of Facebook
they had to switch from Oracle to Hadoop.
And then for those business
analysts, they would have to use
Java to perform this very simple query.
They had to create for MapReduce jobs,
which amounts to 12 Java classes.
And so
they were not going to do that.
And even if they did that,
this would be
slow processing
because you had to read and write from HDFS,
use those dummy mappers
and so on and so forth.
So these are dummy,
these are dummy,
this one is dummy, because the real graph
is what you want to do, is this one.
With all of those
dummy mappers
and without the pitstop at HDFS.
So that was the initial Hive.
So the first thing to do was to
use a second generation
execution engine such as Tez.
Okay, so Tez is a
is not
equivalent to Spark
but it’s from the same generation.
So it uses memory and it’s able to tackle
DAG graphs.
So that was the first improvement.
Then came vectorization.
So instead of processing
one record at a time, now
Hive can process 1024 records at a time.
Then came the ORC format
with which supported vectorization.
So it’s a whole column
of optimized format.
Then came to the caching system
called LLAP then came
transactional tables and materialized views,

and you can see all those improvements
during the years.
So it’s a project
that is still actively being worked on.
Okay.
There was a hiatus at the beginning
which corresponds to the time
where Cloudera started Impala.
I guess people at Hortonworks
wondered what to do for a year
and then started improving Hive.
So vectorizations can be enabled.
ORC, so now this is the default
format for managed tables
because of the way
ORC works,
there’s no more need for indexing.
Okay.
It works using a bloom filter.
So, the data is a split in stripes
and those stripes have a header and a footer
and the bloom
filter shows in the header,
whether or not the record that
you’re looking for is in the
stripes below.
So instead of scanning the
the whole table, you scan the headers
until you find the stripe
that contains your record.
This mechanism is called the bloom filter;
it’s something that also is used in Hbase,
and it removes the need for indexing.
And ORC has been proven
at scale with the famous

So about LLAP:
so it’s a clever caching system.
So the idea of LLAP the
is the same as the one for Impala, it’s to remove
all the other overhead
of SQL processing.
They come at the beginning
of the processing.
So what happens
is the case of young you have to
start JVM.
So starting those JVMs
takes time so instead of starting
JVMs LLAP uses
JVMs that are already started.
And then the next
process that slows down
the performance of your queries
to read the data from HDFS, and LLAP

uses a clever caching system
that allows to, in
most of the cases,
to read the data from the cache.
So when you remove those two steps
starting the JVMs
and reading from the cache, what you get
is only SQL processing
and vectorized on ORC,
so it’s pretty fast.
Then there’s Beeline.
So there used to be a Hive client
that was retired.
Now the editor that should be
used is Beeline.
It uses a JDBC connections string.
So this is the
an example of the use of Beeline.
So you connect, commands
should be prefixed with a bin
So, it’s a bit difficult
to read, so you connect here
and then the
next time you can, when you’re connected,
you have this new prompt
and you can type commands
like "show tables;."
There are two different types of tables
in Hive,
you have the internally managed tables,
which is the default
table is for Hive and the external tables.
So those tables are stored in a directory
that is managed by the Hive superuser
and we call them "Managed" because
when you drop the Managed table,
then you erase the record of the table
in the metastore
and you delete the data in HDFS.
So the lifecycle of the table
is completely managed by Hive.
External table on the contrary have a different behavior.
When you drop an external table,
you erase the record in the metastore,
but you do not touch the data
in HDFS.
It used to be the only difference
between those two types of table, but
with the recent versions of Hive,
managed tables have,
have started to have much more features
than external tables.
And one such feature is
the support for ACID transactions
that’s available for Managed tables.

This is available for single table,
single statement transactions.
So the way this is
implemented
is by introducing a new layer of folders
between the old folder of the table
and folder that contains the data.
For external tables
and older versions of Hive
the layout of the folder was very simple.
You had old folder of the table
and then inside
the folder the files that would
contain the data.
Now because of
to make this
tables transactional, then
there is this new
layer of folders
which is transaction oriented.
Those are the delta,
delta folders.
So there are two types of ACID tables.
This insert-only;
this one is insert-only,
you know, it’s
transactional and insert-only.
And what happens
is that- if one of those, so you see here
three transactions, that translate to three full delta folders
and let’s assume
that the second transaction fails.
What happens is that in the metastore
Hive will keep track of the transactions
that failed
and the transactions that are running.
So when a different user
will ask to select the data in the-
in this table,
the transactions that are running
and those that are failed will be skipped.
And therefore the other users
will see a consistent view
of the table.
This is all I see is achieved.
So for
transactional tables that are not insert-only,
that support, create,
update, and delete,
I add a new metadata column called the row ID
and this allows
Hive to implement updates
and delete.
So let’s look at this in an example.
If you insert those rows in the,

you our table,
the row ID will take those values.
It’s transaction one,
so if you want to do a delete,
the operation will generate
a new folder called "delete delta"
and will put nulls in the value of A and B.
So an update is a delete
followed by insert.
That’s way we can,
for instance, change the value
of 300 from bananas to pears.
First, you delete
the corresponding row
and then you add insert
and your new row is pears.
So you get two folders,
a delete delta and a delta.