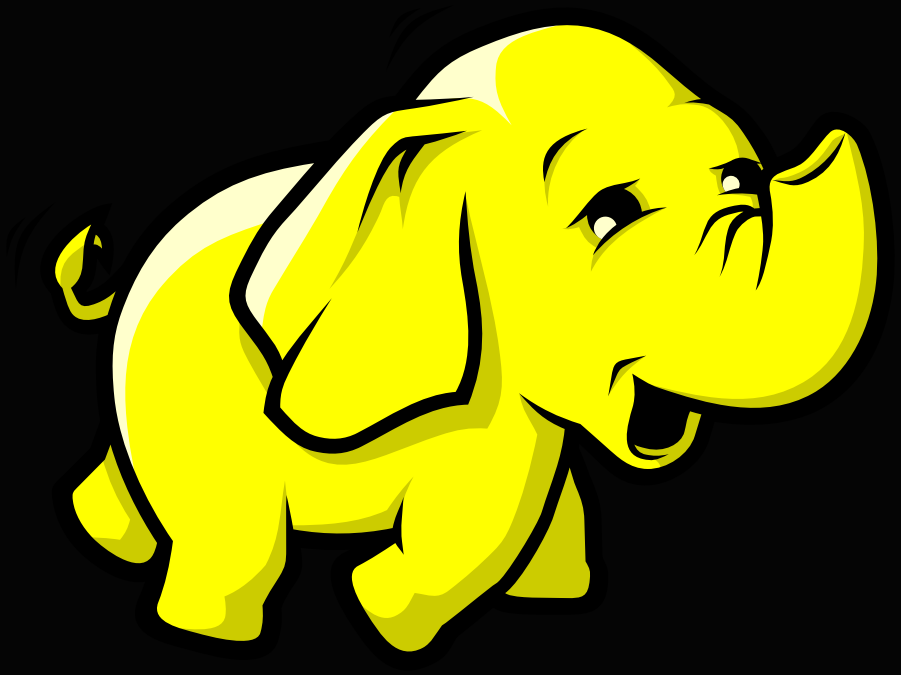
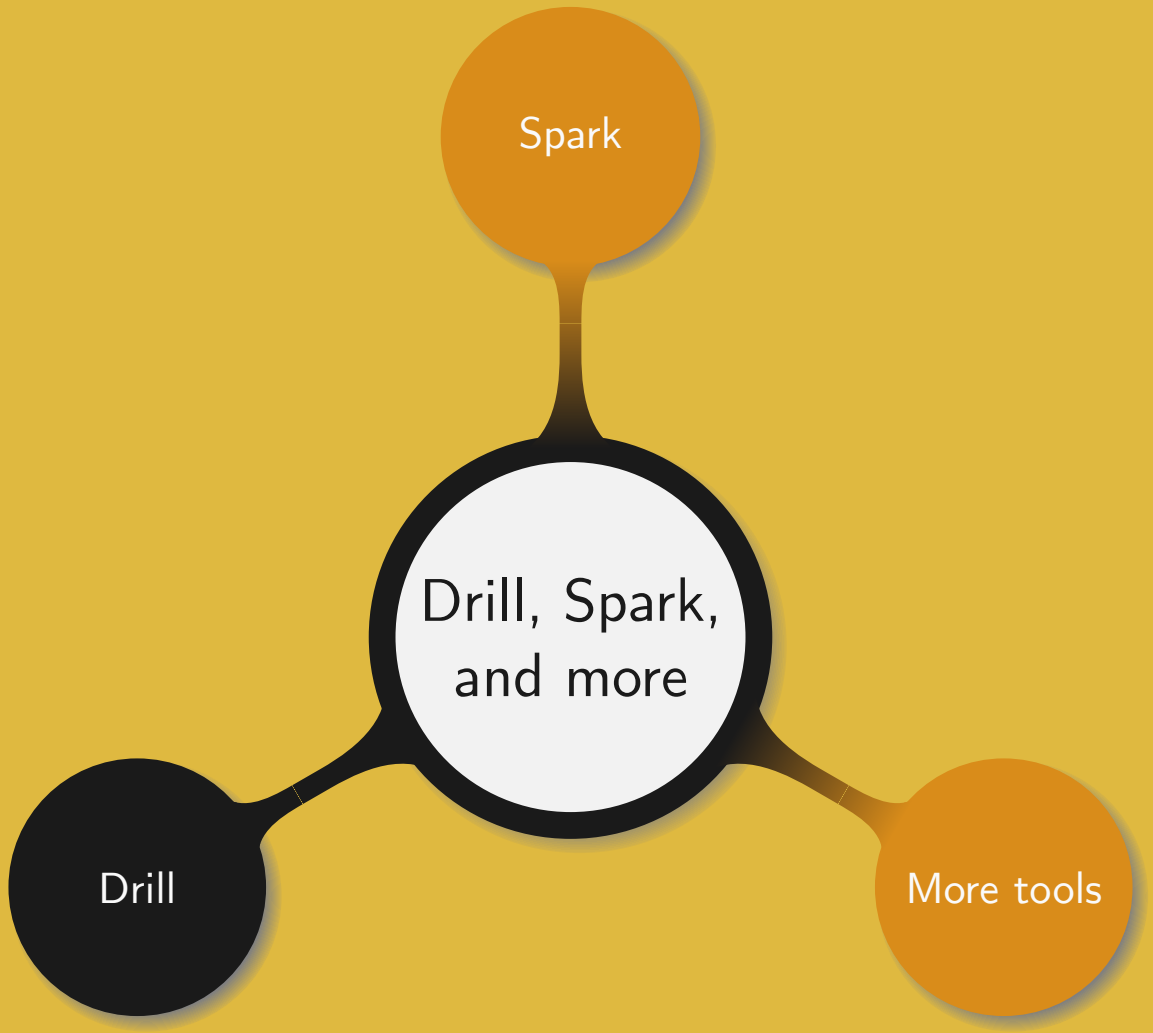


Methods and tools for big data

3. Drill, Spark, and more

Manuel – Summer 2022

Chapter organisation



Drill job

Parties always involved in a Drill job:

- A client which initiates the job
- Zookeeper

Parties optionally involved in a Drill job:

- YARN
- HDFS
- Hive
- HBase

Drill on YARN workflow

Running Drill as a YARN application:

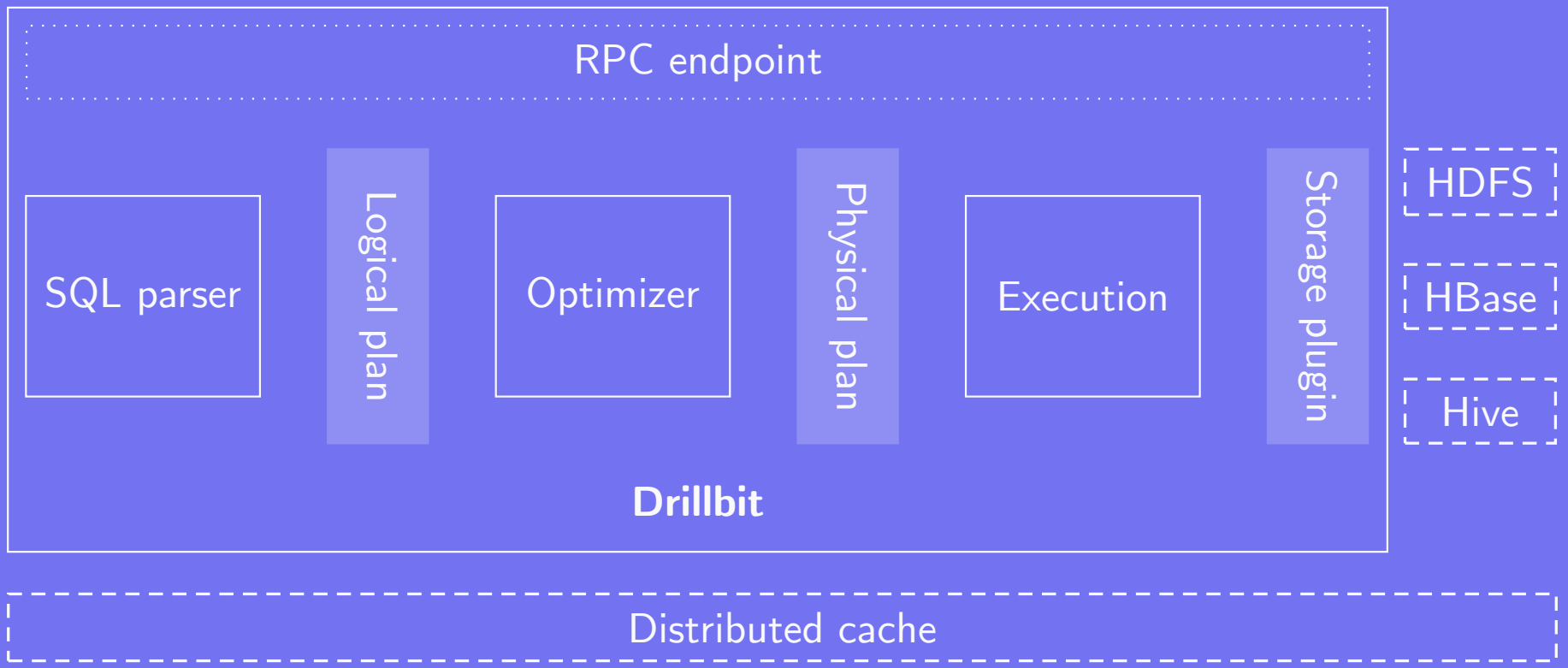
1. Start Drill on the client machine
2. Upload resources to the FS and request resources for the application master
3. Ask a node manager to prepare and start a container for the application master
4. The application master contacts the resource manager to obtain more containers

Drill on YARN workflow

Running Drill as a YARN application:

5. Request the start of Drill software on each assigned node
6. Start a "Drill process" called a *drillbit*
7. Each drillbit starts and registers with Zookeeper
8. The application master checks the health of each drillbit through Zookeeper
9. Use Zookeeper to retrieve information on the drillbits, run queries, etc.

Drillbit



Foreman drillbit:

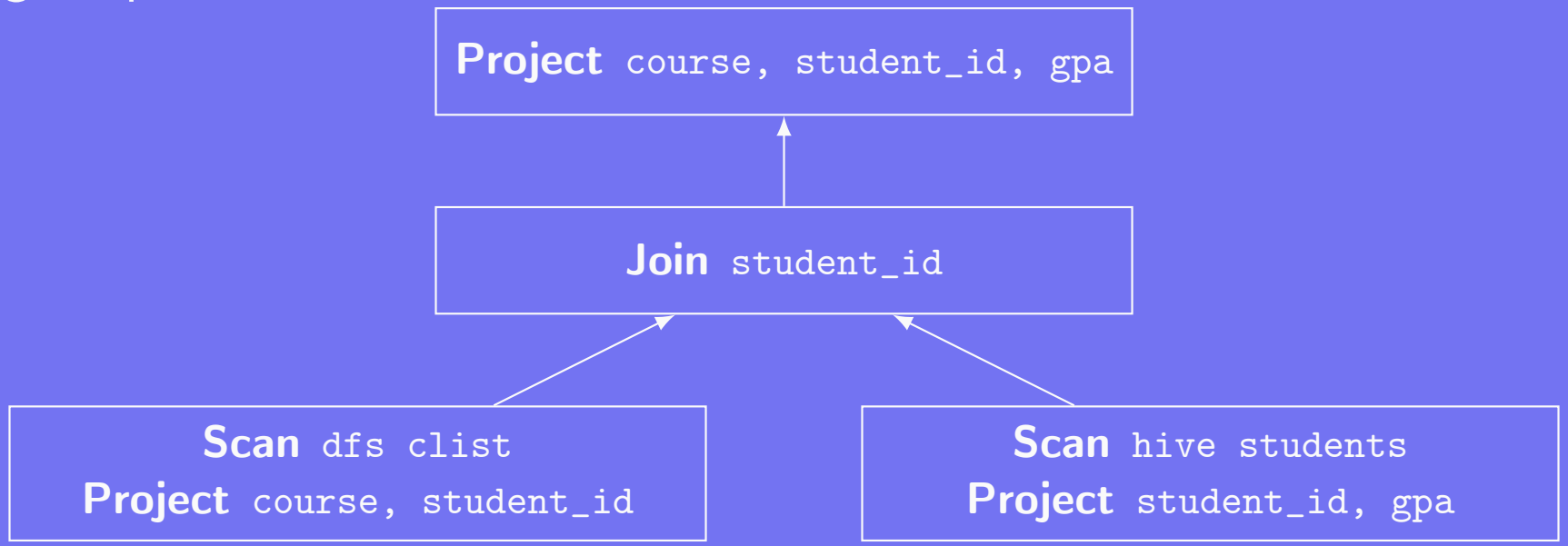
- Drillbit that receives the query
- It drives the entire query

Logical plan

Initial SQL query:

```sql
SELECT course, student_id, gpa
FROM clist.json l, hive.students s
WHERE l.student_id = s.student_id
```

Logical plan:



Drill query execution

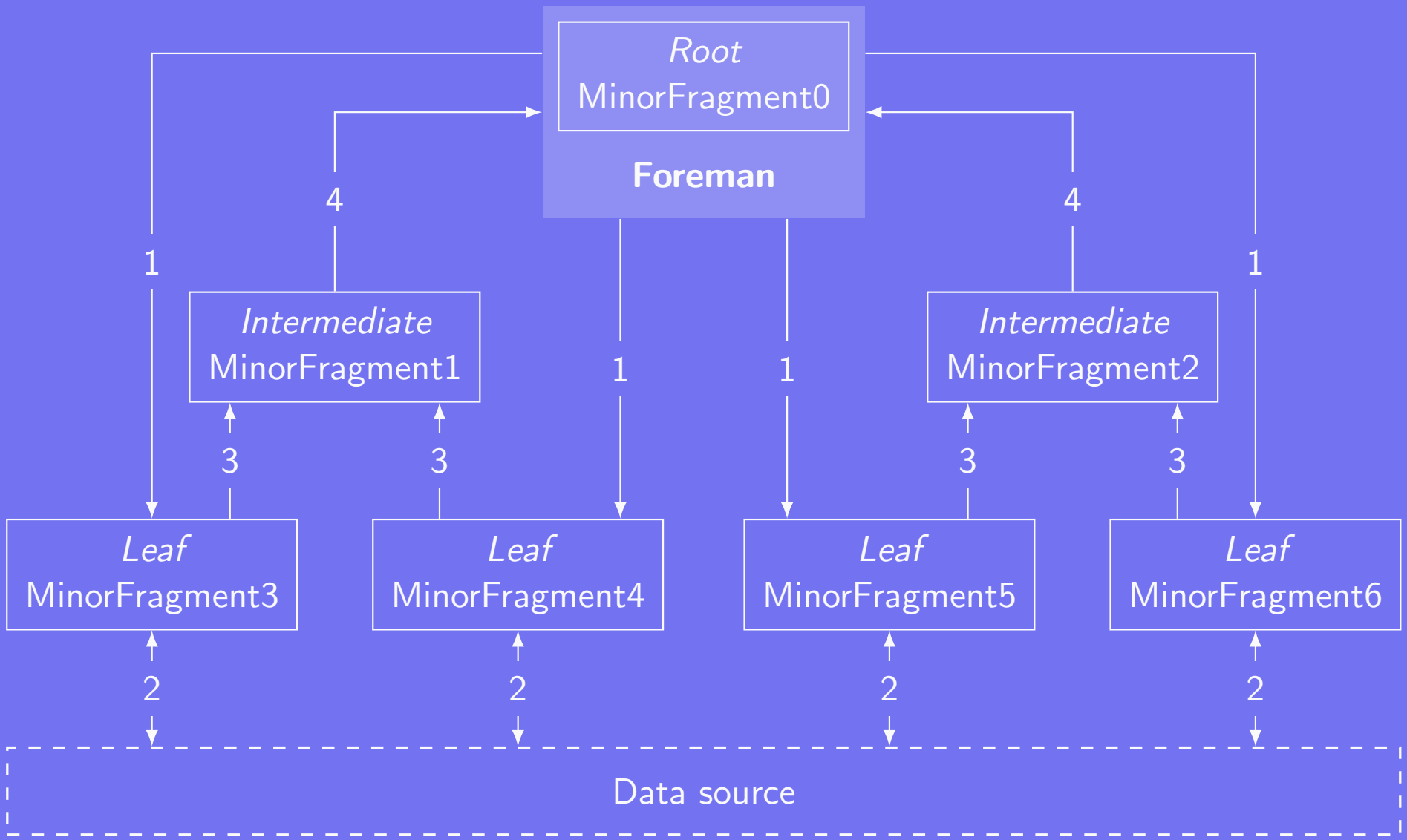
Major fragment:

- Concept representing a phase of the query execution
- Composed of minor fragments

Minor fragment:

- Logical unit of work running in a thread
- Contain one or more relational operators
- Usually as numerous as the number of available drillbits
- Scheduled based on data locality when possible and round-robin otherwise

Minor fragments execution tree



Drill's strength

Architecture:

- No central server, no master-slave concept
- Each drillbit contains all the services and capabilities of Drill
- Nodes can be added or removed at no cost

Columnar execution:

- Avoid access for columns not involved in the query
- Directly performs SQL processing on columns

Drill's strength

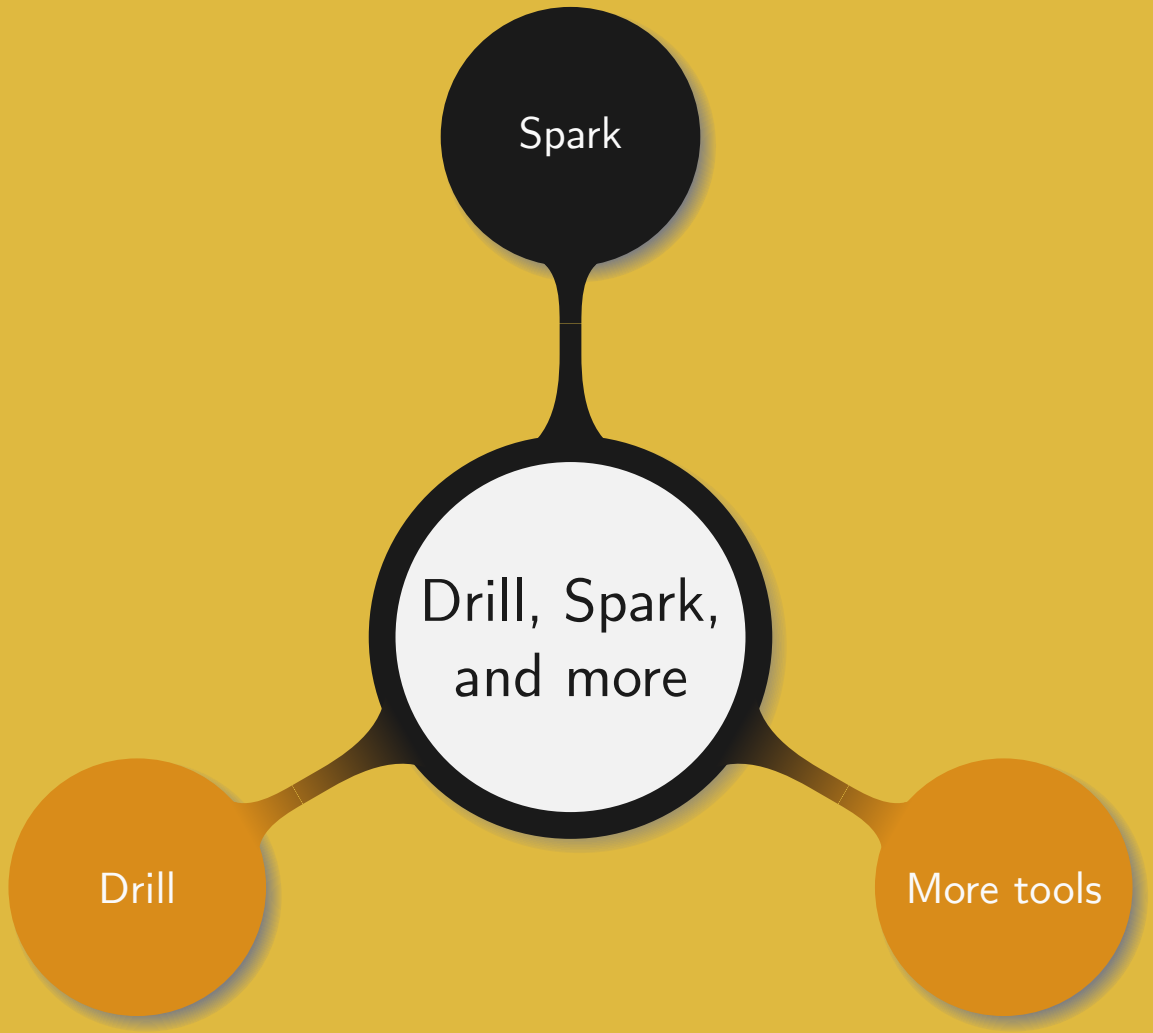
Optimistic query execution:

- Assume no failure will occur during query execution
- Rerun the query in case of failure
- Only write on disk when memory overflows

Vectorization: allow the CPU to operate on vectors

Runtime compilation: generate efficient code for each query

Chapter organisation



Spark job

Spark organisation:

- Application: user program built on Spark and composed of:
  - A driver program: process running the main function
  - Executors: processes launched for an application on worker nodes
- The driver program connects to a cluster manager
  - Standalone: cluster manager provided with Spark
  - YARN
  - Mesos
  - Kubernetes

Spark on YARN workflow

Two modes available:

- Client mode:
  - Driver runs in the client
  - Required in the case of interactive programs
  - Useful when building a Spark program
- Cluster mode:
  - The entire application runs in the cluster
  - Appropriate from production jobs
  - YARN application master failure strategy (slide 2.31) is applied

YARN client mode

Spark job workflow:

1. Start the driver program on a client node
2. The driver requests a container to the resource manager
3. A container starts and runs an Executor Launcher application master
4. The Executor Launcher requests more resources to start Executor backends processes in new containers
5. Each Executor Backend registers with the driver

YARN cluster mode

Workflow similar to client mode but:

- The driver program runs in a YARN application master process
- The client submit a job but does not run any user code
- The application master starts the driver program
- The driver program "replaces" the Executor Launcher

Remark. Data locality:

- Executors are launched before data locality information is available
- The driver can optionally specify preferred locations

Resilient Distributed Dataset

Resilient Distributed Dataset (RDD):

- Core abstraction in Spark
- Collection of objects distributed across a cluster:
  - Read-only: do not alter a dataset, transform it into a new one
  - Resilient: no disk write, reconstruct the RDD in case of partition loss
- Loaded as input:
  - Created from an external dataset
  - From an existing RDD
  - Parallelising an existing collection

RDD operations

Two types of operations on an RDD:

- Transformation:
  - Create a new dataset from an existing one
  - Only compute the result when an action is run
  - Do not return any result to the driver program
- Action:
  - Run a computation on a dataset
  - Return the value to the driver program

Benefits of this approach:

- Transformed RDD is in memory when performing an action
- No large dataset to send back to the driver program

RDD persistence

Datasets are cached in memory across operations:

- An RDD is stored on the node where it was computed
- An old RDD is dropped following the LRU algorithm
- A lost RDD is automatically recomputed if needed

Caching levels:

- Memory only: no compression, lost partitions are recomputed
- Memory and disk: partitions that do not fit in the memory are spilled on disk
- Memory only serialized: compression enabled
- Replication: all the above but also replicate on another node

Functions and variables

Serialization of data and functions:

- Used to share information among the executors
- Transparent to the user

Task closure:

- Cannot share variables among executors
- Determine what variables and methods an executor needs
- Serialize this closure and send it to the executor
- Each executor receives a copy of the original variable
- Variables are not updated on the driver

Shared variables

Broadcast variables:

- Read-only variables broadcasted to each executor
- Data sent in an efficient way to minimize traffic
- Useful for data needed over several stages of the computation

Accumulators:

- Variables that can be added to, using associative and commutative operations
- The driver can retrieve their value
- They are only updated on action tasks
- Update only occurs once, even if an action is rerun

Running of a Spark job

Job submission and execution:

- A job is submitted when an action is performed on an RDD
- The transformations on the RDD are organised into a logical execution plan
- Spark DAG scheduler transforms the logical plan into a execution physical plan
- The physical plan defines stages, split into tasks
- Spark task scheduler constructs a mapping of tasks to executors
- The executor runs the task
- Executors send status updates to the driver when a task is completed or has failed

Higher level abstractions

Newer and higher level abstractions relying on RDD:

- Datasets:
  - Foundational type of the structured APIs
  - Provide type-safety and allow much flexibility
  - Only available in Java and Scala and comes at a performance cost
- DataFrames:
  - Similar to a spreadsheet split into partitions
  - Internally defined as DataSets of type `Row`
  - Most common structured API, supported in all languages
- SQL Tables:
  - Data structure similar to DataFrames but defined within a database
  - Unmanaged table: defined from a file on the disk
  - Managed table: imported in and managed by Spark

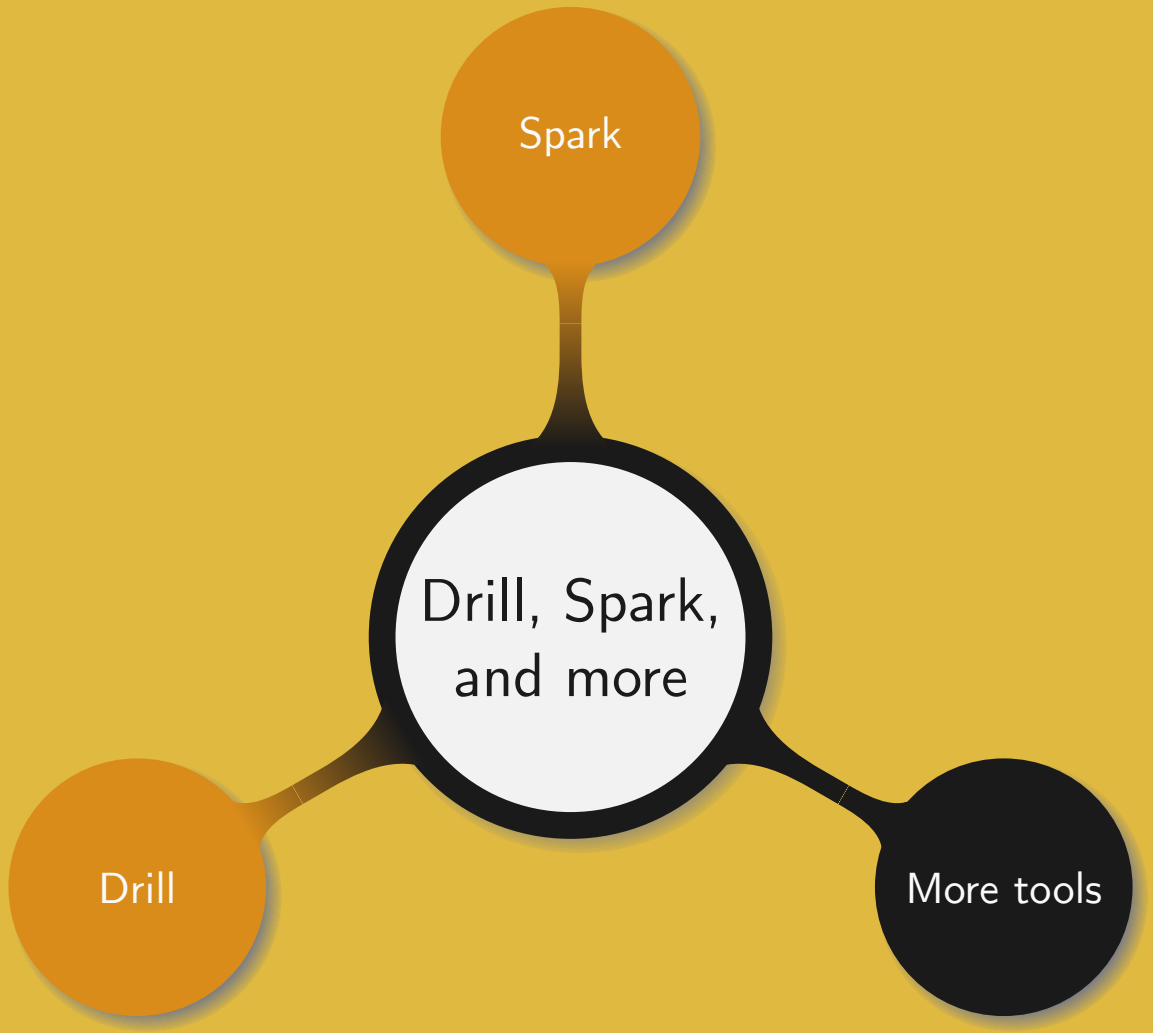
Spark's strength

Advantages of DAG:

- Any lost RDD can easily be recovered
- Offers more possibilities than a simple Map and Reduce approach
- Transformation on RDDs are not directly applied:
  - Allows better optimizations
  - Decreases disk writes and data transfer

Remark. Spark generalises the MapReduce approach, is much faster, and features many more high-level operators

Chapter organisation



Scaling up software

Simple observations:

- Over 20 billions devices are connected to the internet
- The complexity of software keeps increasing
- New security challenges need to be addressed
- Package manager dependencies are complex to handle

Alternative package management systems:

- Flatpak, Snap, AppImage:
  - Distribution agnostic packages
  - Applications are sandboxed, i.e. isolated from each others and the host
- Nix: *all* packages are isolated from each others

Linux containers

LinuX Containers (LXC):

- Operating-system-level virtualization method
- Relies on the kernel's *cgroup* and *namespace* isolation functionalities
- Concurrently run multiple isolated Linux OS on a machine
- Each container must be individually maintained
- Containers can be either privileged or unprivileged
- Containers access the bare machine and rely on the host kernel

LXC requires the setup of a whole OS for each container

Docker

Docker uses a different approach than LXC:

- Initially based on LXC but *now* completely independent
- A daemon manages the docker containers
- A container is an encapsulated environment that runs applications
- An image containing an application and its dependencies is built based on a "configuration file"
- To update simply replace the old image with a new one

Docker needs to be "manually" deployed, managed, and scaled

Kubernetes

Kubernetes is a container orchestration tool:

- Mostly used with, but not limited to Docker
- Initially developed as an internal Google project
- Kubernetes is a cluster application that handles a cluster:
  - *Master:* controls all other machines in the cluster
  - *Nodes:* the machines onto which applications are running
  - *Pods:* single instance of an application or running process

Kubernetes

Main tasks of Kubernetes:

- Monitor the health of the running applications
- Balances the load
- Manages hardware resources allocation
- Eases the deployment of preconfigured applications
- Allows access to storage in the same way as any other resources

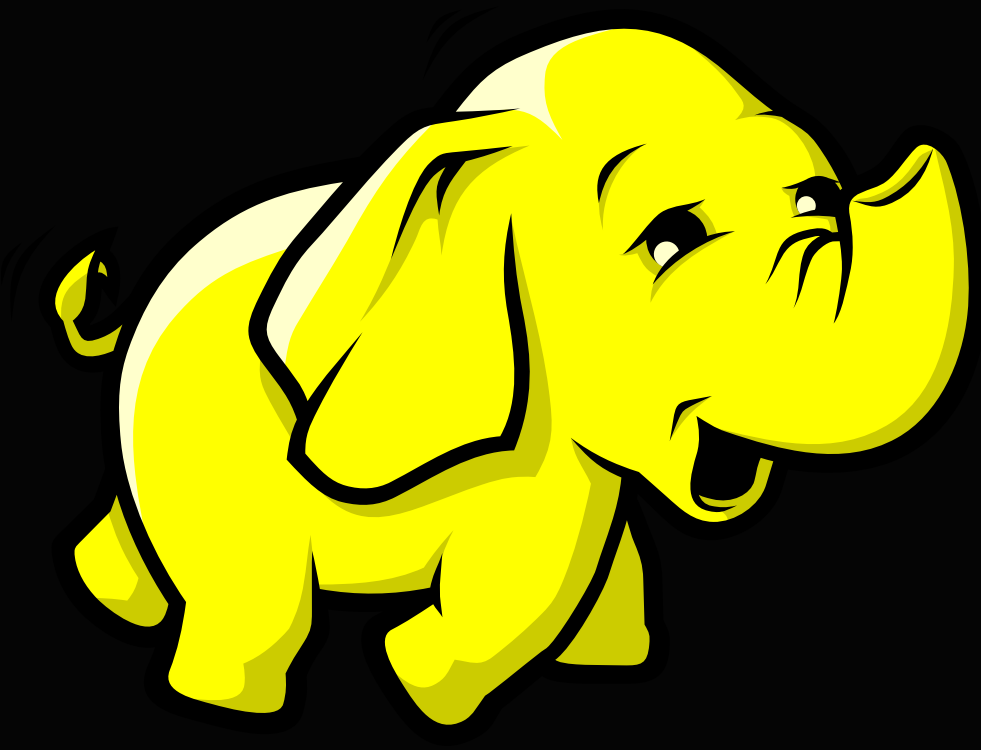
During the lifespan of an application:

- Containers can live, die, be resurrected
- Kubernetes handles everything without any human interaction

Kubernetes can be coupled to Hadoop or work independently

Key points

- When to use Drill?
- Explain the organisation of a drillbit
- When to use Spark?
- How does Spark interact with YARN?
- What are containerized solutions?



Thank you!