DATA ENGINEERING 2

LINUS STUHLMANN

TYPES OF BIG DATA ANALYTICS

Predictive Analytics	Descriptive Analytics	Prescriptive Analytics
• Predicting the future based on historical patterns. • What could happen?	• Mining historical data to provide business insights. • What has happened?	• Enabling smart decisions based on data. • What should we do?

DATA WAREHOUSE VS. DATA LAKE

DATA WAREHOUSE

- Zentrale Datenbank
- Verbindet Daten aus verschiedenen Quellen
- Strukturierte vorverarbeitete Daten (ETL)
- Unterstützt Entscheidungsfindung von Unternehmen

DATA LAKE

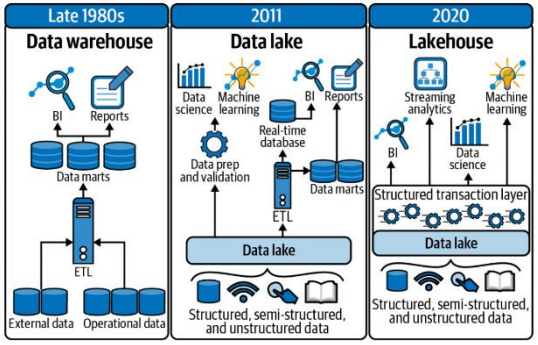
- Zentraler Speicherort
- Grosse Mengen an Rohdaten in nativer Struktur
- Sowohl strukturierte als auch unstrukturierte Daten
  - JSON, CSV, XML …
  - Texte, Bilder, Audio …
- Da nicht vorverarbeitet gehen keine Informationen verloren
  - Vollständigkeit
  - Flexibilität
  - Agilität

DATA SWAMP

Trotz der «unstrukturierten» Speicherung, muss trotzdem ein Metadata Management betrieben werden, da sonst der Aufwand der Datengewinnung schnell exponentiell zunimmt.

DATA LAKE HOUSE

- Modernes Datenarchitekturkonzept
- Vereint Vorteile von Data Lake und Warehouse
- Offene Formate



BIG DATA TOOLS

HADOOP

TRADITIONAL DATA PROCESSING

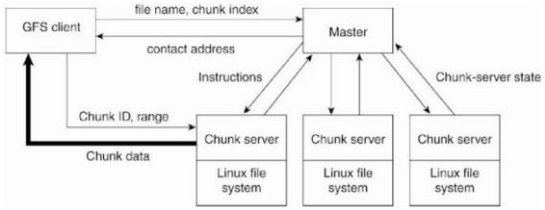
Grösstes Performance Bottleneck ist lese/schreiben in **persistenten Speicher**.

- Nur ~ 150 MB/s
- RAM/Cache viel schneller
  - Begrenzte Grösse

Wenn grosse Datenmengen verarbeitet werden sollen, macht es Anstelle von Scale Up mehr Sinn, Scale Out

DISTRIBUTED FILE SYSTEM

Verteiltes File System von Google (GFS)



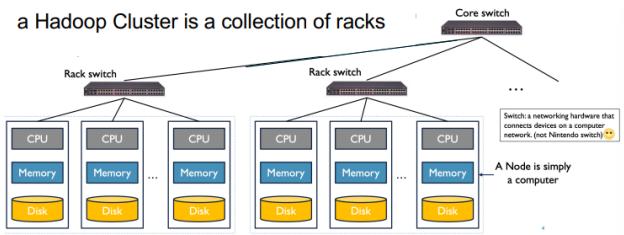
- Kontinuierliches Hinzufügen von Knoten / Ressourcen möglich

WAS IST HADOOP?

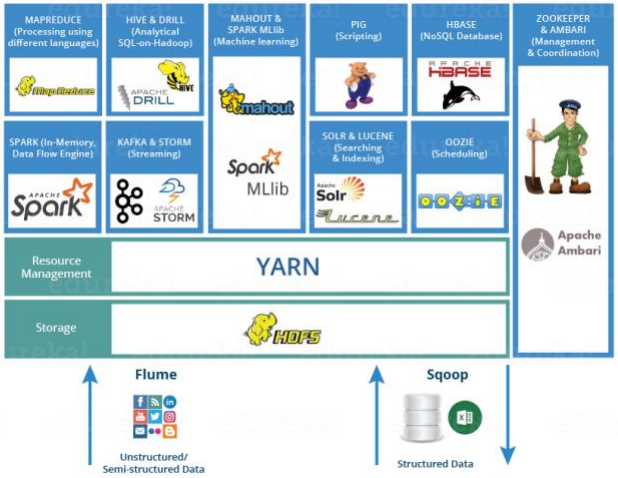
- Framework zum Speichern von Daten auf verteilten Systemen
- Applikationen verteilt laufen lassen.
- Cluster/ Gruppe von verbundenen Computern (Nodes)
- Netzwerke aus bezahlbaren Computern zu mehr Power zusammenschliessen
-

HARDWARE

Ein Hadoop-Cluster ist ein Netzwerk aus Racks.



ECOSYSTEM



TYPISCHE PROBLEME BEI IO CLUSTERN

- Node-Fail ~ 1 von 1000 Nodes failen pro Tag
  - Duplizieren der Daten
- Netzwerk Bottleneck - typischerweise 1-10 GB/s
- Verteilte Programmierung ist sehr kompliziert

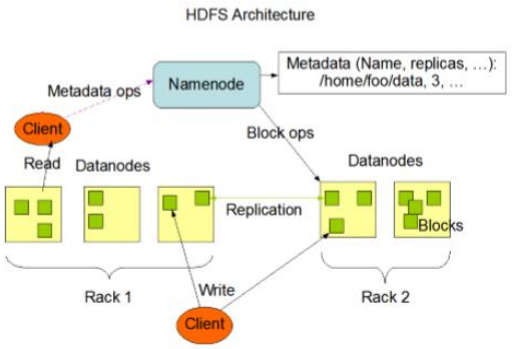
HADOOP KEY FEATURES

- «Shared Nothing»-Architektur
  - Cluster aus unabhängigen Nodes
- Fehlertoleranz
  - Daten werden repliziert gespeichert
  - Typischerweise 3x
  - Bei Ausfall wird Kopie aus anderem Node verwendet
- Commodity Hardware
  - Benötigt keine teuren Hardwarekomponenten
  - Kostengünstige Computer
- Horizontale Skalierung
  - Es können einfach neue Nodes hinzugefügt werden

HDFS

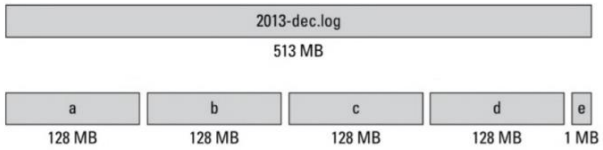
Hadoop Distributed File System (HDFS): Ein **skalierbares und verteiltes Dateisystem**, das Daten über mehrere Maschinen verteilt.

- «write once, read often»-Ansatz.
- Files können nicht geändert werden.
- Data Node = Computer/ Server im Cluster
- Rack = mehrere Computer in einem Rack



DATA BLOCKS

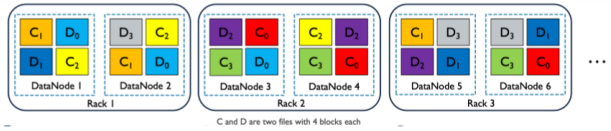
- HDFS teilt die Daten in gleich Blöcke à **128 MB** auf



DATA BLOCK REPLICATION

Folgende Regeln sind bei der Replikation der Datenblöcke notwendig:

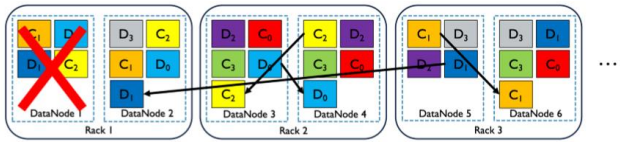
1. Kein Data Node enthält mehr als ein Replika von einem Block.
2. Kein Rack enthält mehr als zwei Replikas aus demselben Block



- Erhöht Sicherheit
- Dadurch kann bei Request auf Nodes mit geringster Latenz zurückgegriffen werden.

AUSFALL EINES DATA NODES

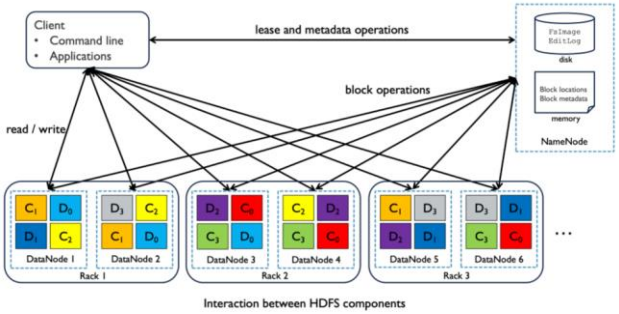
Neue Duplikate werden in verfügbaren Racks erstellt, indem die Blöcke aus den «nächsten» Nodes geholt werden. Ist der Node wieder online, wieder gelöscht.



NAME NODE

- Zentraler Metadaten Server für HDFS
- Verwaltet alle Adressen des HDFS
  - Welcher Block enthält was
  - Wo liegen zu welchem Block die Replikas
- Pflegt zwei Files
  - FSImage
    - Alle Informationen über Datenblöcke
  - EditLog
    - Alle Änderungen seit dem letzten Checkpoint

INTERAKTION DER HDFS KOMPONENTEN

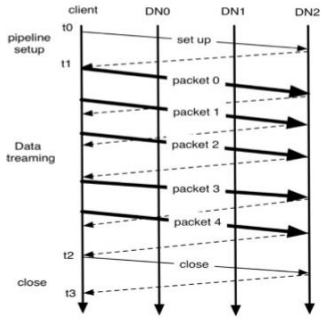


Interaction between HDFS components

- Durch Heartbeat (kurzes Signal) geben Data Nodes «Lebenszeichen»
  - 10 min, kein Response → Failure
- Alle 6 Stunden senden die Data Nodes einen Blockreport, welche Blöcke auf dem Knoten sind

WRITE PROZESSE

- Name Node teilt Block mit ID zu
  - Bestimmt Liste von Data Nodes für Replikas
  - Replika Algorithmus folgt den vorher genannten Block Regeln
  - Es wird die minimale Netzwerkdistanz zum Client berücksichtigt
- Jeder Data Node in der Pipeline:
  - Erhält Datenpacket
  - Speichert in lokalem Repository
  - Leitet das Packet weiter zum nächsten Data Node
- Acknowledgement der Data Nodes
  - Client schliesst File und meldet dem Name Node



READING PROZESSE

- Client sendet einen Request an Name Node für ein File
  - Name Node bestimmt die Liste von Blocks und den Standort von jedem Replika sortiert nach Distanz
  - Client probiert kleinste Distanzen zuerst.

DATEN INTEGRITÄT

- Client prüft Inhalt der Blöcke durch Checksum
- Wenn fehlerhaft
  - Client fragt anderen Data Node an
- Checksum wird auch durch Block Scanner benutzt um regelmässig Integrität zu prüfen

REBALANCING

Um unausgeglichene Verteilungen der Blöcke auf den Data Nodes zu vermeiden wird Rebalancing durchgeführt durch das HDFS.

$$\left| \frac{\text{used space at the node}}{\text{total capacity of the node}} - \frac{\text{used space in the cluster}}{\text{total capacity of the cluster}} \right| > \text{threshold}$$

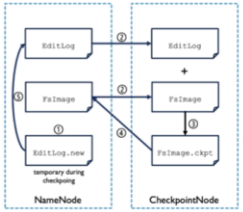
DECOMMISIONING

Soll ein Data Node abgeschaltet werden, werden nur noch Lese Zugriffe zugelassen und keine Replika Platzierungen mehr. Dann platziert der Name Node die Replika auf anderen Data Nodes bis der Node abgeschaltet werden kann.

CHECKPOINT

Es werden regelmässig Checkpoints erstellt von dem Name Node.

- 1. the NameNode creates a new file EditLog.new to accept the journaled file system changes
- 2. As a result, EditLog accepts no further changes and is copied to the CheckpointNode, along with the FsImage file
- 3. The CheckpointNode merges these two files, creating a file named FsImage.ckpt
- 4. The CheckpointNode copies the FsImage.ckpt file to the NameNode. The NameNode overwrites the file FsImage with FsImage.ckpt
- 5. The NameNode renames the EditLog.new file to EditLog.



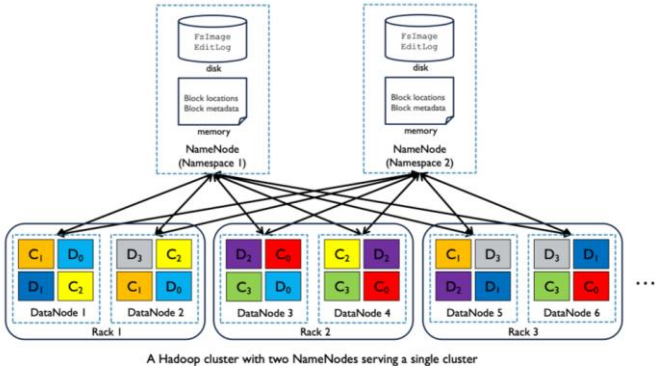
BACKUP NODE

Alternativ kann ein Backup Node erstellt werden, der immer synchronisiert werden muss.

- Real time Backup
- Schutz vor Datenverlust
- Höhere Verfügbarkeit (bei Ausfall)
- Benötig mehr Rechenleistung

FEDERATION

- Mehr als ein Name Node.
- Höhere Performance
- Isolation
  - Ein einzelner Name Node bietet keine Isolation in einer Multiuser Umgebung
  - Experimentelle Anwendungen können auf einem der Name Nodes durchgeführt werden
- Vermeidung von Bottlenecks



A Hadoop cluster with two NameNodes serving a single cluster

HBASE

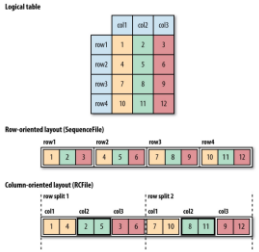
HBase ist eine Open-Source nicht relationale verteilte Datenbank nach dem Vorbild von **Googles Big Table**. Es läuft auf **HDFS**.

- Nicht relational
- Cluster von low-cost Commodity-Server
  - Flexibel
- grosse Datenmengen unstrukturierte Daten.
- Key - Value Pairs
- Spalten-orientiert
- Es werden keine leeren Zellen (Null) gespeichert
  - Es werden nur Key-Value Pairs gespeichert.

SPALTEN-ORIENTIERUNG

Bei herkömmlichen relationalen Datenbanken werden die Einträge Zeilenorientiert gespeichert. HBase speichert seine Einträge spaltenorientiert.

- Spaten werden zusammen gespeichert
  - Schnellerer **Spaltenzugriff**
  - Ermöglicht effizientere **Aggregation** von Daten über die Spalten hinweg.
  - Besser **skalierbar**, da einfacher Spalten über mehrere Knoten hinweg zu speichern



HBASE DATA TYPE

Jede Spalte besteht aus vier Elementen:

- Column Familiy
  - Fasst verschiedene Key-Value Paare zusammen.
  - In einer Column sollten immer Daten zusammen gespeichert werden die häufig zusammen abgefragt werden.
    - Vorname, Nachname
  - Am besten <=3 Column Families pro Tabelle
- Column Qualifier
  - Namen zur Identifikation von Einträgen
- Version
  - Timestamp (Unix)
  - Zur Versionskontrolle
- Values

Row Key	Column Family: {Column Qualifier: Version: Value}
00001	CustomerName: {'FN': 1383859182496: 'John', 'LN': 1383859182858: 'Smith', 'MN': 1383859183001: 'Timothy', 'MN': 1383859182915: 'T'} ContactInfo: {'EA': 1383859183030: 'john.Smith@xyz.com', 'SA': 1383859183073: '1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103: 'Jane', 'LN': 1383859183163: 'Doe', ContactInfo: {'SA': 1383859185577: '7 HBase Ave, CA 22222'}

- Es können Einträge weggelassen werden
- Durch Version kann eine History erstellt werden

QUERY – BEISPIEL

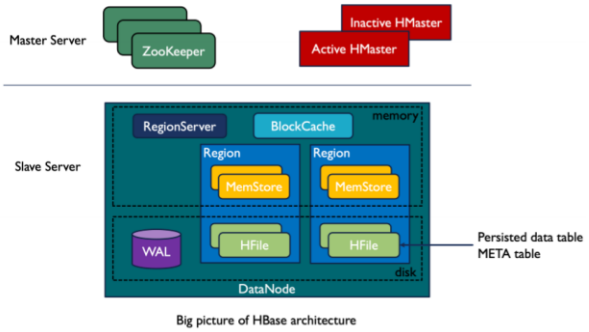
Row-Key = Primary Key

```
Row Key:(Column Family: Column Qualifier: Version) => Value
```

```
'00001:CustomerName:MN' => 'Timothy'
```

```
'00001:CustomerName:MN:1383859182915' => 'T'
```

ARCHITEKTUR

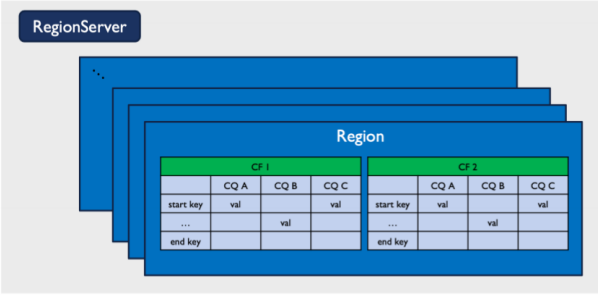


Big picture of HBase architecture

- Horizontale Aufteilung der Tables nach Zeilenschlüssel (Row Key) in Regionen
- Regionen haben default 1GB
- Region Server verwaltet 1000 Regions

REGION SERVER

- RegionServer sind Daemons
  - Speichert und stellt Daten bereit
  - Wenn zu viele Regions auf einem Regionserver verwaltet werden, werden die Daten automatisch an einen weiteren Server übertragen
  - Auto-Sharding

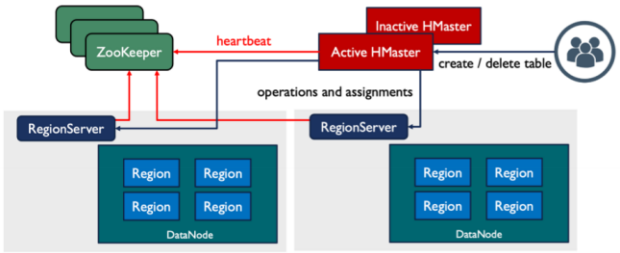


HMASTER

- Koordination und Organisation der RegionServer
- DDL-Operationen
- Monitort alle Instanzen im RegionServer im Cluster
- Interface um
  - Create
  - Delete
  - Update

ZOOKEEPER

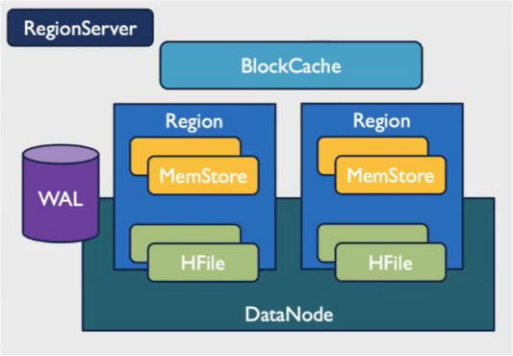
- Koordinator innerhalb HBase
- Hilft bei Maintainment
- Jeder RegionServer sendet Heardbeat zu HMaster → ZooKeeper überwacht das



READING UND WRITING IN HBASE

Die vier Schlüsselkomponenten übernehmen dies:

- BlockCache
- MemStore
- Write Ahead Log (WAL)
- HFile



BLOCKCACHE

- Daten werden in Blöcken gelesen und im BlockCache gespeichert
  - Speichert häufig zugegriffene Datenblöcke im RAM
  - Wenn neue Daten kommen, werden die am wenigsten genutzten Daten gelöscht

MEMSTORE

Speichert die Daten im RAM zwischen, bevor sie auf die Disk oder den persistenten Speicher geladen werden.

- Flush – wenn MemStore gewisse Grösse

WAL – WRITE AHEAD LOG

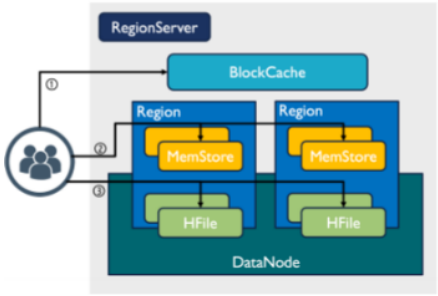
Die Write-Ahead Log (WAL) ist eine Protokolldatei, die alle Änderungen an den Daten in HBase aufzeichnet, bevor sie tatsächlich in den Speicher (MemStore) geschrieben werden.

- Datenverluste verhindern
- Datenintegrität gewährleisten

LESEMECHANISMUS

Je nach Speicherung wird ein anderer Lesemechanismus ausgelöst.

1. direkt Blockcache
2. Über MemStore
3. Über HFile (standard)



COMPACTION

Ist ein Prozess in HBase der durch die Kombination von HFiles probiert Speicher zu optimieren und die Abfrage schneller zu machen.

CRASH AND DATEN RECOVERY

- ZooKeeper erkennt, wenn ein Knoten ausfällt durch das Aussetzen des Heardbeats.
- HMaster verschiebt die Regionen auf einen anderen Server

EIGENSCHAFTEN VON HBASE

- Atomares Schreiben und Lesen
- Konsistentes Schreiben und Lesen
- Lineare und modulare Skalierbarkeit

EINSATZ VON HBASE

- Wir sollten HBase verwenden, wo wir **grosse Datenmengen** haben und wir **schnellen, zufälligen und Echtzeit-Lese- und Schreibzugriff** auf die Daten benötigen.
- Die Datensätze sind über **verschiedene Cluster verteilt**, und wir benötigen eine hohe Skalierbarkeit, um Daten zu verwalten.
- Die Daten werden aus verschiedenen Datenquellen gesammelt, und es handelt sich entweder um **halbstrukturierte** oder **unstrukturierte** Daten oder eine Kombination von beidem.
- Sie benötigen oder möchten **spaltenorientierte** Daten speichern.
- Sie haben **viele Versionen** der Datensätze, und Sie müssen alle speichern.

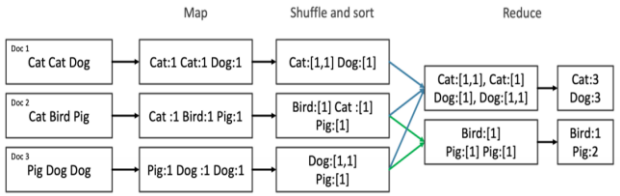
MAP REDUCE

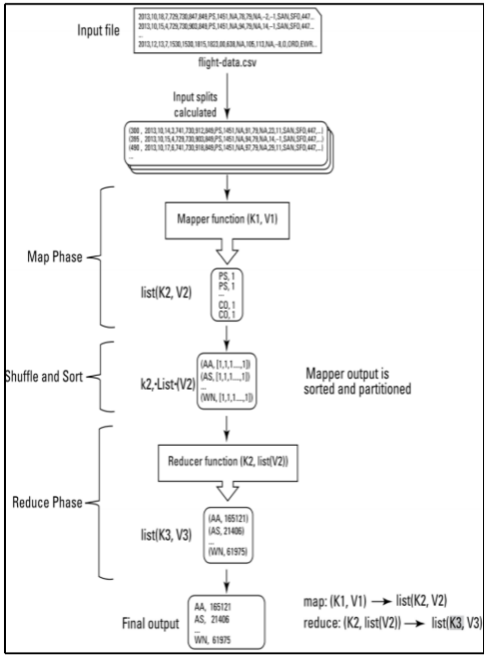
Map Reduce ist ein **Programmiermodell**, das Programmierern erlaubt, verteilte Applikationen zu entwickeln, ohne sich Gedanken zu machen, wie die darunterliegende verteilte Infrastruktur aussieht. Es ist kein Algorithmus der direkt eingesetzt werden kann, sondern eine **Spezifikation wie programmiert** werden sollte.

BEISPIEL: WORD COUNT

Mehrere Dokumente sind auf mehreren Knoten verteil und sollen parallel verarbeitet werden.

- Map Phase
  - Jedes Wort im Dokument wird zu einem Key-Value Paar gemacht (Wort: 1)
- Shuffle Phase
  - Alle Key-Value Paare werden umverteilt, dass alle Keys jeweils demselben Reducer zugewiesen werden.
- Sort Phase
  - Nach der Verteilung werden die Key-Value Paare sortiert, so dass immer alle Paare mit demselben Key nebeneinander sind.
- Reduce Phase
  - Daten werden aggregiert und für jeden Key ausgegeben.





MAP PHASE

- Input für die Map Funktion ist ein in Key-Value Paar. (K1, V1)
- Map Funktion generiert eine Liste mit Key-Value Paaren. List(K2, V2)

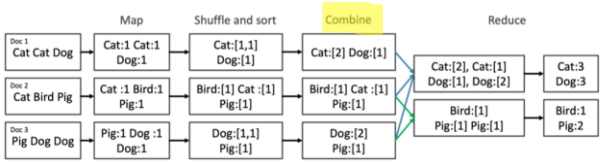
SHUFFLE AND SORT PHASE

- Immer noch auf dem Mapper Node werden die K2 Werte zusammengefasst und der jeweiligen Liste mit Werten V2 angehängt. K2, list(V1)

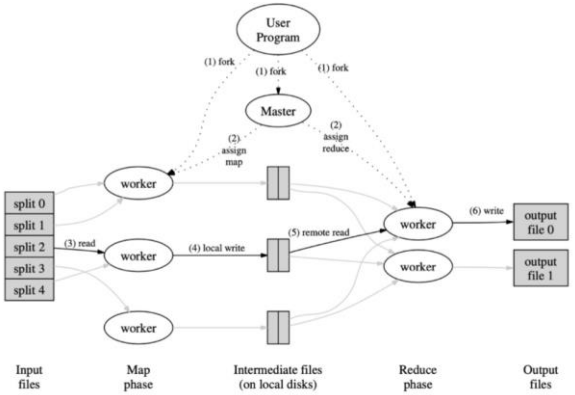
REDUCE PHASE

- Kombiniert alle K2, list(V2) → list(K2, list(V2)) und gibt eine Liste list(K3, V3) zurück.

OPTIONAL: COMBINE FUNCTION



GESAMTER PROZESS



AUSFALLTOLERANZ BEI WORKER-AUSFALL:

- Der Master sendet **regelmäßig Signale** (Pings) an jeden Worker.
- **Keine Antwort** innerhalb einer bestimmten Zeit führt dazu, dass der Worker als **ausgefallen** markiert wird.
- Abgeschlossene **Map-Aufgaben** des ausgefallenen Workers werden zurückgesetzt und **anderen Workern zur Neuausführung zugewiesen**.
- Laufende Map- oder Reduce-Aufgaben auf dem ausgefallenen Worker werden ebenfalls zurückgesetzt und neu geplant.

AUSFALLTOLERANZ BEI MASTER-AUSFALL

- Der Master kann periodische Checkpoints seiner Datenstrukturen schreiben.
- Stirbt der Master, kann eine neue **Kopie vom letzten Checkpoint gestartet** werden.
- Aktuell wird die MapReduce-Berechnung abgebrochen, wenn der Master ausfällt, Kunden können den MapReduce-Vorgang jedoch erneut versuchen.

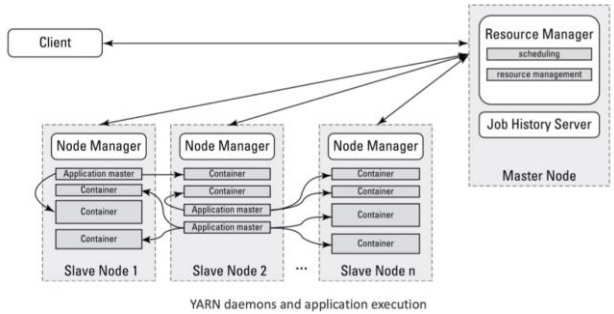
AUFGABENGRANULARITÄT:

- Die Map-Phase wird in M Teile und die Reduce-Phase in R Teile unterteilt.
- **M** und **R** sollten idealerweise **viel größer** als die **Anzahl der Worker-Maschinen** sein, um dynamisches Load Balancing und schnelle Wiederherstellung bei Ausfall zu ermöglichen.

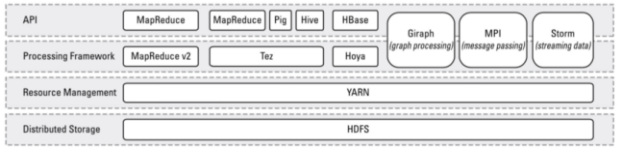
BACKUP-AUFGABEN:

- Um das Problem von Nachzüglern ("stragglers") zu lösen, die den Prozess verlangsamen, werden Backup-Ausführungen für verbleibende Aufgaben geplant, wenn eine MapReduce-Operation fast abgeschlossen ist.
- Eine Aufgabe gilt als abgeschlossen, sobald entweder die primäre oder die Backup-Ausführung fertig ist.
- Dies erhöht die genutzten Rechenressourcen nur um wenige Prozent, reduziert aber signifikant die Zeit, die für große MapReduce-Operationen benötigt wird.

HADOOP YARN (YET ANOTHER RESOURCE NEGOTIATOR)



YARN daemons and application execution



- YARN ist ein Werkzeug, das es **anderen Datenverarbeitungs-Frameworks** (Spark, MapReduce, HBase, …) ermöglicht, auf Hadoop zu laufen.
- Es ermöglicht das Ressourcenmanagement und führt gleichzeitig MapReduce- und HBase-Aufgaben von verschiedenen Clients aus.
- Jede Anwendung verfügt über einen eigenen Master, der speziell für ihre Aufgaben zuständig ist.

HADOOP YARN ARCHITEKTUR:

- YARN bietet eine **effizientere** und **flexiblere** Arbeitslastplanung und ein **Ressourcenmanagement**.
- Damit kann Hadoop **mehr als nur MapReduce**-Jobs ausführen.

VERTEILTE SPEICHERUNGS- UND RESSOURCENMANAGEMENT

- HDFS bleibt die Speicherschicht für Hadoop.
- YARN trennt das Ressourcenmanagement von der Datenverarbeitung, sodass es Ressourcen für jedes für Hadoop geschriebene Framework bereitstellen kann.

RESOURCE MANAGER:

- Der **Resource Manager** ist das Kernstück von YARN und verwaltet alle Ressourcen für die Datenverarbeitung im Hadoop-Cluster.
- Er hat eine globale Übersicht über alle Ressourcen und kümmert sich um die Bearbeitung von Ressourcenanfragen, Planung und Zuweisung.

UNABHÄNGIGKEIT DES RESOURCE MANAGERS:

- Der Resource Manager ist neutral in Bezug auf Anwendungen und Frameworks, kennt keine Map- oder Reduce-Aufgaben, überwacht nicht den Fortschritt von Jobs und kümmert sich nicht um Failover.
- Seine einzige Aufgabe ist die **Planung von Arbeitslasten**, was er sehr effizient durchführt.

NODE MANAGER:

- **Ressourcen Management auf Node Ebene**
- Jeder Slave-Knoten hat einen Node Manager-Daemon, der dem Ressource Manager untergeordnet ist und regelmäßig über die verfügbaren Ressourcen berichtet.
- Überwachung der Container

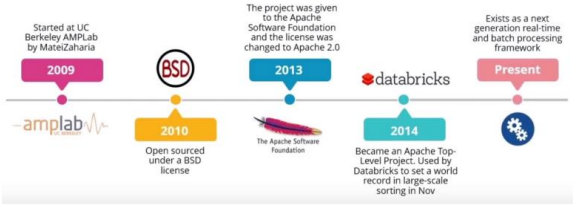
CONTAINER:

- Die Ressourcen im Hadoop-Cluster werden in kleinen Einheiten, sogenannten Containern, verbraucht, die alle notwendigen Ressourcen zur Ausführung einer Anwendung enthalten.

APPLICATION MASTER:

- **Jede Anwendung** auf dem Hadoop-Cluster hat eine eigene, dedizierte **Application Master-Instanz**, die in einem Container-Prozess auf einem Slave-Knoten läuft und den Zustand und Ressourcenbedarf der Anwendung an den Resource Manager meldet.

APACHE SPARK



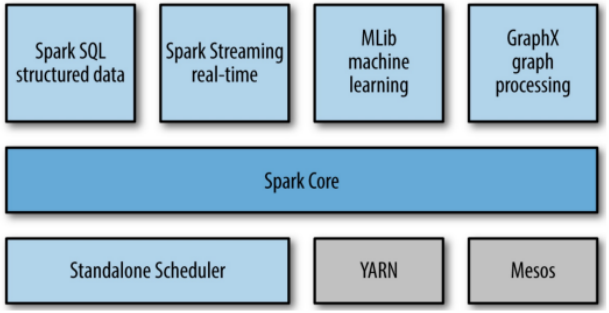
NACHTEILE VON HADOOP

- Kurzlebige MapReduce-Jobs
  - Für Jobs über mehrere Iterationen müssen immer neue Jobs erstellt werden
- Kann Zwischenergebnisse nicht im Memory speichern
  - Muss jedes Zwischenergebnis in den HDFS Speicher laden.
  - Nächste Iteration muss sie von dort wiederholen
- Nur MapReduce Jobs
- Ineffizienz

VORTEILE VON APACHE SPARK

- Performance
  - In-Memory-Computing
  - 100x schneller als Hadoop MapReduce
- Polyglot
  - Einheitliche API (Java, Python, Scala, R)
- Lazy Evaluation
- Echtzeitverarbeitung
- Fehlertolerant
- Machine Learning

KOMPONENTEN



ANWENDUNGEN VON SPARK

STREAMING

Real-Time Data Streaming

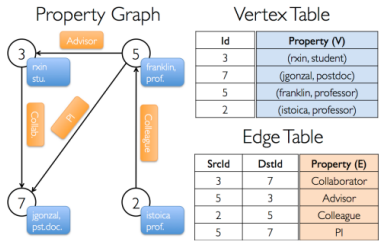
- Twitter
- Stock Market
- Geografische Daten

SQL

Relationale Datenbanken können eingebunden werden.

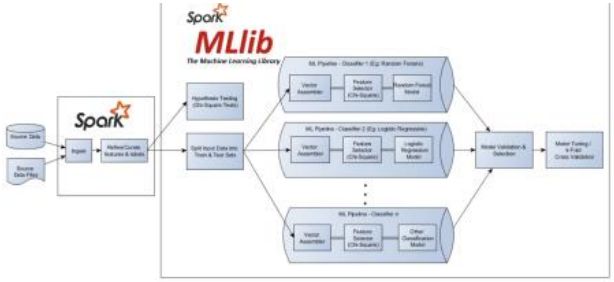
GRAPHX

Graph-Verarbeitung ist unterstützt



MLLIB

Verteilte Machine Learning Jobs sind auch unterstützt und kann bis zu 9x schneller sein.

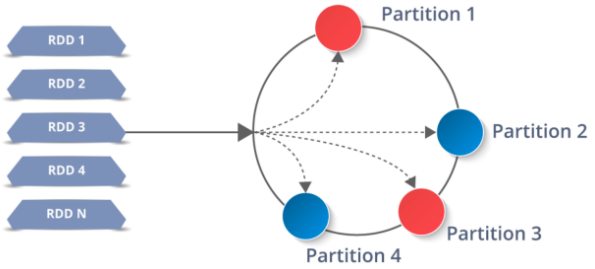


- Klassifikation
- Regression
- Clustering
- Collaborative Filtering

RDD

Resilient Distributed Dataset

- **Resilient**: Hohe Fehlertoleranz und Fähigkeit Daten wiederherzustellen bei Ausfall
- **Distributed**: Daten sind über mehrere Konten verteilt und können auch verteilt verarbeitet werden.
- **Dataset**: Sammlung von partitionierten Daten



- RDDs sind immutable, also nicht veränderbar und «read only». Wenn ein RDD verändert werden soll, muss es durch Transformation von alten RDDs neu erzeugt werden.
- RDDs werden **In-Memory** verarbeitet. Bei Shut-Down werden die RDDs in den persistenten Speicher geschrieben.
- RDDs werden gecachet

UNTERSCHIED RDD ZU HDFS

RDD wird für verteilte **Datenverarbeitung** verwendet, während HDFS für verteilte **Datenspeicherung** verwendet wird.

- RDD – Memory
- HDFS – Disc

WORKFLOW

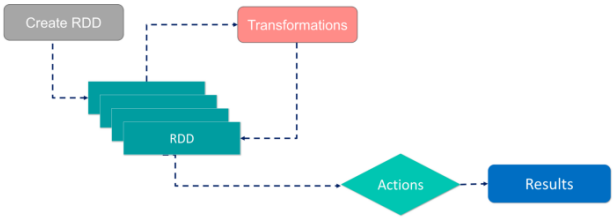
Jedes Datenset in RDD ist aufgeteilt auf verschiedene lokale Partitionen, welche auf unterschiedlichen Knoten im Cluser verarbeitet werden.

Ein RDD kann durch 3 Arten erzeigt werden

1. **Daten in RDD laden**, sc.parallelize(data)
2. Ein Datenset aus einem **externen Storage System wie HDFS oder HBase laden**
3. **Aus** einem **bestehendem RDD** erzeugen

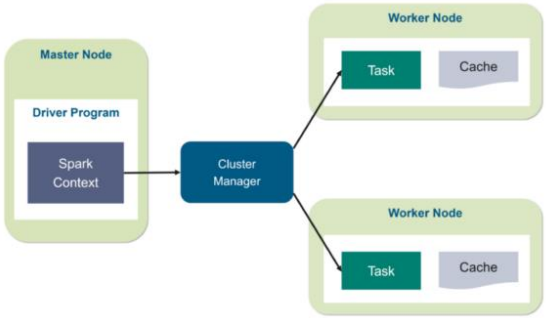
Es können **zwei Operationen** durchgeführt werden:

1. **Transformationen**
   a. 1 RDD rein, 1 oder mehrere raus
2. **Actions**
   a. Ergebnisse aus **Berechnungen mit RDD**



SPARK ARCHITEKTUR

Master Node managet das Hauptprogramm, das Driver Programm.



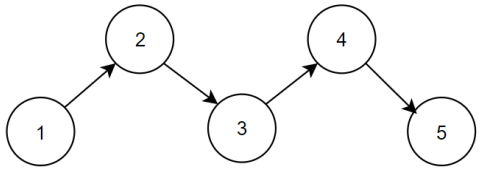
- Alle auszuführenden Befehle laufen über den sogenannten **Spark-Kontext**.
- Dieser arbeitet mit dem **Cluster-Manager** zusammen, um die verschiedenen Jobs zu organisieren.
- Die **Worker Nodes** sind **Slave Nodes** die die Jobs vom Cluster-Manager bekommen und verteilt ausführen.
- Die Ergebnisse werden zurück an den Spark-Kontext gegeben
- Die Nodes können horizontal skaliert werden, um Performance zu erhöhen.

DIRECTED ACYCLIC GRAPH – DAG

- DAGs in Spark repräsentieren Abfolgen von Transformation, die auf RDDs angewendet werden.
- Knoten → **RDD**
- Kanten → Operationen auf RDDs
  - Kein Kreis
  - Gerichtet

In Spark repräsentiert ein DAG:

```
rdd1 = sc.textFile("hdfs://path/to/file")
rdd2 = rdd1.map(lambda x: x.split(" "))
rdd3 = rdd2.filter(lambda x: len(x) > 3)
rdd4 = rdd3.map(lambda x: (x[0], 1))
rdd5 = rdd4.reduceByKey(lambda a, b: a + b)
```

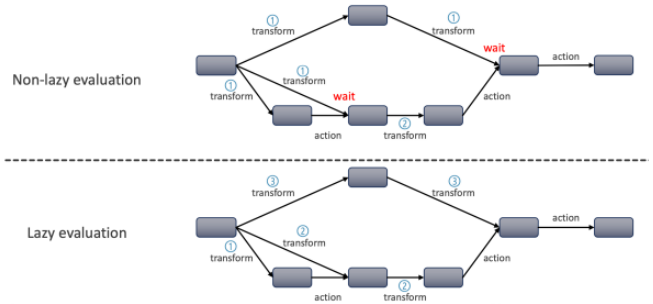


- Diese Transformation werden durch das Lazy Evaluation erst bei Anwendung von Aktionen auf den Graphen ausgelöst.

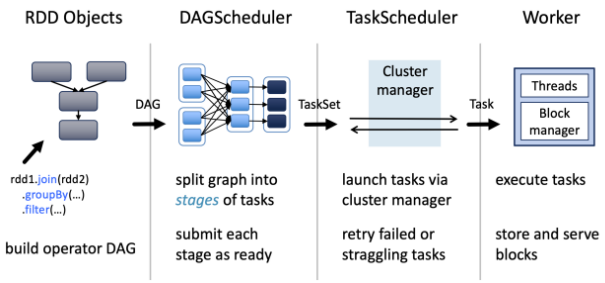
```
result = rdd5.collect()
```

LAZY EVALUATION

- Wird nur ausgeführt, wenn es wirklich benötigt wird. Erhöht Effizienz weil:
  - Weniger Wartezeiten
  - Bessere Verteilung der Auslastung



WORKFLOW



RDD LINEAGE – OPERATION GRAPH

- Da RDDs aus bestehenden RDDs erstellt werden, und die RRDs jweils einen Pointer zu ihrem Parent RDD haben, kann aus den RDDs wiederum ein Graph erzeugt werden.
- RDD Lineage enthält alle Abhängigkeiten zwischen den RDDs.
- Durch die Lineage können die Partitionen durch diese Informationen wieder hergestellt werden, solange der ursprüngliche RDD noch verfügbar ist.

APACHE FUNKTIONEN

BASIC RDD TRANSFORMATIONEN

Gibt RDD zurück!

- map(func)
  - wendet Funktion auf jedes Element in RDD an.
  - Gibt neues tranformiertes RDD zurück
- flatMap(func)
  - gleich wie map(), kann aber auch mehr oder weniger Output Items zurückgeben
- filter(func)
  - gibt RDD zurück mit nur den gefilterten Items
- distinct()
  - entfernt Duplikate in RDD und gibt neues RDD zurück
- union(RDD)
  - gibt RDD aus Vereinigungsmenge von RDDs zurück
- intersection(RDD)
  - Schnittmenge von RDDS und gibt neues RDD zurück
- sample()
  - gibt einen Teilsample des RDDs

BASIC RDD ACTIONS

Gibt nur Ergebnisse zurück!

- count()
  - Zählt Elemente in RDD
- countByValue()
  - Zählt wie oft jedes Element in RDD vorkommt
- collect()
  - erzeugt alle Daten im RDD als Array
- reduce()
  - Aggregiert Datenelemente in RDD
- take(n)
  - bezieht die ersten n Elemente
- top(n)
  - gibt die obersten n Elemente zurück
- takeOrdered(n)
  - gibt die n geordneten Elemente zurück
- takeSample(n)
  - gibt n gesamplete Elemente zurück
- aggregate()
  - aggregiert alle Elemente über alle Partitionen
- foreach()
  - wendet Funktion auf jedes Element an

RDD PERSISTIEREN /CACHING

Wird ein RDD mehrmals oder iterativ verwendet, das heisst derselbe Prozess wird mehrmals auf ein RDD angewandt, kann das sehr rechenaufwändig sein.

- Dafür können in Spark RDDs in einem Cache zwischengespeichert werden.
- Zwischenergebnisse können im Memory oder in Memory und im persistenten Speicher gespeichert.
  - chache(): Memory only
  - persist(StorageLevel.MEMORY_ONLY)
    - Persistieren im RAM
  - persist(StorageLevel.MEMORY_AND_DISK)
    - RAM & Festplatte
  - persist(StorageLevel.DISK_ONLY)
    - nur Festplatte
- _SER → Speicherung als Byte-Array
  - Datenmenge wird reduziert

RDD Storage Level	Store Format	When size of RDD is Greater Than Memory	Memory Usage	CPU Time
MEMORY_ONLY (default)	Deserialized Java object	Recompute	Very high	Low
MEMORY_AND_DISK		Store on the disk	High	Medium
MEMORY_ONLY_SER	Serialized Java object (one-byte array per partition)	Recompute	Low	High
MEMORY_AND_DISK_SER		Store on the disk	Low	High
DISK_ONLY	-	-	Very low	Very high

PAIRED RDDS

Paired RDDs bestehen aus Schlüssel-Wert-Paaren (Tuples) und bieten spezielle Transformationen, die auf diesen Paaren operieren.

```
('apple', 10), ('banana', 20), ('apple', 15)
```

TRANFORMATIONEN AUF PAIRED RDDS

- reduceByKey()
  - kombiniert Werte mit demselben Key
- groupByKey()
  - Gruppiert Werte mit demselben Key
- combineByKey(createCombiner, value, mergeCombiner)
  - Kombiniert Werte mit demselben Key
- mapValues()
  - wendet auf jedes Element in RDD eine Funktion an, ohne Key zu verändern
- flatMapValues()
- keys()
  - nur Keys
- vlaues()
  - nur Werte
- sortByKey()
  - nach Keys sortiert

TRANSFORMATION AUF ZWEI GEPAARTEN RDDS

- subtractByKey(otherRDD)
  - entfernt alle Elemente mit gleichem Key und gleichem Value
- join(otherRDD)
  - inner Join zwischen RDDs
- leftOuterJoin(otherRDD)
- rightOuterJoin(otherRDD)
- fullOuterJoin(otherRDD)
- cogroup(otherRDD)
  - Gruppiert Daten aus beiden RDDs mit demselben Key

AKTIONEN AUF GEPAARTE RDDS

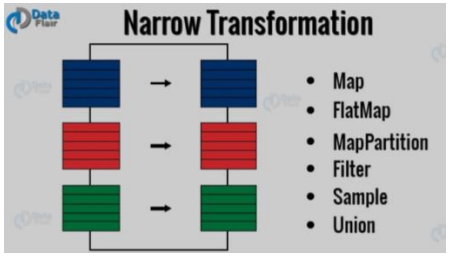
- countByKey()
  - Anzahl Elemente mit Key
- collectAsMap()
  - erzeugt HashMap aus RDD Key, Value
- lookup(key)
  - Gibt Elemente mit Key zurück

RDD TRANFORMATION TYPES

Transformationen, die aus RDDs in Spark angewendet werden, werden in zwei Kategorien eingeteilt.

- Narrow
  - Daten können in derselben Partition verarbeitet werden
  - **Effizient**
- Wide
  - Daten müssen über verschiedene Partitionen umverteilt werden.
  - **Ineffizient**

NARROW TRANSFORMATIONS

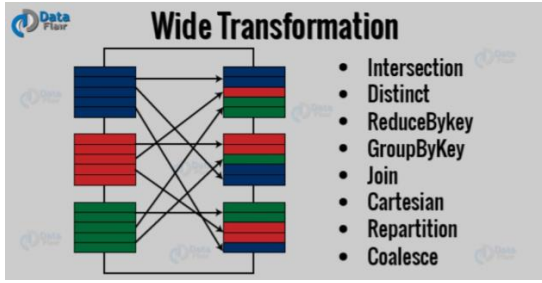


Wenig Daten müssen verschoben werden

WIDE TRANSFORMATION

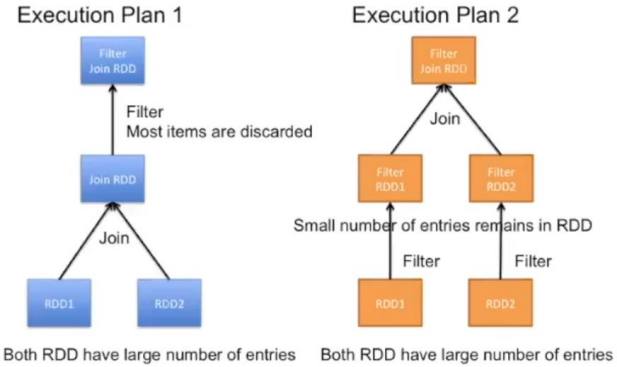
Viele Daten müssen aus verschiedenen Partitionen verschoben werden.

- Shuffling vor der Verarbeitung



EFFIZIENTE JOBPLANUNG

Wide Transformations sollen, wenn möglich so selten wie möglich ausgeführt werden:



In Plan 1 wird erst gejoint und dann gefiltert. Effizienter ist es erst zu filtern und dann zu joinen, da Joining eine Wide-Transformation ist. Wenn erst gefiltert (Narrow) wird, können die Kosten für den Join verringert werden.

PARTITIONEN

RDDs sind auf Spark clustern auf ein oder mehreren Knoten verteilt, wobei **jede Partition** einen **Teil der Daten** enthält. Das birgt folgende Vorteile:

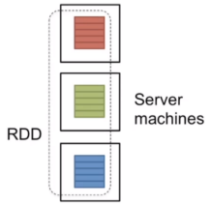
- **parallele** und **verteilte** Verarbeitung der Daten

Wird nicht explizit angegeben, auf wie viele Partitionen das RDD verteilt werden soll, wird die optimale Anzahl Partitionen von Spark bestimmt.

```
rdd = sc.textFile("hdfs://path/to/file", minPartitions=10)
```

Folgende Eigenschaften weisen die Partitionen auf:

- Jedes Key-Value Paar ist garantiert auf derselben Maschine (Knoten)
- Jeder Knoten kann mehrere Partitionen enthalten
- Je mehr Partitionen umso höhere Parallelisierung



- glom()
  - gibt Elemente innerhalb jeder Partition eines RDDs als Liste aus
- coalesce(numPartitions)
  - reduziert Menge der Partitionen in RDD auf die übergebene Anzahl
- repartition(numPartitions)
  - Daten werden neu angeordnet um (mehr oder weniger Partitionen) um die Last auszugleichen. Durch repartition() kann die Anzahl Partitionen verändert werden und die optimale Evrteilung gefunden werden.

```
# glom
rdd = sc.parallelize([1, 2, 3, 4], 2)
rdd.glom().collect()
```

```
[[1, 2], [3, 4]]
```

```
# coalesce
print(sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect())
print(sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect())
```

```
[[1], [2, 3], [4, 5]]
[[1, 2, 3, 4, 5]]
```

```
# repartition
rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
print(rdd.glom().collect())
print(rdd.repartition(2).glom().collect())
print(rdd.repartition(10).glom().collect())
```

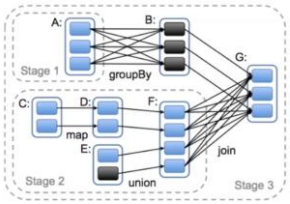
```
[[1], [2, 3], [4, 5], [6, 7]]
[[1, 4, 5, 6, 7], [2, 3]]
[[], [1], [4, 5, 6, 7], [2, 3], [], [], [], [], [], []]
```

- partitionBy()
  - nur für Pair RDDs (also Key-Value)
  - explizite Festlegung der Partitionierung
  - Gleicher Schlüssen zu gleicher Partition

STAGES

Das Konzept der Stages in Apache Spark ist zentral für die Ausführung und Optimierung von Transformationen auf Resilient Distributed Datasets. Operatoren werden in Stages eingeteilt, dabei wird darauf geachtet, dass die **«Narrow»-Transformationen zusammen gruppiert** werden, um «Wide»-Transformation zu verringern.

- Narrow-Transformationen in Stages unterteilen
- Wide-Transformationen definieren Ende einer Stage.
- Welche Partitionen sind schon im Memory
- Bestmöglich optimierter Ausführungsplan wird gesucht



GRAPHS IN SPARK

Nutzen von Graphen:

- Communities finden
- Kürzesten Pfad finden
- Ranking von Webseiten

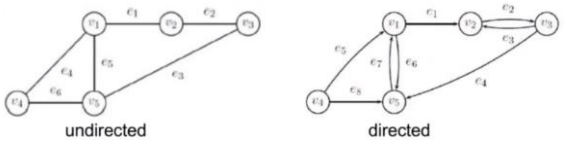
DEFINITION VON GRAPHEN (REPETITION)

Ein Graph besteht aus Knoten V (Vertices) und Kanten E (Edges).

$$G = (V, E)$$

$$V(G) = \{v_1, v_2, v_3, \dots, v_n\}$$

$$E(G) = \{e_1, e_2, \dots, e_n\}$$



- Ein Graph kann gerichtet und ungerichtet sein.

RELATIONALE VS. GRAPH DATENBANKEN

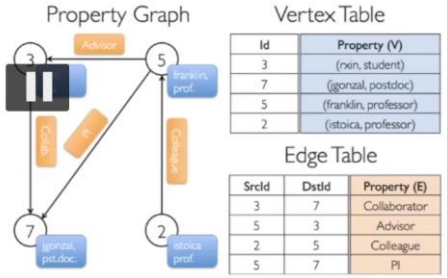
Sobald Abfragen mit mehreren «Hops» gemacht werden, eignen sich RDB nicht mehr, da dabei mehrfache Joins gemacht werden müssen, was sehr rechenintensiv ist.

- Graph Datenbanken sind designt für komplexe Netzwerkabfragen

GRAPHFRAMES

Der Aufbau eines Graph-Frames besteht aus zwei Data Frames:

1. Vertex Table
   a. ID
2. Edge Table
   a. Sources
   b. Destinations



VOTEILE

Drei verschiedene Abstraktionen können umgesetzt werden:

- Graph-Algorithmen
- Pattern-Matching
- Relationale Abfragen

ARCHITEKTUR

- Graph Frames sind als Spark SQL Engine implementiert
- Graph Operationen sind als relationale Joins implementiert
- Es werden unterschiedliche Datenquellen unterstützt
  - JSON, Parquet, CSV, Avro
  - Dokumente aus ElasticSearch/Solr

AUFBAUEN EINES GRAPHFRAMES

Besteht aus den relationalen Tabellen Vertices & Edges

- Vetices
  - Attribute
  - IDs
- Edges
  - Relationships
    - weitere Attribute
  - Sources
  - Destinations

IN PYTHON

```
from graphframes import *
```

```
vertices = sqlContext.createDataFrame([
  ("a", "Alice", 34),
  ("b", "Bob", 36),
  ("c", "Charlie", 30),
  ("d", "David", 29),
  ("e", "Esther", 32),
  ("f", "Fanny", 36)], ["id", "name", "age"])
```

Fixed column name for vertex-DataFrame

```
edges = sqlContext.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
  ("f", "b", "follow"),
  ("f", "c", "follow"),
  ("e", "f", "follow"),
  ("e", "d", "friend"),
  ("d", "a", "friend"),
  ("a", "e", "friend")
], ["src", "dst", "relationship"])
```

Fixed column names for edge-DataFrame

```
g = GraphFrame(vertices, edges)
```

```
vertices.show()
```

(2) Spark Jobs

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
+---+-------+---+
```

```
edges.show()
```

(2) Spark Jobs

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
|  f|  b|      follow|
|  f|  c|      follow|
|  e|  f|      follow|
|  e|  d|      friend|
|  d|  a|      friend|
|  a|  e|      friend|
+---+---+------------+
```

GRAPH OPERATIONS

EIN- UND AUSGEHENDE KANTEN

- Indegree
  - Anzahl eingehende Kanten zu Knoten
- Outdegree
  - Anzahl ausgehende Kanten zu Knoten

```
display(g.inDegrees)
```

(2) Spark Jobs

id	inDegree
f	1
e	2
d	1
c	2
b	3
a	1

FILTER NACH RELATIONSHIP

Wie viele Relationships in Graph?

```
numFollows = g.edges.filter("relationship = 'follow'").count()
print "The number of follow edges is", numFollows
```

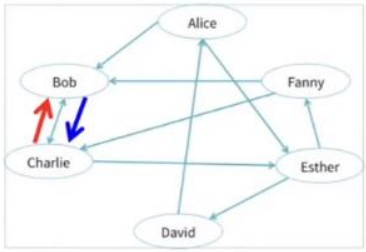
Wie viele «Friend»-Relationships?

```
numFriends = g.edges.filter("relationship = 'friend'").count()
print "The number of friend edges is", numFriends
```

PATTERN FINDING

Finde alle Knoten mit Kanten in beide Richtungen haben.

```
# Search for pairs of vertices with edges in both directions between them
motifs = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v1)")
display(motifs)
```



Das Ergebnis kann auch gefiltert werden:

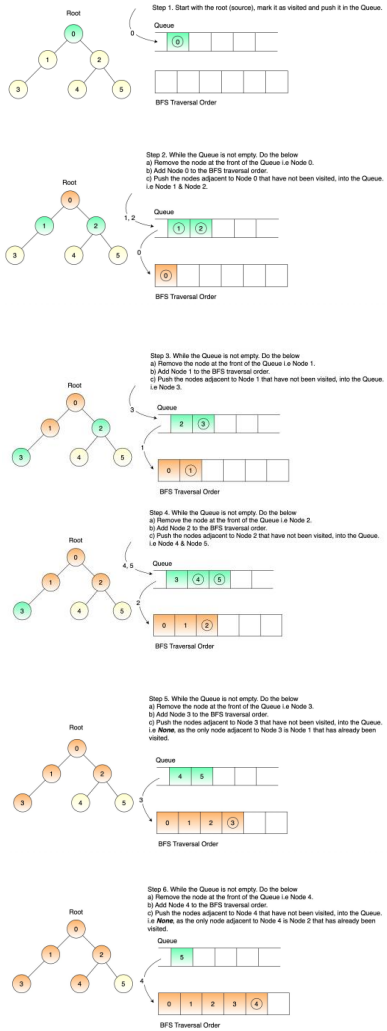
```
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends()  # Get example graph

# Select subgraph of users older than 30, and edges of type "friend"
v2 = g.vertices.filter("age > 30")
e2 = g.edges.filter("relationship = 'friend'")
g2 = GraphFrame(v2, e2)
```

BREADTH FIRST SEARCH

Es wird beim Root-Node gestartet und jeder benachbarte Knoten wird hintereinander

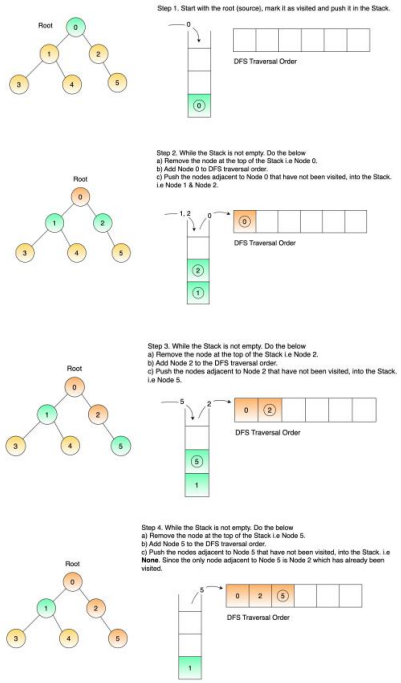
- Queue: Ist die Warteschlange in der gesucht werden soll.
- BFS: ist eine Liste, in der die Reihenfolge der Suche gespeichert wird.

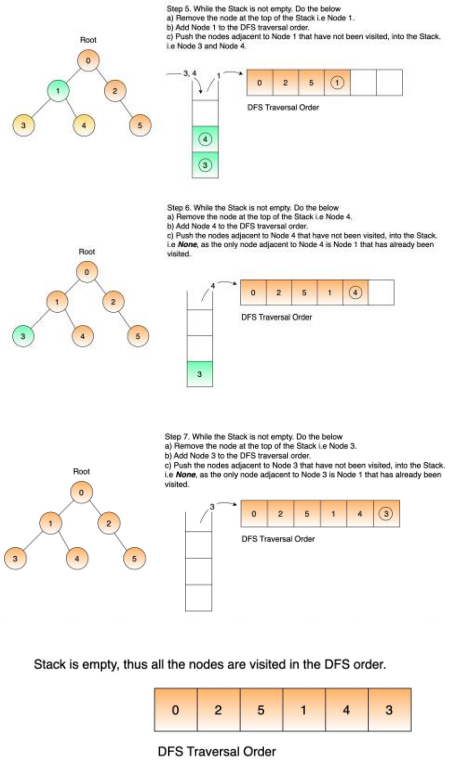


DEPTH FIRST SEARCH

Bei der Tiefensuche wird erst in die Tiefe gegangen, also jeden Pfad in die Tiefe abgesucht, bevor ein neuer abgesucht wird.

- Stack
  - Der Stapel auf dem die Knoten gestapelt werden.
- DFS Traversal Order
  - Reihenfolge des Durchlaufs



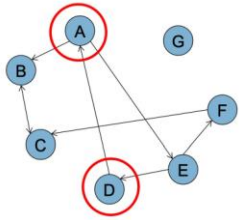


KÜRZESTER PFAD

Was sind die kürzesten Pfade von allen Knoten zu a und d?

```
shortestPath = graph.shortestPaths(landmarks=["a", "d"])
shortestPath.select("id", "distances").show()
```

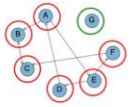
```
+---+----------------+
| id|       distances|
+---+----------------+
|  g|              {}|
|  f|              {}|
|  e|{a -> 2, d -> 1}|
|  d|{a -> 1, d -> 0}|
|  c|              {}|
|  b|              {}|
|  a|{a -> 0, d -> 2}|
+---+----------------+
```



CONNECTED COMPONENTS

Cluster bzw. Komponenten, die miteinander verbunden sind, also alle Knoten die Verbindungen zueinander haben.

```
sc.setCheckpointDir("/tmp")
result = g.connectedComponents()
display(result)
```



(16) Spark Jobs

id	name	age	component
a	Alice	34	1
b	Bob	36	1
c	Charlie	30	1
d	David	29	1
e	Esther	32	1
f	Fanny	36	1
g	Gabby	60	7

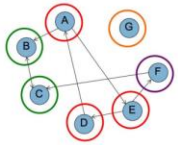
STRONGLY CONNECTED COMPONENTS

Ist ein Cluster aus Knoten die einen Pfad von jedem Knoten zu jedem Knoten haben. Also man erreich jeden Knoten von jedem Startknoten aus.

```
result = g.stronglyConnectedComponents(maxIter=10)
display(result.select("id", "component"))
```

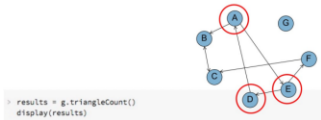
(2) Spark Jobs

id	component
g	7
f	6
e	1
a	1
c	2
b	2
d	1



TRIANGLE COUNT

Wenn alle drei Knoten miteinander verbunden sind wird es als Triangle gezählt.

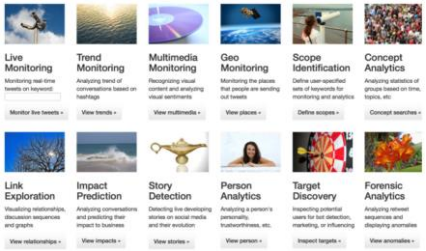



```
results = g.triangleCount()
display(results)
```

(4) Spark Jobs

count	id	name	age
0	g	Gabby	60
0	f	Fanny	36
1	e	Esther	32
1	d	David	29
0	c	Charlie	30
0	b	Bob	36
1	a	Alice	34

SPARK STREAMING



Stream Processing benötigt typischerweise:

- Hohes Volumen verarbeiten
- Real-time Datenverarbeitung (niedrige Latenz)
- Effizientes Recovering von Ausfällen

STREAMING ALGORITHMEN

- Offline-Algorithmen
  - Zugriff auf die gespeicherten Daten die aus einem Stream in einer Datenbank gespeichert wurden.
- Online-Algorithmen
  - Kontinuierliche Verarbeitung eingehenden Daten
  - Immer nur **zeitliche Ergebnisse**
- Streaming-Algorithmen
  - Sequenzielle Verarbeitung von Daten
  - **Ergebnisse erst am Ende** der Durchführung

ZU ANALYSIERENDE STREAMING PROBLEME

- Häufig vorkommende Elemente
- Sortieren
- Filtern
- Sampling
- Clustering
- Event Detection
- Online Learning

SPARK STREAMING

- Input
  - Datenströme wie Kafka, Flume, TCP-Sockets
- Output
  - Verarbeitete Daten abgelegt in:
    - Filesystem
    - Datenbanken
    - Live-Dashboards

- Input Data Streams sind unterteilt in Batches
  - aus Zeitintervall
- Jeder Batch ist ein RDD
- Die Resultate werden als RDD ausgegeben

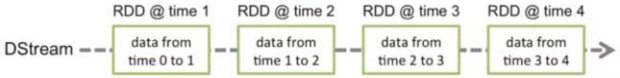
VORTEILE VON SPARK STREAMING

- Unterstützt Batch und Streaming Workloads
  - Sehr ähnlich zu Spark Core
- Dynamische Load Balancierung
- Schnelle Recovery (Checkpoints)
- Machine Learning
- Performance

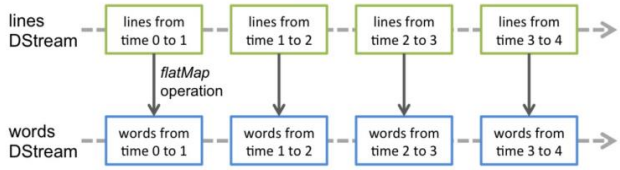
DSTREAM

Ein DStream (Discretized Streams) ist eine Repräsentation eines kontinuierlichem Datenstrom bestehend aus RDDs die ein Zeitintervall repräsentieren. Entweder als:

- Eingangsdaten
- Transformierte Daten



- Jede Operation die auf einen DStream ausgeführt wird, wir auf jedes in Stream enthaltene RDD ausgeführt.



TRANSFORMATIONEN AUF DSTREAMS

- wie bei RDD Transformationen: map(), filter(),…
- Key-Value RDD Transformations: cogroup(), join()

STATELESS- VS. STATEFUL-TRANSFORMATIONEN

- Stateless
  - Speichert keine Daten über die Batches hinweg
- Stateful
  - Speichert Informationen über die Batches hinweg
  - Ergebnisse können vorherige Informationen miteinfliessen lassen
  - Der State kann mit updateStatebyKey() geupdated werden

```
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)  # add the new values with the previous running count to get the new cou
nt
```

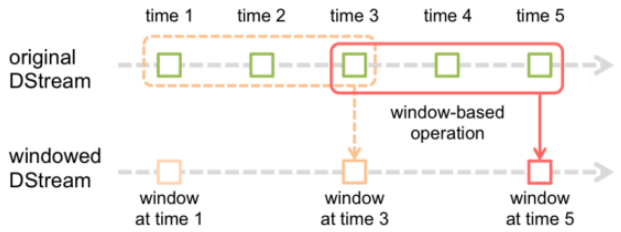
```
runningCounts = pairs.updateStateByKey(updateFunction)
```

WINDOWED OPERATIONS

Es werden Tranformationen auf ein sich verschiebendes Fenster angewendet.

Es werden zwei Parameter benötigt:

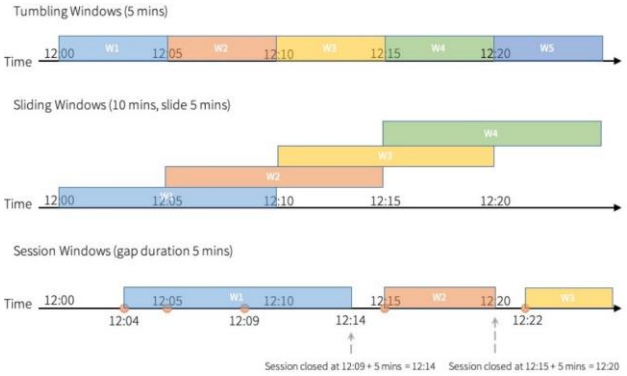
- Window Länge
  - Wie gross ist das Fenster
- Sliding Intervall
  - Wie weit Soll das Fenster sliden.




```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```

Es gibt unterschiedliche Window Arten:

1. Tumbling-Windows
2. Sliding-Window
3. Session-Window



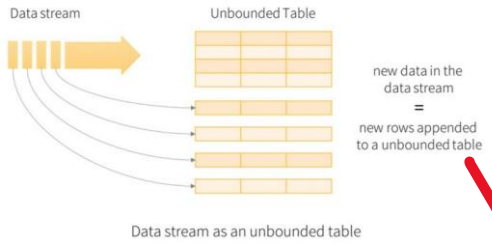
STRUKTURIERTES STREAMING

Kombination aus:

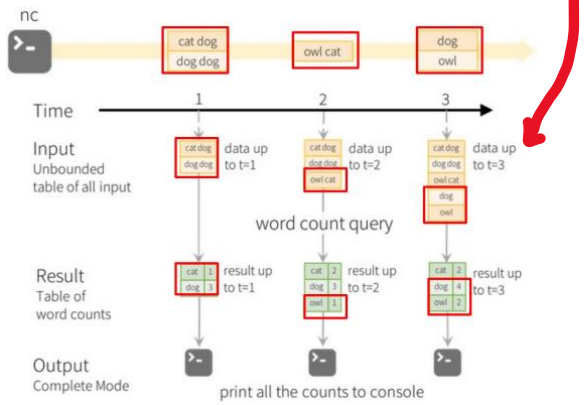
- Stream Processing
  - Spark Streaming
- Batch-Processing
  - Relational DB, NoSQL DB

STREAMING-DF

- Der Live-Stream wird als **Tabelle kontinuierlich upgedatet**
  - Darauf können normale Queries wie auf Standardbatches durchführen
  - Einfachheit
  - Resilient
  - Performance von Spark kann übernommen werden



PROGRAMMIER-MODELL

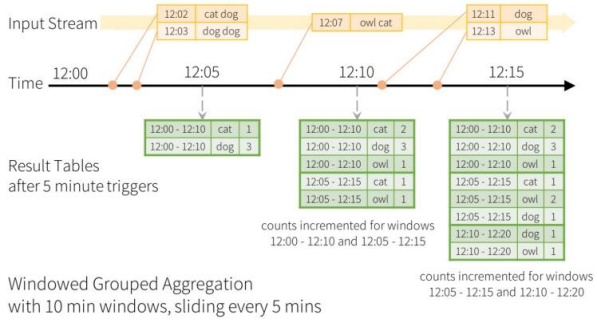


- Die neuen Zeilen des Streaming-DF werden in jedem Trigger-Intervall geschrieben.

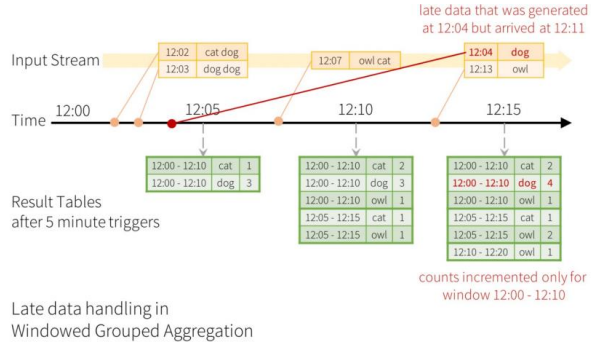
OUTPUT MODELLE

- Complete Mode
  - Das gesamte Ergebnis wird ausgegeben
  - t2 → cat:2, dog:3,owl:1 …
  - t3 → cat:2, dog:4, owl:2
- Append Mode
  - Nur neue Zeilen werden ausgegeben
  - t3 → dog:1, owl:1
- Update Mode
  - Nur upgedatete Zeilen werden ausgegeben
  - t3 → dog:4, owl:2

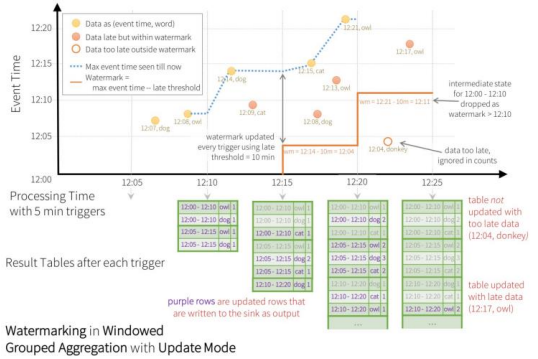
BEISPIEL STREAMING DF



Zu spät eintreffende Daten werden an der richtigen Stelle rückwirkend eingefügt.



Watermark: Zeitdifferenz, in der Daten noch berücksichtigt werden. Später eintreffende Daten werden ignoriert!



UNSTRUCTURED VS. STRUCTURED STREAMING

Unstructured Streaming

- **Pros:**
  - More flexible to different schemas of messages arriving on the same stream (or cases where there is no apriori schema – e.g. natural language text)
- **Cons:**
  - Need to handle state explicitly
    - E.g. with an additional global variable or via updateState API
  - Parsing data "by hand" is error-prone and tedious

Structured Streaming

- **Pros:**
  - Convenient to use SQL syntax
  - Easier to visualize
  - Convenient to understand schema
  - Easier for Spark to optimize => more efficient
- **Cons:**
  - Not very convenient for deeply nested structures
  - Not very convenient for heterogeneous data OR data without an apriori known schema