Fachhochschule Münster University of Applied Sciences

Wirtschafts-informatik

Masterstudiengang Wirtschaftsinformatik

Big Data Engineering

FH Münster
Master Wirtschaftsinformatik
Wintersemester 2015
Dozent: Lars George

Einheit 4

- Rückblick auf Einheit 3
- Dateiformate und Serialisierung
- Zugriff auf Daten: Abfrageschnittstellen
- Suche in Daten
- **Hauptziel**: Dateiformate und Abfrageschnittstellen kennenlernen

- **Übung 4**:
  - Daten in verschiedenen Formaten ablegen
  - Abfrage der Daten mit Hive und Impala

Einheit 4

- **Rückblick auf Einheit 3**
- Dateiformate und Serialisierung
- Zugriff auf Daten: Abfrageschnittstellen
- Suche in Daten

Rückblick auf Einheit 3

- Fragen?
  - Debugging versucht?
  - Local Modus ausprobiert?
- Übung 3: TF-IDF?
  - Was geben die Jobs aus? Was bedeuten die Zahlen?
  - Wie kann man das Endergebnis nutzen?

Einheit 4

- Rückblick auf Einheit 3
- **Dateiformate und Serialisierung**
- Zugriff auf Daten: Abfrageschnittstellen
- Suche in Daten

Dateiformate

In Hadoop **existieren** bereits **mehrere** Datei-formate. Sollte diese **nicht** reichen, so können Eigene **hinzugefügt** werden. Innerhalb der Formate gibt es Unterstützung für **verschiedene** Merkmale wie **Block**- oder **Spaltenorientierung**, **Kompression**, und so weiter.

Abhängig von der **weiteren** Verwendung sollte ein **geeignetes** Format ausgewählt werden, denn das kann sich **stark** auswirken.

Serialisierung von Daten

Unter **Serialisierung** versteht man das **definierte** Umwandeln von Daten aus einem Programm in eine **binäre** Form, welche über Rechner- und Betriebssystemgrenzen **ausgetauscht** und weiterhin **verstanden** werden.

Dies ist nötig für die **verteilte** Kommunikation **zwischen** Prozessen, zum Beispiel mit RPCs (Remote Procedure Calls) oder Dateien.

Ein **weiterer** Anwendungsfall ist die Unterstützung von **verschiedenen** Programmversionen.

Serialisierung von Daten

Auch in Hadoop werden Daten **serialisiert** und dann in die Dateien **geschrieben**. Dabei bieten **verschiedene** Formate Unterstützung für entweder **mehrere** oder nur **spezielle** Serialisierungsmethoden.

Im folgen **schauen** wir uns die Formate an und **diskutieren** später, wie und wo diese genutzt werden (können).

Dateiformattypen

In Hadoop **fallen** die Dateiformate in **zwei** Typen, oder Klassen: **Einfache** und **Container**-dateien.

Die **einfachen** Dateien sind zum Beispiel **Text** Dateien (.txt, .csv). Diese müssen bei jedem Lesen neu geparsed werden.

Containerdateien sind im **Gegensatz** dazu schon mit einer **bekannten** internen Struktur aufgebaut und können **direkt** gelesen werden.

Komprimierung

Innerhalb der Dateien kann dann noch **wahlweise** eine Komprimierung der Daten stattfinden. Dies geschieht **entweder** für die **ganze** Datei in einem, oder aber in **kleineren** Abschnitten. Hier kommt das **blockorientierte** Format der Containerdateien zum Zuge.

Wichtig für die Verarbeitung ist die **Frage**: kann die komprimierte Datei in **Splits** zerlegt werden?

Komprimierung

Bei Dateien, die **nicht** in einem **nativen** Hadoop Format abgelegt sind, **kann** es vorkommen, dass die Hadoop `InputFormat` Klassen diese **nicht** blockweise **lesen** können. Ein Beispiel ist eine Textdatei komprimiert mit GZip.

Dann muss ein **einziger** Mapper die **ganze** Datei lesen. Eine **nebenläufige** Bearbeitung ist dann **nicht** möglich.

Komprimierung

Auf der **anderen** Seite kann die Auswahl einer Komprimierungsmethode auch **helfen**, weniger oft benötigte Daten **höher** – aber eben auch aufwendiger (mehr CPU und Zeit) – zu **verdichten**.

Im folgenden sind **einige** Eigenschaften der im Hadoop Umfeld gebrauchten Komprimierungs-formate **aufgelistet**.

Komprimierungsformate

Format	Tool	Algorithmus	Endung	Teilbar
DEFLATE	-	DEFLATE	.deflate	Nein
gzip	gzip	DEFLATE	.gz	Nein
bzip2	bzip2	bzip2	.bz2	Ja
LZO	lzop	LZO	.lzo	Nein (Ja)
LZ4	-	LZ4	.lz4	Nein
Snappy	-	Snappy	.snappy	Nein

Hinweise:

- DEFLATE ist in der „zlib“ Bibliothek implementiert
- Die Endung „.deflate“ ist eine Hadoop Konvention
- LZO kann „indiziert“ werden, bedeutet aber einen extra Verarbeitungsschritt
- Hadoop ordnet die Codecs den Endungen zu wenn es sonst keine Hinweise gibt, kann aber über Job Parameter explizit gesetzt werden

Komprimierungsformate

Hadoop implementiert die Formate in „Codec“ Klassen.

Format	Hadoop Komprimierungs-Codec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
LZ4	`org.apache.hadoop.io.compress.Lz4Codec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

Implementierungen

Manche der Codecs sind in Java und/oder in nativem Code (C oder C++) implementiert. Diese ist in native Code wesentlich effektiver. Es benötigt aber eine externe Bibliothek.

Format	Java Implementierung	Native Implementierung
DEFLATE	Ja	Ja
gzip	Ja	Ja
bzip2	Ja	Nein
LZO	Nein	Ja
LZ4	Nein	Ja
Snappy	Nein	Ja

Konfiguration

Parameter Name	Typ	Vorgabewert	Beschreibung
`mapred.output.compress`	boolean	`false`	Ausgabe komprimieren
`mapred.output.compression.codec`	Klassenname	`org.apache.hadoop.io.compress.DefaultCodec`	Codec für die Ausgabe
`mapred.output.compression.type`	String	`RECORD`	NONE, RECORD oder BLOCK – für SequenceFile
`mapred.compress.map.output`	boolean	`false`	Map Ausgabe komprimieren
`mapred.map.output.compression.codec`	Klassenname	`org.apache.hadoop.io.compress.DefaultCodec`	Codec für die Map Ausgabe

Hinweise zur Komprimierung

Allgemeine Hinweise:

- Am besten sind Containerformate mit Komprimierung, denn sie sind teilbar
- Danach kommen teilbare Formate, welche auch Komprimierung unterstützen
- Ansonsten muss die Anwendung die Daten in Blöcke zerlegt vorbereiten
- Letzter Ausweg ist die Dateien unkomprimiert abzulegen

SequenceFiles

Ein sehr **praktisches** Dateiformat in Hadoop ist das **SequenceFile**. Dieses speichert Schlüssel/ Werte **Paare** ab, so wie sie auch in MapReduce verarbeitet werden. Damit **bietet** sich an, SequenceFiles als Ausgabe **zwischen** Phasen im Job und zwischen **mehreren** Jobs zu wählen.

Die Wertepaare sind die schon **bekannten** Klassen, abgeleitet von **Writable** (mehr dazu gleich).

SequenceFiles

Die SequenceFile Klasse bietet eine Reader und mehrere Writer Implementierung an, wobei es drei Writer Arten gibt, abhängig vom gewählten Komprimierungstyp, gesetzt durch z. B. die `SequenceFileOutputFormat.setOutputCompressionType()` Methode:

Typ	Writer	Beschreibung
`NONE`	`Writer`	Schreibt unkomprimiert
`RECORD`	`RecordCompressWriter`	Komprimiert pro Wertepaar
`BLOCK`	`BlockCompressWriter`	Komprimiert Wertepaare in Blöcken

SequenceFiles

Geschrieben wird immer ein **Header** mit Metadaten wie Version, Wertepaartypen, Komprimierung und Codec Name, etc.

Darauf folgen, abhängig vom gewählten Writer, die **Wertepaare** mit optionalen Sync-Marker oder **Blöcke** von Paaren mit Sync-Marker dazwischen.

Auch Wertepaar oder Block haben zuerst **Metadaten** welche die Größe des Datensatzes oder Blocks beschreiben.

Hadoop Writables

Jeder Schlüssel oder Wert in Hadoop, der **irgendwie** verarbeitet wird, stammt von der `Writables` Grundklasse ab. Diese **definiert** eine API zum **Schreiben** und **Lesen** der enthaltenen Rohdaten.

Dabei werden die Java **eigenen** `DataInput` und `DataOutput` Klassen aus dessen Serialisierungsbibliothek **unterstützt**.

Es folgt ein **Beispiel** einer eigenen Klasse.

Hadoop Writables

```
public class MyWritable implements Writable {
  // Some data
  private int counter;
  private long timestamp;

  public void write(DataOutput out) throws IOException {
    out.writeInt(counter);
    out.writeLong(timestamp);
  }

  public void readFields(DataInput in) throws IOException {
    counter = in.readInt();
    timestamp = in.readLong();
  }

  public static MyWritable read(DataInput in) throws IOException {
    MyWritable w = new MyWritable();
    w.readFields(in);
    return w;
  }
}
```

Hadoop Writables

Writables definieren **zusätzlich** eine Vergleichs-API, genannt `WritableComparable`, welche `Writable` und die Java `Comparable` API kombinieren. Der Vergleich ist **wichtig**, denn er wird in MapReduce **häufig** gebraucht zum Sortieren und sollte deshalb **effektiv** sein.

Writables können **ohne** Objektinstanziierung die Rohdaten **direkt** vergleichen und erfüllen so die Vorgabe.

Hadoop Writables

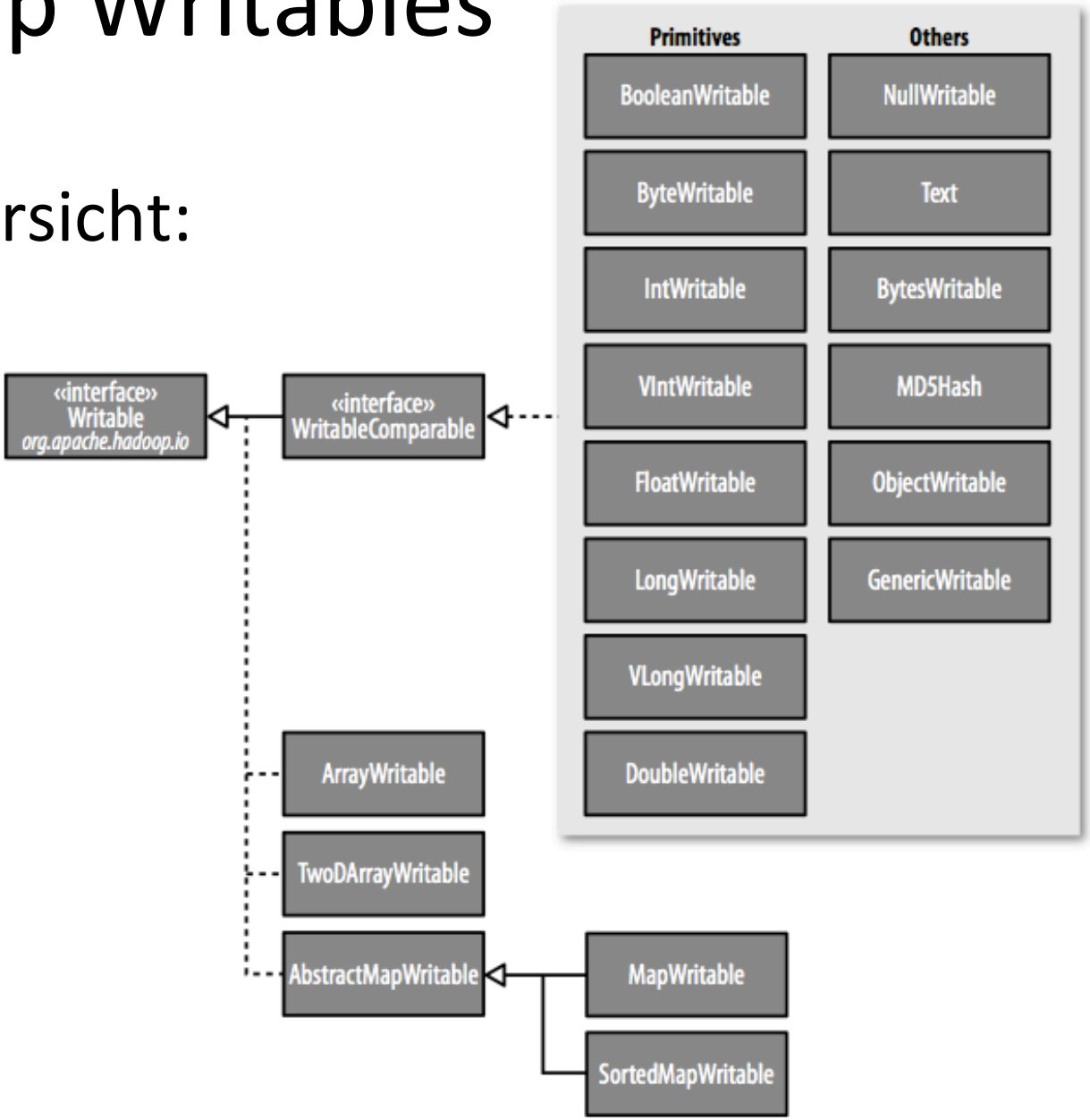
Es gibt **viele** mitgelieferte Klassen, für (fast) **alle** nativen Java Typen, Kollektionen, komplexe Objekte und so weiter. Wir haben die `NullWritable` Klasse in der **letzten** Übung gesehen, welche erlaubt „nichts“ auszugeben.

Es gibt auch **Varianten** wie `IntWritable` und `VIntWritable`, welche den Wert mit einer **variablen** Länge enkodiert.

Auch hier ist eine **geeignete** Auswahl wichtig.

Hadoop Writables

Eine Übersicht:



Serialisierung

Generell ist Serialisierung sehr **verbreitet**, speziell in Umgebungen, welche **Daten** entweder über Rechnergrenzen **austauschen**, oder **persistieren** für den späteren, erneuten Gebrauch.

Dazu sollte die **gewählte** Methode einige **wichtige** Eigenschaften mitbringen, speziell im Rahmen der hochperformanten, verteilten Verarbeitung von **großen** Datenmengen.

Serialisierung

Eigenschaften der Serialisierung:

- Kompakt
  - Nutzt Resources optimal aus, speziell z. B. das Netzwerk
- Schnell
  - Sollte wenig Auswirkungen haben aber trotzdem so effektiv wie möglich sein
- Erweiterbar
  - Zukünftige Änderungen sollten möglich sein, so dass z. B. eine Methodensignatur verändert werden kann
- Interoperativ
  - Verschiedene Systeme sollten unterstützt werden

Serialisierung

Die Hadoop Writable Klassen **erfüllen** diese Vorgaben. Deswegen wurde auch **nicht** die Java eigene, wesentlich **weniger** kompakte Serialisierungs-API benutzt.

Hadoop unterstützt aber auch **andere** Formate, welche über Parameter **deklariert** werden können. Dies nutzt auch Apache Avro aus, um direkt von Hadoop **unterstützt** zu werden.

Serialisierung

Viele der Serialisierungsprogrammiergerüste verwenden **IDLs** (Interface Definition Languages) um die **Struktur** eines Datensatzes zu **definieren**. Dies erlaubt die **externe** Schemadefinition auf allen **benötigten** Plattformen zu **verstehen** und dort mit einem nativen Codegenerator zu **übersetzen**.

Beispiele sind hier Avro, Thrift und Protocol Buffers.

Apache Avro

Avro ist eine **sprachunabhängiges** Serialisierungssystem, welches von Doug Cutting (Vater von Hadoop) **erschaffen** wurde, um eben die Java **Abhängigkeit** der Hadoop Writables zu **überwinden**.

Dazu **definiert** Avro Datensätze in IDLs, **anstatt** in Code. Dies ist besonders **wichtig** für komplexe Strukturen und Objekte, aber auch mit Hinblick auf die **Wartbarkeit**.

Apache Avro

Im Gegensatz zu **anderen** Systemen braucht Avro aber keine **explizite** Codegenerierung, sondern **nimmt** an, dass die Schemadefinition beim Lesen und Schreiben **vorliegt**. Dies ermöglicht, dass die Kodierung noch **effizienter** ist, denn Werte brauchen keine weitere Trenner oder sonstige IDs.

Die Schemas in Avro sind im **JSON** Format abgelegt.

Apache Avro

Schemas werden **sorgfältig** evaluiert und **unterstützen** so das Fortführen von Schemas. Beim Lesen und Schreiben müssen nicht die **gleichen** Schemas vorliegen, sondern können sich **weiterentwickeln**. Damit kann ein Programm Daten in einem **neueren** Schema lesen, welches mit einem **älteren** geschrieben wurde. So können zum Beispiel **Felder** einem Datensatz **hinzugefügt** werden.

Apache Avro

Als Dateiformat **definiert** Avro ein Containerformat, **ähnlich** dem des bereits besprochenen Hadoop SequenceFile. Es **speichert** das Schema und macht damit die Dateien **selbstbeschreibend**. Außerdem sind die Avrodateien **teilbar** und unterstützen **Komprimierung**.

Zusätzlich kann Avro auch für RPCs benutzt werden (hier aber nicht diskutiert).

Apache Avro - Beispiel

Schauen wir uns ein **Beispiel** an, zuerst als **generisches** Programm mit **direkter** Schema Übersetzung.

Danach das Gleiche, aber mit **generiertem** Java Code.

Das Beispiel **definiert** einen Datensatz mit **mehreren** Feldern eines primitiven Java Typs.

Apache Avro – Schema #1

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings.",
  "fields": [
    {"name": "left",
     "type": "string"},
    {"name": "right",
     "type": "string"}
  ]
}
```

Apache Avro – Schreiben #1

```
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(getClass().
  getResourceAsStream("StringPair.avsc"));
GenericRecord datum =
  new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");
ByteArrayOutputStream out =
  new ByteArrayOutputStream();
DatumWriter<GenericRecord> writer =
  new GenericDatumWriter<GenericRecord>(schema);
Encoder encoder = EncoderFactory.get().
  binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
```

Apache Avro – Lesen #1

```
DatumReader<GenericRecord> reader =
  new GenericDatumReader<GenericRecord>(schema);
Decoder decoder = DecoderFactory.get().
  binaryDecoder(out.toByteArray(), null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
```

Apache Avro – Schreiben #2

```
StringPair datum = new StringPair();
datum.left = "L";
datum.right = "R";
ByteArrayOutputStream out =
  new ByteArrayOutputStream();
DatumWriter<StringPair> writer =
  new SpecificDatumWriter<StringPair>(
    StringPair.class);
Encoder encoder = EncoderFactory.get().
  binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
```

Apache Avro – Lesen #2

```
DatumReader<StringPair> reader =
  new SpecificDatumReader<StringPair>(
    StringPair.class);
Decoder decoder = DecoderFactory.get().
  binaryDecoder(out.toByteArray(), null);
StringPair result = reader.read(null, decoder);
assertThat(result.left.toString(), is("L"));
assertThat(result.right.toString(), is("R"));
```

Schema Erweiterung

Dadurch dass der lesende Prozess ein **eigenes** Schema hat, nämlich das für den Prozess zur Zeit **aktuelle**, kann er die Daten mit deren Schema **abgleichen**. Neue oder fehlende Datenfelder können dann **ignoriert** oder **ergänzt** werden.

Im folgen Beispiel **fügen** wir ein weiteres Feld hinzu. Man beachte den „Vorgabewert“, ohne den sonst ein Fehler **auftreten** würde.

Apache Avro – Schema #2

```
{
   "type": "record",
   "name": "StringPair",
   "doc": "A pair of strings with \
      an added field.",
   "fields": [
     {"name": "left", "type": "string"},
     {"name": "right", "type": "string"},
     {"name": "description",
      "type": "string", "default": ""}
   ]
}
```

Wichtig!

Apache Avro – Lesen #3

```
DatumReader<GenericRecord> reader =
  new GenericDatumReader<GenericRecord>(
    schema, newSchema);
Decoder decoder = DecoderFactory.get().
  binaryDecoder(out.toByteArray(), null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertThat(result.get("description").toString(),
is(""));
```

Beide Schemas angegeben

Schema Projektion

Ein Vorteil der Schemaauflösung ist die sogenannte „**Projektion**“. Damit kann zum Beispiel ein sehr **großes** Schema in ein **kleineres** Überführt werden, damit nur **benötigte** Felder zur Verfügung stehen. Dies ist **ähnlich** den Views in SQL.

Fehlende Felder werden entweder **ignoriert** oder mit Vorgabewerten **eingesetzt**, je nachdem ob der Leser oder Schreiber aktueller ist.

Weitere Avro Merkmale

Über das Genannte hinaus unterstützt Avro auch noch sogenannte **Feldaliase**, welche es erlauben Felder **umzubennen**.

Die eingebaute **Sortierung** erlaubt es Avro Datensätze sehr **effektiv** (binär) zu vergleichen und zu **ordnen**. Da diese Funktion schon **mitgeliefert** wird, kann man im Schema die Sortierunsgreihenfolge **elegant** setzen. Ansonsten gilt die **natürliche** Reihenfolge.

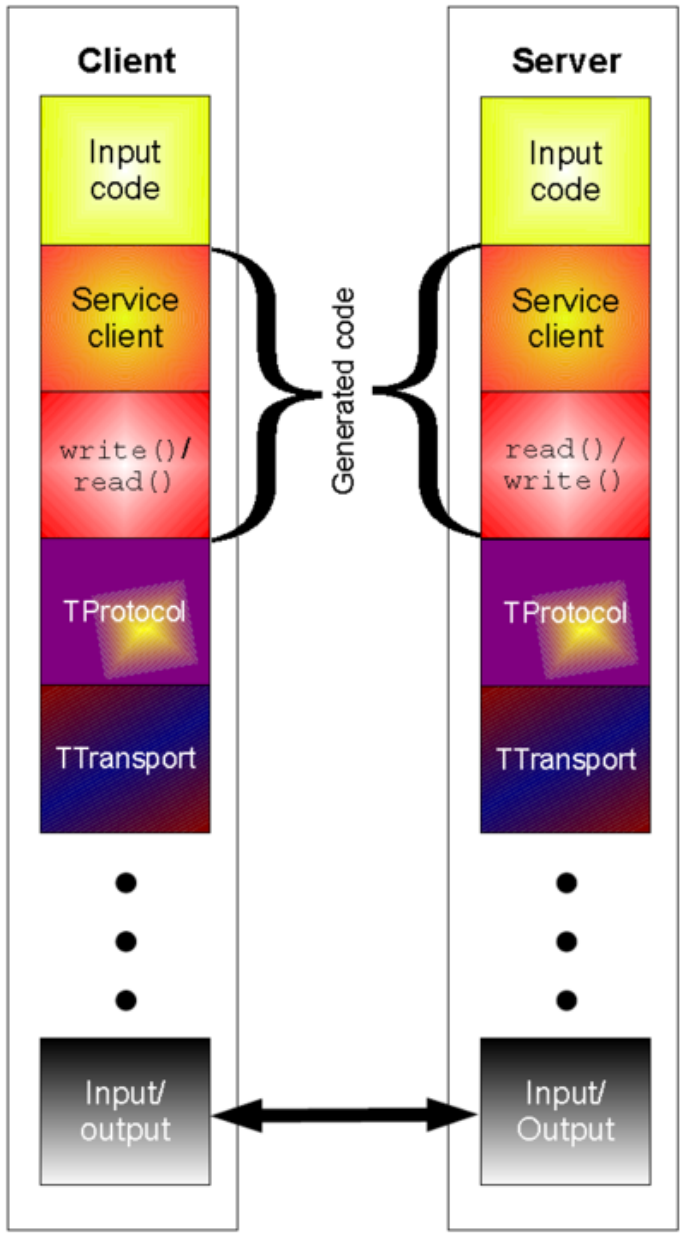
Apache Thrift

Thrift wurde von **Facebook** entwickelt, damit Systeme geschrieben in **verschiedenen** Sprachen **miteinander** kommunizieren konnten. Es wurde später unter Apache Thrift als quelloffenes Projekt **freigegeben**.

Ähnlich wie Avro **definiert** es dessen Schemas **sprachunabhängig** und kann viele Sprachen **unterstützen**. Zusätzlich definiert es den **ganzen** RPC Stack für Client/Server Kommunikation.

Thrift Architektur

Für Thrift muss zuerst Code erzeugt werden. Dieser wird dann an der entsprechenden Stelle in der gesamten Kommunikation aufgerufen. Viele Ebenen in Thrift sind in mehrfachen Varianten verfügbar und können je nach Anwendungsfall ausgewählt werden.



Thrift Architektur

Protokoll

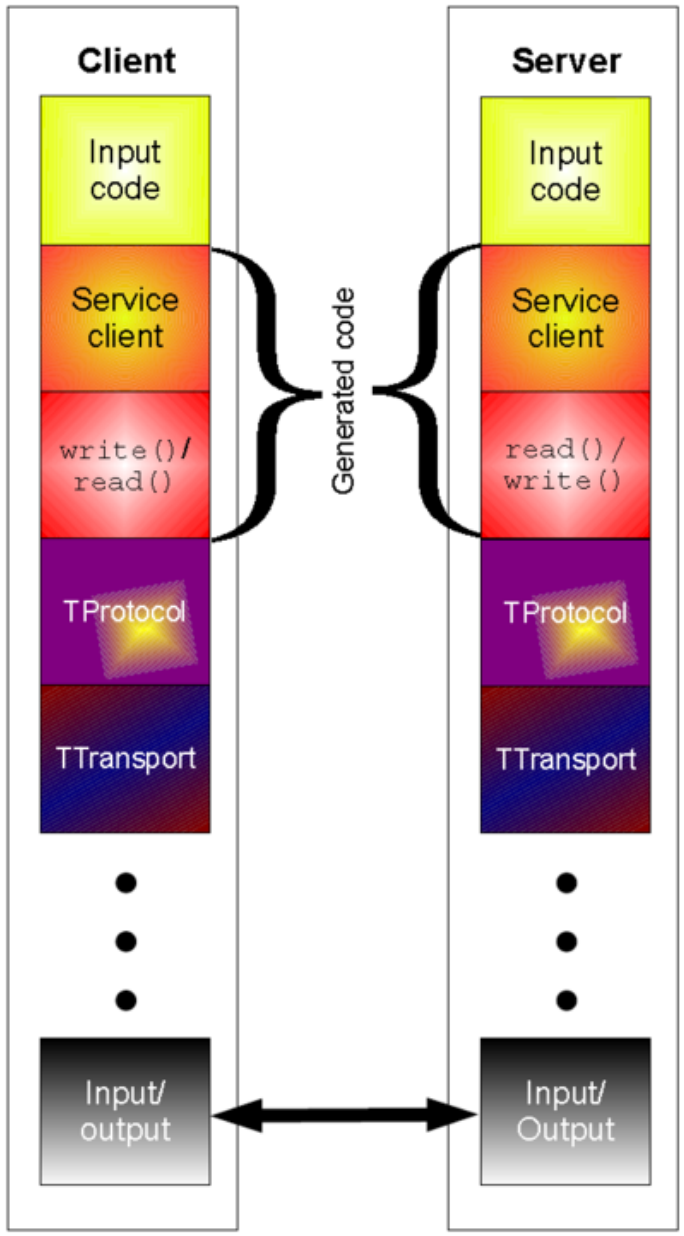
Definiert wie Daten serialisiert werden (binär, kompakt, JSON)

Transport

Definiert die eigentliche Verbindung (Netzwerk, Speicher, Datei)

Server

Fertige Klassen welche einen Serverdienst aufbauen



Thrift Schema

Im Gegensatz zu Avro, braucht Thrift (und auch Protocol Buffers) **mehr** Informationen in der Definition eines Schemas. Jedes Feld braucht eine **eindeutige** ID, welche sich **nicht** ändern darf! Neue Felder bekommen **neue** IDs, **fehlende** Felder, d. h. IDs, werden **ignoriert**.

Zusätzlich definiert Thrift **Dienste**, welche für den Aufbau der mitgelieferten Server Klassen dienen.

Thrift Schema

Schema Definition:

```
struct UserProfile {
  1: i32 uid,
  2: string name,
  3: string blurb
}
service UserStorage {
  void store(1: UserProfile user),
    UserProfile retrieve(1: i32 uid)
}
```

Thrift Client – Python Beispiel

```
# Make an object
up = UserProfile(uid=1, name="Test User",
  blurb="Thrift is great")

# Talk to server via TCP sockets, using a binary protocol
transport = TSocket.TSocket("localhost", 9090)
transport.open()
protocol = TBinaryProtocol.TBinaryProtocol(transport)

# Use service already defined
service = UserStorage.Client(protocol)
service.store(up)

# Retrieve something
up2 = service.retrieve(2)
```

Thrift Schema

Im Gegensatz zu Avro, braucht Thrift (und auch Protocol Buffers) **mehr** Informationen in der Definition eines Schemas. Jedes Feld braucht eine **eindeutige** ID, welche sich **nicht** ändern darf! Neue Felder bekommen **neue** IDs, **fehlende** Felder, d. h. IDs, werden **ignoriert**.

Zusätzlich definiert Thrift **Dienste**, welche für den Aufbau der mitgelieferten Server Klassen dienen.

Serialisierungsformate

Alle der ange- und besprochenen Systeme sind sich sehr **ähnlich**, unterscheiden sich aber in der **Fülle** der besprochenen Merkmale, also wie kompakt, schnell, erweiterbar und interaktiv sie sind. Mal sind die Protokolle ein wenig **effizienter**[1], mal gibt es **mehr** Sprachauswahl und so weiter.

Alleine das letzte Merkmal ist schon eine **lange** Liste:

[1]Siehe: https://code.google.com/p/thrift-protobuf-compare/

Unterstützte Sprachen

Sprache	Avro	Thrift	ProtoBuf
C	Ja	Ja	Ja
C++	Ja	Ja	Ja
C#	Ja	Ja	Ja
Java	Ja	Ja	Ja
PHP	Ja	Ja	Ja
Python	Ja	Ja	Ja
Ruby	Ja	Ja	Ja
Cocoa	Nein	Ja	Nein
D	Nein	Ja	Ja
Delphi	Nein	Ja	Nein
Erlang	Nein	Ja	Ja

Unterstützte Sprachen

Sprache	Avro	Thrift	ProtoBuf
Haskell	Nein	Ja	Ja
OCaml	Nein	Ja	Ja
Perl	Nein	Ja	Ja
Smalltalk	Nein	Ja	Nein
Action Script	Nein	Nein	Ja
Clojure	Nein	Nein	Ja
Common Lisp	Nein	Nein	Ja
Dart	Nein	Nein	Ja
Go	Nein	Nein	Ja
Haxe	Nein	Nein	Ja
JavaScript	Nein	Nein	Ja

Unterstützte Sprachen

Sprache	Avro	Thrift	ProtoBuf
Lua	Nein	Nein	Ja
Matlab	Nein	Nein	Ja
Mercury	Nein	Nein	Ja
Objective C	Nein	Nein	Ja
R	Nein	Nein	Ja
Scala	Nein	Nein	Ja
Vala	Nein	Nein	Ja
Visual Basic	Nein	Nein	Ja

Serialisierungsformate

Im Rahmen von Hadoop ist Avro am **besten** unterstützt. Aber **auch** Thrift und **andere** stehen zur Verfügung über **externe** Quellen. Deswegen muss man auch hier sich wieder genau **informieren** und das Passende **auswählen**.

Protocol Buffer wird zum Beispiel in Hadoop **intern** eingesetzt (weil es vor Avro zur Verfügung stand) für alle RPC Aufrufe. Und auch Thrift ist auch in Bereichen **verfügbar**.

Serialisierungsformate

Wenn wir uns in den Bereich der **dedizierten** Serialisierungsformate zurückbegeben, dann steht neben dem **generischen** Hadoop SequenceFile (und anderen mitgelieferten) aber auch **weitere**, auf Anwendungsfälle **optimierte** Dateiformate zur Verfügung.

Im folgenden schauen wir uns **Parquet** an, welches auf **analytische** Anwendungen **optimiert** wurde.

Parquet

Für **analytische** Aufgaben ist es meistens **besser** die Daten so abzulegen, dass sie beim Lesen möglichst **optimal** gelesen werden können. Dies ist **gleichbedeutend** mit „so wenig I/O wie möglich“.

Parquet ist eine quelloffenes Dateiformat welches **genau** dieses Ziel verfolgt, d. h. ein **effizientes** Layout der Daten für analytische Abfragen **bereitzustellen**.

Beispiel: Twitter

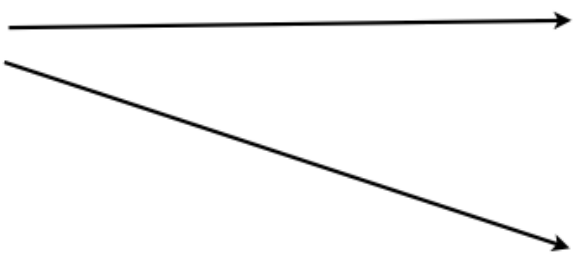
- Twitter's **Daten**
  - **230M+** monatlich aktive Nutzer, welche mehr als **500M** Tweets pro Tag generieren und konsumieren
  - **100TB+** pro Tag an komprimierten Daten
- **Analytische** Infrastruktur
  - Mehrere **1K+** Knoten Hadoop Clusters
  - Log Collection Pipeline
  - Verarbeitungswerkzeuge

Beispiel: Twitter

Twitter's Anwendungsfall

- Logs werden in HDFS abgespeichert
- Thrift wird zum Speichern der Logs benutzt
- Beispielschema: 87 Spalten, bis zu 7 Stufen tief

```
struct LogEvent {
  1: optional logbase.LogBase log_base
  2: optional i64 event_value
  3: optional string context
  4: optional string referring_event
...
  18: optional EventNamespace event_namespace
  19: optional list<Item> items
  20: optional map<AssociationType,Association> associations
  21: optional MobileDetails mobile_details
  22: optional WidgetDetails widget_details
  23: optional map<ExternalService,string> external_ids
}
```



```
struct LogBase {
  1: string transaction_id,
  2: string ip_address,
  ...
  15: optional string country,
  16: optional string pid,
}
```

Beispiel: Twitter

Goal:

> *"To have a state of the art columnar storage available across the Hadoop platform"*

- Hadoop ist sehr verlässlich für große, lang-laufende Abfragen, aber auch I/O lastig
- Sukzessive Umstellung auf die Vorteile der spaltenorientierten Speicherung
- Nicht an ein bestimmtes Programmiergerüst gebunden

Spaltenorientierte Speicherung

Die spaltenorientierte Speicherung **begrenzt** I/O auf die **wirklich** benötigten Daten, denn sie liest **nur** die Spalten die benötigt werden.

Zusätzlich **spart** sie Platz, den Daten im Spalten Layout können **besser** komprimiert und durch typspezifische Enkodierung **weiter** verdichtet werden.

Außerdem ermöglicht sie die Ausführung auf Vektormaschinen (vgl. Großrechner).

Spalten- vs. Zeilenorientierung

Hier ein Beispiel in dem ein logisches Tabellen-schema in eine physikalische Speicherung abgebildet wird.

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

In dem zeilenbasierten Layout folgt Zeile auf Zeile:

Im Gegensatz dazu speichert das spaltenorientierte Layout eine Spalte nach der anderen:

Parquet Übersicht

Parquet definiert ein allgemeines Datei Format, welches

- **sprachunabhängig** und
- **formal spezifiziert** ist.

Implementierungen existieren in Java für MapReduce und in C++, welches von Impala genutzt wird.

Parquet Details

- Der **Algorithmus** ist von Google Dremel‘s **ColumnIO** ausgeborgt
- Das Schema wird in einem **bekannten** Format definiert
- Unterstützt **verschachtelte** Strukturen
- Jede Zelle ist als Triplet **enkodiert**: Wiederholungsebene, Definitionsebene und Wert
- Die Werte für die Ebenen sind **begrenzt** durch die Tiefe des aktuellen Schemas
  - Speicherung in sehr **kompakter** Form

Parquet Details

- Das Schema **ähnelt** Protocol Buffers, **aber** mit Vereinfachungen (z. B. **keine** Maps, Lists oder Sets)
  - Diese komplexen Typen **können** durch die anderen Merkmale ausgedrückt werden
- Die **Wurzel** des Schemas ist eine **Gruppe** von **Feldern** welche **Nachricht** genannt wird
- Feldtypen sind entweder „**Gruppe**“ oder „**primitiver Typ**“ mit Wiederholung als „benötigt“, „optional“ oder „wiederholt“
  - Genau Eins, Keins oder Eins, oder Keins, Eins oder Mehr

Beispiel Schema

```
message AddressBook {
   required string owner;
   repeated string ownerPhoneNumbers;
   repeated group contacts {
      required string name;
      optional string phoneNumber;
   }
}
```

Lists/Sets Umsetzen

Das Fehlen der Lists/Sets kann mit Hilfe von Weiderholungen umgangen werden:

Schema: List of Strings	**Data**: ["a", "b", "c", ...]

message ExampleList {
repeated string list;
}
	{
	list: "a",
	list: "b",
	list: "c",
	…
	}

Maps Umsetzen

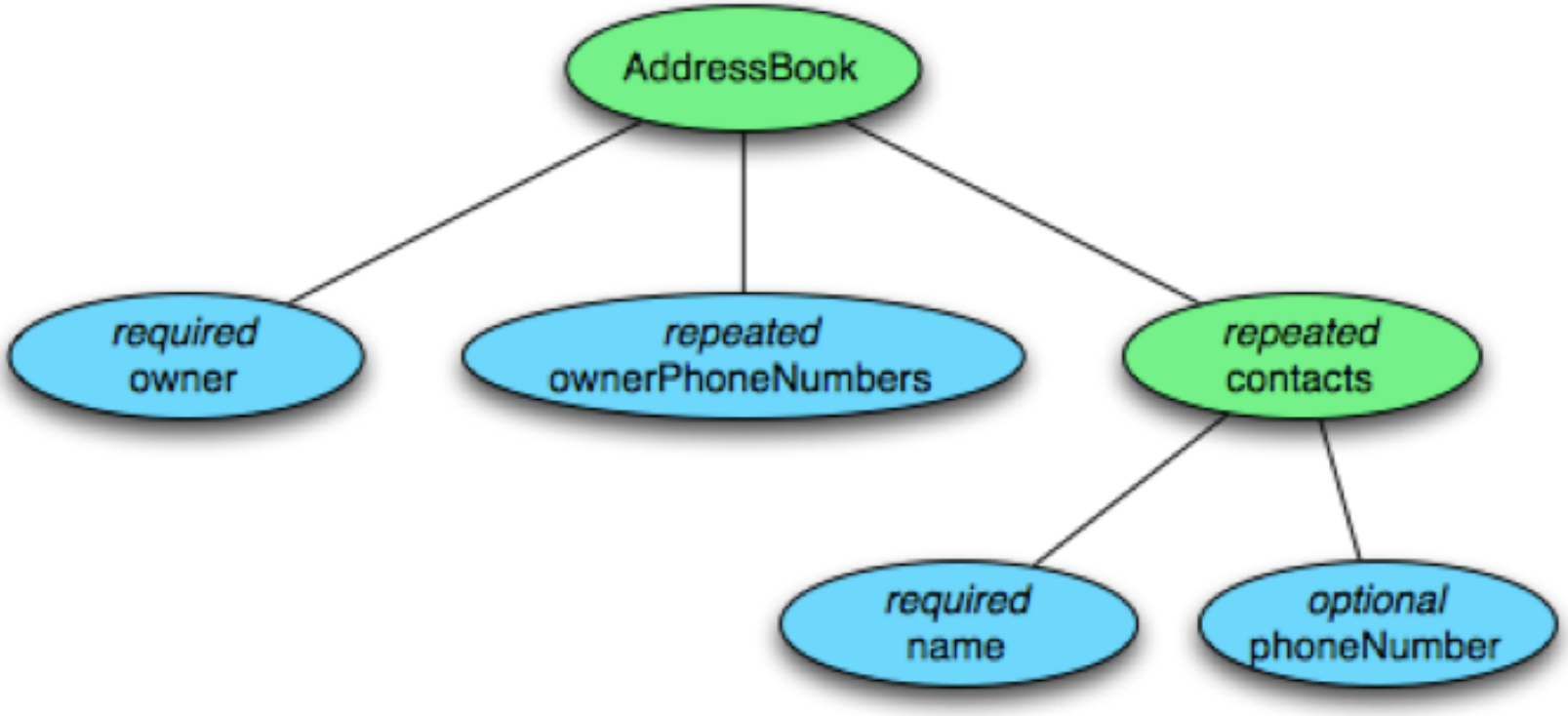
Ähnlich geht das für Maps:

Schema: Map of strings to strings	**Data**: `{"AL" => "Alabama", ...}`

`message ExampleMap {`
`  repeated group map {`
`  required string key;`
`  optional string value;`
`  }`
`}`
	`{`
	`  map: {`
	`  key: "AL",`
	`  value: "Alabama"`
	`  },`
	`  map: {`
	`  key: "AK",`
	`  value: "Alaska"`
	`  },`
	`  …`
	`}`

Schema als Baum

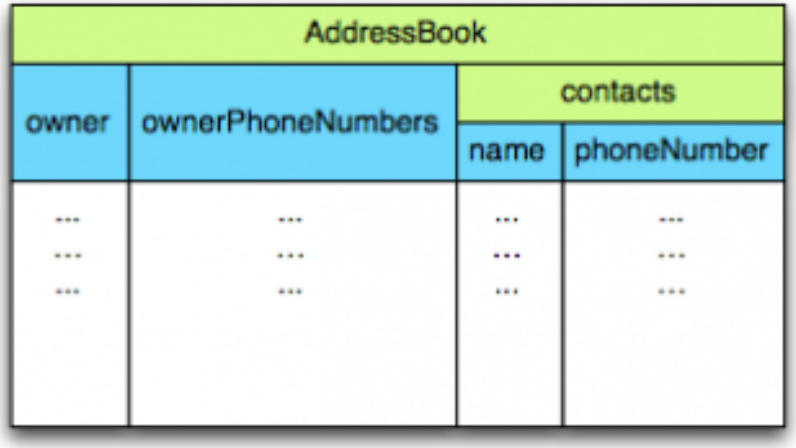
Das gleiche Schema als Baumstruktur ausgedrückt. Blau sind echte Datenfelder.



Feld pro Primitiver Wert

Primitive Felder werden in Spalten des spaltenorientierten Formats umgesetzt (wieder in Blau dargestellt):

Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string



Stufen(Levels)

Die **Struktur** des Datensatzes wird für jeden Wert mit Hilfe von **zwei** Integer festgehalten, welche **Wiederholungs**- und **Definitionsstufen** genannt werden.

Mit diesen beiden Stufen können **verschachtelte** Strukturen **komplett** rekonstruiert werden, während die einfachen Werte trotzdem **separat** gespeichert werden können.

Definitionsstufe

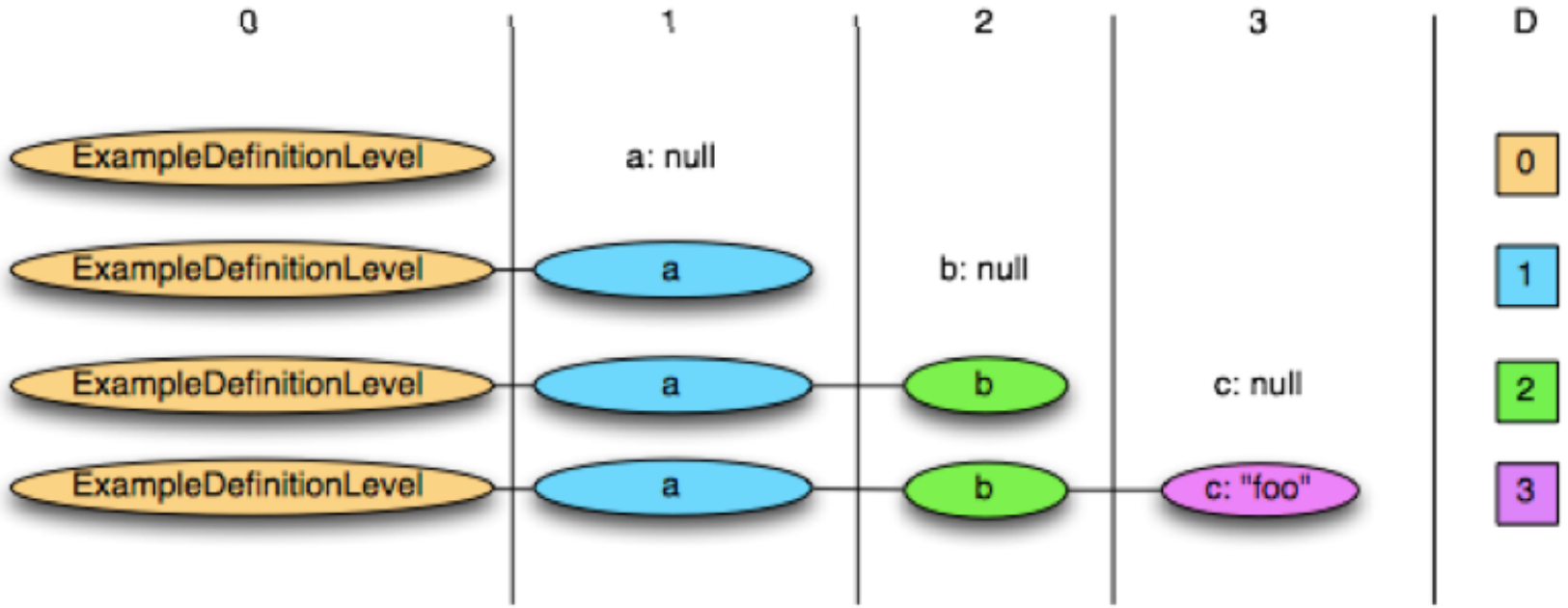
Beispiel:

```
message ExampleDefinitionLevel {
   optional group a {
      optional group b {
         optional string c;
      }
   }
}
```

Beinhaltet eine Spalte “a.b.c” dessen Felder alle optional sind und leer (null) sein können.

Definitionstufe

Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: "foo" } }`	3 (actually defined)



Definitionstufe

Beispiel mit benötigtem Feld:

```
message ExampleDefinitionLevel {
   optional group a {
      required group b {
         optional string c;
      }
   }
}
```

Value	Definition Level
`a: null`	0
`a: { b: null }`	Impossible, as `b` is required
`a: { b: { c: null } }`	1
`a: { b: { c: "foo" } }`	2 (actually defined)

Wiederholungsstufe

Wiederholte Felder **benötigen**, dass gespeichert wird, **wo** eine Liste in einer Spalte an Werten beginnt, denn diese sind ja ansonsten **sequentiell** hintereinander abgelegt.

Die Wiederholungsstufe gibt an **pro** Wert wo eine **neue** Liste anfängt und sind **prinzipiell** Markierungen, welche **gleichzeitig** die Stufe an der die neue Liste anfängt angibt.

Nur Stufen welche **wiederholt** werden brauchen eine Wiederholungsstufe, d. h. optionale oder benötigte Felder werden **nicht** wiederholt und können deshalb „**ignoriert**“ werden.

Wiederholungsstufen

Schema:

```
message nestedLists {
   repeated group level1 {
      repeated string level2;
   }
}
```

Data: `[[a,b,c],[d,e,f,g]],[[h],[i,j]]`

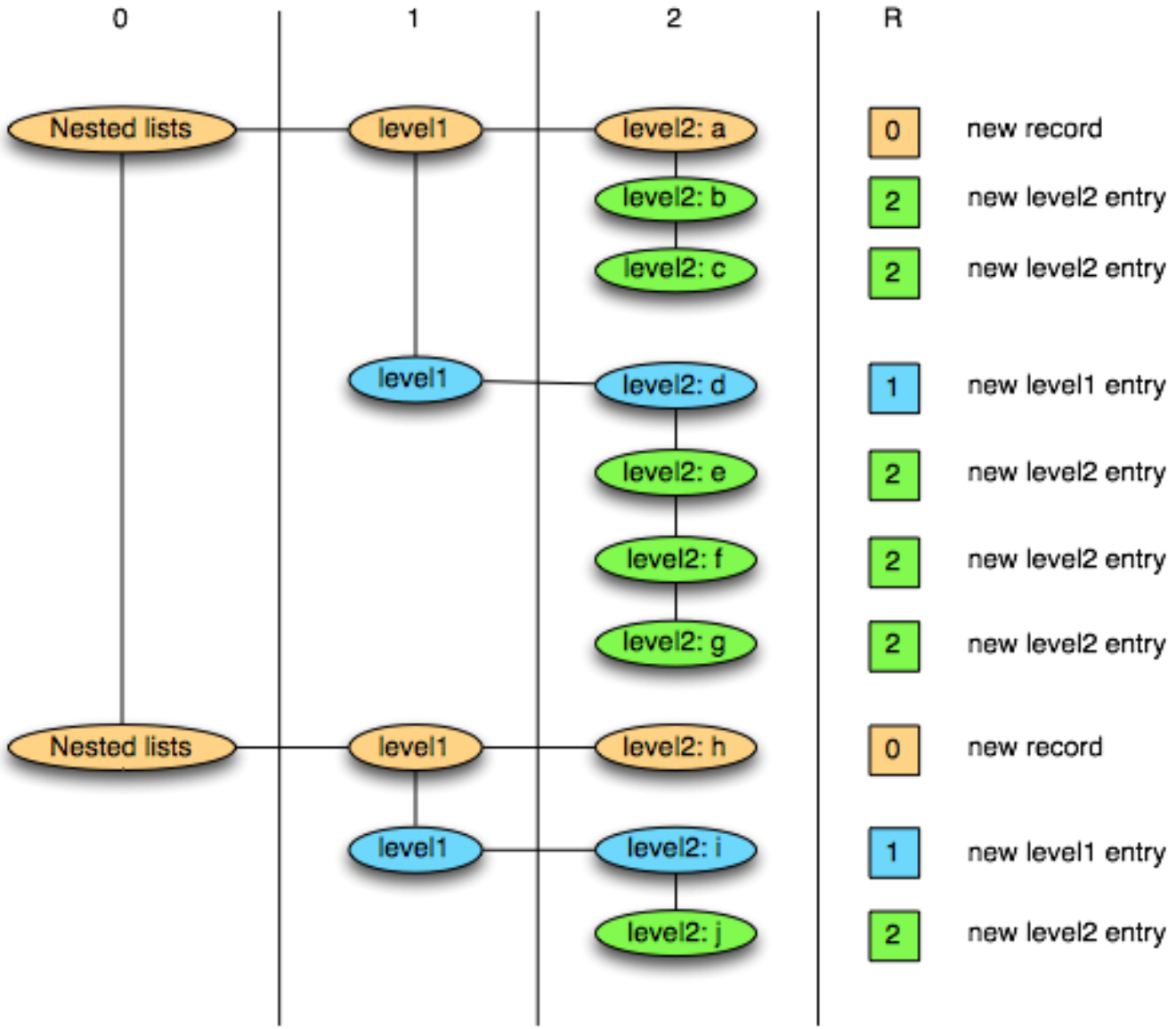
```
{
      level1: {
            level2: a
            level2: b
            level2: c
      },
      level1: {
            level2: d
            level2: e
            level2: f
            level2: g
      }
}
{
      level1: {
            level2: h
      },
      level1: {
            level2: i
            level2: j
      }
}
```

Wiederholungsstufen

Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

- 0 markiert jeden neuen **Datensatz** und bedeutet auch das Erstellen einer **Liste** für Stufe 1 und 2
- 1 markiert eine neue Stufe 1 **Liste** und bedeutet auch das Erstellen einer **Liste** für Stufe 2
- 2 markiert jedes neue **Element** in einer Stufe 2 Liste

Wiederholungsstufen



Kombination der Stufen

Beide Stufen angewendet auf das AddressBook Beispiel:

Column	Max Definition level	Max Repetition level
`owner`	0 (`owner` is *required*)	0 (no repetition)
`ownerPhoneNumbers`	1	1 (*repeated*)
`contacts.name`	1 (`name` is *required*)	1 (`contacts` is *repeated*)
`contacts.phoneNumber`	2 (`phoneNumber` is *optional*)	1 (`contacts` is *repeated*)

Speziell für die Spalten “contacts.phoneNumber” hat eine definierte Telefonnummer eine maximale Definitionsstufe von 2 und ein Eintrag ohne Telefonnummer hat eine Stufe von 1. Wenn keine Eintrag vorliegt ist sie 0.

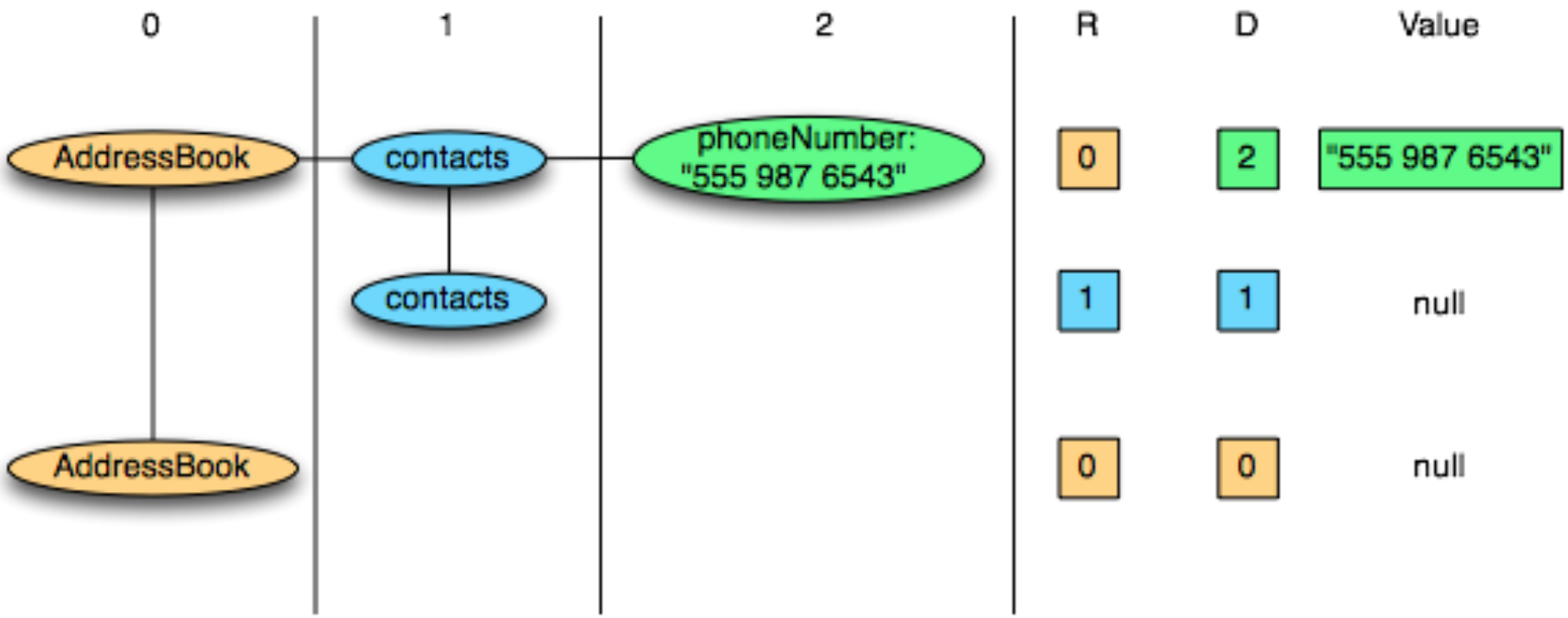
Beispiel: AddressBook

```
AddressBook {
  owner: "Julien Le Dem",
  ownerPhoneNumbers: "555 123 4567",
  ownerPhoneNumbers: "555 666 1337",
  contacts: {
    name: "Dmitriy Ryaboy",
    phoneNumber: "555 987 6543",
  },
  contacts: {
    name: "Chris Aniszczyk"
  }
}
AddressBook {
  owner: "A. Nonymous"
}
```

Beispiel: AddressBook

```
AddressBook {
   contacts: {
     phoneNumber: "555 987 6543"
   }
   contacts: {
   }
}
AddressBook {
}
```

Beispiel: AddressBook



Beispiel: AddressBook

Beim Schreiben:

- contacts.phoneNumber: “555 987 6543”
  - Neuer Datensatz: R = 0
  - Wert definiert: D = max (2)
- contacts.phoneNumber: NULL
  - Wiederholter Kontakt: R = 1
  - Nur bis contacts definiert: D = 1
- contacts: NULL
  - Neuer Datensatz: R = 0
  - Nur definiert bis AddressBook: D = 0

R	D	Value
0	2	"555 987 6543"
1	1	NULL
0	0	NULL

Beispiel: AddressBook

Beim Lesen

- **R=0, D=2, Value = “555 987 6543”:**
  - R = 0 bedeutet neuer Datensatz. Erstellt alle verschachtelten Datensätze von der Wurzel bis zur Definitionsstufe (hier 2)
  - D = 2 und damit das Maximum. Der Wer ist definiert und wird eingefügt.
- **R=1, D=1:**
  - R = 1 bedeutet neuer Eintrag in der contacts Liste auf Stufe 1.
  - D = 1 bedeutet contacts ist definiert, aber nicht phoneNumber, deshalb wird nur ein leerer contacts erstellt.
- **R=0, D=0:**
  - R = 0 bedeutet neuer Datensatz. Erstellt alle verschachtelten Datensätze von der Wurzel bis zur Definitionsstufe.
  - D = 0 => contacts ist nicht gesetzt, deswegen bleibt ein leeres AddressBook übrig.

Beispiel: AddressBook

```
AddressBook {                                    Record 1
   contacts: {
     phoneNumber: "555 987 6543"
   }                                             Subrecord 1
   contacts: {
   }                                             Subrecord 2
}
```

```
AddressBook {                                    Record 2
}
```

Stufen Speicherung

Jeder primitiver Typ besteht aus **drei** Spalten, dennoch ist der Aufwand relativ **gering**. Das liegt an der Begrenzung der Werte durch die Tiefe des Schemas. Es werden nur **wenige** Bits wirklich benötigt.

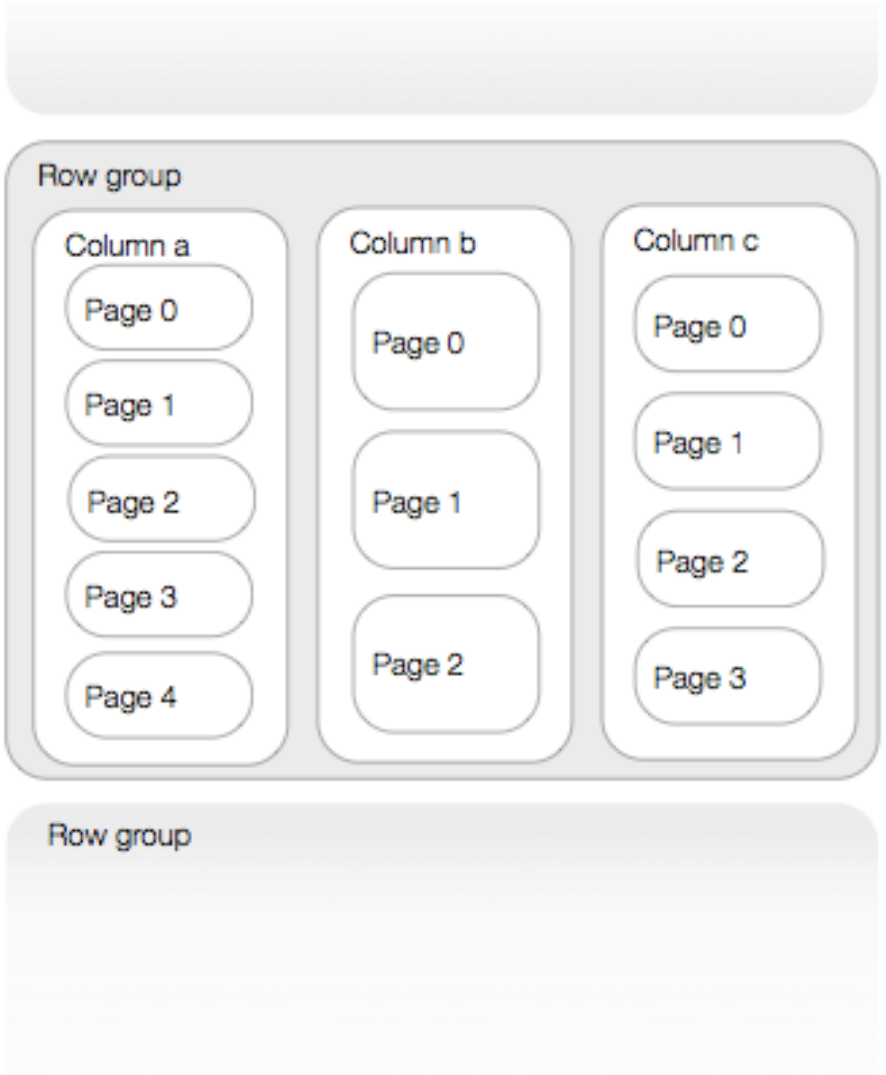
Wenn alle Felder in einem **flachen** Schema als **benötigt** definiert sind, dann können alle Stufen Informationen **weggelassen** werden, denn sie sind alle **null**.

Ansonsten kümmert sich die **Komprimierung** (z. B. RLE) darum die Daten **effizient** zu verdichten.`

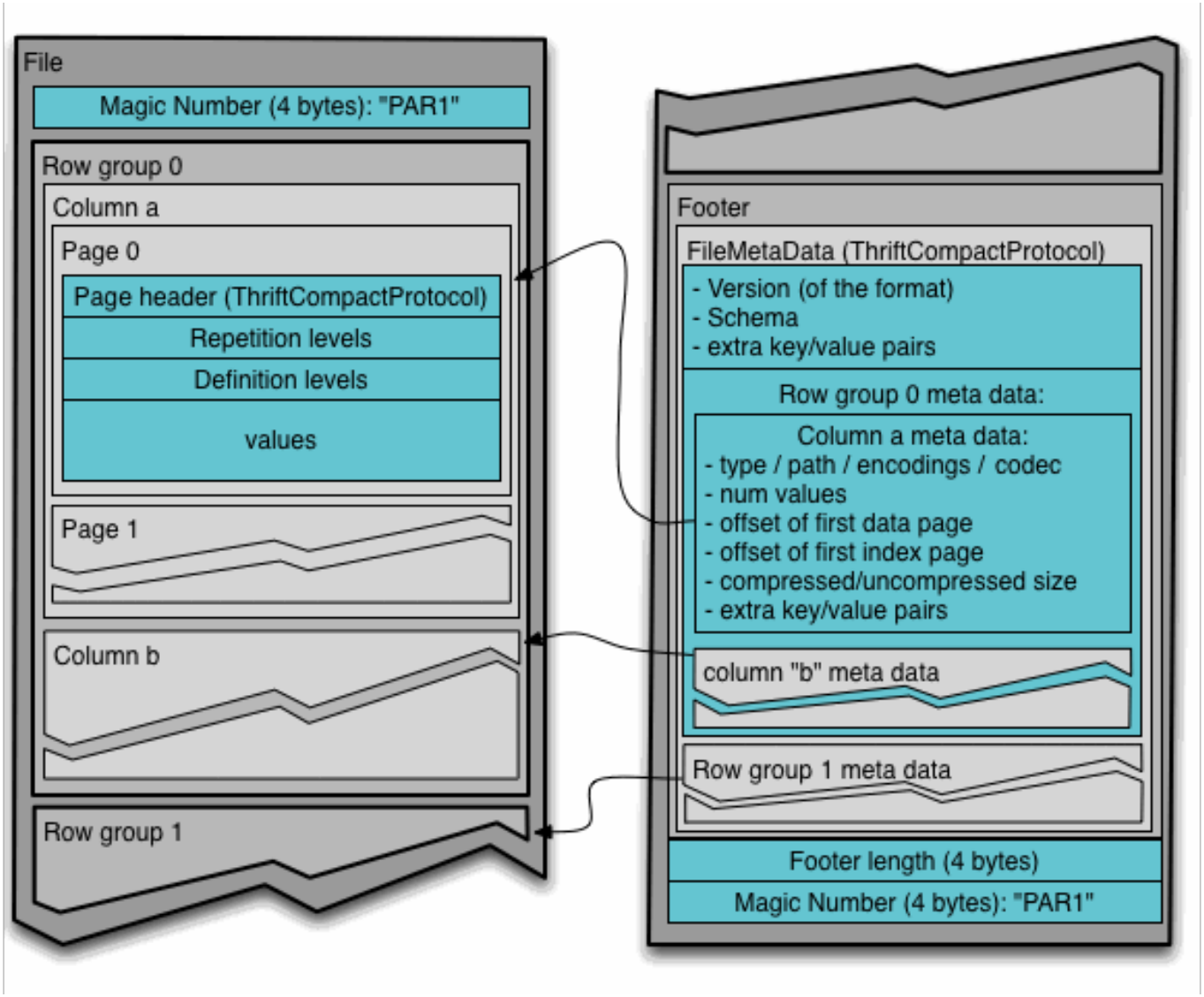
Dateiformat

- **Zeilengruppen** (Row Group):Eine Gruppe an Zeilen in Spaltenformat
  - Maximale Größe wird im Speicher gehalten während des Schreibens
  - Eine (oder mehr) pro Split beim Lesen
  - Ungefähr 50MB < Zeilengruppe < 1GB
- **Spaltenabschnitt** (Chunks): Daten für eine Spalte in einer Zeilengruppe
  - Spaltenabschnitte können unabhängig gelesen werden für effizientes durchlaufen
- **Seite** (Page): Einheit des Zugriffs in einem Spaltenabschnitt
  - Sollte gross genug sein für eine gute Komprimierung
  - Minimale Größe des Lesens eines einzelnen Datensatzes
  - Ungefähr 8KB < Seite < 1MB

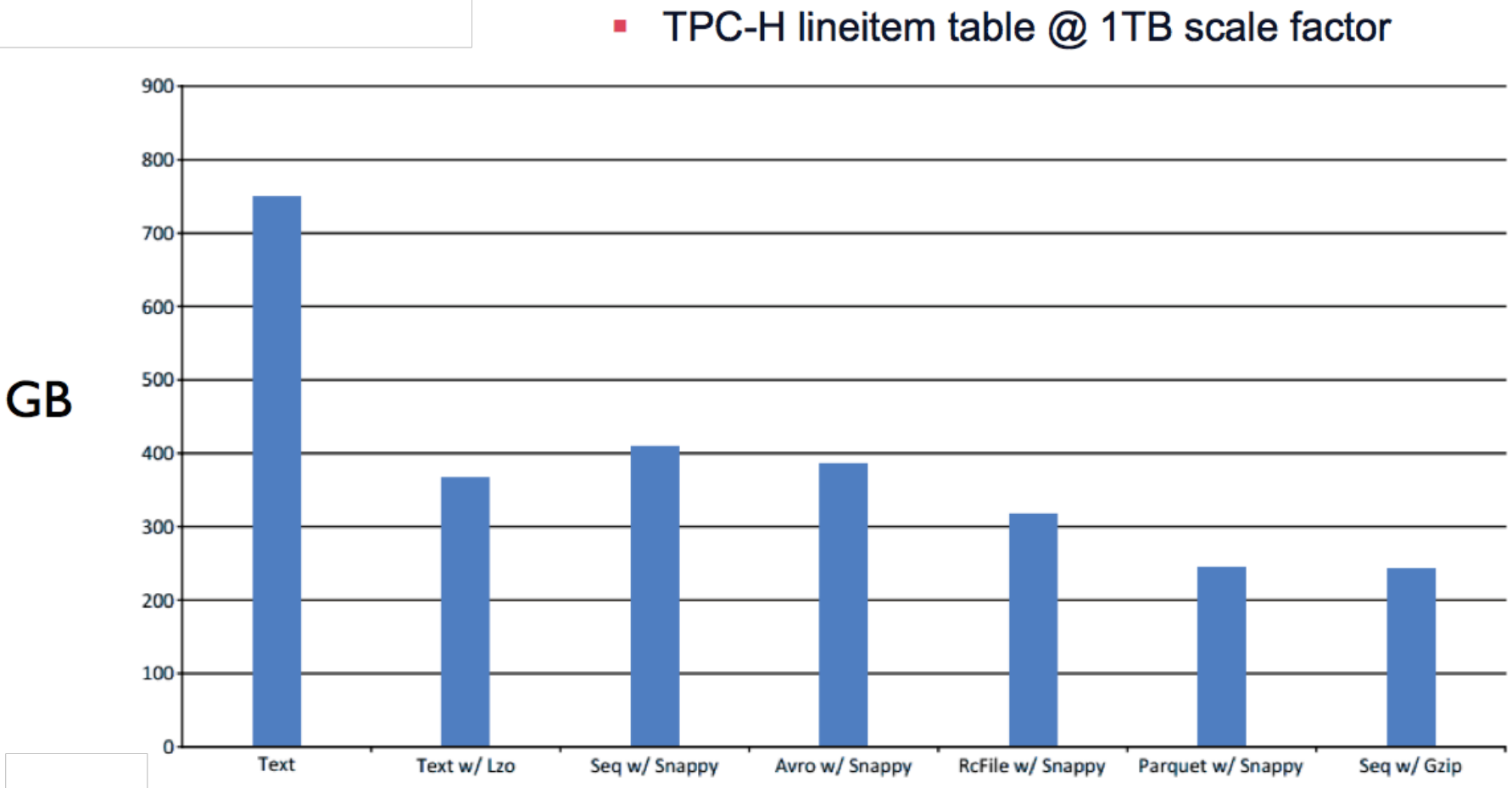
Dateiformat



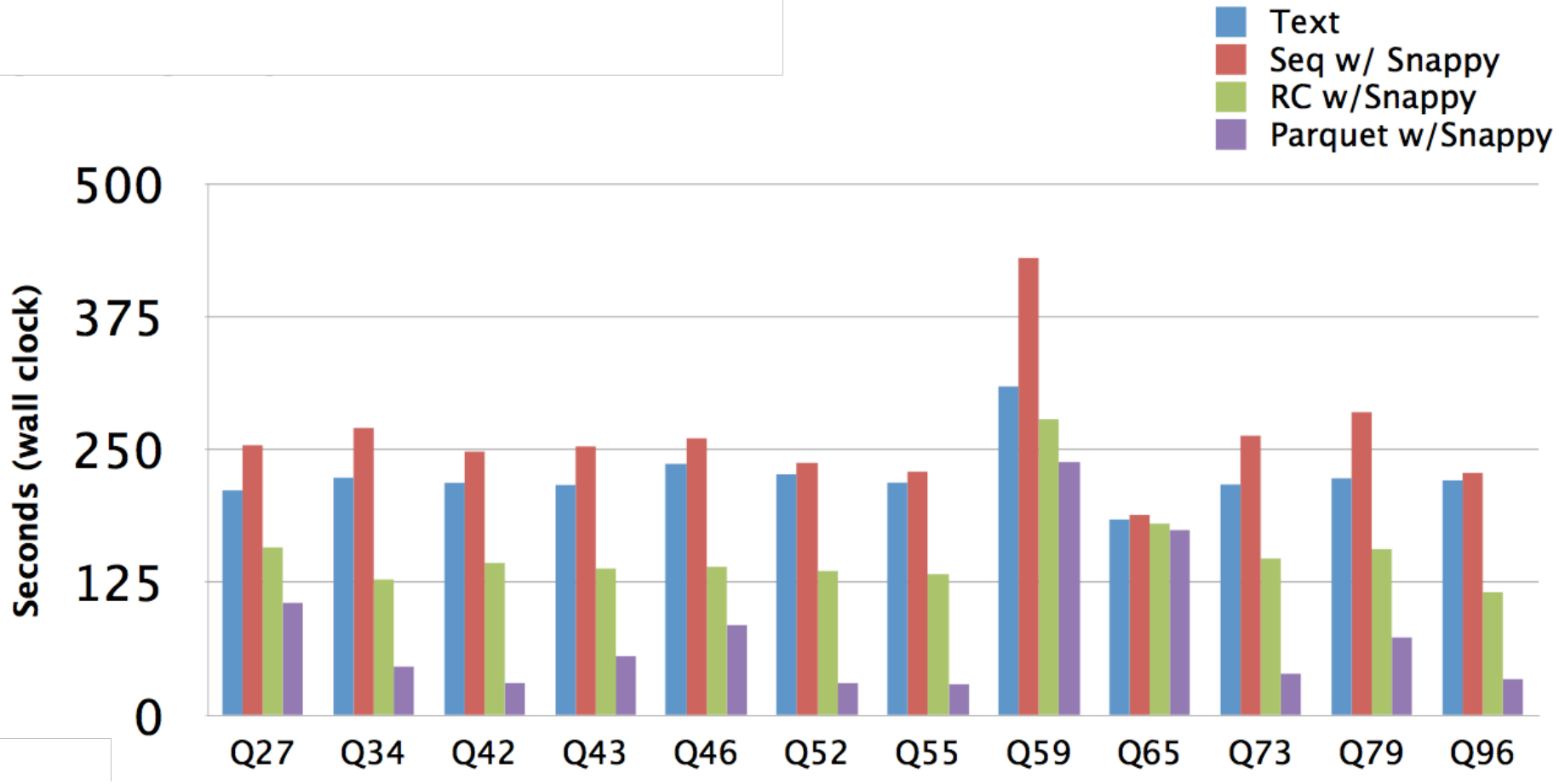
Dateiformat



Beispiel: Impala Ergebnisse



Beispiel: Impala TPC-DS



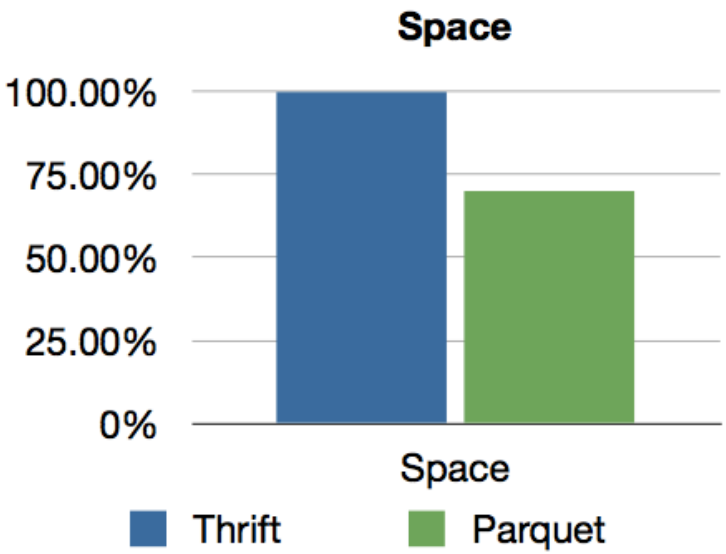
Performance

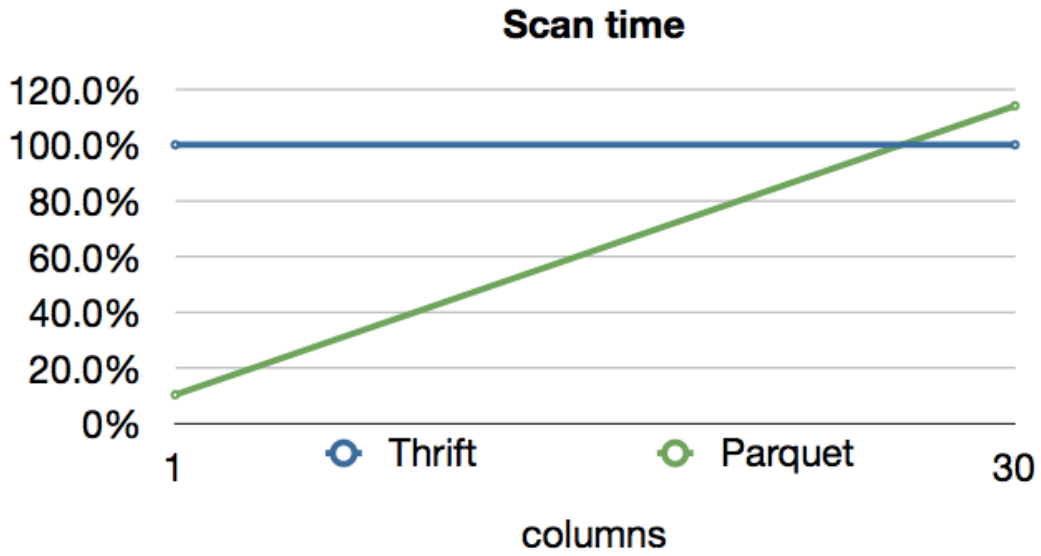
Twitter: production results

Data converted: similar to access logs. 30 columns.

Original format: Thrift binary in block compressed files (LZO)

New format: Parquet (LZO)





- **Space saving:** 30% using the same compression algorithm
- **Scan + assembly time compared to original:**
  - One column: 10%
  - All columns: 110%

Beispiel: Twitter

- Petabytes an Speicherplatz gespart
- Beispiel Job der „Projection push down“ ausnutzt:
  - Job 1 (Pig): Lesen von 32% weniger Daten -> 20% an Zeit gespart
  - Job 2 (Scalding): Lesen von 14 aus insgesamt 35 Spalten, liest 80% weniger Daten -> 66% Zeit gespart
  - Terabytes beim Lesen jeden Tag gespart

Einheit 4

- Rückblick auf Einheit 3
- Dateiformate und Serialisierung
- **Zugriff auf Daten: Abfrageschnittstellen**
- Suche in Daten

Schnittstellen

Es gibt einige Wege auf Daten zuzugreifen. Diese fallen innerhalb von Hadoop in zwei Kategorien:

Batch Zugriff

Hier werden direkt MapReduce oder aber oberhalb liegende Vereinfachungen wie Pig oder Hive eingesetzt.

Echtzeit Zugriff

Die Daten werden direkt aus HDFS gelesen. Dies macht Impala oder Solr.

Pig

Pig wurde innerhalb von Yahoo! entwickelt, um MapReduce einer **größeren** Menge an Anwendern **zugänglich** zu machen.

Es definiert Abfragen in **Pig Latin**, einer **imperativen** Sprache, welche anderen Programmiersprachen ähnelt.

Pig ist **erweiterbar** durch User-defined Functions (**UDFs**) und bringt ein **Shell** mit, welche **Grunt** heißt.

Pig

Es gibt zwei Ausführungsarten, lokal und verteilt:

```
/* local mode */
$ pig -x local ...

/* mapreduce mode */
$ pig ...
```

oder

```
$ pig -x mapreduce ...
```

Pig Beispiel

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt‘
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999
AND (quality == 0 OR quality == 1 OR quality == 4 OR
quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

Pig Ausführung

Bei der Ausführung des Skripts wird dieses in **einen** oder **mehrere** MapReduce Jobs übersetzt.

Diese führen dann die mitgelieferte Pig JAR Datei aus, welche Daten liest und schreibt.

Für den Entwickler ist Pig **einfach** zu lesen, denn es werden **Datenverarbeitungsschritte** angegeben, welche einfach zu deuten sind.

Ein Schema wird **erst** zur Laufzeit und **innerhalb** des Skripts definiert.

Pig Datenmodell

Es gibt **Relations**, **Bags** und **Tuples**, sowie einfache Datentypen (**Fields**). Relations sind Tabellen ähnlich, und enthalten Bags, welche wiederum Tuples enthalten. Dort sind die eigentlichen Felder enthalten. Also sind die Tuples eine **Zeile** in einer Tabelle.

Pig liefert Klassen mit um Daten aus Dateien in die obigen Konzepte zu überführen.

Hive

Facebook hatte ein **ähnliches** Problem wie Yahoo, d.h. es wollte MapReduce und die Daten **leichter** zugänglich machen. Es entwickelte Hive, welches **SQL** als Abfrageschnittstelle anbietet. Die Syntax ist an MySQL **angelehnt** und sollte sehr **einfach** zu verstehen sein.

HiveQL (Hive‘s SQL) hat **einige** Erweiterungen, aber auch Einschränkungen im Vergleich zu Standard SQL.

Impala

Während die vorhergenannten Systeme alle MapReduce Jobs **ausführen**, ist Impala mehr ein System mit **MPP** (massively parallel processing) Eigenschaften einer **verteilten** Datenbank.

Es unterstützt **HiveQL** und den Hive **Metastore** nativ und verhält sich von außen gesehen wie Hive, läuft aber **permanent** und fragt Daten **direkt** aus HDFS ab. MapReduce wird **nicht** benutzt und deswegen gibt es auch nicht dessen Latenzzeiten.

Impala

HiveQL ist eine **Untermenge** des **SQL92** Standards und **unterstützt** select, project, join, union, subqueries, aggregation, insert, order by (mit Einschränkungen).

Wie auch in SQL gibt es eine **DDL** und **DML**, also Definitions- und Abfragesprache.

Impala kann Daten aus **HDFS** und **HBase** lesen. Für HDFS z. B. Text und SequenceFile Dateien, sowie Avro und Parquet.

Impala Architektur

Zwei Prozesstypen: **impalad** und **statestored**

- Impala Daemon (impalad)
  - Nimmt Client **Anfragen** entgegen und **bearbeitet** alle **internen** Anfragen bezüglich der Abfrageausführung
  - Bietet Thrift **Dienste** an für diese Anfragen
- State Store Daemon (statestored)
  - Bietet Namensdienste und Metadaten Verteilung an
  - Bietet Thrift Dienst für diese an

Impala Architektur

- Abfrageausführungsphasen
  - Abfrage kommt über ODBC/Beeswax Thrift API an
  - Der Planer zerlegt die Anfrage in Fragmente
  - Der Koordinator führt die Abfrage auf den verteilten Prozessen aus
- Während der Abfrage
  - Zwischenergebnisse werden zwischen Prozessen ausgetauscht (streaming, im Speicher)
  - Abfrageergebnisse werden an den Aufrufer zurückgeschickt (streaming)
  - Dies kann aber durch bestimmte, blockierende Operatoren (top-n, aggregation) eingeschränkt sein

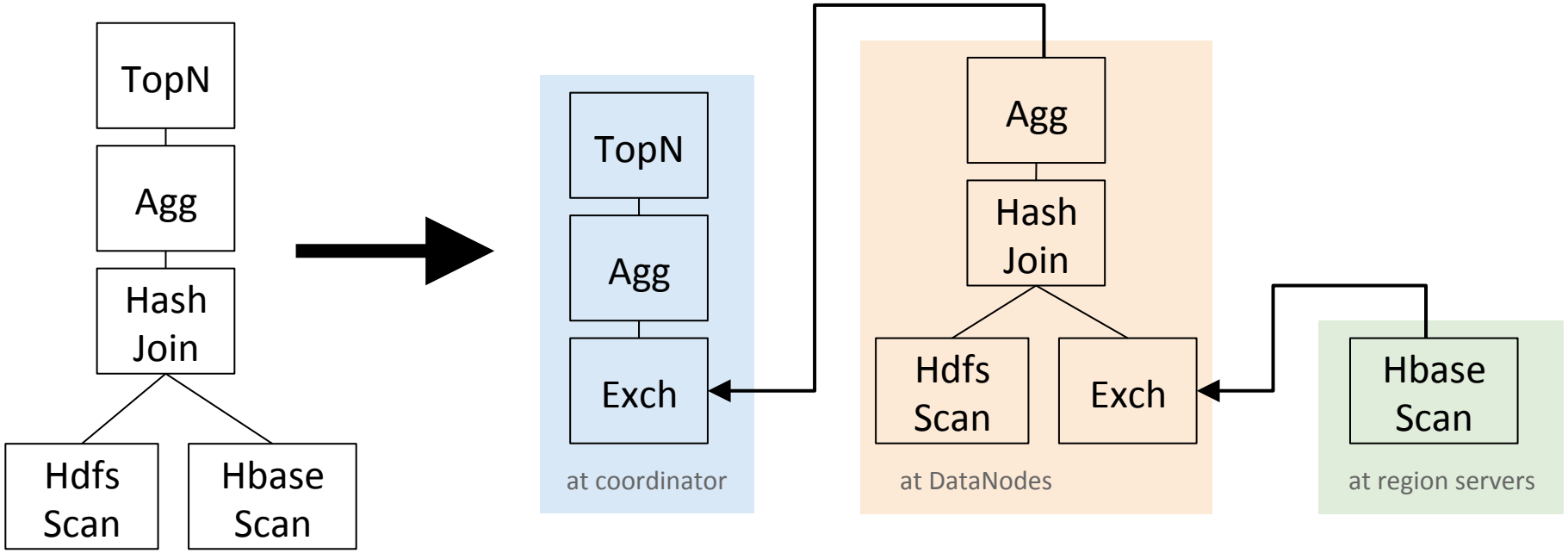
Impala Architektur: Planer

- 2-Phasen Planungsprozess:
  - Einzelknoten Plan: Left-deep Tree der Plan Operatoren
  - Plan Partitionierung: Partitioniere Einzelknoten Plan für maximale Lokalität und minimale Datenbewegung
- Plan Operatoren: Scan, HashJoin, HashAggregation, Union, TopN, Exchange
- Verteilte Aggregation: Voraggregation auf allen Knoten, mische Aggregation auf einem einzelnen Knoten
- Die JOIN Order entspricht der FROM Clause Order

Query Planner

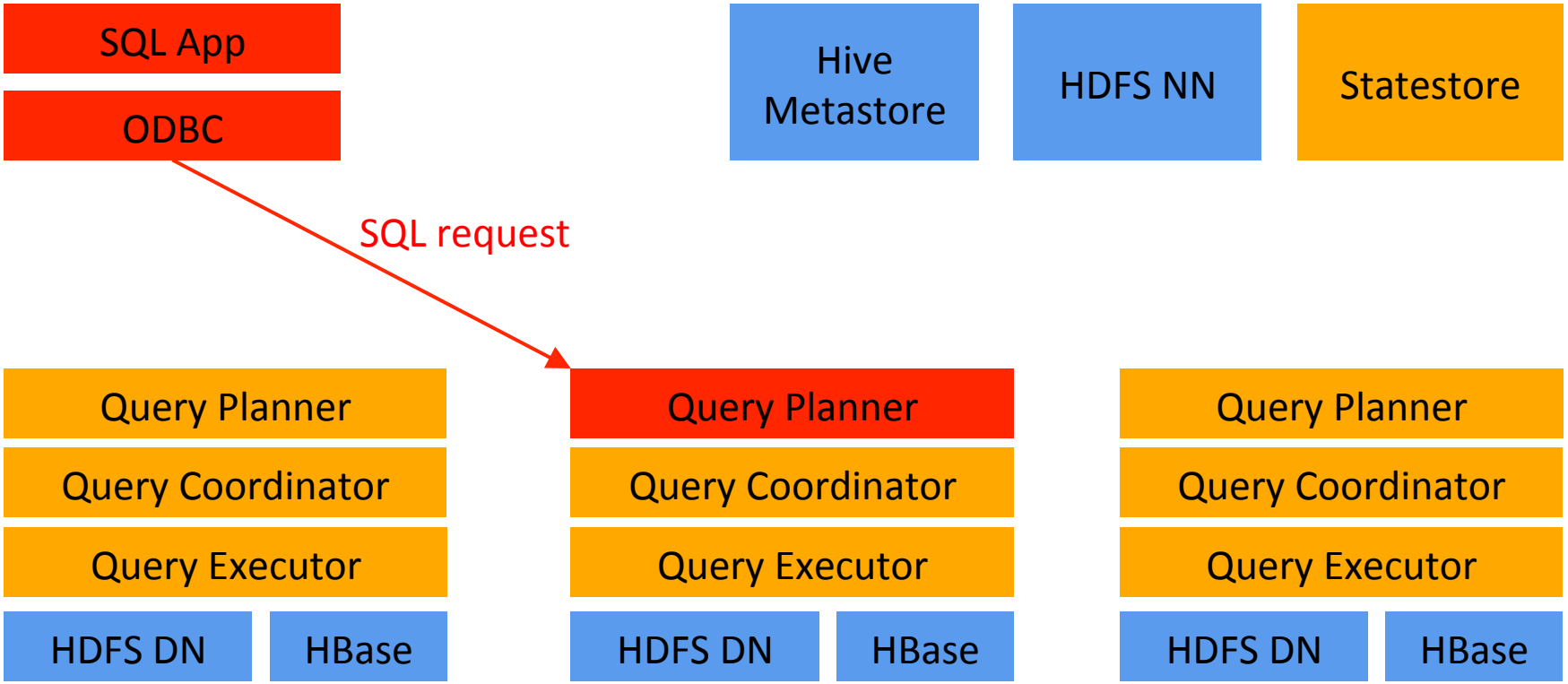
Beispiel: Abfrage mit JOIN und Aggregation

```
SELECT state, SUM(revenue)
FROM HdfsTbl h JOIN HbaseTbl b ON (...)
GROUP BY 1 ORDER BY 2 desc LIMIT 10
```



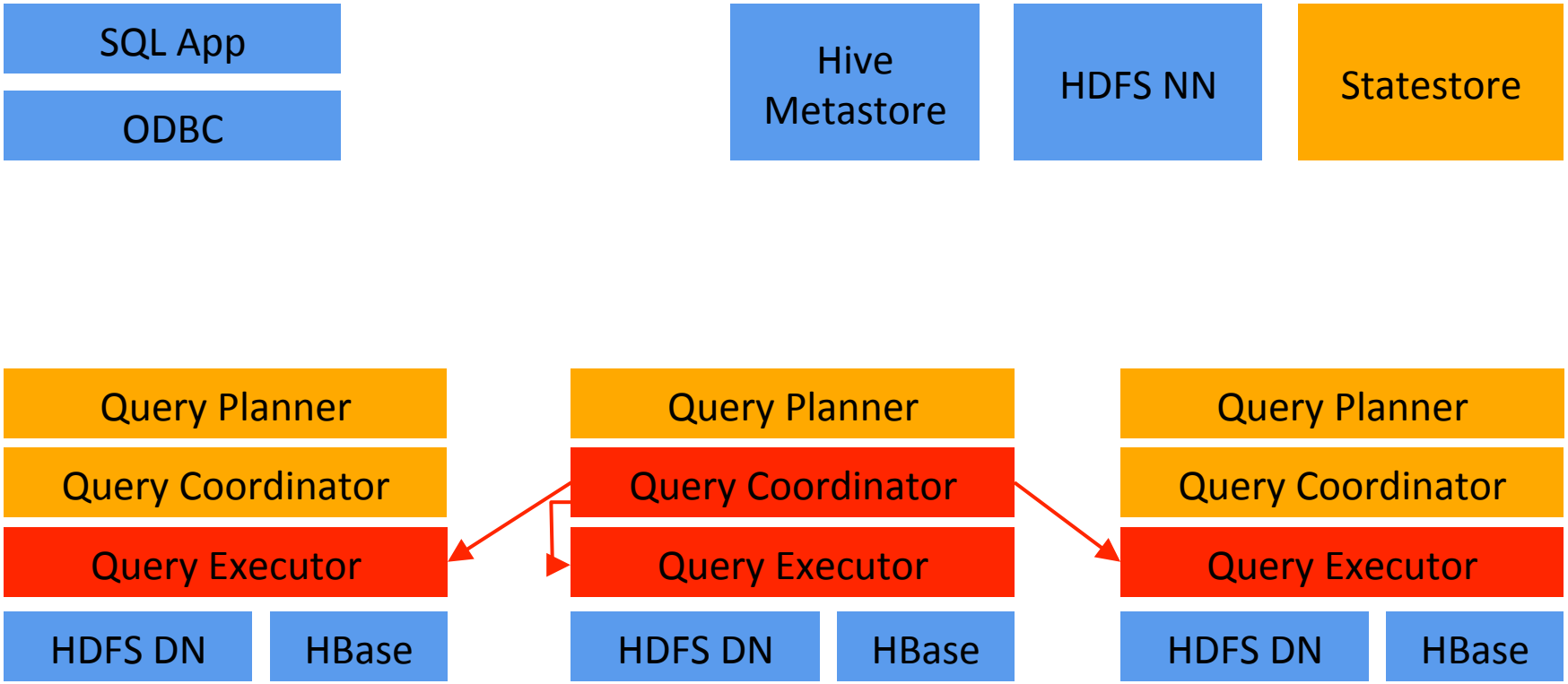
Abfrage Ausführung

Abfrage kommt über ODBC Thrift API



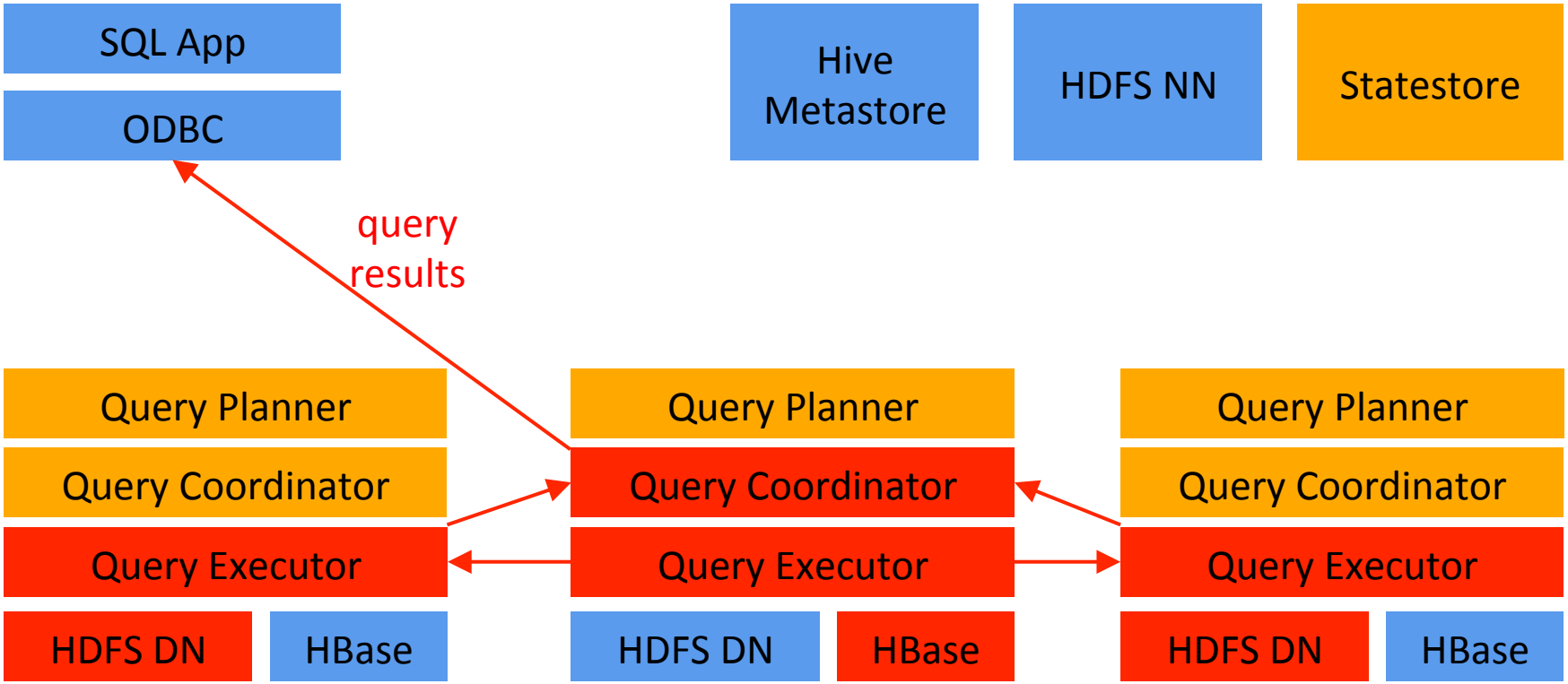
Abfrage Ausführung

Planner zerlegt Plan in Fragmente und der Coordinator führt diese auf den verteilten Prozessen aus



Abfrage Ausführung

Zwischenergebnisse werden unter Prozesses ausgetauscht (streaming) und das Gesamtergebnis an den Aufrufer zurückgeleitet



Einheit 4

- Rückblick auf Einheit 3
- Dateiformate und Serialisierung
- Zugriff auf Daten: Abfrageschnittstellen
- **Suche in Daten**

Apache Lucene

Neben Hadoop (und Avro) hat Doug Cutting ein **weiteres** wichtiges Projekt gegründet: **Lucene**. Dieses bietet eine quelloffene **Textsuche** Implementierung, welche viele **interessante** Merkmale mitbringt, z. B. Boolesche Logik (AND, OR usw.), Term Boosting, Fuzzy Matching. Lucene **hilft** bei der Volltextsuche in Datenbeständen und ist **deshalb** auch im Big Data Umfeld sehr **interessant**.

Apache Solr

Lucene alleine ist aber nur auf **eine** Maschine ausgelegt. Für Big Data brauchen wir aber etwas **Skalierbares** und dies bietet **Solr** an. Es fügt Lucene einen **Serverprozess** hinzu, welcher über eine **REST** API angesteuert werden kann.

SolrCloud ist eine Erweiterung, welche Lucene auf mehrere Rechner verteilt und so Lucene skalierbar macht.

Einheit 4

An dieser Stelle endet die vierte Einheit mit dem Abschluss in die Einführung in MapReduce. In der nächsten Einheit werden wir uns die NoSQL Technologien anschauen, welche die Welt des Batchbetriebes um „Echtzeit“-Abfragen erweitern und ergänzen.

Bis bald!

Übung 4

Ziele:

- Variable Dateiformate für MapReduce Jobs unterstützen
- Ausprobieren welches wann Sinn macht
  - Binäre vs. Text Formate? Komprimierung? Größe?
- Daten über Impala und Hive abfragen
- Solr Index über MapReduce generieren und mit Ergebnissen des TF-IDF Beispiels vergleichen
  - Siehe src/main/resources/books für Daten

Übung 4

Code:

https://github.com/larsgeorge/fh-muenster-bde-lesson-4

Quellen

- Serialisierung
  - Avro: http://avro.apache.org/
  - Thrift: http://thrift.apache.org/
  - Protocol Buffers: https://code.google.com/p/protobuf/
  - Parquet: http://parquet.io/
  - Benchmark: https://code.google.com/p/thrift-protobuf-compare/
- Dateiformate
  - „Hadoop - The Definitive Guide“ von Tom White http://shop.oreilly.com/product/0636920021773.do
- Komprimierung
  - Benchmark: https://github.com/ning/jvm-compressor-benchmark/wiki