Hadoop I/O

- Datenintegrität
- Kompression
- Serialisierung
- Datei-basierte Datenstrukturen

Data I/O und Hadoop

- Allgemeine Techniken „Data I/O“
  - Datenintegrität
  - Kompression
- aber es gibt Besonderheiten bei Multi-Terrabyte großen Daten
- Verteilte Systeme
  - Serialisierungs-Frameworks
  - On-disk data structures

Datenintegrität

Data Corruption

- Erkennen von korrumpierten (verdorbenen) Daten
- *Checksum*: zur Kontrolle ob Daten fehlerhaft
- CRC-32 (Cyclic redundancy check)
  - 32 bit Integer Checksum

HDFS Daten-Integität

- `io.bytes.per.checksum`
  - Parameter: default 512 bytes
  - 4 bytes für CRC-32 Checksum =>
    - storage overhead < 1%
- Verantwortlich zur Kontrolle: DataNodes
  - Bei *pipeline-write* Replikation: Letzte Datanode kontrolliert
- Status wird in log-Files festgehalten
  - Erkennen von fehlerhaften Disks

HDFS Daten-Integrität

- Client liest Daten
  - Bei Fehler: Erkennen und Lesen aus anderem Namenode
  - Falls keine unkorrumpierten Daten verfügbar `ChecksumException`
  - Check kann ausgeschaltet werden (`setVerifyChecksum(false)` oder `-ignoreCrc` Schalter)
- In einem Hintergrund-Thread laufen auf den Datanodes „Datanode block scanner“
- Bei fehlerhaften Blocks
  - Löschen fehlerhafter Blocks
  - Erneute Replikation

Lokale Datenintegrität

- Hadoops `LocalFileSystem` überprüft auf dem Client
  - Bei Schreiben des Files „ex1“
  - verborgene Datei „.ex1.crc“ mit Checksumme der Datei-Chunks wird geschrieben
- Overhead für Checksumming gering
  - kann aber ausgeschaltet werden
    - `RawLocalFileSystem`
    - Global über Property `fs.file.impl`
- `ChecksumFileSystem` als Wrapper für `(nonchecksum)` Dateisysteme

Kompression

Datenkompression

- Daten können in HDFS komprimiert werden
- Verschiedenste Formate möglich
  - deflate, gzip, bzip2, LZO, Snappy
  - Einige Formate unterstützen splitting
    - Wichtig für große Files – Datenlokalität!
- Kompression über Klassen wie `CompressionCodec`

Kompression in MapReduce

- Input Kompression kann über Dateierweiterung bestimmt werden: `CompressionCodecFactory`
- `Bei Output`

```
FileOutputFormat(job, true);
FileOutputFormat.setOutputCompressionClass
    (job, GzipCodec.class);
```

Serialization

Serialisierung

- **Serialisierung** ist der Prozess strukturierte Objekte in einen byte-stream umzuwandeln, z.B. für Übertragung im Netzwerk oder Schreiben zum persistenten Speicher
- **Deserialisierung** ist der umgekehrte Prozess
- Bei *distributed data processing* wichtig für:
  - Interprozess-Kommunikation
  - Persistente Speicherung

Interprozess Kommunikation in Hadoop

- Per *RPC (Remote Procedure Calls)*
  - *messages* werden mittels Serialisierung in einen *binary stream* umgewandelt
  - Ziele von RPC
    - Kompact
    - Schnell
    - Erweiterbar
    - Interoperabel

`Writable` Interface

- Hadoop eigenes Serialisierungsformat
- Zwei Methoden für
  - Schreiben zu `DataOutput` *binary stream*
  - Lesen von `DataOutput` *binary stream*

Writable

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

/** ....
*/
public interface Writable {
  /**
    .. Serialize the fields of this object to out ..
 */
  void write(DataOutput out) throws IOException;
  /** ..
    Deserialize the fields of this object from in ...
 */
 void readFields(DataInput in) throws IOException;
}
```

WritableComparable

- Erweitert Interface `Writable` und `java.lang.Comparable`

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends
Writable, Comparable<T> {
}
```

- Wichtig für Key-Sortierung

RawComparator WritableComparator

- `RawComparator` ist Erweiterung des Java Interface `Comparator` zur Optimierung
  - Ermöglicht Vergleich ohne dass das ganze Objekt deserialisiert werden muss (Overhead der Objekt-Erzeugung wird vermieden)
- Klasse `WritableComparator` ist eine allgemeine Implementierung des `RawComparators`

`WritableComparable` Wrapper Klassen für *Java Primitives*

Java primitive	Implementierung von Writable	Größe der Serialisieung in Bytes
`boolean`	`BooleanWritable`	1
`byte`	`ByteWritable`	1
`short`	`ShortWritable`	2
`int`	`IntWritable` `VIntWritable`	4 1-5
`float`	`FloatWritable`	4
`long`	`LongWritable` `VLongWritable`	4 1-9
`double`	`DoubleWritable`	8

Text

- Standard UTF-8 (nicht Javas modifizierte Version von UTF-8 für Objekt Serialisierung)
- Maximale Länge 2 GB
- Indexing über byte-Offset!

```
@Test
public void text(){ //     3byte 4byte
  Text myText = new Text("\uD801\uDC00");
  assertThat(myText.find("\uDC00", is(3));
}
```

- `Text` ist *mutable* (Java `String` nicht)

ObjectWritable

- Wrapper für JavaPrimitives, `String`, `enum`, `Writable`, `null` und Felder (Arrays) dieser Typen
- Schreibt Klassennamen der gewrappen Typen pro Eintrag
  - → zusätzlicher Speicherverbrauch
  - Alternative für feste Typen (statisches Array fester Typen) per `GenericWritable`

weitere Writables

- `byteWritable`: Wrapper für Array von binären Daten
- `nullWritable`: Platzhalter für 0 bytes
  - Immutable Singleton, Instanz über: `NullWritable.get()`
-

WritableCollections

- `ArrayWritable`
- `ArrayPrimitiveWritable`
- `TwoDArrayWritable`
- `MapWritable`
- `SortedMapWritable`
- `EnumMapWritable`
- keine *sets* (außer `EnumSetWritable`) oder *lists*
  - Sets über `MapWritable` mit `NullWritable` Values
  - Listen z.B. über `ArrayWritable` (gleicher Typ), `GenericWritable` (unterschiedliche Typen)

Serialisierung für eigene Typen

- Eigene Implementierungen des `WritableComparators<XX>`
  - Kontrolle über Sortierordnung
  - Tuning der binären Repräsentation wichtig für Performanz
- Keine *Java Object Serialization* (zu allgemein)
- API für *pluggable* Serialisierungsframeworks
- Apache Avro (oder mit begrenzter Unterstützung Apache Thrift, Google Protokol Buffers)

Datei-basierte Datenstrukturen

NLineinputFormat

- NLineinputFormat
  - fasst n-Zeilen für einen Mapper-Input zusammen

Datei-basierte Daten-Container

- Jedes binäre Blob in ein eigenes File skaliert nicht.
  - → Datei-basierte Daten-Container für binäre Daten
- Zwei Typen
  - `SequenceFile`
  - `MapFile`

SequenceFiles

- Für binäre Key-Value Paare
- Container für kleine Dateinen

MapFiles

- „Sortiertes SequenceFile“
- Persistente Version von `java.util.Map`, die größer als die *in-memory* Version werden kann.

Literatur

- Tom White, „Hadoop The Definite Guide“, third edition, 2012, O'Reilly