Freitag: Große Datenmengen

12 Große Datenmengen

- Big Data
- Map Reduce
- Hadoop
- Aufteilung der Daten
- Komplexität
- Datenkompression
- Monitoring

Große Datenmengen

Was ist "Big Data"?[11]

- viele Unternehmen arbeiten mit großen Datenmengen[10]
  - jeden Tag werden auf Facebook 300 Millionen Fotos hochgeladen und über vier Millarden Inhalte geteilt
  - jede Sekunde werden Videos im Umfang von sechs Stunden auf YouTube hochgeladen
  - jeden Tag werden 500 Millionen Tweets auf Twitter gepostet
  - ...
- wie können wir mit diesen großen, sich schnell verändernden Datenmengen umgehen?

---

[10] http://thefinancialbrand.com/58321/big-data-banking-applications/

[11] Diese Slides sind von `http://lintool.github.io/UMD-courses/bigdata-2015-Spring` adaptiert.

Vertikales und horizontales Skalieren I

Zwei Varianten:

Vertikales Skalieren

- mehr RAM/Festplattenspeicher/CPUs
- stößt an Grenzen, man kann nicht “on the fly” die Performance steigern

Vertikales und horizontales Skalieren II

Horizontales Skalieren

- benutze zusätzliche Rechner
- komplexer als vertikales Skalieren
- heutzutage existieren Umgebungen und Programmierparadigmen (wie MapReduce), welche die Komplexität minimieren

Teile und herrsche

Programmieransätze für Parallelisierung benutzen das Konzept “teile und herrsche” (divide and conquer).

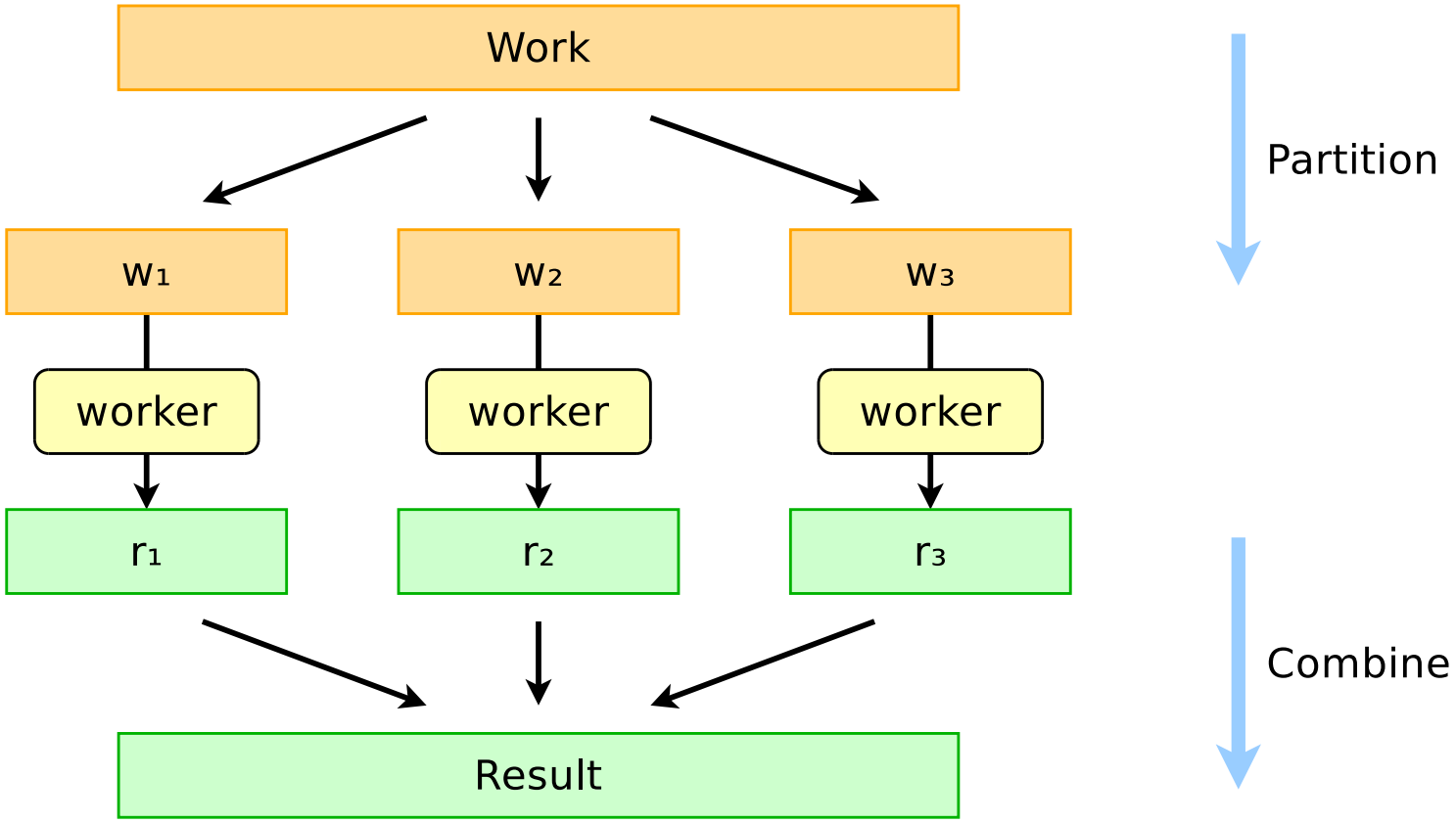


Abbildung: Teile und herrsche.

Herausforderungen

- Wie unterteilen wir den Input in einzelne Einheiten?
- Wie verteilen wir diese Einheiten auf einzelne Rechner?
- Was machen wir, wenn wir mehr Rechner als Einheiten haben?
- Was ist, wenn die Rechner gegenseitig auf Resultate zugreifen müssen?
- Wie aggregieren wir Resultate?
- Wie finden wir heraus, ob alle Rechner ihre Arbeit erledigt haben?
- Was machen wir, wenn ein Rechner ausfällt?

Eine Lösung

Benutze die richtige Abstraktionsebene!

- Benutze die richtige Abstraktionsebene: verberge System-Interna vor dem Entwickler
- Trenne das *was* vom *wie*
  - der Entwickler legt nur fest, welche Berechnung durchgeführt werden soll
  - ein *execution framework* kümmert sich um die Durchführung der Berechnung

Große Datenmengen

MapReduce

- MapReduce ist ein Programmiermodell für verteiltes Rechnen mit sehr großen Datenmengen
- inspiriert durch Paradigma der funktionalen Programmierung
- arbeitet in drei Schritten: *map*, *shuffle* und *reduce*

Funktionale Programmierung I

Eine *map-Operation* erstellt eine neue Liste, indem es eine Funktion f auf jedes Element der Eingabeliste anwendet.

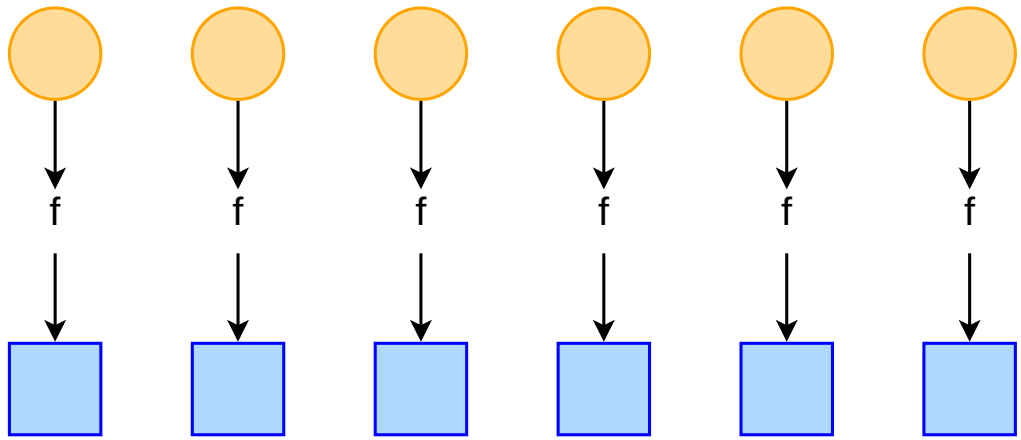


Abbildung: Die map-Operation.

Funktionale Programmierung II

Eine *reduce-Operation* berechnet einen Wert für eine Liste, indem die Elemente der Liste mit einer Funktion g kombiniert werden.

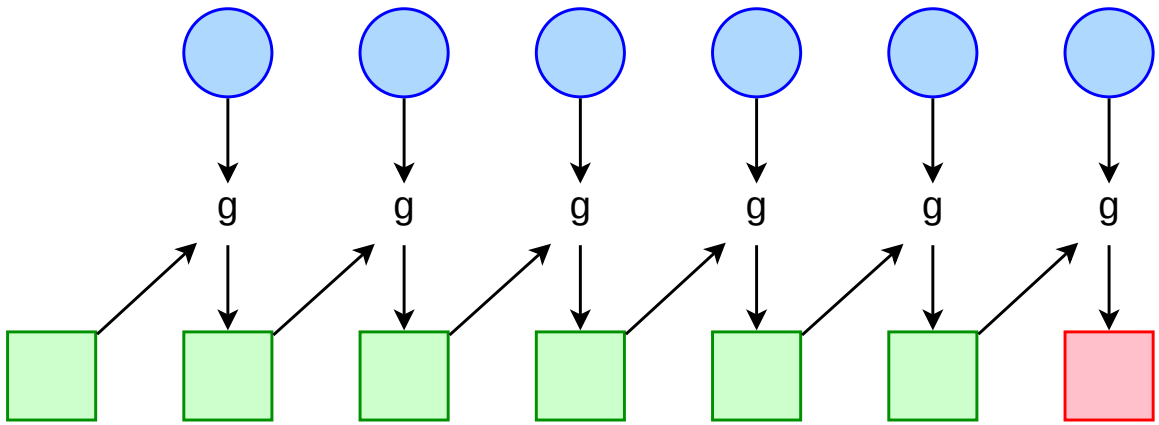


Abbildung: Die reduce-Operation.

Funktionale Programmierung III

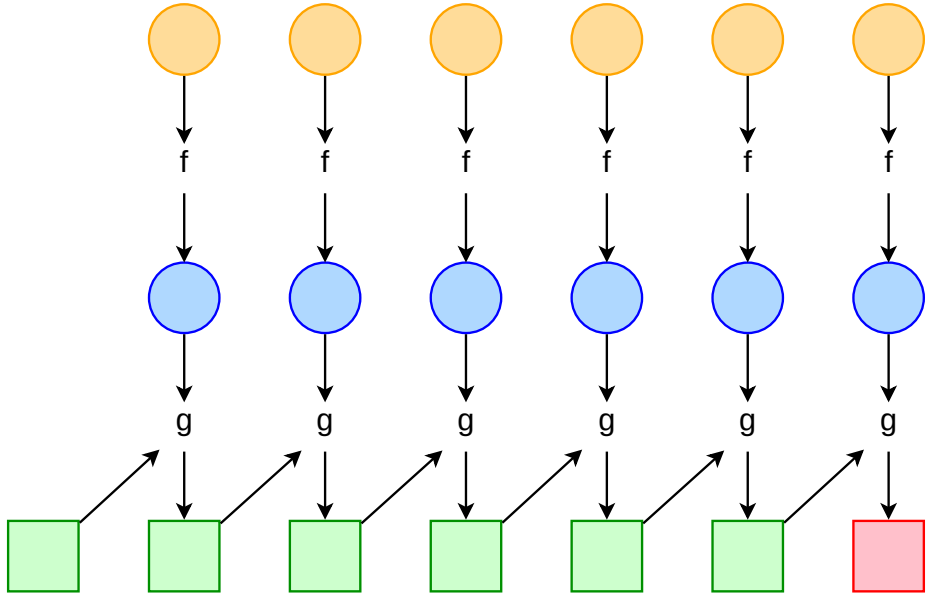


Abbildung: map-reduce.

- Die map-Operation ist trivial parallelisierbar: die Berechnungen sind Unabhängig
- wenn es nicht wichtig ist, in welcher Reihenfolge die Operationen durchgeführt werden, können wir die Operationen umordnen

MapReduce Framework I

Im MapReduce-Framework müssen zwei Funktionen angegeben werden:

1. `map(key, value)` $\rightarrow$ `[(key2, value2)]`
   - Eingabedaten werden von der map-Funktion bearbeitet
   - Ausgabe: eine Liste von von Zwischenresultaten, bestehend aus einem Key und einem Wert
2. `reduce(key2,[value2]) -> [(key2,value2)]`
   - Alle Paare mit dem gleichen Key werden vom selben Reducer verarbeitet
   - Diese aggregiert die Zwischenresultate

MapReduce Framework II

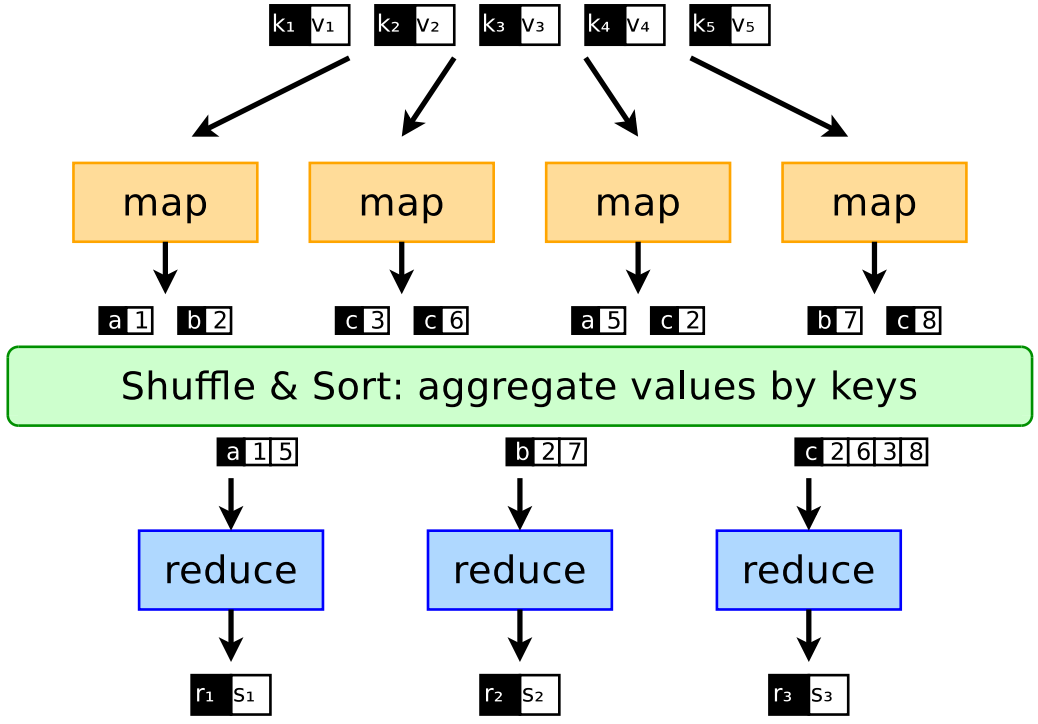


Abbildung: MapReduce.

Beispiel: Zählen von Wörtern I

```
def map(docid, text):
    for each w in text:
        emit(w, 1)

def reduce(term, values) {
    int result = 0

    for each v in values:
        result += v

    emit(term, result)
}
```

Beispiel: Zählen von Wörtern II

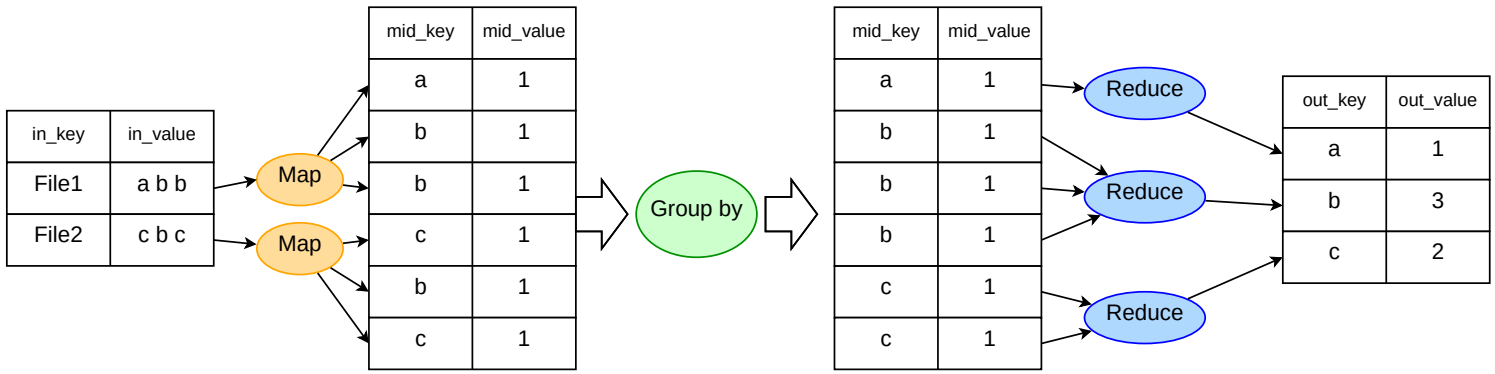


Abbildung: Berechne Worthäufigkeiten mit MapReduce. Quelle: Chen und Schlosser, *Map-Reduce Meets Wider Varieties of Applications*, 2008

Framework

Das MapReduce Framework...

- ...kümmert sich um die Zuteilung von Rechnern zu Aufgaben (map und reduce)
- ...übernimmt die Verteilung der Daten auf Rechner
- ...übernimmt die Synchronisierung (sort und shuffle)
- ...erkennt und behandelt Fehler wie z.B. Rechnerausfall

Große Datenmengen

Hadoop

- Google verfügt über eine proprietäre C++-Implementierung von MapReduce
- Hadoop ist eine open source Implementierung in Java
  - ursprünglich von Yahoo! entwickelt
  - jetzt: Apache
  - viele Erweiterungen, HBase, Hive, Spark
- `http://hadoop.apache.org/`

Wer benutzt Hadoop/MapReduce?

- Amazon/A9
- Facebook
- Google
- IBM
- Intel Research
- Joost
- Last.fm
- New York Times
- Yahoo!
- Baidu

Ein verteiltes Dateisystem

- Schicke nicht die Prozesse zu den Rechnern – schicke die Prozesse zu den Daten!
  - Daten werden auf den Festplatten der Rechner im Cluster gespeichert
  - Prozesse, die Daten benötigen, die auf einem bestimmten Rechner liegen, werden auf diesem Rechner gestartet
- Warum macht man das?
  - bei großen Datenmengen ist der RAM nicht groß genug, um alle Daten im Speicher zu halten
  - Es ist zu langsam, die Daten durch das Netzwerk zu schicken
- Wir benötigen also ein *verteiltes Dateisystem*
  - GFS (Google File System) für Google's MapReduce
  - HDFS (Hadoop Distributed File System) für Hadoop

Annahmen

- Standard-Hardware anstelle von “exotischer” Hardware
  - “Scale out” anstatt “scale up”
- möglicherweise hohe Ausfallrate der Komponenten
  - billige Standard-Hardware fällt vergleichsweise oft aus
- vergleichsweise geringe Anzahl sehr großer Dateien
- Dateien werden einmal erstellt, dann üblicherweise Daten angehängt (evtl. parallel)
- es werden üblicherweise Sequenzen von Daten gelesen

Designenscheidungen

- Dateien werden als “Chunks” gespeichert
  - feste Größe (64MB)
- Zuverlässigkeit durch Replikation
  - jeder Chunk wird auf mindestens drei “Datanodes” gespeichert
- ein “Master” um Zugriffe und Metadaten zu verwalten
  - einfaches zentralisiertes Management
- kein Caching der Daten
  - kein großer Vorteil, da üblicherweise große Sequenzen von Daten gelesen werden

HDFS-Architektur

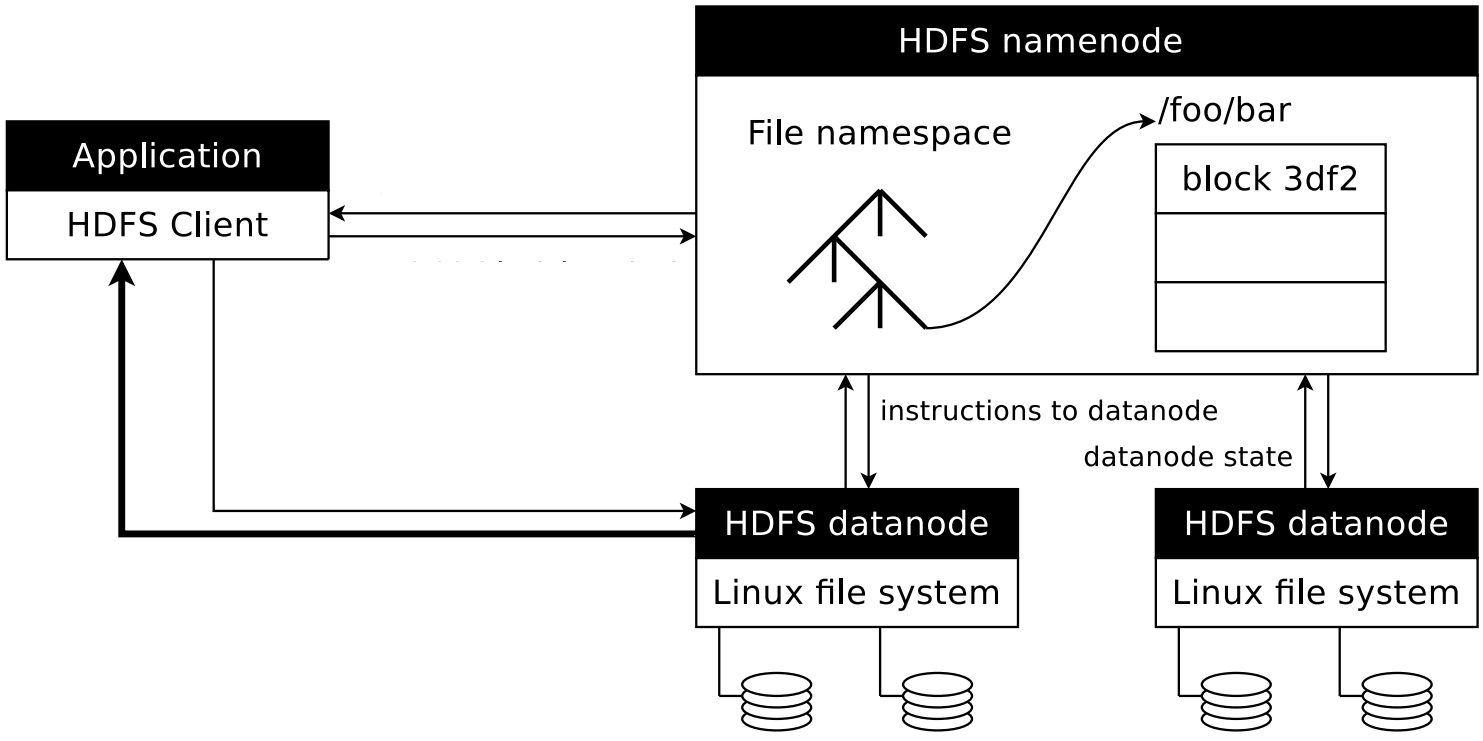


Abbildung: HDFS-Architektur

Verteiltes Dateisystem

- Datenkohärenz
  - Klient kann Daten an existierende Dateien anhängen
  - kein anderes Verändern der Daten möglich
- Dateien sind in Blöcke unterteilt
  - typischerweise von 64 MB größe
  - jeder Block ist auf mehreren Datanodes repliziert
- intelligenter Klient
  - Klient kann Speicherort von Blöcken finden
  - Klient kann direkt auf Datanode zugreifen

Namenode

- verwalte Eigenschaften des Dateisystems:
  - speichere Datei-/Verzeichnisstruktur, Metadaten, Datei-zu-Block-Mapping, Zugriffsrechte, ...
- koordiniere Dateioperationen:
  - verweise Klienten auf Datanodes für Lesen und Schreiben
  - es werden keine Daten durch die Namenode geschickt
- guten Zustand des Clusters erhalten:
  - kommuniziere in festgelegten Intervallen mit den Datanodes
  - Blockreplizierung und Blockrebalancing
  - Garbage collection

Namenode-Metadaten

- Metadaten sind im Arbeitspeicher
- Typen von Metadaten
  - Liste von Dateien
  - Liste von Blöcken für jede Datei
  - Liste von Datanodes für jeden Block
  - Dateiattribute, z.B. Erstellungszeitpunkt, Replikationsfaktor
- Transaktions-Log
  - speichert Dateierstellungen, Dateilöschungen, ...

Datanode

- Block-Server
  - speichert Daten im lokalen Dateisystem (z.B. ext4)
  - speichert Metadaten eines Blocks (z.B. checksum)
  - stellt Daten und Metadaten Klienten zur Verfügung
- Statusberichte
  - sende in festgelegten Intervallen einen Bericht über die Blöcke an die Namenode
- Datensicherung
  - sende Daten an andere spezifierte Namenodes

Block-Zuweisen und Korrektheit von Daten

- standardmäßige Strategie
  - eine Datenkopie auf lokalem Datanode
  - zweite Kopie in einer anderen "logischen Partition" des Clusters
  - tritte Kopie auf auf dieser Partition
  - weitere Kopien werden zufällig plaziert
- Klient liest von Kopie, die am nächsten ist
- benutze Checksums, um Daten zu validieren
- Dateierstellung
  - Klient berechnet Checksum
  - Datanode speichert die Checksum
- Dateizugriff
  - Klient holt Daten und Checksum von Datanode
  - falls Checksum-basierte Validierung fehlschlägt, werden andere Kopien ausprobiert

Namenode-Ausfall

- “single point of failure”
- Transaktions-Log wird in mehreren Verzeicnis gespeichert
  - auf lokalem Dateisystem
  - auf “remote” Dateisystem
- es gibt typischerweise auch noch eine zweite Namenode
  - *keine* Backup-Namenode!
  - erstellt Checkpoints (Images) um eine kaputte Namenode wiederherzustellen

Übersicht

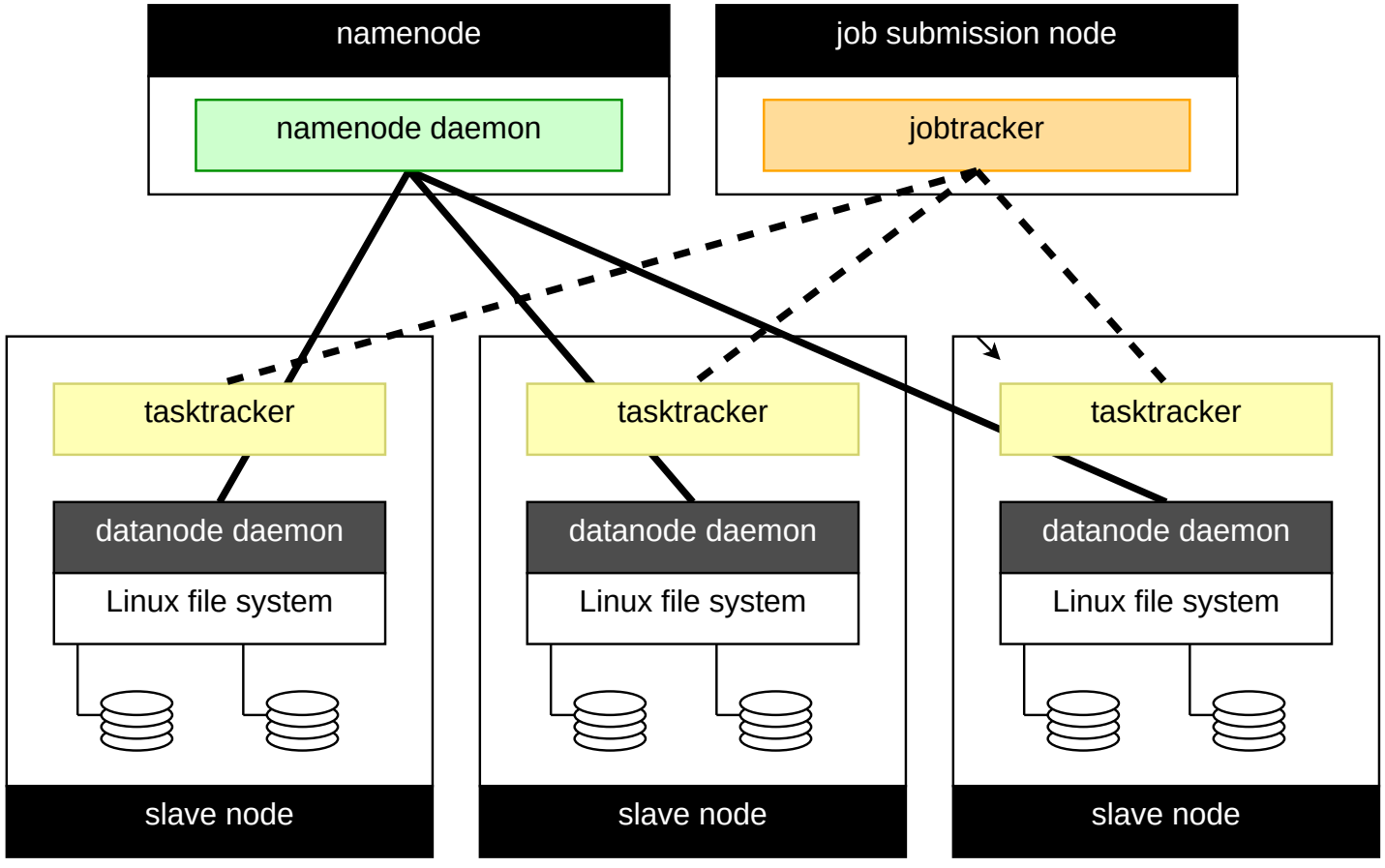


Abbildung: Übersicht von Hadoop-Komponenten.

Referenzen und Literatur

Jimmy Lin und Chris Dyer. *Data-intensive text processing with MapReduce*. Morgan & Claypool, 2010.

Tom White. *Hadoop: The Definitive Guide*, 4th Edition. O'Reilly (2015).

Große Datenmengen

Developmentset

- Während der Entwicklung eines Programms:
  - Untersuchen der Daten: Welche Fälle kommen vor, wie kann ich mit ihnen umgehen? etc.
- Untersuchung des gesamten Datensatz zu aufwändig/unmöglich

  $\rightarrow$ Entwickeln auf repräsentativer Teilmenge

  $\rightarrow$ **Developmentset**

Testset

- Nach der Entwicklung eines Programms:
  - Testen: Stimmt erwartetes Ergebnis mit Ausgabe überein?
- Für Testlauf muss ein Teil der ursprünglichen Daten zurückgehalten werden

  $\rightarrow$ **Testset**

Aufteilung

Also: Ursprünglicher Datensatz wird aufgeteilt in:

- Developmentset
- Trainingsset
- Testset

Die beste Aufteilung variiert von Anwendung zu Anwendung (z.B. 10%, 70%, 20%)

Große Datenmengen

Komplexität

Definition

Unter der **Komplexität** (auch Aufwand oder Kosten) eines Algorithmus versteht man in der Komplexitätstheorie seinen maximalen Ressourcenbedarf (Laufzeit und Speicherplatz).

[Quelle: `http://de.wikipedia.org/wiki/Komplexitaet_(Informatik)`]

Man kann anhand der Beschreibung eines Algorithmus (z.B. durch Pseudocode) abschätzen, wie lange er für die Verarbeitung einer Eingabe von bestimmter (großer) Länge braucht.

O-Notation

Θ , Ω , O usw. beschreiben Mengen von Funktionen.

$f \in \Theta(g)$	f wächst genauso schnell wie g
$f \in \Omega(g)$	f wächst nicht wesentlich langsamer als g
$f \in \omega(g)$	f wächst schneller als g
$\mathbf{f \in O(g)}$	f **wächst nicht wesentlich schneller als** g
$f \in o(g)$	f wächst langsamer als g

Man schreibt auch: $f = O(g)$ (ungenaue Notation)

Große Datenmengen

Zeitkomplexität

Aufwand in Anzahl Rechenschritten in Abhängigkeit von der Länge der Eingabe.

Bsp. 1 - Arrayzugriff

```
el = array[4]
```

- Zugriff auf 4. Element eines Arrays

→ Zugriff auf i-tes Element eines Arrays immer konstanter Aufwand, unabhängig von i, also $O(1)$

Zeitkomplexität

Bsp. 2 - Rekursiv Fibonaccizahlen berechnen

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- In jedem Schritt verdoppelt sich die Anzahl der zu berechnenden Werte (weil ein gesuchter Wert immer die beiden vorhergehenden benötigt)

$\rightarrow$ Aufwand $O(2^n)$

Bsp. 3 - Kontextvektoren erstellen

```
def get_context(corpus, word):
 1    windowSize = 25
 1    context = FreqDist()
 n    for i in xrange(len(corpus)): # len(corpus) = n
 1        if corpus[i] == word:
 1            if i - windowSize < 1:
 1                start = 0
          else:
 1            start = i - windowSize
 1        if i + windowSize > len(corpus):
 1            end = len(corpus) + 1
          else:
 1            end = i + windowSize + 1
25        for contextWord in corpus[start:i] + corpus[i+1:end]:
 1            context.inc(contextWord)
      return context
```

z. B.

$1 + 1 + n * (1 + 1 + 1 + 1 + 1) + 25 * 1 = 5n + 27$

$\rightarrow$ Proportional zu n, also $O(n)$

Große Datenmengen

Platzkomplexität

Unter der **Platzkomplexität** eines Problems versteht man den (minimalen) Bedarf an Speicherplatz eines Algorithmus zur Lösung dieses Problems, in Abhängigkeit von der Länge der Eingabe.

Große Datenmengen

Datenkompression

Eine Datei mit einem Textinhalt wie:

```
Auch ein kleiner Coli ist ein Coli.
```

kann direkt abgespeichert werden, oder in komprimierter Form.
Das spart Platz: Bei typischen Textdateien bis zu 80 - 90%.

Beispiel für eine Kodierung:

```
Auch ein kleiner Coli ist -4 -3.
```

$\rightarrow$ Viele verschiedene Algorithmen und Kombinationen von ihnen

Beispiel gzip

```
:~$ cat Druckversion_jezerte.htm
```

```
... <P> Und sieh, es traf sich, da&szlig; Athmas, der K
&ouml;nig, aus dem Palaste ging, der Morgenk&uuml;hle
zu genie&szlig;en, bevor der Tag anbrach; und wandelte
 den breiten Weg daher auf gelbem Sand und wurde der
Dirne gewahr, trat nahe zu und stand betroffen &uuml
;ber ihre Sch&ouml;nheit, begr&uuml;&szlig;te die
Erschrockene und k&uuml;&szlig;t' ihr die Stirn. <P>
 ...
```

```
:~$ ls -l
```

```
-rw-r--r-- 1 knapp students 20096 3. Feb 16:49
Druckversion_jezerte.htm
```

```
:~$ gzip Druckversion_jezerte.htm
```

```
:~$ ls -l
```

```
-rw-r--r-- 1 knapp students 8355 3. Feb 16:49
Druckversion_jezerte.htm.gz
```

Kompressionsformate

- Verschiedene Verfahren zur Kompression ergeben verschiedene Dateiformate:
  - gzip (.gz)
  - bzip2 (.bz2; .bz)
  - ZIP (.zip)
  - RAR (.rar)
  - 7-ZIP (.7z)
- Für verschiedene Zwecke geeignet

$\rightarrow$ Das richtige Tool für das entsprechende Format

Zwischenergebnisse speichern

Welche Möglichkeiten bietet meine Programmiersprache?

- Java: Klasse `Deflater`
- Python: Module `gzip`, `bz2`, `zipfile`, `pickle`…
- …

Zwischenergebnisse einsehen

Auch wenn Eure Daten nur komprimiert vorliegen, könnt Ihr sie einsehen, ohne sie entpackt speichern zu müssen (funktioniert nur für bestimmte Formate).

```
:~$ bzmore <file>
```

```
:~$ bzless <file>
```

```
:~$ zcat <file>
```

Weiterführender Link

Algorithmen und Datenstrukturen Vorlesung SS 2008 bei U. Köthe

- Wiki zum Thema Effizienz: `http://alda.iwr.uni-heidelberg.de/index.php/Effizienz`

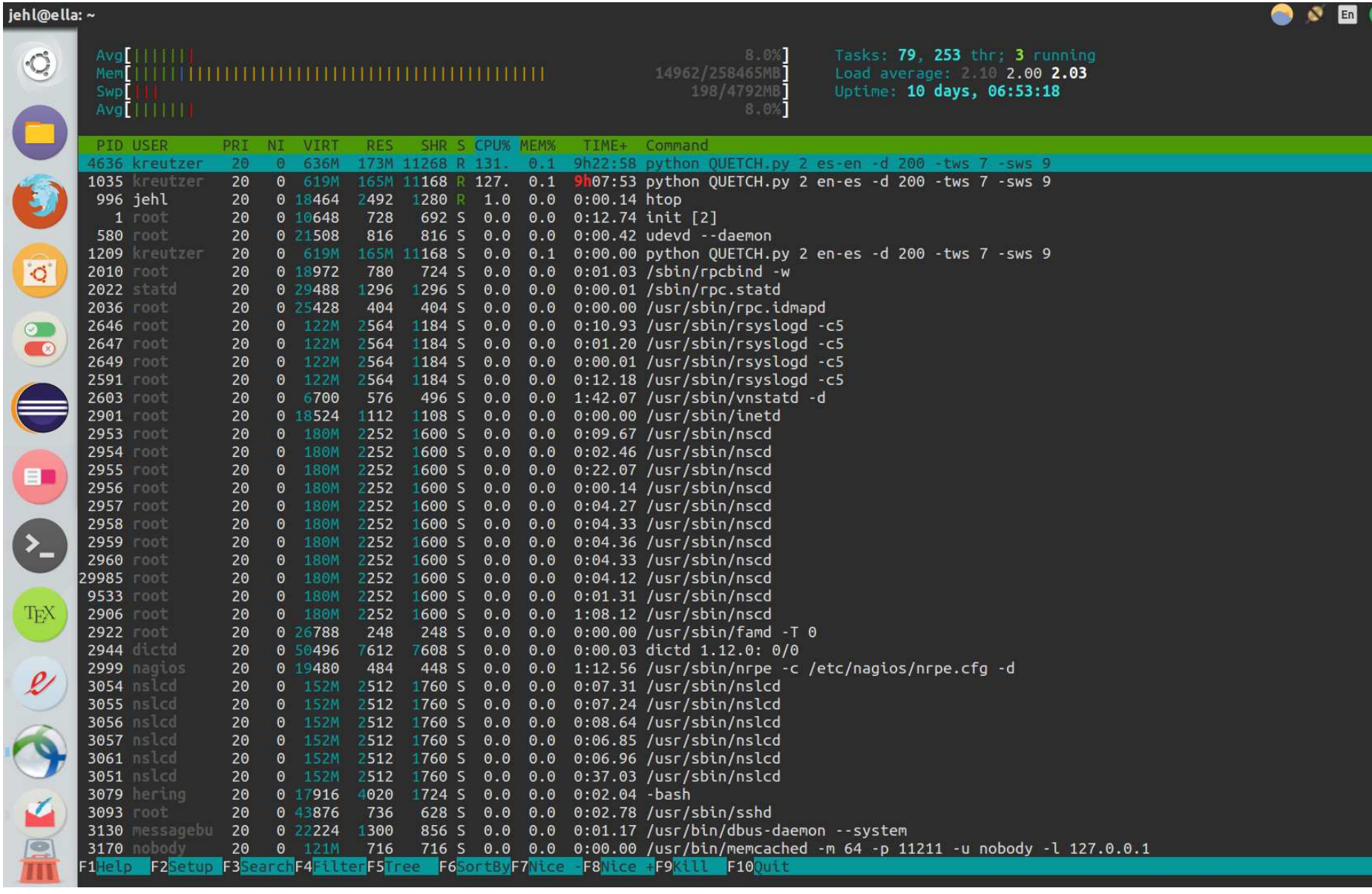
Große Datenmengen

Auf Ressourcenverbrauch achten

Da auch andere auf den Compute Servern arbeiten, sollte man rechen- oder speicherintensive Jobs im Auge behalten. Folgende Befehle helfen hier:

- Überblick über die gibt `:~$ df -h`. Hier können Sie sehen, wie voll `/home/students` ist.
- Größe eines Ordners kann man mit `:~$ du -sh <dirname>` sehen.
- Jobs kann man „nicen“, indem man den Auruf auf der Shell mit dem Keyword `nice` beginnt, z.B. `:~$ nice sleep 10h`. Der Prozess läuft dann mit niedrigerer Priorität.
- `htop` zeigt die laufenden Prozesse und ihren Speicherverbrauch an.

htop


jehl@ella: ~
Avg[|||||| 8.0%]
Mem[|||||||||||||||||||||||||||||||||||| 14962/258465MB]
Swp[||| 198/4792MB]
Avg[|||||| 8.0%]
Tasks: 79, 253 thr; 3 running
Load average: 2.10 2.00 2.03
Uptime: 10 days, 06:53:18
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
4636 kreutzer 20 0 636M 173M 11268 R 131. 0.1 9h22:58 python QUETCH.py 2 es-en -d 200 -tws 7 -sws 9
1035 kreutzer 20 0 619M 165M 11168 R 127. 0.1 9h07:53 python QUETCH.py 2 en-es -d 200 -tws 7 -sws 9
996 jehl 20 0 18464 2492 1280 R 1.0 0.0 0:00.14 htop
1 root 20 0 10648 728 692 S 0.0 0.0 0:12.74 init [2]
580 root 20 0 21508 816 816 S 0.0 0.0 0:00.42 udevd --daemon
1209 kreutzer 20 0 619M 165M 11168 S 0.0 0.1 0:00.00 python QUETCH.py 2 en-es -d 200 -tws 7 -sws 9
2010 root 20 0 18972 780 724 S 0.0 0.0 0:01.03 /sbin/rpcbind -w
2022 statd 20 0 29488 1296 1296 S 0.0 0.0 0:00.01 /sbin/rpc.statd
2036 root 20 0 25428 404 404 S 0.0 0.0 0:00.00 /usr/sbin/rpc.idmapd
2646 root 20 0 122M 2564 1184 S 0.0 0.0 0:10.93 /usr/sbin/rsyslogd -c5
2647 root 20 0 122M 2564 1184 S 0.0 0.0 0:01.20 /usr/sbin/rsyslogd -c5
2649 root 20 0 122M 2564 1184 S 0.0 0.0 0:00.01 /usr/sbin/rsyslogd -c5
2591 root 20 0 122M 2564 1184 S 0.0 0.0 0:12.18 /usr/sbin/rsyslogd -c5
2603 root 20 0 6700 576 496 S 0.0 0.0 1:42.07 /usr/sbin/vnstatd -d
2901 root 20 0 18524 1112 1108 S 0.0 0.0 0:00.00 /usr/sbin/inetd
2953 root 20 0 180M 2252 1600 S 0.0 0.0 0:09.67 /usr/sbin/nscd
2954 root 20 0 180M 2252 1600 S 0.0 0.0 0:02.46 /usr/sbin/nscd
2955 root 20 0 180M 2252 1600 S 0.0 0.0 0:22.07 /usr/sbin/nscd
2956 root 20 0 180M 2252 1600 S 0.0 0.0 0:00.14 /usr/sbin/nscd
2957 root 20 0 180M 2252 1600 S 0.0 0.0 0:04.27 /usr/sbin/nscd
2958 root 20 0 180M 2252 1600 S 0.0 0.0 0:04.33 /usr/sbin/nscd
2959 root 20 0 180M 2252 1600 S 0.0 0.0 0:04.36 /usr/sbin/nscd
2960 root 20 0 180M 2252 1600 S 0.0 0.0 0:04.33 /usr/sbin/nscd
29985 root 20 0 180M 2252 1600 S 0.0 0.0 0:04.12 /usr/sbin/nscd
9533 root 20 0 180M 2252 1600 S 0.0 0.0 0:01.31 /usr/sbin/nscd
2906 root 20 0 180M 2252 1600 S 0.0 0.0 1:08.12 /usr/sbin/nscd
2922 root 20 0 26788 248 248 S 0.0 0.0 0:00.00 /usr/sbin/famd -T 0
2944 dictd 20 0 50496 7612 7608 S 0.0 0.0 0:00.03 dictd 1.12.0: 0/0
2999 nagios 20 0 19480 484 448 S 0.0 0.0 1:12.56 /usr/sbin/nrpe -c /etc/nagios/nrpe.cfg -d
3054 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:07.31 /usr/sbin/nslcd
3055 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:07.24 /usr/sbin/nslcd
3056 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:08.64 /usr/sbin/nslcd
3057 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:06.85 /usr/sbin/nslcd
3061 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:06.96 /usr/sbin/nslcd
3051 nslcd 20 0 152M 2512 1760 S 0.0 0.0 0:37.03 /usr/sbin/nslcd
3079 hering 20 0 17916 4020 1724 S 0.0 0.0 0:02.04 -bash
3093 root 20 0 43876 736 628 S 0.0 0.0 0:02.78 /usr/sbin/sshd
3130 messagebu 20 0 22224 1300 856 S 0.0 0.0 0:01.17 /usr/bin/dbus-daemon --system
3170 nobody 20 0 121M 716 716 S 0.0 0.0 0:00.00 /usr/bin/memcached -m 64 -p 11211 -u nobody -l 127.0.0.1
F1Help F2Setup F3SearchF4FilterF5Tree F6SortByF7Nice -F8Nice +F9Kill F10Quit