ADMIN:DOCS

LPIC-1: Suchen und Extrahieren von Daten aus Dateien

Entdecke in Teil 7 der LPIC-1-Serie, wie du mit find, locate, grep & Regex effizient Dateien suchst und Text extrahierst. Praktische Workflows, Automatisierung und Tipps für Linux-Admins.

Kategorie: LPIC-1 Serie **Veröffentlicht:** 14.11.2025 **Zuletzt geprüft:** 02.11.2026

In den ersten sechs Teilen unserer LPIC-1-Serie hast du die Linux-Kommandozeile kennengelernt, dich im Dateisystem bewegt, Dateiinhalte angezeigt und bearbeitet sowie Text mit Shell-Kommandos verarbeitet. Mit Streams, Pipes und Umleitungen hast du anschließend gelernt, die Ein- und Ausgaben verschiedener Programme miteinander zu verbinden. Im vorherigen Teil ging es schließlich darum, Dateien und Verzeichnisse zu archivieren und zu komprimieren.

Damit kannst du Daten anzeigen, verändern, weiterleiten und platzsparend ablegen. Im administrativen Alltag fehlt jedoch noch ein entscheidender Schritt: Du musst die benötigten Dateien und Informationen zunächst finden. Auf einem überschaubaren Testsystem mag das noch mit einigen Verzeichniswechseln und einem prüfenden Blick funktionieren. Auf einem gewachsenen Server mit mehreren Dateisystemen, zahlreichen Konfigurationen und umfangreichen Logbeständen endet dieser Ansatz recht zuverlässig in Zeitverschwendung.

In diesem siebten Teil der Serie lernst du deshalb, Dateien und Verzeichnisse mit `find`, `locate` und `plocate` gezielt zu suchen. Mit `grep` durchsuchst du Dateiinhalte nach festen Zeichenketten oder regulären Ausdrücken, während `sed` ausgewählte Inhalte weiterverarbeiten kann. Außerdem verbinden wir die Werkzeuge mit Pipes und `xargs` und greifen damit die Konzepte der vorherigen Lektionen wieder auf.

Dabei geht es nicht nur um passende Befehle, sondern auch um Suchbereiche, Berechtigungen, Exitcodes, sichere Dateinamenverarbeitung und die Grenzen der einzelnen Werkzeuge.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

> **Wichtiger Hinweis:** Diese Serie ersetzt keinen offiziellen Vorbereitungskurs für die LPIC-1-Zertifizierung. Sie ergänzt die Prüfungsinhalte um praktische Beispiele und technische Zusammenhänge, die dir beim Lernen und in der späteren Linux-Administration helfen.

Warum Suchen und Extrahieren so zentral ist

Linux-Systeme verteilen ihre Informationen über zahlreiche Verzeichnisse und Dateien. Konfigurationen liegen überwiegend unter `/etc`, Protokolle häufig unter `/var/log`, Anwendungsdaten unter `/var/lib` und benutzerspezifische Einstellungen in den jeweiligen Home-Verzeichnissen. Hinzu kommen Paketdateien, temporäre Daten, Backups und anwendungsspezifische Verzeichnisstrukturen. Zu wissen, dass eine Information irgendwo vorhanden sein müsste, reicht deshalb selten aus.

In der Administration suchst du grundsätzlich auf zwei Ebenen. Entweder benötigst du eine bestimmte Datei oder ein Verzeichnis, oder du suchst innerhalb von Dateien nach einer Zeichenkette beziehungsweise einem Muster. Für die erste Aufgabe stehen dir Werkzeuge wie `find`, `locate` und `plocate` zur Verfügung. Dateiinhalte durchsuchst du dagegen mit `grep` und regulären Ausdrücken. `sed` kann die gefundenen Inhalte anschließend auswählen oder verändern.

Typische Aufgaben sind beispielsweise:

- eine unbekannte oder verschobene Konfigurationsdatei auffinden,
- große Dateien identifizieren, wenn ein Dateisystem vollzulaufen droht,
- kürzlich geänderte Dateien nach einer fehlerhaften Aktualisierung ermitteln,
- Dateien anhand von Eigentümer oder Berechtigungen prüfen,
- Fehlermeldungen in umfangreichen Logdateien finden,
- bestimmte Werte aus Konfigurationen oder Befehlsausgaben extrahieren,
- komprimierte Protokolle durchsuchen, ohne sie dauerhaft zu entpacken.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Die Werkzeuge ergänzen sich dabei. `find` durchsucht den aktuellen Zustand des Dateisystems und kann Treffer anhand zahlreicher Eigenschaften auswählen. `locate` und `plocate` greifen auf einen zuvor erzeugten Index zurück und liefern dadurch meist schneller Ergebnisse, können jedoch veraltete Einträge enthalten. `grep` arbeitet nicht mit Dateieigenschaften, sondern mit dem Inhalt einer Datei oder eines Datenstroms.

Gerade die Kombination mit Streams, Pipes und Umleitungen macht diese Befehle für die Administration wichtig. Suchergebnisse lassen sich an weitere Programme übergeben, filtern oder in Dateien schreiben. Damit entsteht aus mehreren kleinen Werkzeugen ein nachvollziehbarer Arbeitsablauf, ohne dass dafür ein umfangreiches Skript erforderlich ist.

> **Tipp:** Lege vor einer Suche fest, ob du nach einem Pfad, einer Dateieigenschaft oder einem Inhalt suchst. Diese Unterscheidung entscheidet darüber, welches Werkzeug sinnvoll ist und verhindert unnötige Suchläufe über große Verzeichnisbäume.

Suchen ist allerdings nicht automatisch eine harmlose Leseoperation.

Ein unbeschränktes `find /` kann zahlreiche eingebundene Dateisysteme durchlaufen und erhebliche I/O-Last verursachen. Aktionen wie `-delete`, `-exec` oder eine nachgeschaltete Verarbeitung mit `xargs` können Treffer unmittelbar verändern oder entfernen. Deshalb trennen wir in diesem Artikel konsequent zwischen dem Ermitteln von Treffern und den darauf ausgeführten Aktionen.

> ⚠ **Warnung:** Prüfe Suchergebnisse zuerst mit einer reinen Ausgabe, bevor du Dateien löschst, verschiebst oder ihre Inhalte veränderst. Ein syntaktisch korrekter Suchbefehl kann trotzdem mehr Dateien erfassen als beabsichtigt.

Einordnung in die LPIC-1-Zertifizierung

Das Suchen nach Dateien und das Filtern von Dateiinhalten ist in mehreren Prüfungszielen des LPIC-1 Exam 101 verankert. Grundlage sind die derzeit veröffentlichten Prüfungsziele der Version 5.0 für den Exam-Code 101-500.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Für diesen Artikel sind insbesondere folgende Prüfungsziele relevant:

- **103.3 – Grundlegende Dateiverwaltung:** Dieses Prüfungsziel umfasst den Einsatz von `find`, um Dateien anhand ihres Typs, ihrer Größe oder ihrer Zeitstempel zu finden und anschließend Aktionen auf die Treffer anzuwenden. Das Prüfungsziel hat die Gewichtung 4.
- **103.7 – Textdateien mit regulären Ausdrücken durchsuchen:** Hierzu gehören einfache und erweiterte reguläre Ausdrücke sowie deren Verwendung mit `grep` und `sed`. Auch das Suchen, Löschen, Ändern und Ersetzen von Text anhand eines Musters ist Bestandteil dieses Prüfungsziels. Die Gewichtung beträgt 3.
- **104.7 – Systemdateien finden und am richtigen Ort ablegen:** Dieses Prüfungsziel behandelt typische Verzeichnisse nach dem Filesystem Hierarchy Standard sowie die Werkzeuge `find`, `locate` und `updatedb`. Es hat die Gewichtung 2.

Darüber hinaus greift der Artikel Inhalte aus bereits behandelten Prüfungszielen wieder auf. `sed` gehört auch zur Verarbeitung von Textströmen aus **103.2**. Pipes und `xargs`, mit denen du Suchergebnisse an weitere Befehle übergibst, sind Bestandteil von **103.4**.

Die Gewichtung eines Prüfungsziels gibt dessen relative Bedeutung innerhalb der Prüfung an. Eine höhere Gewichtung bedeutet, dass dazu voraussichtlich mehr Fragen gestellt werden. Aus den Gewichtungen lässt sich jedoch kein fester prozentualer Anteil für diesen Artikel ableiten, weil sich seine Inhalte über mehrere Prüfungsziele verteilen.

> **Prüfungshinweis:** Lerne nicht nur einzelne Optionen auswendig. Du solltest erkennen können, ob eine Aufgabe eine Suche nach Dateieigenschaften, eine Suche nach Dateiinhalten oder eine Kombination mehrerer Werkzeuge verlangt. Genau diese Abgrenzung entscheidet häufig darüber, ob `find`, `locate`, `grep` oder eine Pipe mit `xargs` die passende Lösung ist.

Was dich in diesem Artikel erwartet

In den folgenden Abschnitten lernst du die wichtigsten Werkzeuge und Konzepte kennen, mit denen du unter Linux Dateien, Verzeichnisse und Inhalte gezielt suchst:

- Grundlagen der Dateisuche und die Auswahl des passenden Werkzeugs
- `find` für die Suche nach Namen, Dateitypen, Größen, Zeitstempeln, Eigentümern und Berechtigungen
- logische Verknüpfungen und Aktionen innerhalb eines `find`-Ausdrucks
- `locate` und `plocate` für schnelle Suchvorgänge über eine Pfaddatenbank
- `grep` für feste Zeichenketten und reguläre Ausdrücke
- grundlegende und erweiterte reguläre Ausdrücke
- Auswahl und Veränderung von Text mit `sed`
- Suche in komprimierten Dateien und Archiven
- sichere Kombinationen aus `find`, `grep`, Pipes und `xargs`
- Übungen zur Vertiefung und Prüfungsvorbereitung

Die Werkzeuge werden nicht isoliert betrachtet. Du lernst, wie sie zusammenarbeiten, welche Grenzen sie haben und wie du ihre Ergebnisse überprüfst, bevor daraus weitere Aktionen entstehen.

Praktische Relevanz für den Administrator-Alltag

Suchen gehört zu den häufigsten Aufgaben auf einem Linux-System. Oft beginnt eine Fehleranalyse nicht mit einer Änderung, sondern mit der Frage, wo sich eine relevante Datei befindet und welche Informationen sie enthält.

Typische Einsatzbereiche sind:

- Konfigurationsdateien eines Dienstes auffinden,
- Fehlermeldungen in aktuellen und älteren Logs untersuchen,
- große oder lange nicht veränderte Dateien identifizieren,
- Dateien mit bestimmten Eigentümern oder Berechtigungen prüfen,

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

- geänderte Dateien nach einem Update eingrenzen,
- bestimmte Direktiven in mehreren Konfigurationsdateien vergleichen,
- Daten aus Befehlsausgaben oder Textdateien extrahieren,
- komprimierte Protokolle und Archive untersuchen,
- Suchergebnisse sicher an weitere Befehle übergeben.

Dabei geht es nicht darum, möglichst lange Befehlsketten zu bauen. Eine gute Suche beginnt mit einem klar begrenzten Suchbereich und eindeutigen Kriterien. Je genauer du weißt, wonach du suchst, desto leichter lassen sich Ergebnisse bewerten und unbeabsichtigte Aktionen vermeiden.

Wie gewohnt findest du im gesamten Artikel praxisnahe Markierungen zur Orientierung:

- **Tipps und Hinweise** für eine nachvollziehbare und effiziente Arbeitsweise
- ⚠ **Warnungen und Stolperfallen** bei belastenden oder verändernden Befehlen
- **Praktische Beispiele** zum direkten Nachvollziehen auf der Kommandozeile
- **Typische Fehlerquellen** mit fundierter Erklärung der technischen Ursache

So holst du das meiste aus diesem Artikel heraus

Arbeite die Beispiele am besten in einer eigenen Testumgebung durch. Für die meisten Übungen genügt ein temporäres Verzeichnis in deinem Home-Verzeichnis. Dort kannst du Dateien mit unterschiedlichen Namen, Inhalten, Größen und Zeitstempeln anlegen, ohne produktive Daten zu berühren.

Gerade bei `find`, regulären Ausdrücken und `xargs` reicht es nicht, einen Befehl nur zu lesen. Kleine Unterschiede bei Anführungszeichen, Klammern oder der Reihenfolge von Ausdrücken können das Ergebnis deutlich verändern. Führe deshalb zunächst nur Suchbefehle aus, die Treffer anzeigen. Ergänze verändernde Aktionen erst, nachdem du die Ausgabe geprüft hast.

> **Wichtiger Hinweis:** Verwende für die Übungen keine produktiven Verzeichnisse und führe verändernde Befehle nicht als `root` aus. Ein falsch gesetztes Suchkriterium bleibt auch mit administrativen Rechten falsch – es richtet dann lediglich gründlicher Schaden

> an.

Achte bei jedem Beispiel auf drei Fragen:

Welcher Bereich wird durchsucht, nach welchen Kriterien wird ausgewählt und was geschieht anschließend mit den Treffern? Wenn du diese drei Punkte beantworten kannst, hast du den Befehl verstanden und musst ihn nicht bloß auswendig lernen.

Grundlagen der Datensuche unter Linux

Bevor wir die einzelnen Werkzeuge betrachten, müssen wir zwei unterschiedliche Aufgaben voneinander trennen: die Suche nach Dateien und Verzeichnissen sowie die Suche nach Informationen innerhalb von Dateien.

Diese Unterscheidung klingt zunächst selbstverständlich, ist aber für die Auswahl des richtigen Befehls entscheidend. `find` kann eine Datei anhand ihres Namens, Typs oder ihrer Metadaten auswählen. Ob in dieser Datei eine bestimmte Fehlermeldung steht, weiß `find` jedoch nicht. Dafür benötigst du ein Werkzeug wie `grep`.

Dateien finden und Dateiinhalte durchsuchen

Bei einer Suche nach Dateien oder Verzeichnissen arbeitest du mit Eigenschaften des Dateisystemeintrags. Dazu gehören beispielsweise:

- Name und Pfad,
- Dateityp,
- Größe,
- Zeitstempel,
- Eigentümer und Gruppe,
- Berechtigungen.

Ein typisches Suchziel lautet:

Finde unter `/var/log` alle regulären Dateien, die auf `.log` enden und innerhalb der letzten sieben Tage verändert wurden. Hier wird nicht geprüft, was in den Dateien steht. Entscheidend sind ausschließlich Pfad, Typ, Name und Zeitstempel.

Bei einer Inhaltssuche sieht die Aufgabe anders aus:

Finde in den Logdateien alle Zeilen, die das Wort `error` enthalten. Nun müssen die Dateien geöffnet und ihre Inhalte gelesen werden. Der Dateiname spielt dabei möglicherweise eine Rolle, ist aber nicht das eigentliche Suchkriterium. Diese Aufgabe übernimmt `grep`.

In der Praxis werden beide Ebenen häufig kombiniert. Zuerst wählst du mit `find` eine begrenzte Menge von Dateien aus. Anschließend übergibst du diese Dateien an `grep`, um ihre Inhalte zu durchsuchen.

Dateisuche (find / locate)

Suchziel
: Dateinamen, Dateipfade, Typen, Rechte und Zeitstempel

Werkzeuge
: find (Echtzeit-Dateisystem), locate (Index-Datenbank)

Dateizugriff
: Liest nur Metadaten und Inodes des Dateisystems

Performance
: Extrem schnell bei Inode- und Index-Filtern

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Inhaltssuche (grep / sed / awk)

Suchziel
: Textinhalte, Logzeilen, Fehlermuster in Dateien

Werkzeuge
: grep (Musterfilter), sed (Stream-Editor), awk (Spalten)

Dateizugriff
: Öffnet und liest den tatsächlichen Dateiinhalt Zeile für Zeile

Performance
: Abhängig von Dateigröße und Regex-Komplexität

Gemeinsam: Über Pipes, -exec oder xargs lassen sich beide Welten nahtlos zu mächtigen Pipelines verketten

Das Diagramm zeigt die grundlegende Aufteilung: `find`, `locate` und `plocate` ermitteln Pfade. `grep` untersucht Inhalte. `sed` kann passende Textstellen auswählen oder verändern. Über Pipes, `-exec` oder `xargs` lassen sich beide Ebenen verbinden.

Das passende Werkzeug auswählen

Die Werkzeuge überschneiden sich teilweise, arbeiten aber nach unterschiedlichen Prinzipien:

Werkzeug	Durchsucht	Grundlage	Typischer Einsatz
`find`	Dateien und Verzeichnisse	aktueller Zustand des Dateisystems	präzise Suche nach Name, Typ, Größe, Zeit oder Berechtigungen
`locate`	gespeicherte Pfade	zuvor erzeugte Datenbank	schnelle Suche nach bekannten Datei- oder Verzeichnisnamen
`plocate`	gespeicherte Pfade	komprimierter Suchindex	schnelle indexbasierte Pfadsuche auf Systemen mit `plocate`
`grep`	Textdateien und Datenströme	Zeichenketten oder reguläre Ausdrücke	Logs, Konfigurationen und Befehlsausgaben durchsuchen
`sed`	Textdateien und Datenströme	Adressen und reguläre Ausdrücke	Text auswählen, ersetzen oder umformen

Werkzeug	Durchsucht	Grundlage	Typischer Einsatz
`xargs`	keine eigenen Daten	Eingabe über `stdin`	Eingabewerte als Argumente an einen anderen Befehl übergeben

`find` und `locate` sind deshalb keine schnellere und langsamere Variante desselben Befehls. `find` prüft den aktuellen Zustand eines Verzeichnisbaums. `locate` und `plocate` lesen aus einer Datenbank, deren Inhalt vom Zeitpunkt der letzten Aktualisierung abhängt.

Auch `grep` und `find` ersetzen einander nicht. `find` entscheidet, welche Dateien ausgewählt werden. `grep` entscheidet, welche Zeilen oder Textstellen zu einem Muster passen.

> **Tipp:** Formuliere die Aufgabe vor dem Befehl als Satz. „Finde Dateien, die …“ führt meist zu `find`. „Finde Zeilen, die … enthalten“ führt zu `grep`. Enthält die Aufgabe beide Aussagen, benötigst du wahrscheinlich eine Kombination.

Suchbereiche, Berechtigungen und Systemlast

Jede Dateisuche beginnt an einem definierten Punkt. Bei `find` ist das der angegebene Startpfad. Je größer dieser Bereich ist, desto mehr Verzeichnisse müssen gelesen und desto mehr Einträge müssen geprüft werden.

Eine Suche unter `/etc` bleibt auf einen vergleichsweise klaren Bereich beschränkt. Eine Suche ab `/` kann dagegen lokale Dateisysteme, eingebundene Datenträger, Netzlaufwerke und virtuelle Dateisysteme erreichen. Welche Bereiche tatsächlich durchlaufen werden, hängt von den vorhandenen Mountpoints und den verwendeten Optionen ab.

Große Suchbereiche haben mehrere Folgen:

- Die Suche benötigt mehr Zeit und erzeugt zusätzliche I/O-Last.
- Fehlende Leserechte führen zu Fehlermeldungen oder unvollständigen Ergebnissen.
- Eingebundene Netzwerkdateisysteme können die Suche erheblich verzögern.
- Verändernde Aktionen erfassen möglicherweise mehr Dateien als beabsichtigt.
- Flüchtige Verzeichnisse können sich während der Suche verändern.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Nicht jeder Berechtigungsfehler ist ein Grund, den gesamten Befehl mit `sudo` auszuführen. Prüfe zuerst, ob die nicht lesbaren Verzeichnisse für deine Aufgabe überhaupt relevant sind. Administrative Rechte erweitern den Suchbereich, erhöhen bei nachfolgenden Aktionen aber zugleich den möglichen Schaden.

Fehlermeldungen lassen sich über die Standardfehlerausgabe ausblenden:

```
find /var -name "*.conf" 2>/dev/null
```

Damit verschwinden Meldungen wie `Permission denied` aus der Anzeige. Der Befehl erhält dadurch jedoch keine zusätzlichen Rechte. Nicht lesbare Verzeichnisse werden weiterhin nicht durchsucht, und die Ausgabe kann unvollständig sein.

> ⚠ **Warnung:** Das Umleiten von `stderr` nach `/dev/null` behebt keinen Fehler. Es macht ihn lediglich unsichtbar. Verwende diese Umleitung nur, wenn du bewusst auf die Meldungen verzichten kannst und weißt, dass fehlende Treffer für die jeweilige Aufgabe akzeptabel sind.

Für große Verzeichnisbäume solltest du den Suchbereich möglichst früh begrenzen. Geeignete Startpfade, eine kontrollierte Suchtiefe und die Beschränkung auf ein Dateisystem reduzieren Laufzeit und Last. Die dafür vorgesehenen Optionen von `find` behandeln wir im nächsten Abschnitt.

find – Dateien und Verzeichnisse gezielt suchen

`find` durchsucht einen oder mehrere Verzeichnisbäume und prüft jeden gefundenen Eintrag gegen einen Ausdruck. Dieser Ausdruck kann Tests, Optionen, logische Operatoren und Aktionen enthalten. Dadurch eignet sich `find` nicht nur zum Auffinden von Dateien, sondern auch für Prüfungen und kontrollierte Verarbeitungsschritte.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Anders als `locate` arbeitet `find` nicht mit einem zuvor erzeugten Index. Der Befehl liest den aktuellen Zustand des Dateisystems. Neue, verschobene oder gelöschte Dateien werden daher unmittelbar berücksichtigt, sofern der aufrufende Benutzer die betreffenden Verzeichnisse durchsuchen darf.

Funktionsweise und grundlegende Syntax

Die grundlegende Syntax lautet:

```
find [Startpfad] [Ausdruck]
```

Der Startpfad legt fest, an welcher Stelle die Suche beginnt. Der Ausdruck beschreibt, welche Einträge passen und was mit ihnen geschehen soll.

Ein einfaches Beispiel:

```
find /etc -type f -name "*.conf"
```

Der Befehl beginnt unter `/etc`, berücksichtigt nur reguläre Dateien und wählt Dateinamen aus, die auf `.conf` enden. Unterverzeichnisse werden standardmäßig rekursiv durchsucht.

Ein `find`-Ausdruck besteht aus mehreren Elementen:

- **Tests** prüfen Eigenschaften wie Name, Typ, Größe oder Zeitstempel.
- **Optionen** beeinflussen den Suchvorgang, beispielsweise die maximale Tiefe.
- **Operatoren** verknüpfen mehrere Bedingungen miteinander.
- **Aktionen** geben Treffer aus oder übergeben sie an andere Befehle.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Wenn du keine andere Aktion angibst, gibt `find` die Pfade der passenden Einträge aus. Diese Ausgabe entspricht im einfachen Fall der Aktion `-print`.

1. **Startpfad festlegen (z. B. /var/log)** find öffnet das Ausgangsverzeichnis im Dateisystem
2. **Verzeichnishierarchie rekursiv durchlaufen** Inodes und Unterverzeichnisse per Tiefensuche abtasten
3. **Filterkriterien anwenden** Jeden Eintrag gegen Typ (-type), Name (-name), mtime und Rechte prüfen
4. **Aktion auf Treffer ausführen** Pfad auf stdout ausgeben (-print) oder Befehl ausführen (-exec)

`find` wertet den Ausdruck für jeden erreichbaren Eintrag aus. Welche Dateien am Ende ausgegeben oder verarbeitet werden, hängt deshalb nicht nur von den einzelnen Tests, sondern auch von ihrer Verknüpfung ab.

Suche nach Namen und Dateitypen

Mit `-name` suchst du nach dem Namen eines Verzeichniseintrags. Das Muster gilt nur für den letzten Bestandteil des Pfades, nicht für den vollständigen Pfad.

```
find /etc -name "sshd_config"
```

Für Muster stehen dieselben grundlegenden Platzhalter zur Verfügung, die du bereits vom Shell-Globbing kennst:

- `*` steht für beliebig viele Zeichen,
- `?` steht für genau ein Zeichen,
- `[abc]` steht für eines der angegebenen Zeichen.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Alle Dateien mit der Endung `.conf` findest du beispielsweise so:

```
find /etc -name "*.conf"
```

Die Anführungszeichen sind wichtig. Ohne sie kann die Shell das Muster bereits vor dem Start von `find` expandieren. `find` erhält dann nicht mehr das Muster, sondern möglicherweise eine Liste vorhandener Dateinamen aus dem aktuellen Verzeichnis.

> **Typische Fehlerquelle:** Schreibe Suchmuster wie `"*.conf"` oder `"log?.txt"` in Anführungszeichen. Andernfalls verarbeitet unter Umständen die Shell den Platzhalter, obwohl er für `find` bestimmt ist.

`-name` unterscheidet zwischen Groß- und Kleinschreibung. Für eine Suche ohne diese Unterscheidung stellt GNU `find` den Test `-iname` bereit:

```
find /etc -iname "*.conf"
```

Mit `-path` prüfst du den vollständigen Pfad des Eintrags:

```
find /etc -path "*/systemd/*.conf"
```

Für die Auswahl eines Dateityps verwendest du `-type`:

Test	Bedeutung
`-type f`	reguläre Datei
`-type d`	Verzeichnis
`-type l`	symbolischer Link
`-type b`	blockorientiertes Gerät
`-type c`	zeichenorientiertes Gerät
`-type p`	benannte Pipe
`-type s`	Socket

Eine Suche nach regulären Logdateien sieht damit so aus:

```
find /var/log -type f -name "*.log"
```

Die Einschränkung auf `-type f` verhindert, dass gleichnamige Verzeichnisse oder andere Dateitypen als Treffer erscheinen.

Suche nach Größe und Zeitstempeln

Mit `-size` wählst du Dateien anhand ihrer logischen Größe aus:

```
find /var -type f -size +100M
```

Das `+` bedeutet größer als der angegebene Wert. Ein vorangestelltes `-` bedeutet kleiner als der Wert. Ohne Vorzeichen sucht `find` nach der angegebenen Anzahl von Einheiten.

GNU `find` unterstützt unter anderem folgende Einheiten:

Suffix	Einheit
`c`	Byte
`k`	Kibibyte mit 1024 Byte
`M`	Mebibyte mit 1024 × 1024 Byte
`G`	Gibibyte mit 1024 × 1024 × 1024 Byte
kein Suffix	Blöcke mit jeweils 512 Byte

Die Größe wird auf ganze Einheiten aufgerundet. `-size 1M` bedeutet deshalb nicht, dass eine Datei exakt ein Mebibyte groß sein muss. Es erfasst Dateien, deren auf ganze Mebibyte aufgerundete Größe eins beträgt.

Für eine Suche nach leeren regulären Dateien kannst du `-empty` verwenden:

```
find /tmp -type f -empty
```

Zeitbasierte Tests unterscheiden zwischen mehreren Zeitstempeln:

- `-mtime` prüft die letzte Änderung des Dateiinhalts,
- `-ctime` prüft die letzte Änderung der Inode-Metadaten,
- `-atime` prüft den letzten Zugriff,
- `-mmin`, `-cmin` und `-amin` arbeiten entsprechend mit Minuten.

Dateien, deren Inhalt innerhalb der letzten sieben vollständigen 24-Stunden-Zeiträume geändert wurde, findest du mit:

```
find /var/log -type f -mtime -7
```

Die Schreibweise folgt einem festen Schema:

Schreibweise	Bedeutung
`-mtime 7`	vor sieben vollständig vergangenen 24-Stunden-Zeiträumen
`-mtime -7`	vor weniger als sieben vollständig vergangenen 24-Stunden-Zeiträumen
`-mtime +7`	vor mehr als sieben vollständig vergangenen 24-Stunden-Zeiträumen

`find` rundet die Zeitspanne bei `-mtime`, `-ctime` und `-atime` auf vollständige 24-Stunden-Zeiträume ab. Für genauere Grenzen eignen sich die Minutenvarianten:

```
find /etc -type f -mmin -60
```

Dieser Befehl sucht Dateien, deren Inhalt vor weniger als 60 Minuten geändert wurde.

Mit `-daystart` berechnet GNU `find` Tagesangaben ab dem Beginn des aktuellen Kalendertages statt relativ zum Startzeitpunkt des Befehls:

```
find /var/log -daystart -type f -mtime 0
```

> **Tipp:** Verwende `-mmin`, wenn die genaue Zahl vergangener Minuten wichtig ist. `-mtime` arbeitet mit vollständigen 24-Stunden-Zeiträumen und entspricht nicht automatisch einem Kalendertag.

Suche nach Benutzer, Gruppe und Berechtigungen

Mit `-user` und `-group` wählst du Dateien anhand ihres Eigentümers oder ihrer Gruppe aus:

```
find /srv -type f -user www-data
```

```
find /srv -type f -group www-data
```

Für numerische IDs stehen `-uid` und `-gid` zur Verfügung. Dateien, deren gespeicherte Benutzer- oder Gruppen-ID auf dem System keinem bekannten Eintrag mehr zugeordnet werden kann, findest du mit `-nouser` beziehungsweise `-nogroup`:

```
find /home -xdev \( -nouser -o -nogroup \) -print
```

Solche Dateien können beispielsweise nach dem Löschen eines Benutzerkontos zurückbleiben.

Der Test `-perm` prüft die gesetzten Berechtigungsbits. Dabei ist die genaue Schreibweise entscheidend:

```
find /srv -type f -perm 0644
```

Dieser Befehl findet ausschließlich Dateien, deren Berechtigungsbits exakt `0644` entsprechen. Mit einem vorangestellten Minus müssen alle angegebenen Bits gesetzt sein. Andere Bits dürfen zusätzlich vorhanden sein:

```
find /srv -type f -perm -0002
```

Damit findest du reguläre Dateien, bei denen das Schreibrecht für andere Benutzer gesetzt ist. Ob darüber hinaus weitere Rechte bestehen, spielt für diesen Test keine Rolle.

Mit einem Schrägstrich genügt es, wenn mindestens eines der angegebenen Bits gesetzt ist:

```
find / -xdev -type f -perm /6000
```

Dieser Befehl findet reguläre Dateien, bei denen das SUID- oder SGID-Bit gesetzt ist.

Schreibweise	**Prüfung**
`-perm 0644`	Berechtigungen entsprechen exakt `0644`
`-perm -0002`	alle angegebenen Bits sind gesetzt
`-perm /6000`	mindestens eines der angegebenen Bits ist gesetzt

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

> ⚠ **Warnung:** Eine Suche nach auffälligen Berechtigungen ist zunächst nur eine Bestandsaufnahme. Ändere Rechte nicht pauschal mit einem nachgeschalteten `chmod`. Prüfe vorher, warum die jeweilige Datei diese Berechtigungen besitzt und welcher Dienst davon abhängt.

Logische Verknüpfungen mit AND, OR und NOT

Mehrere Tests werden standardmäßig mit einem logischen AND verknüpft. Der folgende Befehl wählt nur Einträge aus, die sowohl reguläre Dateien sind als auch auf `.log` enden:

```
find /var/log -type f -name "*.log"
```

Das ausgeschriebene `-a` ist möglich, aber normalerweise nicht erforderlich:

```
find /var/log -type f -a -name "*.log"
```

Für eine OR-Verknüpfung verwendest du `-o`:

```
find /var/log \( -name "*.log" -o -name "*.err" \)
```

Die Klammern gruppieren den Teilausdruck. Sie müssen vor der Shell geschützt werden, beispielsweise mit einem Backslash. Wenn nur reguläre Dateien mit einer der beiden Endungen

gesucht werden sollen, lautet der vollständige Ausdruck:

```
find /var/log -type f \( -name "*.log" -o -name "*.err" \) -print
```

Eine Bedingung negierst du mit `!`:

```
find /etc -type f ! -name "*.conf"
```

Damit findest du unter `/etc` reguläre Dateien, deren Name nicht auf `.conf` endet.

Die Operatoren haben unterschiedliche Prioritäten:

1. Klammern gruppieren Ausdrücke ausdrücklich.
2. `!` negiert den folgenden Ausdruck.
3. AND beziehungsweise das implizite `-a` wird vor OR ausgewertet.
4. `-o` bildet die OR-Verknüpfung.

-a (AND) | Beide Bedingungen müssen zutreffen (Standardverhalten bei Aneinanderreihung)

-o (OR) | Mindestens eine der Bedingungen muss zutreffen

! oder -not | Negiert das nachfolgende Kriterium

\(... \) | Klammert Ausdrücke zur Steuerung der Bindungsreihenfolge

Ohne Klammern können Ausdrücke anders ausgewertet werden als beim ersten Lesen erwartet:

```
find /var/log -type f -name "*.log" -o -name "*.err"
```

Dieser Ausdruck entspricht sinngemäß:

Ein Verzeichnis mit der Endung `.err` könnte deshalb ebenfalls als Treffer erscheinen. Gruppiere OR-Ausdrücke immer ausdrücklich, wenn weitere Bedingungen für alle Alternativen gelten sollen.

> **Typische Fehlerquelle:** AND bindet stärker als OR. Wenn mehrere Dateinamenmuster gemeinsam mit weiteren Bedingungen gelten sollen, gehören die OR-Bedingungen in geschützte Klammern.

Suchtiefe und Dateisystemgrenzen

`find` durchsucht Unterverzeichnisse standardmäßig ohne festgelegte Tiefengrenze. Mit `-maxdepth` begrenzt du, wie weit der Befehl unterhalb des Startpfades absteigt:

```
find /etc -maxdepth 2 -type f -name "*.conf"
```

Der Startpfad selbst besitzt die Tiefe 0. Direkt darin enthaltene Einträge liegen auf Tiefe 1.

Mit `-mindepth` legst du fest, ab welcher Tiefe Einträge geprüft werden:

```
find /tmp/testdaten -mindepth 1 -maxdepth 1 -print
```

Damit wird der Startpfad `/tmp/testdaten` selbst nicht ausgegeben. Berücksichtigt werden nur seine direkt enthaltenen Einträge.

Die Option `-xdev` verhindert, dass `find` in Verzeichnisse auf anderen Dateisystemen wechselt:

```
find / -xdev -type f -size +1G
```

Die Suche bleibt damit auf dem Dateisystem des Startpfades. Andere eingehängte Dateisysteme müssen bei Bedarf als eigene Startpfade angegeben werden.

`-prune` schließt bestimmte Verzeichnisbäume von der Traversierung aus. Das folgende Beispiel überspringt `/var/cache`:

```
find /var -path /var/cache -prune -o -type f -name "*.log" -print
```

Trifft `find` auf `/var/cache`, verhindert `-prune` das Hinabsteigen in dieses Verzeichnis. Für alle anderen Einträge wird die rechte Seite des OR-Ausdrucks ausgewertet.

> **Tipp:** Verwende einen möglichst konkreten Startpfad. `-xdev`, `-maxdepth` und `-prune` helfen bei der Eingrenzung, ersetzen aber keine vorherige Überlegung darüber, welcher Teil des Dateisystems tatsächlich relevant ist.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Symbolische Links werden standardmäßig nicht verfolgt. GNU `find` verwendet ohne weitere Angabe das Verhalten von `-P`. Mit `-L` folgt der Befehl symbolischen Links und prüft die Eigenschaften ihrer Ziele:

```
find -L /srv -type f -name "*.conf"
```

Das kann zusätzliche Verzeichnisbäume in die Suche einbeziehen. Außerdem verändert `-L`, worauf sich Tests wie `-type` beziehen.

> ⚠ **Warnung:** Verwende `-L` nur bewusst. Ein symbolischer Link kann auf einen großen oder unerwarteten Verzeichnisbaum zeigen. In Verbindung mit verändernden Aktionen entstehen zudem schwer überschaubare Folgen.

Aktionen mit `-print`, `-exec`, `-ok` und `-delete`

Tests bestimmen, welche Einträge passen. Aktionen legen fest, was anschließend mit diesen Treffern geschieht.

Die grundlegende Ausgabeaktion ist `-print`:

```
find /etc -type f -name "*.conf" -print
```

`-ls` gibt zusätzliche Metadaten aus:

```
find /etc -type f -name "*.conf" -ls
```

GNU `find` bietet mit `-printf` eine formatierbare Ausgabe. Das folgende Beispiel zeigt die Größe in Byte und den vollständigen Pfad:

```
find /var/log -type f -printf "%s %p\n"
```

Mit `-exec` übergibst du gefundene Pfade an einen weiteren Befehl. Die Zeichenfolge `{}` wird durch den jeweiligen Pfad ersetzt.

```
find /etc -type f -name "*.conf" -exec file {} \;
```

Die Schreibweise `\;` beendet den auszuführenden Befehl. In dieser Form wird `file` für jeden Treffer einzeln aufgerufen.

Effizienter ist häufig die Variante mit `+`:

```
find /etc -type f -name "*.conf" -exec file {} +
```

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Dabei sammelt `find` mehrere Pfade und übergibt sie gemeinsam in möglichst wenigen Aufrufen. Die Argumentlisten werden so aufgeteilt, dass die vom System vorgegebene maximale Größe nicht überschritten wird.

Einzelaufruf (-exec befehl {} \;)

Prozessaufrufe
: Startet für jede gefundene Datei einen eigenen Kindprozess

Performance
: Hoher Fork-Overhead bei vielen Treffern

Exitcode
: Gibt Fehler des letzten Einzelaufrufs zurück

Anwendungsfall
: Wenn Argumente zwingend einzeln übergeben werden müssen

Gebündelter Aufruf (-exec befehl {} +)

Prozessaufrufe
: Bündelt tausende Treffer in möglichst wenige Prozessaufrufe

Performance
: Maximal performant und ressourcenschonend (analog zu xargs)

Exitcode
: Schlägt fehl, sobald ein Teilaufruf einen Fehlercode liefert

Anwendungsfall
: Standard für Batch-Verarbeitung, grep, chown oder chmod

Gemeinsam: {} dient in beiden Varianten als Platzhalter für gefundene Pfadnamen

Für Befehle, die mehrere Dateinamen gleichzeitig akzeptieren, ist `{} +` meist die bessere Wahl. Wenn jeder Treffer einzeln verarbeitet werden muss, bleibt `{} \;` erforderlich.

`-ok` arbeitet ähnlich wie `-exec`, verlangt jedoch vor jedem Aufruf eine Bestätigung:

```
find /tmp/testdaten -type f -name "*.tmp" -ok rm -- {} \;
```

Das kann bei wenigen Treffern hilfreich sein, eignet sich aber nicht für unbeaufsichtigte Skripte.

Mit `-delete` löscht GNU `find` passende Dateien und leere Verzeichnisse direkt. Vor dem Löschen solltest du exakt denselben Ausdruck zunächst mit `-print` prüfen:

```
find /tmp/testdaten -type f -name "*.tmp" -mtime +7 -print
```

Erst wenn die Ausgabe stimmt, wird die Aktion ersetzt:

```
find /tmp/testdaten -type f -name "*.tmp" -mtime +7 -delete
```

`-delete` aktiviert automatisch eine tiefenweise Traversierung. Inhalte eines Verzeichnisses werden damit vor dem Verzeichnis selbst verarbeitet. Diese Eigenschaft macht die Kombination mit `-prune` ungeeignet, weil `-prune` bei aktivem `-depth` keine Wirkung hat.

> ⚠ **Warnung:** `-delete` besitzt keine Rückfrage und keinen Papierkorb. Kontrolliere Startpfad, Tests und Trefferliste unmittelbar vor dem Löschen. Übernimm nicht lediglich einen ähnlichen Befehl aus dem Verlauf deiner Shell.

Bei `-exec` solltest du Pfade nicht in eine ungeschützte Shell-Zeichenkette einsetzen. Diese Konstruktion ist fehleranfällig:

```
find /tmp/testdaten -type f -exec sh -c "echo {}" \;
```

Der gefundene Dateiname wird Bestandteil des Shell-Codes. Sonderzeichen im Namen können dadurch anders interpretiert werden als beabsichtigt. Übergib Pfade stattdessen als Argumente:

```
find /tmp/testdaten -type f -exec sh -c 'for path do printf "%s\n" "$path"; done' sh {} +
```

Innerhalb der Shell stehen die gefundenen Pfade anschließend als korrekt getrennte Argumente zur Verfügung.

Praktische Beispiele für die Systemadministration

Praxis-Szenario: Große Dateien auf einem Dateisystem finden

```
find /var -xdev -type f -size +500M -print
```

Der Befehl sucht unter `/var` nach regulären Dateien mit mehr als 500 Mebibyte. Durch `-xdev` werden darunter eingehängte andere Dateisysteme nicht betreten.

Für eine nach Größe sortierbare Ausgabe kannst du `-printf` verwenden:

```
find /var -xdev -type f -size +500M -printf "%s %p\n" | sort -n
```

Praxis-Szenario: Kürzlich geänderte Konfigurationen suchen

```
find /etc -type f -mmin -1440 -print
```

Damit findest du reguläre Dateien, deren Inhalt innerhalb der letzten 1440 Minuten verändert wurde. Das kann nach einem Update oder einer manuellen Konfigurationsänderung bei der Eingrenzung helfen.

Praxis-Szenario: Dateien ohne bekannten Eigentümer oder bekannte Gruppe finden

```
find /home -xdev \( -nouser -o -nogroup \) -ls
```

Die Ausgabe zeigt Dateien, deren numerische Benutzer- oder Gruppen-ID keinem aktuellen Konto beziehungsweise keiner aktuellen Gruppe zugeordnet werden kann.

Praxis-Szenario: Konfigurationsdateien nach einer Direktive durchsuchen

```
find /etc -type f -name "*.conf" -exec grep -HnF "Listen" {} +
```

`find` wählt reguläre Konfigurationsdateien aus. `grep` durchsucht deren Inhalte nach der festen Zeichenkette `Listen` und gibt Dateinamen sowie Zeilennummern aus.

Praxis-Szenario: Leere Dateien in einem temporären Verzeichnis anzeigen

```
find /tmp/testdaten -type f -empty -print
```

Auch hier wird zunächst ausschließlich ausgegeben. Ob leere Dateien tatsächlich fehlerhaft oder entbehrlich sind, hängt von der jeweiligen Anwendung ab.

> **Typische Fehlerquelle:** Ein Treffer ist kein Beweis dafür, dass eine Datei gelöscht oder verändert werden darf. `find` prüft nur die angegebenen Kriterien. Die fachliche Bedeutung einer Datei kennt der Befehl nicht.

locate und plocate – Suche über einen Dateiindex

`locate` verfolgt einen anderen Ansatz als `find`. Statt bei jeder Suche das Dateisystem zu durchlaufen, liest der Befehl aus einer zuvor erzeugten Datenbank. Diese Datenbank enthält Pfade, die während eines früheren Scanlaufs erfasst wurden.

Dadurch eignet sich `locate` besonders für schnelle Suchen nach bekannten Datei- oder Verzeichnisnamen. Die höhere Geschwindigkeit hat jedoch einen Preis: Das Ergebnis beschreibt den Stand der Datenbank und nicht zwingend den aktuellen Zustand des Dateisystems.

Auf vielen aktuellen Linux-Systemen wird der Befehl `locate` durch `plocate` bereitgestellt. Andere Systeme verwenden weiterhin `mlocate`, GNU `locate` oder eine weitere Implementierung. Die grundlegende Arbeitsweise ist ähnlich, einzelne Optionen, Standardpfade und das Verhalten bei mehreren Suchmustern können sich jedoch unterscheiden.

Mit folgenden Befehlen prüfst du, welche Implementierung auf deinem System vorhanden ist:

```
command -v locate
locate --version
```

Funktionsweise der Suchdatenbank

Die Datenbank wird mit `updatedb` erzeugt oder aktualisiert. Dabei durchsucht `updatedb` die dafür vorgesehenen Bereiche des Dateisystems und speichert die gefundenen Pfade in einem Index.

Eine spätere Suche mit `locate` liest diesen Index:

find (Dateisystem-Scan)

Datenquelle
: Durchsucht direkt das Live-Dateisystem

Aktualität
: 100% Echtzeit, findet sekundenalte Änderungen

Geschwindigkeit
: Abhängig von Plattengröße und I/O

Berechtigungen
: Prüft Zugriffsrechte beim Durchlaufen

locate / plocate (Datenbanksuche)

Datenquelle
: Liest die vorberechnete Indexdatenbank (/var/lib/plocate)

Aktualität
: Abhängig vom letzten updatedb-Lauf (oft nächtlicher Timer)

Geschwindigkeit
: Nahezu instantan (Millisekunden)

Berechtigungen
: plocate filtert Treffer nach Rechten des aufrufenden Benutzers

Gemeinsam: Beide Werkzeuge unterstützen Wildcards und reguläre Ausdrücke zur Pfadsuche

`locate` öffnet bei einer normalen Suche nicht jeden Verzeichnisbaum erneut. Deshalb kann der Befehl auch auf Systemen mit vielen Dateien schnell Ergebnisse liefern. Die Suche ist allerdings auf die Pfade beschränkt, die beim letzten Lauf von `updatedb` erfasst wurden.

Ein einfaches Beispiel:

```
locate sshd_config
```

Bei `plocate` wird ein Suchmuster ohne Platzhalter standardmäßig als Teilzeichenkette behandelt. Der Befehl kann daher neben `/etc/ssh/sshd_config` auch Sicherungskopien oder andere Pfade ausgeben, die `sshd_config` enthalten.

> **Tipp:** Verwende `locate` für eine schnelle Orientierung. Prüfe einen gefundenen Pfad anschließend mit Werkzeugen wie `ls`, `stat` oder `test`, wenn seine aktuelle Existenz oder seine Eigenschaften für den nächsten Arbeitsschritt entscheidend sind.

Datenbank mit `updatedb` aktualisieren

Auf Systemen mit `plocate` wird die systemweite Datenbank üblicherweise regelmäßig durch einen systemd-Timer aktualisiert. Andere Distributionen oder Locate-Implementierungen können dafür einen Cronjob oder einen anderen Mechanismus verwenden.

Ob eine automatische Aktualisierung eingerichtet ist, kannst du unter systemd mit einer Suche in den vorhandenen Timern prüfen:

```
systemctl list-timers --all | grep -E "locate|updatedb"
```

Wenn eine neu angelegte Datei sofort im Index erscheinen soll, kann die systemweite Datenbank manuell aktualisiert werden:

```
sudo updatedb
```

Administrative Rechte sind bei der systemweiten Datenbank erforderlich, damit `updatedb` die vorgesehenen Verzeichnisbäume lesen und den Index am konfigurierten Ort schreiben kann. Für eine gewöhnliche Suche mit `locate` sind dagegen normalerweise keine Root-Rechte erforderlich.

1. **Dateisystem-Änderung erfolgt**Neue Datei wird angelegt; locate findet sie noch nicht
2. **updatedb-Lauf (systemd-timer oder manuell)**Liest das Dateisystem ein und aktualisiert die Indexdatenbank
3. **Index auf aktuellem Stand**locate liefert die neue Datei sofort in den Suchergebnissen

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Der Ort der Datenbank hängt von der verwendeten Implementierung und der Distribution ab. Pfade wie `/var/lib/plocate/plocate.db` oder `/var/lib/mlocate/mlocate.db` dürfen deshalb nicht als allgemeingültig vorausgesetzt werden.

> ⚠ **Warnung:** Starte `updatedb` nicht unnötig in kurzen Abständen. Das Erstellen beziehungsweise Aktualisieren des Indexes muss Verzeichnisstrukturen lesen und erzeugt entsprechende I/O-Last. Auf einem korrekt eingerichteten System übernimmt diesen Vorgang bereits ein Timer oder Cronjob.

Konfiguration von `updatedb`

Bei `plocate` wird die systemweite Konfiguration üblicherweise aus `/etc/updatedb.conf` gelesen. Darüber lässt sich festlegen, welche Dateisysteme, Verzeichnisse und Pfade beim Aufbau der Datenbank übersprungen werden.

Wichtige Einstellungen sind:

Einstellung	Bedeutung
`PRUNEFS`	Dateisystemtypen, die nicht durchsucht werden
`PRUNENAMES`	Verzeichnisnamen, deren Inhalte nicht erfasst werden
`PRUNEPATHS`	konkrete Verzeichnispfade, deren Inhalte nicht erfasst werden
`PRUNE_BIND_MOUNTS`	legt fest, ob Bind-Mounts übersprungen werden

Die genaue Syntax und das unterstützte Verhalten hängen von der verwendeten `updatedb`-Implementierung ab. Bei `plocate` enthält `PRUNENAMES` beispielsweise Verzeichnisnamen und keine Shell-Glob-Muster. Ein Eintrag wie `*.tmp` schließt deshalb nicht allgemein alle temporären Dateien aus.

Vorhandene Werte sollten nicht ungeprüft durch eine eigene verkürzte Liste ersetzt werden. Distributionen tragen dort häufig bereits virtuelle Dateisysteme, temporäre Bereiche oder andere ungeeignete Suchziele ein.

```
grep -Ev "^[[:space:]]*(#|$)" /etc/updatedb.conf
```

Der Befehl zeigt die wirksamen, nicht auskommentierten Zeilen der Konfiguration an. Er verändert die Datei nicht.

> **Typische Fehlerquelle:** Änderungen an `/etc/updatedb.conf` wirken sich erst auf einen danach erzeugten Index aus. Eine geänderte Ausschlussliste verändert keine bereits bestehende Datenbank rückwirkend.

Suchmuster und wichtige Optionen

Die folgenden Beispiele orientieren sich an `plocate`. Prüfe bei einer anderen Implementierung die lokale Manpage mit `man locate`.

Eine einfache Suche nach einer Teilzeichenkette:

```
locate sshd_config
```

Groß- und Kleinschreibung ignorierst du mit `-i`:

```
locate -i readme.md
```

Mit `-b` beziehungsweise `--basename` wird das Suchmuster nur mit dem letzten Bestandteil des Pfades verglichen:

```
locate -b sshd_config
```

`-b` erzwingt keine exakte Übereinstimmung. Auch ein Basisname wie `sshd_config.backup` kann zum Suchmuster passen.

Mit `-e` beziehungsweise `--existing` prüft `plocate`, ob ein Datenbankeintrag zum Zeitpunkt der Ausgabe noch im Dateisystem existiert:

```
locate -e sshd_config
```

Diese Prüfung kann veraltete Treffer nach gelöschten oder verschobenen Dateien unterdrücken. Sie ergänzt jedoch lediglich eine Existenzprüfung und aktualisiert die Datenbank nicht.

Die Anzahl ausgegebener Treffer begrenzt du mit `-l`:

```
locate -l 20 "*.service"
```

Das Suchmuster steht in Anführungszeichen, damit es nicht durch die Shell expandiert wird.

Mit `-c` gibt `plocate` nur die Anzahl der passenden Datenbankeinträge aus:

```
locate -c "*.conf"
```

Für reguläre Ausdrücke bietet `plocate` unter anderem `--regex`. Ein vollständiger Pfad, der auf `/sshd_config` endet, lässt sich damit so suchen:

```
locate --regex "/sshd_config$"
```

Wichtige Optionen im Überblick:

Option	Bedeutung
`-i`, `--ignore-case`	Groß- und Kleinschreibung ignorieren
`-b`, `--basename`	nur den Basisnamen prüfen
`-w`, `--wholename`	den vollständigen Pfad prüfen; bei `plocate` Standard
`-e`, `--existing`	nur noch vorhandene Pfade ausgeben
`-l N`, `--limit N`	Ausgabe auf `N` Treffer begrenzen
`-c`, `--count`	nur die Anzahl der Treffer ausgeben
`-0`, `--null`	Treffer mit NUL-Zeichen statt Zeilenumbruch trennen
`-S`, `--statistics`	Informationen über die verwendete Datenbank anzeigen

`-S` steht bei `plocate` und GNU `locate` für Statistiken. Die Option sortiert keine Suchergebnisse:

```
locate -S
```

Format und Inhalt der Ausgabe sind implementationsabhängig.

> **Typische Fehlerquelle:** `locate` sucht standardmäßig nicht unabhängig von Groß- und Kleinschreibung. Wenn die Schreibweise unbekannt ist, musst du `-i` ausdrücklich angeben.

Veraltete Treffer und implementationsabhängige Unterschiede

Zwischen zwei Aktualisierungen können drei Arten von Abweichungen auftreten:

- Eine neu angelegte Datei fehlt noch in der Datenbank.
- Eine gelöschte Datei steht weiterhin im Index.
- Eine verschobene oder umbenannte Datei erscheint unter ihrem alten Pfad und fehlt unter dem neuen.

Bei `plocate` reduziert `-e` veraltete Treffer, indem vor der Ausgabe geprüft wird, ob der Pfad noch existiert:

```
locate -e -b sshd_config
```

Wenn du den aktuellen Zustand des Dateisystems vollständig prüfen musst, bleibt `find` das geeignete Werkzeug:

```
find /etc -type f -name "sshd_config"
```

Auch das Verhalten bei mehreren Suchmustern ist nicht über alle Implementierungen hinweg identisch. `plocate` verlangt standardmäßig, dass ein Pfad zu allen angegebenen Mustern passt. Andere Locate-Varianten können mehrere Muster als Alternativen behandeln. Für portable Skripte solltest du dich deshalb nicht auf ein ungeprüftes implementationsspezifisches Verhalten verlassen.

Die Datenbank kann zudem Verzeichnisse bewusst auslassen. Ein fehlender Treffer bedeutet deshalb nicht automatisch, dass die gesuchte Datei auf dem System nicht existiert. Möglicherweise wurde ihr Dateisystem oder Verzeichnis beim Aufbau des Indexes ausgeschlossen.

Sichtbarkeit und Berechtigungen

Eine systemweite Pfaddatenbank enthält grundsätzlich Informationen über zahlreiche Dateien. Moderne Implementierungen wie `plocate` berücksichtigen deshalb bei korrekt eingerichteter Datenbank die Sichtbarkeit für den aufrufenden Benutzer. Ein Pfad soll nur ausgegeben werden, wenn der Benutzer die dafür erforderlichen übergeordneten Verzeichnisse lesen beziehungsweise betreten darf.

Dieses Verhalten hängt von der Eigentümerschaft, den Berechtigungen und den beim Aufbau der Datenbank verwendeten Einstellungen ab. Eine selbst erzeugte Datenbank mit zu weit gefassten Leserechten kann Dateinamen offenlegen, die andere Benutzer normalerweise nicht sehen könnten.

> ⚠ **Warnung:** Behandle eine Locate-Datenbank nicht als harmlose Cache-Datei. Sie enthält eine Übersicht gespeicherter Pfade und kann dadurch vertrauliche Dateinamen oder Verzeichnisstrukturen offenlegen. Achte bei eigenen Datenbanken auf restriktive Berechtigungen.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Eine private Datenbank für einen begrenzten Verzeichnisbaum lässt sich mit `plocate` beispielsweise so erzeugen:

```
updatedb -l 0 -U "$HOME/projekte" -o "$HOME/projekte.db"
```

Die Datenbank wird anschließend ausdrücklich für die Suche angegeben:

```
locate -d "$HOME/projekte.db" suchmuster
```

Alle Benutzer, die diese Datenbank lesen können, können daraus die gespeicherten Pfade ermitteln. Die Datei sollte deshalb nicht allgemein lesbar sein:

```
chmod 600 "$HOME/projekte.db"
```

`locate` und `find` im Vergleich

Eigenschaft	**`locate` beziehungsweise `plocate`**	**`find`**
Datenquelle	zuvor erzeugte Pfaddatenbank	aktuelles Dateisystem
Aktualität	abhängig vom letzten `updatedb`-Lauf	Zustand während der Suche
Suchkriterien	hauptsächlich Pfad- und Namensmuster	Name, Typ, Größe, Zeiten, Rechte und weitere Metadaten

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Eigenschaft	`locate` beziehungsweise `plocate`	`find`
Geschwindigkeit	bei geeigneten Mustern meist sehr schnell	abhängig von Umfang und Struktur des Suchbereichs
nicht indexierte Bereiche	werden nicht gefunden	können bei ausreichenden Rechten durchsucht werden
Aktionen auf Treffern	Ausgabe zur Weiterverarbeitung	integrierte Aktionen wie `-exec` und `-delete`
typischer Einsatz	schnelle Orientierung nach bekanntem Namen	präzise und aktuelle Dateisystemsuche

Beide Werkzeuge haben damit einen klaren Platz. `locate` beantwortet schnell die Frage, wo ein bestimmter Name laut Index vorkommt. `find` prüft, welche Einträge im angegebenen Bereich jetzt tatsächlich vorhanden sind und den geforderten Eigenschaften entsprechen.

□□ Praktisches Beispiel:

Du suchst die Konfigurationsdatei des SSH-Servers. Für eine schnelle Orientierung verwendest du zunächst:

```
locate -e -b sshd_config
```

Bleibt die Suche ohne Ergebnis oder benötigst du einen aktuellen und auf `/etc` begrenzten Befund, prüfst du direkt das Dateisystem:

```
find /etc -type f -name "sshd_config" -print
```

> **Tipp:** Beginne mit `locate`, wenn du einen Namen kennst und schnell mögliche Pfade sehen möchtest. Verwende `find`, wenn Aktualität, vollständige Kontrolle über den Suchbereich oder zusätzliche Dateieigenschaften entscheidend sind.

Weitere Details findest du in den lokalen Manpages mit `man locate`, `man updatedb` und `man updatedb.conf` sowie in den plocate-Manpages.

grep – Inhalte in Dateien durchsuchen

`grep` durchsucht Textdateien oder Datenströme zeilenweise nach einem Muster. Passt das Muster zu einem Teil einer Zeile, gibt `grep` standardmäßig die vollständige Zeile aus.

Damit unterscheidet sich `grep` grundlegend von `find`: `find` wählt Dateien anhand ihrer Eigenschaften aus, während `grep` die Inhalte dieser Dateien untersucht. Beide Werkzeuge lassen sich später kombinieren, sollten aber zunächst getrennt verstanden werden.

Grundlegende Syntax und einfache Suchmuster

Die grundlegende Syntax lautet:

```
grep [Optionen] Muster [Datei...]
```

Eine einfache Suche in einer Datei:

```
grep "error" anwendung.log
```

Der Befehl gibt alle Zeilen aus `anwendung.log` aus, die zur Zeichenfolge `error` passen. Groß- und Kleinschreibung werden dabei unterschieden. `Error` oder `ERROR` sind deshalb keine Treffer.

Mehrere Dateien können gemeinsam durchsucht werden:

```
grep "error" anwendung.log system.log
```

Wenn mehrere Dateien angegeben sind, stellt `grep` den Dateinamen standardmäßig vor jede ausgegebene Zeile. Ohne Dateiangabe liest `grep` von der Standardeingabe.

Dadurch lässt sich der Befehl direkt in einer Pipe verwenden:

```
journalctl -u ssh.service | grep "Failed"
```

`journalctl` schreibt seine Ausgabe auf `stdout`. Die Pipe übergibt diesen Datenstrom an `grep`, das nur passende Zeilen ausgibt.

1. **Datei oder Eingabestrom einlesen**grep verarbeitet Zeile für Zeile aus Dateien oder stdin
2. **Musterprüfung gegen regulären Ausdruck**Zeichenkette gegen definierte Suchmuster (BRE/ERE) vergleichen
3. **Filterentscheidung**Passende Zeilen selektieren (-v invertiert die Bedingung)
4. **Ausgabe formatieren**Trefferzeile, Dateiname (-H) oder Zeilennummer (-n) auf stdout ausgeben

Das Muster sollte in Anführungszeichen stehen. Dadurch verhinderst du, dass Leerzeichen oder für die Shell bedeutungsvolle Zeichen vor dem Start von `grep` verarbeitet werden.

Beginnt das Suchmuster mit einem Bindestrich, kann `grep` es mit einer Option verwechseln. Mit `--` beendest du die Optionsauswertung:

```
grep -F -- "-Xms" jvm.options
```

Feste Zeichenketten mit `grep -F`

`grep` interpretiert sein Suchmuster standardmäßig als regulären Ausdruck. Zeichen wie `.`, `*`, `[` oder `^` können deshalb eine besondere Bedeutung besitzen.

Wenn du eine feste Zeichenkette suchst, verwendest du `-F` beziehungsweise `--fixed-strings`:

```
grep -F "server.name=web01.example.net" anwendung.conf
```

Der Punkt wird mit `-F` als gewöhnliches Zeichen behandelt. Ohne `-F` steht ein Punkt in einem regulären Ausdruck für ein beliebiges Zeichen.

Ein weiteres Beispiel:

```
grep -F "[error]" anwendung.log
```

Mit `-F` sucht `grep` tatsächlich nach den Zeichen `[error]`. Ohne diese Option würde der Ausdruck als Zeichenklasse interpretiert.

> **Tipp:** Verwende `grep -F`, wenn du nach einem bekannten Wort, Pfad, Konfigurationswert oder einer vollständigen Fehlermeldung suchst und keine regulären Ausdrücke benötigst. Das macht die Absicht des Befehls eindeutig und verhindert überraschende Treffer durch Metazeichen.

Mehrere feste Zeichenketten lassen sich mit mehreren `-e`-Optionen angeben:

```
grep -F -e "ERROR" -e "WARNING" anwendung.log
```

Die Zeile wird ausgewählt, wenn mindestens eines der Muster passt.

Muster können auch aus einer Datei gelesen werden:

```
grep -F -f suchmuster.txt anwendung.log
```

Dabei interpretiert `grep` jede Zeile aus `suchmuster.txt` als eigenes Suchmuster.

Groß- und Kleinschreibung, Zeilennummern und Kontext

Mit `-i` ignoriert `grep` die Groß- und Kleinschreibung:

```
grep -i "error" anwendung.log
```

Damit passen unter anderem `error`, `Error` und `ERROR`.

`-n` ergänzt die Zeilennummer:

```
grep -nF "PermitRootLogin" /etc/ssh/sshd_config
```

Eine mögliche Ausgabe sieht so aus:

```
33:PermitRootLogin no
```

Mit `-H` erzwingst du die Ausgabe des Dateinamens:

```
grep -HnF "PermitRootLogin" /etc/ssh/sshd_config
```

`-h` unterdrückt den Dateinamen, wenn mehrere Dateien durchsucht werden:

```
grep -hF "ERROR" app1.log app2.log
```

Häufig reicht eine einzelne Trefferzeile nicht aus, um einen Logeintrag zu beurteilen. Für umgebende Zeilen stehen folgende Optionen zur Verfügung:

Option	Bedeutung
`-A N`	`N` Zeilen nach dem Treffer ausgeben
`-B N`	`N` Zeilen vor dem Treffer ausgeben
`-C N`	jeweils `N` Zeilen vor und nach dem Treffer ausgeben

Drei Zeilen vor und nach einer Fehlermeldung:

```
grep -C 3 "connection refused" anwendung.log
```

Nur die folgenden fünf Zeilen:

```
grep -A 5 "Traceback" anwendung.log
```

Wenn mehrere Trefferbereiche voneinander getrennt sind, kennzeichnet GNU `grep` die Trennung standardmäßig mit `--`.

> **Tipp:** Kontextzeilen sind bei Logs oft aussagekräftiger als eine isolierte Fehlermeldung. Prüfe insbesondere, welche Meldungen unmittelbar vor dem Fehler ausgegeben wurden. Dort steht nicht selten die eigentliche Ursache, während die letzte Zeile nur noch den Zusammenbruch dokumentiert.

Trefferzeilen, Textstellen und Dateinamen unterscheiden

Ohne besondere Ausgabeoptionen gibt `grep` vollständige passende Zeilen aus. Mehrere Optionen verändern, was als Ergebnis erscheint.

Mit `-o` beziehungsweise `--only-matching` gibt GNU `grep` nur den passenden Teil einer Zeile aus:

```
grep -oF "ERROR" anwendung.log
```

Enthält eine Zeile das Muster mehrfach, kann `-o` mehrere Ausgaben für dieselbe Eingabezeile erzeugen.

`-c` zählt dagegen passende Zeilen:

```
grep -cF "ERROR" anwendung.log
```

Der ausgegebene Wert ist nicht zwingend die Anzahl aller Vorkommen. Enthält eine Zeile das Wort `ERROR` dreimal, wird diese Zeile von `grep -c` trotzdem nur einmal gezählt.

Mit `-l` gibt `grep` ausschließlich die Namen der Dateien aus, die mindestens eine passende Zeile enthalten:

```
grep -lF "PermitRootLogin" /etc/ssh/sshd_config /etc/ssh/ssh_config
```

`-L` zeigt entsprechend nur Dateien ohne passenden Inhalt:

```
grep -LF "Managed by Ansible" /etc/*.conf
```

Mit `-q` unterdrückt `grep` die normale Ausgabe vollständig:

```
grep -qF "PermitRootLogin no" /etc/ssh/sshd_config
```

Diese Option ist für Bedingungen in Skripten vorgesehen. Das Ergebnis wird über den Exitcode ausgewertet.

Die wichtigsten Ausgabeoptionen im Überblick:

Option	Ausgabe
`-n`	passende Zeilen mit Zeilennummer
`-H`	Dateiname vor jeder passenden Zeile
`-h`	Dateinamen unterdrücken
`-o`	nur passende Teile einer Zeile
`-c`	Anzahl passender Zeilen

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Option	Ausgabe
`-l`	Namen von Dateien mit Treffern
`-L`	Namen von Dateien ohne Treffer
`-q`	keine normale Ausgabe

> **Typische Fehlerquelle:** `grep -c` zählt ausgewählte Zeilen und nicht allgemein alle einzelnen Treffer. Wenn du Vorkommen zählen möchtest, musst du die passenden Textteile zunächst mit `-o` ausgeben und anschließend zählen.

Ein einfaches Beispiel für die Anzahl einzelner Vorkommen:

```
grep -oF "ERROR" anwendung.log | wc -l
```

Dabei zählt `wc -l` die von `grep -o` ausgegebenen Trefferzeilen.

Treffer ausschließen

Mit `-v` kehrst du die Auswahl um. `grep` gibt dann alle Zeilen aus, die nicht zum Muster passen:

```
grep -vF "DEBUG" anwendung.log
```

Mehrere Ausschlüsse lassen sich kombinieren:

```
grep -vF -e "DEBUG" -e "TRACE" anwendung.log
```

Das ist beispielsweise nützlich, wenn ein Log überwiegend aus Debugmeldungen besteht und du zunächst nur die übrigen Einträge betrachten möchtest.

Leere Zeilen lassen sich mit einem regulären Ausdruck ausschließen:

```
grep -v "^$" anwendung.conf
```

Die genaue Bedeutung von `^` und `$` behandeln wir im Abschnitt zu regulären Ausdrücken.

Kommentare und leere Zeilen können gemeinsam ausgeblendet werden:

```
grep -Ev "^[[:space:]]*(#|$)" anwendung.conf
```

Dieser Ausdruck verwendet bereits erweiterte reguläre Ausdrücke. Er wird im nächsten Hauptabschnitt vollständig zerlegt.

Mehrere Dateien und Verzeichnisse durchsuchen

Mit `-r` durchsucht GNU `grep` Verzeichnisse rekursiv:

```
grep -rF "PermitRootLogin" /etc/ssh
```

Bei GNU `grep` folgt `-r` symbolischen Links nur dann, wenn sie ausdrücklich als Argument auf der Kommandozeile angegeben wurden. `-R` folgt dagegen auch symbolischen Links, die während der rekursiven Suche gefunden werden.

```
grep -RF "PermitRootLogin" /etc/ssh
```

`-R` kann dadurch weitere oder unerwartet große Verzeichnisbäume erreichen.

> ⚠ **Warnung:** Verwende `grep -R` nur, wenn symbolische Links bewusst verfolgt werden sollen. Ein Link kann auf einen anderen Mountpoint, einen umfangreichen Datenbestand oder zurück in einen bereits durchsuchten Bereich zeigen.

Mit `--include` beschränkst du die rekursive Suche auf passende Dateinamen:

```
grep -rF --include="*.conf" "PermitRootLogin" /etc
```

Bestimmte Dateinamen schließt du mit `--exclude` aus:

```
grep -rF --exclude="*.bak" "PermitRootLogin" /etc
```

Verzeichnisse lassen sich mit `--exclude-dir` überspringen:

```
grep -rF --exclude-dir=".git" "example.net" "$HOME/projekte"
```

Mehrere Muster können wiederholt angegeben werden:

```
grep -rF \
  --include="*.conf" \
  --exclude="*.bak" \
  --exclude-dir=".git" \
  "example.net" \
  "$HOME/projekte"
```

Für komplexere Auswahlkriterien ist häufig eine Kombination aus `find` und `grep` übersichtlicher. `grep -r` eignet sich vor allem dann, wenn die Auswahl nach Verzeichnis und Dateinamen genügt.

Binärdateien und ungewöhnliche Eingaben

Stellt GNU `grep` fest, dass eine Datei binäre Daten enthält, gibt es standardmäßig nicht unbedingt die passenden Zeilen aus. Stattdessen kann eine Meldung wie diese erscheinen:

```
Binary file firmware.bin matches
```

Mit `-I` behandelt `grep` Binärdateien so, als enthielten sie keine Treffer:

```
grep -rIF "example.net" /opt/anwendung
```

Mit `-a` beziehungsweise `--text` zwingst du `grep`, eine Binärdatei wie Text zu behandeln:

```
grep -aF "example.net" firmware.bin
```

Das kann bei bestimmten Dateiformaten hilfreich sein, birgt aber Risiken. Binärdaten können Steuerzeichen enthalten und die Terminalausgabe beeinträchtigen.

> ⚠ **Warnung:** Leite ungeprüfte Binärdaten nicht leichtfertig mit `grep -a` an ein interaktives Terminal weiter. Nutze bei unbekannten Dateien zunächst `file`, um den Dateityp zu bestimmen.

```
file firmware.bin
```

Komprimierte Dateien sind ebenfalls keine normalen Textdateien. Ein gewöhnliches `grep` dekomprimiert sie nicht automatisch. Die dafür vorgesehenen Werkzeuge wie `zgrep`, `bzgrep` und `xzgrep` behandeln wir in einem späteren Abschnitt.

Exitcodes und Verwendung in Skripten

`grep` unterscheidet drei grundlegende Ergebnisse über seinen Exitcode:

Exitcode	Bedeutung
`0`	mindestens eine Zeile wurde ausgewählt
`1`	keine Zeile wurde ausgewählt
`2`	ein Fehler ist aufgetreten

Damit ist ein fehlender Treffer kein technischer Fehler. Für Skripte ist diese Unterscheidung wichtig.

```
grep -qF "PermitRootLogin no" /etc/ssh/sshd_config
status=$?

case "$status" in
  0)
    printf '%s\n' "Direktive gefunden"
    ;;
  1)
    printf '%s\n' "Direktive nicht gefunden"
    ;;
  2)
    printf '%s\n' "Datei konnte nicht korrekt durchsucht werden" >&2
    ;;
esac
```

Die Option `-q` unterdrückt die Ausgabe, ändert aber nicht die grundlegende Bedeutung der Exitcodes. Eine Besonderheit bleibt: Sobald GNU `grep -q` einen Treffer gefunden hat, kann es

mit `0` enden, auch wenn beim Zugriff auf eine weitere Eingabedatei ein Fehler auftritt. Wenn alle Eingabefehler zuverlässig erkannt werden müssen, sollte `-q` daher nicht unkritisch über viele Dateien verwendet werden.

```
33:PermitRootLogin no
```

Nach einer Pipe enthält `$?` in der Shell standardmäßig nur den Exitcode des letzten Befehls. Die zuverlässige Auswertung mehrstufiger Pipelines behandeln wir später bei den kombinierten Workflows.

> **Typische Fehlerquelle:** Ein Exitcode von `1` bedeutet bei `grep` nicht, dass der Befehl fehlgeschlagen ist. Er bedeutet lediglich, dass keine passende Zeile gefunden wurde. Ein technischer Fehler wird mit `2` gemeldet.

Praktische Beispiele

Anwendungsfall: Eine SSH-Direktive mit Zeilennummer anzeigen

```
grep -HnF "PermitRootLogin" /etc/ssh/sshd_config
```

Anwendungsfall: Fehler unabhängig von der Schreibweise suchen

```
grep -inF "error" anwendung.log
```

Anwendungsfall: Nur Logdateien rekursiv durchsuchen

```
grep -rHnF --include="*.log" "connection refused" /var/log
```

Anwendungsfall: Kommentare und leere Zeilen einer Konfiguration ausblenden

```
grep -Ev "^[[:space:]]*(#|$)" anwendung.conf
```

Anwendungsfall: Dateien mit einer bestimmten Konfigurationsdirektive ermitteln

```
grep -rlF --include="*.conf" "server_name" /etc
```

Anwendungsfall: Prüfen, ob ein Fehlercode vorkommt

```
if grep -qF "AH01630" apache-error.log; then
  printf '%s\n' "Fehlercode AH01630 wurde gefunden"
else
  status=$?
  if [ "$status" -eq 1 ]; then
    printf '%s\n' "Fehlercode nicht gefunden"
  else
    printf '%s\n' "Logdatei konnte nicht gelesen werden" >&2
  fi
fi
```

> **Tipp:** Beginne mit `grep -F`, wenn du eine konkrete Zeichenkette suchst. Wechsle erst dann zu regulären Ausdrücken, wenn das Suchmuster tatsächlich variabel sein muss. Ein komplizierter Ausdruck ist kein Qualitätsmerkmal. Meist ist er nur kompliziert.

Reguläre Ausdrücke mit `grep` und `sed`

Eine feste Zeichenkette beschreibt einen konkreten Text. Ein regulärer Ausdruck beschreibt dagegen eine Menge möglicher Zeichenfolgen. Damit kannst du beispielsweise Zeilen auswählen, die mit einem Datum beginnen, einen numerischen Fehlercode enthalten oder mehreren zulässigen Schreibweisen entsprechen.

Reguläre Ausdrücke sind keine eigene Programmiersprache und verstehen keine fachliche Bedeutung. Ein Muster kann erkennen, dass eine Zeichenfolge wie eine IPv4-Adresse aufgebaut ist. Ob die einzelnen Werte tatsächlich zwischen 0 und 255 liegen, folgt daraus noch nicht.

Grundlegende und erweiterte reguläre Ausdrücke

POSIX unterscheidet zwei grundlegende Syntaxvarianten:

- **Basic Regular Expressions (BRE)** verwendet `grep` standardmäßig.
- **Extended Regular Expressions (ERE)** aktivierst du mit `grep -E`.

Beide Varianten können unter GNU `grep` im Wesentlichen dieselben Muster beschreiben. Der wichtigste Unterschied liegt in der Schreibweise bestimmter Metazeichen.

Funktion	BRE	ERE
Gruppierung	`\( … \)`	`( … )`
Alternative	`|`	`|`
ein- oder keinmal	`\?`	`?`
ein- oder mehrmals	`\+`	`+`
Wiederholungsbereich	`\{n,m\}`	`{n,m}`

Ein Muster für Zeilen, die mit `ERROR` oder `WARNING` beginnen, ist in BRE vergleichsweise unruhig:

```
grep '^\(ERROR\|WARNING\):' anwendung.log
```

Mit ERE wird derselbe Ausdruck lesbarer:

```
grep -E '^(ERROR|WARNING):' anwendung.log
```

Für neue Befehle ist `grep -E` meist die klarere Wahl, sobald Gruppierungen, Alternativen oder Wiederholungsbereiche benötigt werden. Der ältere Befehl `egrep` sollte dafür nicht mehr verwendet werden.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

> **Tipp:** Wähle BRE oder ERE bewusst und bleibe innerhalb eines Ausdrucks bei dieser Syntax. Viele vermeintlich komplizierte Regex-Fehler entstehen lediglich dadurch, dass Klammern oder Pluszeichen für die falsche Variante escaped wurden.

Literale, Metazeichen und Anker

Die meisten Zeichen stehen in einem regulären Ausdruck für sich selbst. Einige Zeichen besitzen eine besondere Funktion:

Zeichen	Bedeutung
`.`	ein beliebiges Zeichen
`^`	Anfang einer Zeile
`$`	Ende einer Zeile
`*`	vorherigen Ausdruck keinmal oder beliebig oft wiederholen
`[ … ]`	eines der enthaltenen Zeichen
`[^ … ]`	ein Zeichen, das nicht enthalten ist
`\`	Bedeutung des folgenden Zeichens verändern

Das Muster `error` passt überall innerhalb einer Zeile, an der diese Zeichenfolge vorkommt:

```
grep 'error' anwendung.log
```

Soll die vollständige Zeile ausschließlich aus `error` bestehen, werden Anfang und Ende verankert:

```
grep '^error$' anwendung.log
```

Nur Zeilen, die mit `error` beginnen:

```
grep '^error' anwendung.log
```

Nur Zeilen, die mit `error` enden:

```
grep 'error$' anwendung.log
```

Ein Punkt ist ohne Escaping ein Metazeichen. Das Muster `web.01` passt daher nicht nur zu `web.01`, sondern beispielsweise auch zu `web-01` oder `webX01`.

Für einen tatsächlichen Punkt muss das Zeichen escaped werden:

```
grep 'web\.01' hosts.txt
```

```
Binary file firmware.bin matches
```

Mit `grep -E` passt dieses Muster beispielsweise zu `web01.example.net`, aber nicht zu `db01.example.net` oder `web1.example.net`.

```
grep -E '^web[[:digit:]]{2}\.example\.net$' hosts.txt
```

Zeichenklassen und Klammerausdrücke

Ein Klammerausdruck beschreibt ein einzelnes Zeichen aus einer vorgegebenen Menge:

```
grep -E 'gr[ae]y' datei.txt
```

Das Muster passt zu `gray` und `grey`.

Ein Bereich lässt sich mit einem Bindestrich angeben:

```
grep -E '[0-9]' datei.txt
```

Für portable Muster sind POSIX-Zeichenklassen vorzuziehen. Sie berücksichtigen die aktive Locale und machen die beabsichtigte Zeichenart deutlicher:

Zeichenklasse	Bedeutung
`[[:digit:]]`	Ziffer
`[[:alpha:]]`	alphabetisches Zeichen
`[[:alnum:]]`	Buchstabe oder Ziffer
`[[:lower:]]`	Kleinbuchstabe
`[[:upper:]]`	Großbuchstabe
`[[:space:]]`	Leerraum einschließlich Tabulator und Zeilenraum
`[[:blank:]]`	Leerzeichen oder horizontaler Tabulator
`[[:xdigit:]]`	hexadezimale Ziffer
`[[:punct:]]`	Satz- und Sonderzeichen

Die doppelten eckigen Klammern gehören zur Syntax:

```
grep -E '[[:digit:]]{4}' datei.txt
```

Eine negierte Zeichenklasse beginnt innerhalb der Klammern mit `^`:

```
grep -E '[^[:digit:]]' datei.txt
```

Das Muster passt zu einem Zeichen, das keine Ziffer ist.

Die Auswertung von Zeichenklassen und Bereichen kann von der eingestellten Locale abhängen. Wenn ein Skript ausdrücklich byteorientiert und mit ASCII-ähnlicher Sortierung

arbeiten soll, kann die Locale für den einzelnen Aufruf festgelegt werden:

```
LC_ALL=C grep -E '^[A-Z]+$' datei.txt
```

> **Typische Fehlerquelle:** `[0-9]` und `[[:digit:]]` sehen ähnlich aus, drücken aber nicht unter allen Locales zwingend dieselbe Zeichenmenge aus. Verwende POSIX-Zeichenklassen, wenn du die Zeichenart und nicht nur einen sichtbaren Bereich beschreiben möchtest.

Wiederholungen und Quantifizierer

Ein Quantifizierer bezieht sich immer auf den unmittelbar vorhergehenden Ausdruck. Das kann ein einzelnes Zeichen, eine Zeichenklasse oder eine gruppierte Teilstruktur sein.

In ERE stehen folgende Quantifizierer zur Verfügung:

Quantifizierer	Bedeutung
`*`	keinmal oder beliebig oft
`+`	einmal oder öfter
`?`	keinmal oder einmal
`{n}`	genau `n` Wiederholungen
`{n,}`	mindestens `n` Wiederholungen
`{n,m}`	zwischen `n` und `m` Wiederholungen

Eine oder mehrere Ziffern:

```
grep -E '[[:digit:]]+' datei.txt
```

Genau vier Ziffern:

```
grep -E '[[:digit:]]{4}' datei.txt
```

Zwischen zwei und vier Ziffern:

```
grep -E '[[:digit:]]{2,4}' datei.txt
```

Ein optionales Suffix:

```
grep -E '^server(-backup)?$' systeme.txt
```

Das Muster passt zu `server` und `server-backup`.

Die Gruppierung entscheidet, worauf sich eine Wiederholung bezieht:

```
grep -E '^(ab){3}$' datei.txt
```

Dieses Muster passt zu `ababab`. Ohne Klammern würde nur das unmittelbar vorhergehende Zeichen wiederholt.

Besondere Vorsicht ist bei scheinbar lesbaren Wörtern geboten:

```
grep -E 'error{2,}' datei.txt
```

Der Quantifizierer gehört nur zum letzten `r`. Das Muster verlangt also `erro` gefolgt von mindestens zwei weiteren `r`-Zeichen. Es bedeutet nicht, dass das Wort `error` mehrfach vorkommen muss.

Mehrere vollständige Vorkommen werden gruppiert:

```
grep -E '(error){2,}' datei.txt
```

Gruppierungen und Alternativen

Runde Klammern fassen mehrere Teile eines ERE-Ausdrucks zu einer Einheit zusammen. Der senkrechte Strich beschreibt Alternativen:

```
grep -E '^(INFO|WARNING|ERROR):' anwendung.log
```

Der Ausdruck verlangt am Zeilenanfang genau eine der drei Gruppen, gefolgt von einem Doppelpunkt. Alternativen sollten so eng wie möglich gruppiert werden. Der folgende Ausdruck:

```
grep -E '^ERROR|WARNING$' anwendung.log
```

bedeutet:

Exitcode 0 | Mindestens ein Treffer gefunden (Bedingung erfüllt)

Exitcode 1 | Kein Treffer gefunden (Muster passte auf keine Zeile)

Exitcode >1 | Fehler aufgetreten (Datei nicht lesbar oder Syntaxfehler)

Für eine vollständige Zeile mit genau einem der beiden Werte müssen beide Alternativen gemeinsam verankert werden:

```
grep -E '^(ERROR|WARNING)$' anwendung.log
```

^ | Zeilenanfang (Anker) [0-9]{4} | Genau vier Ziffern (z. B. Jahreszahl)

- | Wörtlicher Bindestrich als Trennzeichen .* | Beliebige nachfolgende Zeichenfolge

Gruppierungen dienen außerdem dazu, wiederkehrende Strukturen zu beschreiben. Ein syntaktisches Muster für eine IPv4-ähnliche Zeichenfolge lautet:

```
grep -E '([[:digit:]]{1,3}\.){3}[[:digit:]]{1,3}' zugriffe.log
```

Das Muster erkennt vier durch Punkte getrennte Zahlengruppen. Es prüft nicht, ob jede Gruppe einen gültigen Wert zwischen 0 und 255 enthält.

> ⚠ **Warnung:** Ein regulärer Ausdruck, der die Form einer IP-Adresse, eines Datums oder einer E-Mail-Adresse erkennt, validiert nicht automatisch deren fachliche Gültigkeit. Form und Bedeutung sind zwei verschiedene Prüfungen.

Passende Textteile extrahieren

Mit `grep -oE` kannst du den Teil einer Zeile ausgeben, der zum vollständigen regulären Ausdruck passt.

Aus einer Logzeile wie dieser:

```
2026-08-17T14:05:22Z login user=sebastian result=failed
```

lässt sich der Benutzerwert einschließlich des Schlüssels extrahieren:

```
grep -oE 'user=[[:alnum:]_.-]+' anwendung.log
```

Ausgabe:

```
user=sebastian
```

Soll nur der Wert hinter `user=` weiterverarbeitet werden, genügt eine Capture-Gruppe in `grep` nicht. `grep -o` gibt den vollständigen Treffer aus und keine frei wählbare Untergruppe. Der Schlüssel kann anschließend mit einem weiteren Werkzeug entfernt werden:

```
grep -oE 'user=[[:alnum:]_.-]+' anwendung.log | cut -d= -f2
```

Alternativ übernimmt `sed` Auswahl und Ersetzung in einem Schritt:

```
sed -nE 's/.*user=([[:alnum:]_.-]+).*/\1/p' anwendung.log
```

> **Typische Fehlerquelle:** Runde Klammern sorgen bei `grep` nicht dafür, dass nur der Inhalt der Gruppe ausgegeben wird. Sie strukturieren das Muster. `grep -o` gibt weiterhin den gesamten passenden Ausdruck aus.

Text mit `sed` auswählen

`sed` verarbeitet einen Eingabestrom zeilenweise. Reguläre Ausdrücke können dabei als Adresse festlegen, auf welche Zeilen ein Befehl angewendet wird.

Standardmäßig gibt `sed` jede verarbeitete Zeile aus. Mit `-n` wird diese automatische Ausgabe unterdrückt. Der Befehl `p` gibt anschließend nur gezielt ausgewählte Zeilen aus:

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

```
sed -n '/ERROR/p' anwendung.log
```

Mit erweiterten regulären Ausdrücken:

```
sed -nE '/^(ERROR|WARNING):/p' anwendung.log
```

Für eine reine Auswahl ist `grep` meist kürzer. `sed` wird interessant, wenn gefundener Text zugleich verändert oder umgeformt werden soll.

Ersetzungen mit `sed`

Der Substitutionsbefehl besitzt diese Grundform:

```
s/Suchmuster/Ersetzung/Flags
```

Eine einfache Ersetzung:

```
sed 's/localhost/db01.example.net/' anwendung.conf
```

Ohne weiteres Flag ersetzt `sed` nur das erste passende Vorkommen je Zeile. Mit `g` werden alle Vorkommen innerhalb der jeweiligen Zeile ersetzt:

```
sed 's/localhost/db01.example.net/g' anwendung.conf
```

Der Befehl schreibt das Ergebnis auf `stdout`. Die Eingabedatei bleibt unverändert.

Enthält das Muster viele Schrägstriche, kann ein anderes Trennzeichen verwendet werden:

```
sed 's#/var/www/html#/srv/www#g' anwendung.conf
```

Das ist lesbarer als eine Folge escaped gesetzter Schrägstriche.

In der Ersetzung steht `&` für den gesamten gefundenen Text:

```
sed -E 's/ERROR/[&]/g' anwendung.log
```

Aus `ERROR` wird damit `[ERROR]`.

Gruppen können über `\1` bis `\9` referenziert werden. Das folgende Beispiel ersetzt einen Passwortwert, behält aber den Schlüssel:

```
sed -E 's/^(password=).*/\1[ENTFERNT]/' anwendung.conf
```

Die erste Gruppe enthält `password=`. `\1` fügt diesen Teil in die Ersetzung ein.

Dateien mit `sed` verändern

Die Option `-i` schreibt Änderungen direkt in die angegebene Datei. Das sollte erst geschehen, nachdem derselbe Ausdruck ohne `-i` geprüft wurde.

Zunächst nur die geplante Ausgabe anzeigen:

```
sed -E 's/^(timeout=)[[:digit:]]+/\130/' anwendung.conf
```

Unter GNU `sed` kann `-i` gleichzeitig eine Sicherungskopie anlegen:

```
sed -E -i.bak 's/^(timeout=)[[:digit:]]+/\130/' anwendung.conf
```

Die ursprüngliche Datei bleibt als `anwendung.conf.bak` erhalten.

> ⚠ **Warnung:** `sed -i` ersetzt die Datei und kann dabei Eigenschaften symbolischer Links oder implementationsabhängige Metadaten beeinflussen. Prüfe Ausgabe und Sicherungskopie, bevor du den Befehl auf mehrere Konfigurationsdateien anwendest.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Eine Sicherungskopie ist noch keine Verifikation. Vergleiche anschließend beide Dateien:

```
diff -u anwendung.conf.bak anwendung.conf
```

Erst der Vergleich zeigt, ob ausschließlich die beabsichtigten Zeilen verändert wurden.

Quoting und Escaping in der Shell

Ein regulärer Ausdruck wird nicht direkt an `grep` oder `sed` übergeben. Zuerst verarbeitet die Shell die Kommandozeile. Danach interpretiert das jeweilige Programm den verbleibenden Ausdruck.

Einfache Anführungszeichen sind deshalb für feste Regex-Ausdrücke meist die sicherste Wahl:

```
grep -E '^[[:digit:]]{4}-[[:digit:]]{2}-[[:digit:]]{2}$' daten.txt
```

Innerhalb einfacher Anführungszeichen verändert die Shell weder `$` noch Backslashes oder Sternchen.

Doppelte Anführungszeichen erlauben dagegen Variablenersetzung:

```
muster='ERROR|WARNING'
grep -E "^($muster):" anwendung.log
```

Das kann beabsichtigt sein, verbindet aber Shell- und Regex-Syntax. Der Inhalt der Variablen wird Bestandteil des regulären Ausdrucks und muss deshalb vertrauenswürdig und syntaktisch geeignet sein.

Die wichtigsten Ebenen sind:

^ | Zeilenanfang (ERROR|WARNING) | Gruppe mit Alternativen (Oder-Verknüpfung)

: | Wörtlicher Doppelpunkt .* | Restlicher Meldungstext bis Zeilenende

> **Tipp:** Wenn ein Ausdruck nicht funktioniert, prüfe die Ebenen getrennt. Gib zuerst das tatsächlich übergebene Muster aus, teste danach nur den Regex und ergänze zuletzt die `sed`-Ersetzung. Drei kleine Fehler übereinander ergeben selten ein interessantes Rätsel. Meist ergeben sie nur einen verlorenen Nachmittag.

Portabilität und GNU-Erweiterungen

GNU `grep` und GNU `sed` unterstützen zusätzliche Kurzformen wie `\b`, `\w` oder `\s`. Diese sind bequem, gehören aber nicht in allen Fällen zur portablen POSIX-ERE-Syntax.

Für distributionsübergreifende Shellskripte sind POSIX-Zeichenklassen meist die verlässlichere Wahl:

GNU-nahe Kurzform	Portablere Schreibweise
`\d`	`[[:digit:]]`
`\w`	`[[:alnum:]_]`
`\s`	`[[:space:]]`

Auch `grep -P` für Perl-kompatible reguläre Ausdrücke ist nicht auf jedem System verfügbar und gehört nicht zum LPIC-1-Kern dieses Artikels.

`grep -E` und `sed -E` sind für erweiterte reguläre Ausdrücke die klaren Schreibweisen. Ob ein Ausdruck darüber hinaus GNU-spezifische Erweiterungen verwendet, solltest du bei Skripten

dokumentieren.

Praktische Beispiele

Praxis-Muster: ISO-ähnliche Datumsangaben am Zeilenanfang finden

```
grep -E '^[[:digit:]]{4}-[[:digit:]]{2}-[[:digit:]]{2}[[:space:]]' anwendung.log
```

Der Ausdruck prüft nur die Form. Eine Zeile mit `2026-99-42` würde ebenfalls passen.

Numerische HTTP-Statuscodes extrahieren

```
grep -oE 'status=[1-5][[:digit:]]{2}' zugriffe.log
```

Aktivierte oder deaktivierte Werte erkennen

```
grep -E '^(enabled|disabled)=(yes|no)$' anwendung.conf
```

Pfade in einer Ausgabe vereinheitlichen

```
sed -E 's#/var/www/[^[:space:]]+#/srv/www#g' bericht.txt
```

Sensible Werte in einer erzeugten Ausgabe maskieren

```
sed -E 's/^(token=).*/\1[ENTFERNT]/' anwendung.conf
```

Der Befehl verändert die Eingabedatei nicht, solange `-i` fehlt.

> **Typische Fehlerquelle:** Ein immer umfangreicherer regulärer Ausdruck ist nicht automatisch präziser. Wenn Validierung, Berechnung oder mehrere voneinander abhängige Regeln erforderlich sind, ist ein Regex möglicherweise nicht mehr das passende Werkzeug.

Suchen in komprimierten Dateien und Archiven

Im vorherigen Teil hast du gelernt, wie sich Dateien komprimieren und mit `tar` zu Archiven zusammenfassen lassen. Dieser Abschnitt baut unmittelbar auf dem Artikel zur Archivierung und Komprimierung auf und setzt die dort erklärten Grundlagen zu `tar`, `gzip`, `bzip2` und `xz` voraus. Hier konzentrieren wir uns darauf, gespeicherte Dateinamen und Inhalte gezielt zu durchsuchen. Für die Suche ist die Unterscheidung zwischen Komprimierung und Archivierung entscheidend.

Eine Datei wie `anwendung.log.gz` enthält einen einzelnen komprimierten Datenstrom. Eine Datei wie `logs.tar.gz` enthält dagegen ein mit `gzip` komprimiertes `tar`-Archiv, in dem mehrere Dateien und Verzeichnisse liegen können.

```
2026-08-17T14:05:22Z login user=sebastian result=failed
```

Diese beiden Formate benötigen unterschiedliche Suchwege. `zgrep` eignet sich für gzip-komprimierte Einzeldateien. Bei einem `tar.gz`-Archiv muss zunächst `tar` den gewünschten Archiveintrag bereitstellen.

Gzip-komprimierte Dateien mit `zgrep` durchsuchen

`zgrep` verbindet die Dekomprimierung einer gzip-komprimierten Datei mit einer anschließenden Suche. Die Datei wird während der Verarbeitung gelesen, aber nicht dauerhaft entpackt.

```
zgrep -nF "ERROR" anwendung.log.gz
```

Die Optionen werden an `grep` weitergegeben. In diesem Beispiel ergänzt `-n` die Zeilennummer, während `-F` nach der festen Zeichenkette `ERROR` sucht.

Mehrere rotierte Logdateien können gemeinsam durchsucht werden:

```
zgrep -HnF "connection refused" /var/log/meine-anwendung/*.gz
```

Die Shell erweitert `*.gz` zu den vorhandenen Dateien. `-H` sorgt dafür, dass der jeweilige Dateiname in der Ausgabe erscheint.

Groß- und Kleinschreibung lassen sich wie gewohnt ignorieren:

```
zgrep -HinF "timeout" /var/log/meine-anwendung/*.gz
```

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Für andere Kompressionsformate existieren entsprechende Werkzeuge:

Format	Werkzeug	Beispiel
gzip	`zgrep`	`zgrep -nF "ERROR" datei.gz`
bzip2	`bzgrep`	`bzgrep -nF "ERROR" datei.bz2`
xz	`xzgrep`	`xzgrep -nF "ERROR" datei.xz`

Welche Optionen diese Wrapper unterstützen, hängt von der installierten Implementierung ab. Die jeweilige Manpage zeigt, welche `grep`-Optionen weitergereicht werden.

> **Tipp:** Komprimierte Logs müssen für eine Suche nicht in eine temporäre Datei entpackt werden. Das spart Schreibzugriffe und verhindert, dass neben dem Archiv unbemerkt eine große unkomprimierte Kopie liegen bleibt.

Grenzen der direkten Suche

Eine gzip-Datei wird als Datenstrom dekomprimiert. Um einen Inhalt am Ende der Datei zu prüfen, muss der vorhergehende komprimierte Datenstrom verarbeitet werden. `zgrep` kann deshalb nicht beliebig an eine bestimmte Textzeile springen.

Die Belastung entsteht hauptsächlich durch:

- das Lesen der komprimierten Datei,
- die Dekomprimierung,
- die Auswertung des Suchmusters.

Eine große komprimierte Datei führt nicht automatisch zu einem entsprechend hohen RAM-Verbrauch. Die Verarbeitung erfolgt normalerweise als Datenstrom. Laufzeit und CPU-Aufwand können trotzdem erheblich sein, insbesondere bei vielen Archiven oder komplexen Mustern.

Mit `gzip -t` lässt sich vorab prüfen, ob ein gzip-Datenstrom formal gelesen werden kann:

```
gzip -t anwendung.log.gz
```

Der Befehl erzeugt bei Erfolg keine normale Ausgabe. Sein Exitcode zeigt das Ergebnis:

```
gzip -t anwendung.log.gz
printf '%s\n' "$?"
```

Ein Exitcode von `0` steht für eine erfolgreiche Prüfung. Ein anderer Wert weist auf einen Fehler hin.

> **Typische Fehlerquelle:** Keine Ausgabe von `gzip -t` bedeutet nicht, dass die enthaltenen Logdaten fachlich vollständig oder korrekt sind. Der Befehl prüft den komprimierten Datenstrom, nicht den Inhalt der Protokolle.

Archiveinträge mit `tar` suchen

Bei einem `tar`-Archiv kannst du zunächst dessen Verzeichnis anzeigen:

```
tar -tf logs.tar
```

Für ein gzip-komprimiertes Archiv:

```
tar -tzf logs.tar.gz
```

Die Ausgabe enthält die Namen der gespeicherten Archiveinträge. Sie enthält nicht deren Dateiinhalte. Bestimmte Einträge lassen sich über eine Pipe filtern:

```
tar -tzf logs.tar.gz | grep -E '\.log$'
```

Dieser Befehl sucht nach Namen, die auf `.log` enden. Er beantwortet die Frage, welche Logdateien im Archiv enthalten sind.

```
user=sebastian
```

> **Tipp:** Lass dir das Archivverzeichnis zuerst vollständig oder gefiltert anzeigen. Dadurch kennst du den exakten internen Pfad, den `tar` für eine anschließende Inhaltsausgabe erwartet.

Inhalte eines Archiveintrags durchsuchen

Mit `-O` schreibt GNU `tar` den Inhalt eines ausgewählten Archiveintrags auf `stdout`, statt ihn als Datei auf das Dateisystem zu extrahieren.

Angenommen, das Archiv enthält den Eintrag `var/log/meine-anwendung/error.log`:

```
tar -xOzf logs.tar.gz var/log/meine-anwendung/error.log
```

Die Ausgabe kann unmittelbar an `grep` weitergegeben werden:

```
tar -xOzf logs.tar.gz var/log/meine-anwendung/error.log \
  | grep -nF "ERROR"
```

Der Ablauf besteht aus zwei klar getrennten Schritten:

```
s/Suchmuster/Ersetzung/Flags
```

Der interne Pfad muss so angegeben werden, wie er im Archiv gespeichert ist. Ein führender Schrägstrich gehört häufig nicht zum Eintragsnamen, selbst wenn die ursprüngliche Datei unter einem absoluten Pfad lag.

Den richtigen Namen ermittelst du deshalb zuerst mit:

```
tar -tzf logs.tar.gz
```

Mehrere Archiveinträge als Datenstrom durchsuchen

GNU `tar` kann mehrere passende Einträge auf `stdout` schreiben. Mit `--wildcards` lassen sich dafür Namensmuster verwenden:

```
tar --wildcards -xOzf logs.tar.gz "*.log" \
  | grep -nF "ERROR"
```

Dabei werden die Inhalte der passenden Dateien nacheinander in denselben Datenstrom geschrieben. `grep` kennt die Grenzen zwischen den Archiveinträgen nicht. Dateinamen und Zeilennummern lassen sich dadurch nicht mehr zuverlässig dem ursprünglichen Eintrag zuordnen.

Diese Form eignet sich deshalb nur, wenn die Frage lautet, ob ein Muster irgendwo in den ausgewählten Inhalten vorkommt. Für eine nachvollziehbare Zuordnung solltest du Archiveinträge einzeln untersuchen.

> ⚠ **Warnung:** Wenn mehrere Archiveinträge gemeinsam über `tar -xO` ausgegeben werden, entsteht ein zusammenhängender Datenstrom. Eine Trefferzeile enthält nicht automatisch den Namen der ursprünglichen Datei.

Warum `zgrep` für `tar.gz` nicht genügt

Ein häufiger Fehler ist dieser Befehl:

```
zgrep -F "ERROR" logs.tar.gz
```

`zgrep` entfernt hier lediglich die gzip-Komprimierung. Das Ergebnis ist jedoch kein normaler Textstrom aus einer einzelnen Logdatei, sondern ein `tar`-Archiv mit Headern, Metadaten und den hintereinander gespeicherten Dateiinhalten.

Selbst wenn der Befehl scheinbar einen Treffer liefert, fehlt die saubere Zuordnung zu einem Archiveintrag. Außerdem können Binärdaten und Tar-Metadaten die Auswertung beeinflussen.

Der korrekte Ablauf lautet:

password= | Zu matchender Präfix

([^]*) | Erste Erfassungsgruppe: beliebige Zeichen ohne Leerzeichen

\1 | Rückreferenz auf den Inhalt der ersten Gruppe im Ersetzungsteil

> **Typische Fehlerquelle:** Die Endung `.tar.gz` bezeichnet zwei Verarbeitungsschritte: `tar` bündelt mehrere Einträge, `gzip` komprimiert anschließend das Archiv. `zgrep` berücksichtigt nur die äußere gzip-Schicht.

Archiv vor der Suche prüfen

Bevor du Inhalte aus einem Archiv liest, kannst du prüfen, ob sich dessen Verzeichnis fehlerfrei verarbeiten lässt:

```
tar -tzf logs.tar.gz >/dev/null
```

Anschließend wertest du den Exitcode aus:

```
status=$?

if [ "$status" -eq 0 ]; then
  printf '%s\n' "Archivverzeichnis konnte gelesen werden"
else
  printf '%s\n' "Archiv konnte nicht vollständig gelesen werden" >&2
fi
```

Diese Prüfung zeigt, ob `tar` das Archivverzeichnis verarbeiten konnte. Sie garantiert weder, dass alle archivierten Daten fachlich vollständig sind, noch ersetzt sie einen Restore-Test.

Praktische Beispiele

Praxis-Szenario: Fehler in einer rotierenden gzip-Logdatei suchen

```
zgrep -HnF "database connection failed" \
  /var/log/meine-anwendung/*.gz
```

Konfigurationsdateien in einem Archiv auflisten

```
tar -tzf etc-backup.tar.gz | grep -E '\.conf$'
```

Eine bestimmte archivierte Konfiguration durchsuchen

```
tar -xOzf etc-backup.tar.gz etc/ssh/sshd_config \
  | grep -nF "PermitRootLogin"
```

Einen Fehlercode in xz-komprimierten Logs suchen

```
xzgrep -HnF "status=503" zugriffe.log.xz
```

Mehrere Archive nach enthaltenen Logdateien prüfen

```
for archiv in /backups/logs-*.tar.gz; do
  printf '\n%s\n' "$archiv"
  tar -tzf "$archiv" | grep -E '\.log$'
done
```

Die Schleife zeigt für jedes Archiv getrennt an, welche Einträge auf `.log` enden. Sie durchsucht noch keine Dateiinhalte.

> **Tipp:** Trenne bei der Fehlersuche die Fragen „Welche Datei ist im Archiv enthalten?“ und „Welcher Inhalt steht in dieser Datei?“. Die erste beantwortet `tar -t`, die zweite `tar -xO` in Verbindung mit `grep`.

`find`, `grep` und `xargs` sicher kombinieren

Mit `find` suchst du nach Dateien, während `grep` deren Inhalte untersucht. Sobald beide Werkzeuge zusammenarbeiten sollen, muss die Ausgabe des einen Befehls als Eingabe oder Argumentliste für den nächsten dienen. Die Grundlagen dieses Datenflusses kennst du bereits aus dem Artikel zu Streams, Pipes und Umleitungen.

Eine einfache Suchkette besteht aus drei Schritten:

1. **1. Shell-Ebene** Parst Quotes, expandiert Variablen und schützt Metazeichen
2. **2. Regex-Engine** Kompiliert das Suchmuster und identifiziert Textstellen
3. **3. Werkzeug-Ausführung (sed / grep)** Wendet Ersetzungen, Filter oder Aktionen auf gefundene Gruppen an

Die Pipe überträgt zunächst nur einen Datenstrom. Sie entscheidet nicht, ob eine Zeile ein Dateiname, ein Suchmuster oder gewöhnlicher Text ist. Diese Bedeutung entsteht erst durch den empfangenden Befehl.

Dateien direkt mit `-exec` durchsuchen

Soll jeder gefundene Pfad unmittelbar an `grep` übergeben werden, ist `find -exec` die übersichtlichste Lösung:

```
find "$HOME/lpic-test" -type f -name "*.log" \
  -exec grep -HnF -- "ERROR" {} +
```

`find` sucht alle Dateien mit der Endung `.log`. Die gefundenen Pfade werden an `grep` übergeben, das darin nach der festen Zeichenfolge `ERROR` sucht.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

Die verwendeten Optionen haben folgende Bedeutung:

- `-H` zeigt den Dateinamen vor jedem Treffer an.
- `-n` ergänzt die jeweilige Zeilennummer.
- `-F` behandelt `ERROR` als feste Zeichenfolge und nicht als regulären Ausdruck.
- `--` beendet die Verarbeitung der `grep`-Optionen.
- `{}` steht für die von `find` gefundenen Pfade.
- `+` fasst mehrere Pfade zu möglichst wenigen `grep`-Aufrufen zusammen.

Eine mögliche Ausgabe sieht so aus:

```
/home/user/lpic-test/system.log:14:ERROR ----> Dienst konnte nicht gestartet werden
/home/user/lpic-test/webserver.log:87:ERROR ----> Verbindung abgelehnt
```

> **Tipp:** Wenn genau ein nachgelagerter Befehl ausgeführt werden soll, ist `-exec ... {} +` meist einfacher als eine zusätzliche Pipe mit `xargs`. Die Dateinamen werden direkt als Argumente übergeben und müssen nicht erneut aus einem Textstrom gelesen werden.

Die bereits im Abschnitt zu `find` gezeigte Variante mit `\;` startet den Befehl für jeden Treffer einzeln:

```
find "$HOME/lpic-test" -type f -name "*.log" \
  -exec grep -HnF -- "ERROR" {} \;
```

Das Ergebnis ist vergleichbar, bei vielen Dateien entstehen jedoch deutlich mehr Prozesse. Für `grep`, `stat`, `file` oder `sha256sum` ist die gebündelte Form mit `+` deshalb in der Regel sinnvoller.

Wozu `xargs` benötigt wird

Nicht jeder Befehl kann seine zu verarbeitenden Dateinamen über die Standardeingabe lesen. `grep` erwartet Dateinamen beispielsweise als Argumente hinter dem Suchmuster:

```
grep -HnF -- "ERROR" system.log webserver.log
```

`xargs` liest Einträge aus der Standardeingabe und baut daraus eine solche Argumentliste. Eine einfache Kombination könnte daher so aussehen:

```
find "$HOME/lpic-test" -type f -name "*.log" |
  xargs grep -HnF -- "ERROR"
```

Für gewöhnliche Dateinamen funktioniert dieser Aufruf. Verlässlich ist er trotzdem nicht. Standardmäßig trennt `xargs` seine Eingabe anhand von Leerzeichen und Zeilenumbrüchen. Ein Pfad wie dieser wird dadurch falsch zerlegt:

```
/home/user/lpic-test/alte logs/system.log
```

Aus einem Dateinamen werden zwei Argumente:

```
/home/user/lpic-test/alte
logs/system.log
```

`grep` versucht anschließend, zwei nicht vorhandene Dateien zu öffnen.

> **Typische Fehlerquelle:** `find ... | xargs ...` ist für beliebige Dateinamen ungeeignet. Leerzeichen, Anführungszeichen, Backslashes oder Zeilenumbrüche können die Trennung verändern. Dass der Befehl mit den eigenen Testdateien funktioniert, beweist nur, dass deren Namen gerade höflich waren.

Dateinamen nullterminiert übergeben

Ein Linux-Dateiname darf nahezu jedes Zeichen enthalten. Das Nullbyte ist die entscheidende Ausnahme und eignet sich deshalb als eindeutiges Trennzeichen.

`find` erzeugt mit `-print0` eine nullterminierte Ausgabe. `xargs` liest dieses Format mit `-0` ein:

```
find "$HOME/lpic-test" -type f -name "*.log" -print0 |
  xargs -0 -r grep -HnF -- "ERROR"
```

Der Datenfluss sieht damit so aus:

Komprimierte Einzeldatei (.gz / .xz)

Dateistruktur
: Ein einziger komprimierter Datenstrom

Direktsuche
: zgrep / xzgrep dekomprimieren und filtern direkt

Werkzeuge
: zgrep, bzgrep, xzgrep, zcat

Empfehlung
: Standard bei rotierenden System-Logs (/var/log/*.gz)

Archiv-Container (.tar.gz / .tar.xz)

Dateistruktur
: Tar-Header bündeln mehrere Dateien in einem Archiv

Direktsuche
: zgrep schlägt fehl, da tar-Container-Header vorliegen

Werkzeuge
: tar -tf (Inhalt auflisten) oder tar -xOzf (Stream filtern)

Empfehlung
: Standard bei Datensicherungen und Projektpaketen

Leerzeichen und Zeilenumbrüche bleiben Bestandteile des jeweiligen Dateinamens. Sie werden nicht mehr als Trenner interpretiert.

Die Option `-r` verhindert unter GNU `xargs`, dass `grep` einmal ohne Dateinamen gestartet wird, wenn `find` keinen Treffer liefert. Sie ist eine GNU-Erweiterung und daher nicht auf jedem Unix-System verfügbar. Im üblichen GNU/Linux-Umfeld ist sie dennoch sinnvoll.

Mehrere Suchschritte verbinden

`grep` kann mit `-l` ausschließlich die Namen der Dateien ausgeben, in denen mindestens ein Treffer vorkommt. Mit GNU `grep -Z` wird jeder ausgegebene Dateiname durch ein Nullbyte

abgeschlossen. Damit bleibt die sichere Übergabe auch in einer längeren Befehlskette erhalten:

```
find "$HOME/lpic-test" -type f -name "*.log" -print0 |
  xargs -0 -r grep -lZF -- "ERROR" |
  xargs -0 -r stat --
```

Die Verarbeitung erfolgt in klar getrennten Stufen:

1. `find` liefert alle passenden Logdateien.
2. `grep -lF` filtert die Dateien mit der Zeichenfolge `ERROR`.
3. `-Z` gibt die Treffer weiterhin nullterminiert aus.
4. `stat` zeigt Informationen zu den verbliebenen Dateien an.

Eine Pipeline ist nur so zuverlässig wie ihre schwächste Übergabestelle. Würde `grep` die Dateinamen hier wieder mit gewöhnlichen Zeilenumbrüchen ausgeben, wäre die Nullterminierung unterbrochen.

Anzahl und Parallelität der Aufrufe steuern

`xargs` fasst Eingaben automatisch zu Argumentlisten zusammen, die innerhalb der Systemgrenzen liegen. Mit `-n` kannst du zusätzlich festlegen, wie viele Eingaben höchstens an einen einzelnen Aufruf übergeben werden:

```
find "$HOME/lpic-test" -type f -name "*.log" -print0 |
  xargs -0 -r -n 5 stat --
```

`stat` verarbeitet damit höchstens fünf Dateien pro Aufruf.

Die Option `-P` erlaubt mehrere parallele Prozesse:

```
find "$HOME/lpic-test" -type f -name "*.iso" -print0 |
  xargs -0 -r -n 1 -P 4 sha256sum --
```

Dieser Befehl berechnet bis zu vier Prüfsummen gleichzeitig. Bei vielen großen Dateien kann das die Verarbeitung beschleunigen. Die Ausgabe erscheint dann jedoch nicht zwingend in derselben Reihenfolge wie die Eingabe.

Parallelität ist nur sinnvoll, wenn die einzelnen Aufrufe unabhängig voneinander arbeiten. Mehrere Prozesse, die dieselbe Datei verändern oder dieselbe Ausgabedatei beschreiben, produzieren keine Beschleunigung, sondern ein Glücksspiel mit Prozess-IDs.

> **Tipp:** Beginne ohne `-P` und prüfe zuerst, ob die Befehlskette korrekt arbeitet. Parallelisierung sollte eine funktionierende Verarbeitung beschleunigen, nicht deren Fehler gleichzeitig vervielfachen.

Suchergebnisse vor Änderungen prüfen

Die Kombination aus `find`, `xargs` und einem schreibenden Befehl kann viele Dateien in kurzer Zeit verändern. Ein falsch gesetztes Suchkriterium wirkt dann ebenfalls auf viele Dateien – nur leider mit bemerkenswerter Effizienz.

Prüfe deshalb zunächst ausschließlich die Auswahl:

```
find "$HOME/lpic-test" -type f -name "*.tmp" -print
```

Sollen zusätzlich Dateiinformationen kontrolliert werden, bleibt der Aufruf weiterhin ohne verändernde Wirkung:

```
find "$HOME/lpic-test" -type f -name "*.tmp" \
  -exec stat -- {} +
```

Erst wenn Pfade, Dateityp und Suchbereich stimmen, sollte der eigentliche Zielbefehl ergänzt werden.

> ⚠ **Warnung:** Verbinde ungeprüfte Suchergebnisse niemals direkt mit löschenden oder überschreibenden Befehlen. Kontrolliere vorher dieselbe `find`-Abfrage mit `-print`, `stat` oder einem vergleichbaren lesenden Befehl. Besonders kritisch sind Suchläufe mit erhöhten Rechten oder in Verzeichnissen außerhalb einer Testumgebung.

Für die meisten Aufgaben reicht damit eine einfache Entscheidung:

- Verwende `find -exec ... {} +`, wenn die Treffer direkt an einen einzelnen Befehl gehen.
- Verwende `find -print0 | xargs -0`, wenn du die Argumentbildung mit Optionen wie `-n`, `-P` oder weiteren Verarbeitungsschritten steuern möchtest.
- Verwende keine zeilenbasierte Übergabe, sobald beliebige Dateinamen verarbeitet werden können.

Zusammenfassende Übungen

Die folgenden Übungen verbinden die Werkzeuge aus diesem Artikel zu vollständigen Suchabläufen. Du arbeitest dabei ausschließlich in einem eigenen Übungsverzeichnis und benötigst keine Root-Rechte.

Einige Aufgaben greifen zusätzlich auf Streams, Pipes und Umleitungen sowie auf die im vorherigen Teil behandelte Archivierung und Komprimierung zurück.

Der grundsätzliche Ablauf bleibt bei allen Übungen gleich:

1. **Archivstruktur auflisten (tar -tzf)** Ermittelt alle enthaltenen Dateipfade ohne Entpackung auf Platte

2. **Dateiliste nach Ziel filtern (grep)** Gewünschten Pfad innerhalb des Archivs exakt identifizieren

3. **Gezielten Extraktionspfad definieren** Den identifizierten Pfad an die nachfolgende Streaming-Pipeline übergeben

Testumgebung vorbereiten

Lege zunächst ein separates Verzeichnis mit einigen Log- und Konfigurationsdateien an:

```
uebung="$HOME/lpic-suche-uebung"

mkdir -p "$uebung"/{logs,config,archive}

printf '%s\n' \
  "INFO Anwendung gestartet" \
  "ERROR Verbindung fehlgeschlagen" \
  "INFO Neuer Versuch" \
  > "$uebung/logs/anwendung.log"

printf '%s\n' \
  "WARN Speicherplatz wird knapp" \
  "ERROR Dienst nicht erreichbar" \
  "INFO Prüfung beendet" \
  > "$uebung/logs/system alt.log"

printf '%s\n' \
  "Listen 80" \
  "ServerName example.test" \
  > "$uebung/config/web.conf"

printf '%s\n' \
  "Port 22" \
```

```
  "PermitRootLogin no" \
  > "$uebung/config/ssh.conf"
```

Erstelle anschließend ein komprimiertes Archiv der Logdateien und eine einzelne mit `gzip` komprimierte Datei:

```
tar -czf "$uebung/archive/logs.tar.gz" \
  -C "$uebung" logs

gzip -k "$uebung/logs/anwendung.log"
```

Kontrolliere die vorbereiteten Dateien:

```
find "$uebung" -type f -printf '%P\n'
```

Die Ausgabe sollte diese Dateien enthalten:

```
logs/anwendung.log
logs/anwendung.log.gz
logs/system alt.log
config/web.conf
config/ssh.conf
archive/logs.tar.gz
```

> **Tipp:** Verwende für die Übungen die Variable `$uebung` statt den vollständigen Pfad mehrfach einzutippen. Sie gilt nur in der aktuellen Shell. Öffnest du ein neues Terminal, musst du sie erneut setzen.

Übung 1: Dateien mit mehreren Kriterien suchen

Aufgabe: Suche im Übungsverzeichnis nach allen regulären, nicht komprimierten Dateien, deren Name auf `.log` endet.

Probiere zunächst, den passenden `find`-Befehl selbst zusammenzustellen.

Mögliche Lösung:

```
find "$uebung" -type f -name "*.log"
```

Das Ergebnis besteht aus zwei Dateien:

```
/home/user/lpic-suche-uebung/logs/anwendung.log
/home/user/lpic-suche-uebung/logs/system alt.log
```

Erweitere die Suche anschließend so, dass nur Dateien innerhalb des Verzeichnisses `logs` berücksichtigt werden und die Suche nicht in weitere Unterverzeichnisse hinabsteigt:

```
find "$uebung/logs" -maxdepth 1 -type f -name "*.log"
```

Suche nun ausschließlich nach komprimierten Dateien:

```
find "$uebung" -type f \( -name "*.gz" -o -name "*.xz" -o -name "*.bz2" \)
```

Hier ist die Gruppierung wichtig. Ohne `\(` und `\)` würden die Verknüpfungen anders ausgewertet, wodurch `-type f` nicht zuverlässig für alle Namensbedingungen gelten würde.

> **Typische Fehlerquelle:** Wird `*.log` nicht in Anführungszeichen gesetzt, versucht bereits die Shell, das Muster im aktuellen Verzeichnis aufzulösen. `find` erhält dann nicht mehr zwingend den vorgesehenen Suchausdruck.

Übung 2: Inhalte mit `grep` filtern

Aufgabe: Suche in allen nicht komprimierten Logdateien nach der festen Zeichenfolge `ERROR`. Dateiname und Zeilennummer sollen ausgegeben werden.

Mögliche Lösung:

```
grep -HnF -- "ERROR" "$uebung"/logs/*.log
```

Die Ausgabe sollte zwei Treffer enthalten:

```
/home/user/lpic-suche-uebung/logs/anwendung.log:2:ERROR Verbindung fehlgeschlagen
/home/user/lpic-suche-uebung/logs/system alt.log:2:ERROR Dienst nicht erreichbar
```

Suche anschließend mit einem erweiterten regulären Ausdruck nach Zeilen, die mit `WARN` oder `ERROR` beginnen:

```
grep -HnE -- "^(WARN|ERROR)" "$uebung"/logs/*.log
```

Der Ausdruck setzt sich aus drei Bestandteilen zusammen:

- `^` verankert die Suche am Zeilenanfang.
- `(WARN|ERROR)` beschreibt zwei mögliche Zeichenfolgen.
- `-E` aktiviert erweiterte reguläre Ausdrücke.

Ermittle danach nur die Namen der Dateien, in denen `ERROR` vorkommt:

```
grep -lF -- "ERROR" "$uebung"/logs/*.log
```

Mit `-c` kannst du zusätzlich zählen, wie viele passende Zeilen jede Datei enthält:

```
grep -HcF -- "ERROR" "$uebung"/logs/*.log
```

Eine Ausgabe wie diese bedeutet, dass in jeder Datei genau eine passende Zeile gefunden wurde:

```
/home/user/lpic-suche-uebung/logs/anwendung.log:1
/home/user/lpic-suche-uebung/logs/system alt.log:1
```

Übung 3: Daten mit `sed` extrahieren

Aufgabe: Gib aus den Konfigurationsdateien ausschließlich den jeweils ersten Wert einer Zeile aus. Aus `Listen 80` soll beispielsweise `Listen` werden.

Mögliche Lösung:

```
sed -nE 's/^([^[:space:]]+).*/\1/p' "$uebung"/config/*.conf
```

Die Ausgabe lautet:

- Port
- PermitRootLogin
- Listen
- ServerName

Der Ausdruck arbeitet folgendermaßen:

1. **tar -xOzf archiv.tar.gz dateipfad** Extrahiert ausschließlich die Zieldatei direkt nach stdout
2. **Streaming ohne Festplatten-I/O** Keine temporären Dateien im Dateisystem nötig
3. **grep filtert Textstream** Treffer werden sofort auf dem Terminal oder im Skript ausgegeben

Die Option `-n` unterdrückt die normale Ausgabe von `sed`. Nur Zeilen, auf die die Ersetzung angewendet wurde, erscheinen durch das abschließende `p`.

Filtere nun ausschließlich Zeilen, die mit `ServerName` oder `PermitRootLogin` beginnen:

```
sed -nE '/^(ServerName|PermitRootLogin)[[:space:]]/p' \
  "$uebung"/config/*.conf
```

Diese Aufgabe verändert keine Dateien. `sed` schreibt das Ergebnis lediglich auf die Standardausgabe, solange weder `-i` noch eine Umleitung in die Quelldatei verwendet wird.

> ⚠ **Warnung:** Verwende `sed -i` nicht zum Experimentieren mit wichtigen Konfigurationsdateien. Die Option verändert die Dateien direkt. Prüfe einen Ausdruck zuerst ohne `-i` und arbeite bei echten Konfigurationen mit einer Sicherung oder Versionsverwaltung.

Übung 4: Komprimierte Dateien und Archive durchsuchen

Eine einzelne `.gz`-Datei und ein `.tar.gz`-Archiv sehen an ihrer Endung ähnlich aus, enthalten jedoch unterschiedliche Strukturen. Genau diese Unterscheidung wurde im Artikel zur Archivierung und Komprimierung behandelt.

Suche zunächst mit `zgrep` in der einzelnen komprimierten Logdatei:

```
zgrep -HnF -- "ERROR" "$uebung/logs/anwendung.log.gz"
```

`zgrep` dekomprimiert den Datenstrom intern und übergibt ihn an `grep`. Die Datei bleibt unverändert.

Zeige danach den Inhalt des Archivs an:

```
tar -tzf "$uebung/archive/logs.tar.gz"
```

Die Ausgabe enthält die gespeicherten Verzeichnis- und Dateinamen:

```
logs/
logs/anwendung.log
logs/system alt.log
```

Suche nun innerhalb der archivierten Datei `logs/anwendung.log`, ohne das Archiv dauerhaft zu entpacken:

```
tar -xOzf "$uebung/archive/logs.tar.gz" \
```

```
  "logs/anwendung.log" |
  grep -nF -- "ERROR"
```

`tar -xOzf` schreibt den Inhalt der ausgewählten Archivdatei auf die Standardausgabe. Die Pipe leitet diesen Datenstrom unmittelbar an `grep` weiter.

Die Ausgabe enthält keinen Dateinamen, weil `grep` hier einen Datenstrom und keine benannte Datei verarbeitet:

```
2:ERROR Verbindung fehlgeschlagen
```

> **Tipp:** Prüfe mit `tar -tzf` zuerst den exakten Namen des Archiveintrags. `tar -xOzf` benötigt den im Archiv gespeicherten Pfad und nicht den ursprünglichen absoluten Dateipfad.

Übung 5: `find`, `grep` und `xargs` verbinden

Diese Aufgabe greift den Datenfluss aus Streams, Pipes und Umleitungen erneut auf. Diesmal werden jedoch Dateinamen und nicht nur gewöhnliche Textzeilen weitergegeben.

Suche zunächst mit `find -exec` in allen Logdateien nach `ERROR`:

```
find "$uebung/logs" -type f -name "*.log" \
  -exec grep -HnF -- "ERROR" {} +
```

Führe dieselbe Suche anschließend mit einer nullterminierten Übergabe an `xargs` aus:

```
find "$uebung/logs" -type f -name "*.log" -print0 |
  xargs -0 -r grep -HnF -- "ERROR"
```

Beide Befehle müssen auch die Datei `system alt.log` korrekt verarbeiten. Das Leerzeichen im Dateinamen darf nicht zu einer Aufteilung in mehrere Argumente führen.

Teste zum Vergleich die unsichere Variante:

```
find "$uebung/logs" -type f -name "*.log" |
  xargs grep -HnF -- "ERROR"
```

Für `system alt.log` sollte `grep` Fehlermeldungen ausgeben, weil `xargs` den Pfad am Leerzeichen trennt. Genau dieser kontrollierte Fehler zeigt, weshalb `-print0` und `-0` zusammengehören.

Prüfe zum Abschluss, welche Logdateien tatsächlich `ERROR` enthalten, und übergib nur deren Namen an `stat`:

```
find "$uebung/logs" -type f -name "*.log" -print0 |
  xargs -0 -r grep -lZF -- "ERROR" |
  xargs -0 -r stat --
```

Die Nullterminierung bleibt dabei durch die gesamte Kette erhalten:

1. **find selektiert Dateien**Filtert Pfade nach Typ, Datum und Name (-type f -name "*.conf")

2. **Sichere Trennung (-print0)** NUL-Byte schützt Pfade mit Leerzeichen und Sonderzeichen
3. **xargs -0 bündelt Argumente** Übergibt Pfadlisten in optimierten Blöcken an den Folgeprozess
4. **grep durchsucht Inhalte** Durchsucht parallel oder sequenziell nach Suchbegriffen (-Hn "Port")

Kontrollfragen

Beantworte die folgenden Fragen ohne die vorherigen Abschnitte nachzuschlagen:

1. Warum ist `locate` bei unmittelbar zuvor erstellten Dateien möglicherweise nicht aktuell?
2. Welche `find`-Option beschränkt die Suche auf reguläre Dateien?
3. Wann ist `grep -F` gegenüber einem regulären Ausdruck die bessere Wahl?
4. Welche Bedeutung besitzen `^` und `$` in einem regulären Ausdruck?
5. Warum kann `zgrep` eine `.gz`-Datei durchsuchen, aber nicht ohne Weiteres alle Dateien eines `.tar.gz`-Archivs?
6. Weshalb müssen `find -print0` und `xargs -0` gemeinsam verwendet werden?
7. Wann ist `find -exec ... {} +` einfacher als eine Kombination mit `xargs`?
8. Warum sollte eine Suchabfrage vor einer schreibenden oder löschenden Aktion zunächst nur lesend geprüft werden?

Die Antworten ergeben sich nicht aus dem Auswendiglernen einzelner Optionen, sondern aus dem jeweiligen Datenfluss:

- Woher kommen die Dateinamen?
- Wie werden sie getrennt?
- Welcher Befehl interpretiert sie als Argumente oder Inhalte?

Befehlsreferenz (Cheatsheet)

find /etc -name "*.conf"

$ `Sucht Dateien nach exaktem oder maskiertem Namen im Dateisystem`

find /var/log -iname "*error*"

$ `Case-insensitive Namenssuche (ignoriert Groß- und Kleinschreibung)`

find /tmp -type f -empty

$ `Filtert nach Dateityp (f=Datei, d=Verzeichnis, l=Symbolischer Link) und Leerdateien`

find /var/log -type f -mtime -7

$ `Sucht nach Änderungszeit (-n jünger als n Tage, +n älter als n Tage)`

find /var -type f -size +100M

$ `Sucht Dateien nach Mindest- oder Maximalgröße (c=Bytes, k=KB, M=MB, G=GB)`

find /bin -perm -4000

$ `Filtert nach Berechtigungsbits (z. B. SUID- oder SGID-gesetzte ausführbare Dateien)`

find /etc -name "*.conf" -exec grep -H "Port" {} +

Führt Befehl hocheffizient gebündelt mit den Trefferpfaden als Argumente aus

find . -type f -print0 | xargs -0 grep -l "ERROR"

$ `Sichere Pipeline-Übergabe auch bei Leer- und Sonderzeichen im Pfadnamen`

locate nginx.conf

$ `Schnelle Dateinamensuche über den indizierten Datenbankbestand (plocate)`

locate -e backup.tar.gz

$ Zeigt nur Index-Treffer an, die im Dateisystem tatsächlich noch existieren

sudo updatedb

$ Aktualisiert die locate/plocate-Indexdatenbank manuell

grep -E "^(error|warn)" /var/log/syslog

$ Durchsucht Dateien nach erweiterten regulären Ausdrücken (ERE)

grep -F "[DEBUG]" app.log

$ Feste Zeichenkette ohne Regex-Overhead extrem schnell durchsuchen

grep -i -n -v "^#" /etc/ssh/sshd_config

$ -i ignoriert Groß/Klein, -n zeigt Zeilennummern, -v invertiert das Suchergebnis

grep -rl "PermitRootLogin" /etc/ssh/

$ Rekursive Suche über Verzeichnisbäume, gibt ausschließlich Dateinamen mit Treffern aus

zgrep -i "failed" /var/log/auth.log.1.gz

$ Durchsucht gzip-komprimierte Logdateien direkt ohne Entpacken auf Festplatte

tar -xOzf logs.tar.gz logs/app.log | grep "ERR"

$ Streamt eine Archivdatei direkt nach stdout und filtert den Textstrom über grep

sed -n 's/^user=\([^]\).*/\1/p' auth.log

$ Extrahiert und modifiziert Textteile über Mustersuche und Rückreferenzen

Weiterführende Ressourcen

Fazit

Du hast es geschafft – im siebten Teil der LPIC-1-Serie hast du gelernt, Dateien und Inhalte unter Linux gezielt zu suchen, Ergebnisse einzugrenzen und die gefundenen Daten sicher weiterzuverarbeiten.

Mit `find` kannst du den aktuellen Zustand des Dateisystems anhand von Namen, Typen, Größen, Zeitstempeln und weiteren Eigenschaften durchsuchen. `locate` und `plocate` liefern ihre Ergebnisse dagegen aus einer Datenbank. Das ist schneller, setzt jedoch einen aktuellen Dateiindex voraus.

Für die Inhaltssuche steht `grep` im Mittelpunkt. Feste Zeichenfolgen lassen sich mit `-F` eindeutig behandeln, während reguläre Ausdrücke komplexere Suchmuster ermöglichen. Mit `sed` kannst du passende Daten anschließend nicht nur anzeigen, sondern gezielt extrahieren oder umformen.

Auch komprimierte Daten bleiben durchsuchbar. `zgrep`, `bzgrep` und `xzgrep` arbeiten mit einzelnen komprimierten Dateien. Bei einem `tar`-Archiv musst du dagegen zuerst die enthaltenen Dateien bestimmen und den gewünschten Inhalt anschließend über die Standardausgabe weiterreichen. Dieser Teil baut unmittelbar auf der zuvor behandelten Archivierung und Komprimierung auf.

Die wichtigsten Punkte im Überblick

`find` sucht im Dateisystem, `locate` in einem Index und `grep` innerhalb von Dateien. Reguläre Ausdrücke beschreiben nicht den Suchort, sondern das Muster, das in einem Text erkannt werden soll. Bei komprimierten Dateien entscheidet die Datenstruktur darüber, ob ein Werkzeug wie `zgrep` ausreicht oder zuerst ein Archiv mit `tar` geöffnet werden muss.

Durch die Kombination mit Streams, Pipes und Umleitungen lassen sich aus diesen Werkzeugen vollständige Suchketten aufbauen. `find -exec ... {} +` eignet sich für die direkte Übergabe an einen Befehl. Benötigst du `xargs`, schützt die Kombination aus `-print0` und `-0` Dateinamen mit Leerzeichen oder Zeilenumbrüchen vor einer fehlerhaften Aufteilung.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien

> □□ **Tipp:** Begrenze zuerst den Suchbereich und filtere anschließend die Inhalte. Eine enge Suche liefert nicht nur übersichtlichere Ergebnisse, sondern vermeidet auch unnötige Zugriffe auf Verzeichnisse und Dateien, die für die eigentliche Aufgabe keine Rolle spielen.

Die behandelten Werkzeuge gehören sowohl für die LPIC-1-Prüfung als auch in der praktischen Linux-Administration zum grundlegenden Handwerkszeug. Entscheidend ist nicht, jede Option auswendig zu kennen. Du solltest erkennen können, welche Datenquelle durchsucht wird, wie aktuell sie ist und in welcher Form das Ergebnis an den nächsten Befehl übergeben wird.

Alle bisher veröffentlichten Teile findest du in der Übersicht der LPIC-1-Serie.

□□ **Zur Kursübersicht:** Alle LPIC-1-Artikel & Module

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-suchen-und-extrahieren-von-daten-aus-dateien