Einführung in relationale Datenbanken und semantische Dateisysteme

Stand: 23. Juli 2008

Inhaltsverzeichnis

1 Vorwort

Dieses Dokument enthält im ersten Teil eine Einführung in relationale Datenbanken und semantische Dateisysteme. In der heutigen Zeit fließen Konzepte aus beiden Welten in aktuelle Speichersysteme ein. Oft sind jedoch Personen, die sich damit befassen (müssen), nur mit einer der beiden Denkweisen vertraut. An diese Gruppe richtet sich das vorliegende Dokument.

Darüber hinaus werden weiterführende Informationen vermittelt, und zwar zu sog. Metadaten, die in vielen Dateiformaten insbesondere aus dem Multimedia-Bereich enthalten sind (→ *4*). Leider werden diese Informationen innerhalb des jeweiligen Dateikörpers gespeichert und sind deshalb für das Dateisystem unsichtbar. Nur Programme, die den Aufbau der internen Dateiformate kennen, können diese Zusatzinformationen extrahieren und nutzen. Semantische Dateisysteme (→ *5*) können eingesetzt werden, um dieses Problem zu lösen.

2 Datenbanktechnik

Datenbanksysteme kommen immer dann zum Einsatz, wenn an die Datenhaltung besondere Anforderungen betreffend der Zuverlässigkeit, des zu speichernden Volumens, der Ausfallsicherheit, des gleichzeitigen Zugriffs oder der Komplexität der Datenbeschreibung gestellt werden. Viele Grundkonzepte der Datenbanktechnologie wurden in den 1970er-Jahren entwickelt und seitdem beständig weiterentwickelt. Nach einer Stabilisierung in den 1980er-Jahren wird die aktuelle Entwicklung von speziellen Anforderungen aus den Bereichen Multimedia oder Dateisysteme geprägt **[Saa05]**.

Datenbanken bestehen aus einer logischen Ebene und einer physikalischen Ebene, die die logische Struktur speichert und verwaltet. Operationen auf einer Datenbank dürfen nur von einem DBMS (Datenbank-Management-System) durchgeführt werden, ein »direkter« Zugriff auf Datensätze oder Dateien ist nicht vorgesehen. Dadurch können an einer zentralen Stelle Zugriffs- und Konsistenz-Kontrollen vorgenommen werden **[Mat03]**. Eine Datenbank zusammen mit einem DBMS wird Datenbanksystem genannt; theoretisch können mehrere DBMS für eine konkrete Datenbank existieren, die sich dann in ihrem Funktionsumfang unterscheiden:

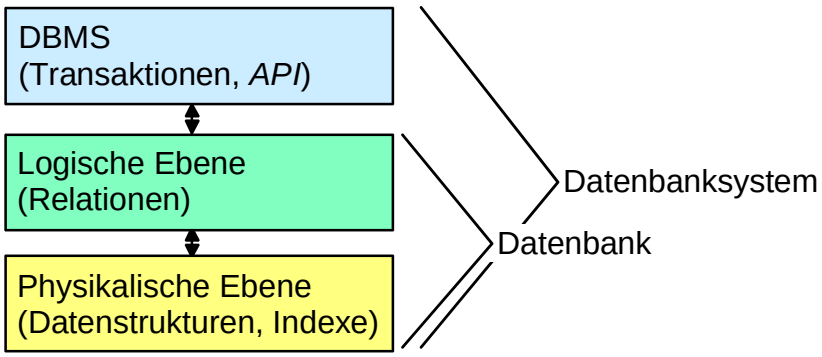


Abb. 2-1: Begriffsdefinition

Erschwerend kommt hinzu, dass der Begriff »Datenbank« umgangssprachlich ein Synonym für »Datenbanksystem« ist, und nicht im Sinne von → *Abb. 2-1* gebraucht wird.

2.1 Relationen (logische Ebene)

Es gibt verschiedene Datenbankmodelle, die für die Datenbeschreibung eingesetzt werden. Die Konzepte eines hierarchisches Datenmodells oder eines Netzwerkmodells wurden zunehmend von Systemen mit relationalem Modell verdrängt, so dass heutige Systeme meistens relationale Datenbanken sind. Der Begriff »Datenbank« wird sogar häufig stellvertretend für relationale Modelle verwendet **[Saa05]**.

Die Grundlagen der Theorie der relationalen Datenbank wurden von Codd in **[Cod70]** beschrieben. Die erste kommerziell erfolgreiche relationale Datenbank wurde Ende der 1970er-Jahre von der Firma Oracle auf den Markt gebracht **[Wik05h]**.

Das grundlegende Element einer relationalen Datenbank ist die *Relation*. Vereinfacht ausgedrückt handelt es sich dabei um eine Matrix oder Tabelle mit Tupeln (Datensätzen) gleicher Struktur (tatsächlich ist der Begriff *Tabelle* belegt und beschreibt eine Verallgemeinerung der Relation). Eine relationale Datenbank ist eine Sammlung mehrerer Relationen.

Eine Spalte einer Relation enthält ein Attribut *A*, das aus einem Namen und einem Wertebereich *dom*(*A*) besteht, etwa **String** oder **int**. Die Attribute bilden die Struktur der Relation. Der Name eines Attributs muss innerhalb der Relation eindeutig sein, zwei Relationen dürfen aber jeweils ein Attribut mit gleichem Namen besitzen. Folgendes Beispiel zeigt eine Relation für Adressen:

R	**Vorname** **String**	**Nachname** **String**	**Straße** **String**	**Hausnummer** **int**	**Postleitzahl** **int**	**Ort** **String**
r(R)	Anton	Meier	Parkstraße	1	44139	Dortmund
	Berta	Müller	Schlossallee	1	44227	Dortmund

Abb. 2.1-1: Relation für Adressen

Die oben dargestellte Relation kann keine Adresse mit der Hausnummer »1a« aufnehmen, da *dom*(**Hausnummer**) = **int** keine Buchstaben zulässt. Die blau gefärbte Titelzeile in → *Abb. 2.1-1* entspricht der Struktur, die bereits beim Erstellen der Relation festgelegt wurde und nicht mehr geändert werden kann. Folgende Abbildung zeigt die Struktur einer Tabelle, die Teil einer vom Autor betreuten Datenbank ist:

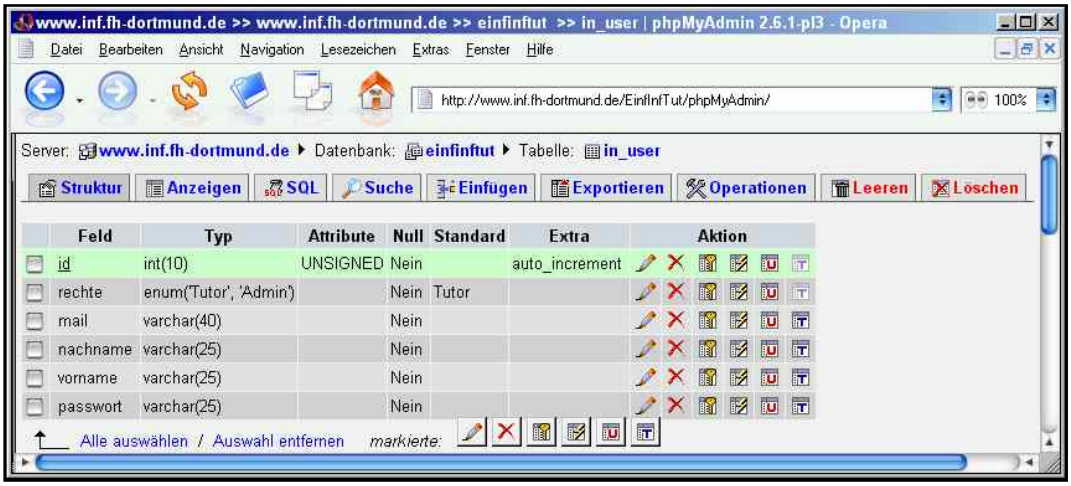


Abb. 2.1-2: Struktur einer Tabelle

In einer Relation dürfen keine identischen Tupel enthalten sein, während dies bei einer Tabelle sehr wohl erlaubt ist. Zwischen einer Relation und einer Tabelle besteht also derselbe Unterschied wie

zwischen einer Menge und einer Multimenge in der Mathematik. Folgende Abbildung zeigt eine Tabelle, die keine Relation ist, da die ersten beiden Tupel identisch sind:

R	Vorname String	Nachname String	Straße String	Hausnummer int	Postleitzahl int	Ort String
t(R)	Anton	Meier	Parkstraße	1	44139	Dortmund
	Anton	Meier	Parkstraße	1	44139	Dortmund
	Berta	Müller	Schlossallee	1	44227	Dortmund

Abb. 2.1-3: Tabelle, aber keine Relation

Eine wichtige Eigenschaft von Relationen und Tabellen ist, dass sie bestimmte Operationen unterstützen, die zusammen die Relationenalgebra bilden.

2.2 Relationenalgebra (logische Ebene)

Die Relationenalgebra definiert Verknüpfungen zwischen Relationen, Attributen und Tupeln. Der Zugriff auf Daten erfolgt bei relationalen Datenbanken ausschließlich durch solche Operationen; ein direkter Zugriff auf Dateien ist weder möglich noch erwünscht. Die folgenden Beispiele erläutern die wichtigsten Verknüpfungen und verwenden dabei die Relationen *r* und *s* als Eingabe:

r	A	B	C
	1	2	3
	4	5	6

s	A	B	C
	7	8	9
	4	5	6

Abb. 2.2-1: Beispielrelationen **[Wik05f]**

Die folgenden Beispiele wurden **[Wik05f]** entnommen, anhand von **[Mat03]** und **[Saa05]** überprüft und der in diesem Dokument verwendeten Terminologie angepasst.

2.2.1 Vereinigung

Bei der Vereinigung $r \cup s$ werden alle Tupel der Relation *r* mit allen Tupeln der Relation *s* zu einer einzigen Relation vereint. Voraussetzung dafür ist, dass *r* und *s* die gleiche Struktur haben:

$$r \cup s := \{t | t \in r \vee t \in s\}$$

r ∪ s	A	B	C
	1	2	3
	4	5	6
	7	8	9

Abb. 2.2-2: Vereinigung **[Wik05f]**

2.2.2 Differenz

Bei der Differenzoperation *r - s* werden aus der ersten Relation alle Tupel entfernt, die auch in der zweiten Relation vorhanden sind:

$$r - s := \{t | t \in r \wedge t \notin s\}$$

	A	B	C
r - s	1	2	3

Abb. 2.2-3: Differenz **[Wik05f]**

2.2.3 Durchschnitt

Das Ergebnis der Durchschnittsoperation $r \cap s$ ist eine Relation mit allen Tupeln, die sich sowohl in *r* als auch in *s* finden lassen. Der Mengendurchschnitt lässt sich darüber hinaus auch als Mengendifferenz ausdrücken: $r \cap s = r - (r - s)$:

$$r \cap s := \{t | t \in r \wedge t \in s\}$$

	A	B	C
r ∩ s	4	5	6

Abb. 2.2-4: Durchschnitt **[Wik05f]**

2.2.4 Projektion

Die Projektion extrahiert einzelne Attribute aus einer Relation, sie blendet also Spalten aus. Die Projektion ist definiert bezüglich einer Projektionsliste *ß*, die die gewünschten Attribute enthält:

$$\pi_{ß}(r) := \{t_{ß} | t \in r\}$$

$\pi_{A,B}(r)$	A	B
	1	2
	4	5

$\pi_{A}(s)$	A
	1
	4

Abb. 2.2-5: Projektion **[Wik05f]**

2.2.5 Selektion

Bei der Selektion wird mit einem Vergleichsausdruck eine Auswahl von Tupeln festgelegt, die dann in die Ergebnisrelation übernommen werden. Es werden also Zeilen ausgeblendet:

$$\sigma_{Ausdruck}(r) := \{t | t \in r \wedge t \ \ erfüllt \ \ Ausdruck\}$$

$\sigma_{C>4}$(r)

A	B	C
4	5	6

Abb. 2.2-6: Selektion **[Wik05f]**

2.2.6 Kreuzprodukt

Das Kreuzprodukt $r \times s$ ähnelt stark dem Kartesischen Produkt der Mengenlehre. Das Resultat des Kreuzprodukts ist die Menge aller Kombinationen aller Tupel aus *r* und *s*, d.h. jede Zeile der einen Relation wird mit jeder Zeile der anderen Relation kombiniert. Dazu ist es jedoch erforderlich, dass die beiden Relationen keine gleichnamigen Attribute enthalten. Aus diesem Grund können die Beispielrelationen *r* und *s* aus → *Abb. 2.1-4* hier nicht verwendet werden:

$$r \times s := \{(a_1, a_2, ..., a_n, b_1, b_2, ..., b_m) | (a_1, a_2, ..., a_n) \in r \wedge (b_1, b_2, ..., b_m) \in s\}$$

r

A	B	C
1	2	3
4	5	6
7	8	9

s

D	E
1	2
8	9

r × s

A	B	C	D	E
1	2	3	1	2
1	2	3	8	9
4	5	6	1	2
4	5	6	8	9
7	8	9	1	2
7	8	9	8	9

Abb. 2.2-7: Kreuzprodukt **[Wik05f]**

Werden zwei Relationen kombiniert, die gleichnamige Attribute enthalten, so handelt es sich bei der Operation nicht um ein Kreuzprodukt, sondern um einen Join.

2.2.7 Join

Die Join-Operation verbindet zwei Relationen, bei denen ein Attribut in beiden Relationen mit identischem Namen und Datentyp vorhanden ist. Die Join-Operation liefert wie das Kreuzprodukt als Ergebnis eine Relation, deren Struktur aus den Attributen beider Ursprungsrelationen gebildet wird.

Im Gegensatz zum Kreuzprodukt, das nur auf Relationen mit disjunkten Attributnamen angewendet werden darf, wird das Attribut mit gemeinsamen Namen nur einmal in den Join-Verbund übernommen, damit die Relationseigenschaft für das Ergebnis erhalten bleibt. Das Ergebnis ist wohl definiert, da nur Tupel übernommen werden, die im gemeinsamen Attribut identische Werte enthalten. Beispiel:

$$r := (a_1, a_2, ..., a_n, c_1, c_2, ..., c_x)$$
$$s := (b_1, b_2, ..., b_m, c_1, c_2, ..., c_x)$$
$$r \bowtie s := \{x \cup y_{[b_1, b_2, ..., b_m]} \mid x \in r \wedge y \in s \wedge x_{[c_1, c_2, ..., c_x]} = y_{[c_1, c_2, ..., c_x]}\}$$

r

A	B	C
1	2	3
4	5	6
7	8	9

s

A	D
1	5
7	4

r ⋈ s

A	B	C	D
1	2	3	5
7	8	9	4

Abb. 2.2-8: Inner Join **[Wik05f]**

Der oben dargestellte Join-Verbund heißt »Inner Join«. Wenn leere Attribute in einem Tupel erlaubt sind, sog. *Nullmarken*, so kann auch ein »Outer Join« gebildet werden. Beim Inner Join enthält die Ergebnisrelation nur Tupel, die aus Tupeln der beiden Eingangsrelationen gebildet wurden; wenn für ein Tupel der einen Relation kein Gegenstück in der anderen Relation vorhanden ist, so erscheint es nicht im Ergebnis. Beim Outer Join ist das anders: wird für ein Tupel kein Gegenstück gefunden, so werden die nicht vorhandenen Attribute dieses Tupels mit Nullmarken gefüllt. Dadurch erscheinen dennoch alle Tupel im Ergebnis.

Wenn alle Tupel der linken Eingaberelation ggf. durch Nullmarken ergänzt werden sollen, so spricht man von einem »Left Outer Join«, im anderen Fall von einem »Right Outer Join«. Eine Kombination wird »Full Outer Join« genannt. Folgendes Beispiel zeigt einen Left Outer Join für die Relationen aus → *Abb. 2.2-8*:

r

A	B	C
1	2	3
4	5	6
7	8	9

s

A	D
1	5
7	4

r ⟕ s

A	B	C	D
1	2	3	5
4	5	6	
7	8	9	4

Abb. 2.2-9: Left Outer Join

2.2.8 Umbenennung

Attribute können umbenannt werden, um u.A. Joins von disjunkten Relationen oder Kreuzprodukte von Relationen mit sonst gleichen Attributnamen zu ermöglichen.

$\beta_{[B \to X]}(r)$

A	X	C
1	2	3
4	5	6

Abb. 2.2-10: Umbenennen von Attributen **[Wik05f]**

2.2.9 Eigenschaften der Relationenalgebra

Die Relationenalgebra ist orthogonal, d.h. das Ergebnis einer Verknüpfung zweier Relationen ist wieder eine Relation. Operationen können also beliebig tief geschachtelt werden, so dass sich sehr komplexe Ausdrücke bilden lassen. Als Beispiel sei die Struktur folgender Relationen gegeben **[Wik05f]**:

- *KUNDE*: *Kundennr, Name, Wohnort, Kontostand*
- *LIEFERANT*: *Lieferantennr, Name, Ort, Telefon*
- *WARE*: *Warennr, Bezeichnung, Lieferantennr, Preis*

Mittels Ausdrücken der Relationenalgebra können nun verschiedene Suchanfragen über die Relationen formuliert werden **[Wik05f]**:

- Die Preise aller Waren:
  $\pi_{Bezeichnung,\ Preis}(WARE)$
- Alle Lieferanten aus Bremen:
  $\sigma_{Ort="Bremen"}(LIEFERANT)$
- Ort in *LIEFERANT* umbenennen, z.B. um Mengenoperationen anwenden zu können:
  $\beta_{Ort \to Wohnort}(LIEFERANT)$
- Die Telefonnummer alle Lieferanten, die Gemüse in Bremen liefern:
  $\pi_{Telefon}(\sigma_{Bezeichnung="Gemüse" \wedge Ort="Bremen"}(LIEFERANT \bowtie WARE))$
- Alle Orte, die wenigstens einen Lieferanten und wenigstens übrer einen Kunden verfügen:
  $\pi_{Ort}(\beta_{Wohnort \to Ort}(KUNDE)) \cap \pi_{Ort}(LIEFERANT)$

Verschiedene Operationen können durch andere ersetzt werden: ein Join lässt sich beispielsweise als Bildung des Kreuzprodukts und anschließende Selektion auffassen. Eine Datenbankanfrage kann also auf mehrere Weisen formuliert werden. Vor der eigentlichen Ausführung sollte die Anfrage daher optimiert werden, also eine möglichst effiziente äquivalente Form gefunden werden. Zum Beispiel sind folgende Terme algebraisch äquvalent **[Saa05]**:

1. $\sigma_{A=Konstante}(r \bowtie s)$ (A sei ein Attribut von r)
2. $(\sigma_{A=Konstante}(r)) \bowtie s$

Nehmen wir nun an, dass die Relation r 100 Tupel und s 50 Tupel enthält. Nehmen wir ferner an, dass die Tupel sequenziell, etwa in der Einfügereihenfolge, abgelegt sind. Ein Join wird von ineinander geschachtelten Schleifen durchgeführt, hat also den Aufwand $n * m$ bei den Relationsgrößen n und m. Außerdem soll angenommen werden, dass eine Selektion über eine Konstante 10% der Tupel selektiert und dass der Join hier alle Kombinationen liefert, da die beiden Relationen nur ein Attribut gemeinsam haben. Unter diesen Annahmen ergibt sich etwa folgender Aufwand (gemessen in der Anzahl der Zugriffe auf Einzeltupel) **[Saa05]**:

1. Im ersten Fall enthält das Zwischenergebnis der Join-Ausführung 50∗100 = 5000 Operationen. Diese Tupel müssen alle für die Selektion durchlaufen werden. Das ergibt insgesamt 10000 Operationen **[Saa05]**.
2. Laut Annahme erfüllen 10 Tupel in *r* die Bedingung *A=Konstante*. Die Ausführung der Selektion erfordert 100 Zugriffe und die Ausführung des Verbundes nun zusätzlich 10∗50 = 500 Operationen. Insgesamt werden somit 600 Operationen benötigt **[Saa05]**.

Die beiden unterschiedlichen Ausführungen ergeben in diesem Beispiel Aufwandsabschätzungen, die fast um den Faktor 20 differieren. Eine wichtige Heuristik besteht daher darin, Projektionen und vor allem Selektionen möglichst früh auszuführen, um die Größe von Zwischenergebnissen zu minimieren. Weitere Regeln ergeben sich jeweils aus einer konkreten Implementierung **[Saa05]**.

2.3 Relationale DBMS

Ein weiteres Merkmal relationaler Datenbanken besteht nach **[Cod70]** darin, dass die Auswahl von Tupeln nur über die Daten selbst erfolgt. Es gibt also keine Zeiger oder Referenzen, sondern nur die Möglichkeit der Selektion. Zur Steuerung des Datenbanksystems kommunizieren Client und Server in einer bestimmten Abfragesprache miteinander. Vor allem *SQL* spielt als solche Sprache eine große Rolle. Die drei Buchstaben »SQL« sind Namensbestandteil vieler Datenbank-Server, wie zum Beispiel mySQL, PostgreSQL oder Microsoft SQL Server **[Wik05i]**. SQL wurde immer wieder ergänzt und erweitert, so dass es mehrere Versionen gibt, die als ISO- bzw. ANSI-Standard vorliegen und lizenziert werden können **[Wik05g]**:

	Name	Alias
1986	SQL-86	SQL-87
1989	SQL-89	
1992	SQL-92	SQL2
1999	SQL:1999	SQL3
2003	SQL:2003	

Abb. 2.3-1: SQL-Versionen **[Wik05g]**

SQL operiert standardmäßig auf Tabellen und nicht auf Relationen, da so bestimmte Operationen effizienter implementiert werden können: nach einer Projektion müssen beispielsweise keine identischen Tupel gesucht und aus dem Ergebnis entfernt werden.

2.3.1 SQL-Syntax

SQL sieht in einer Datenbank eine Sammlung von Tabellen. Da ein Datenbank-Server mehrere Datenbanken verwalten kann, wird mit dem USE-Befehl eine Standard-Datenbank ausgewählt **[Mat03]**:

```
USE MeineDatenbank
```

In einer Datenbank kann mit CREATE TABLE eine neue Tabelle angelegt und nach Gebrauch mit DROP TABLE wieder gelöscht werden **[Mat03]**:

```
CREATE TABLE Adressen
(
  Nummer INT NOT NULL,
  Vorname VARCHAR(50) NOT NULL,
  Name VARCHAR(50) NOT NULL,
  Strasse VARCHAR(50),
  Hausnummer INT,
  Geburtstag DATE
)

DROP TABLE Adressen
```

NOT NULL legt fest, dass Nullmarken im entsprechenden Attribut nicht erlaubt sind. In eine existierende Tabelle können nun mit INSERT INTO neue Tupel eingefügt werden. Die Tupel müssen dabei natürlich der Tabellenstruktur entsprechen, andernfalls würde der Befehl vom SQL-Server zurückgewiesen werden **[Mat03]**:

```
INSERT INTO Adressen VALUES (1,'Petra','Mustermann','Schlossallee',1,'01.01.1980')
INSERT INTO Adressen (Nummer,Vorname,Name) VALUES (2,'Peter','Mustermann')
```

Mit DELETE können Tupel aus einer Tabelle entfernt werden. Die Auswahl der zu löschenden Tupel erfolgt dabei gemäß **[Cod70]** durch die Daten selbst **[Mat03]**:

```
DELETE * FROM Adressen WHERE Strasse='Schlossallee'
```

Durch obigen SQL-Befehl werden aus der Tabelle Adressen alle Tupel entfernt, deren Attribut Strasse den Inhalt »Schlossallee« hat. Neben dem Verändern von Daten stellen Abfragen die wichtigste Operation dar. In SQL werden Selektionen (→ *2.2.5*) und Projektionen (→ *2.2.4*) mit dem Befehl SELECT realisiert, ggf. einschließlich einer Sortierung mit ORDER BY **[Mat03]**:

```
SELECT * FROM Adressen WHERE Strasse='Schlossallee'          //Selektion
SELECT (Name,Vorname) FROM Adressen ORDER BY Name            //Projektion
SELECT (Name,Vorname) FROM Adressen WHERE Strasse='Schlossallee'   //Beides
```

Da SQL mit Tabellen arbeitet, können bei einer Selektion, vor allem bei gleichzeitiger Projektion, identische Tupel im Suchergebnis enthalten sein. Dies wird durch Angabe des Schlüsselwortes DISTINCT im SQL-Befehl verhindert **[Mat03]**:

```
SELECT DISTINCT (Name,Vorname) FROM Adressen ORDER BY Name
```

Enthält die Tabelle Adresse also zwei Personen mit gleichem Namen, aber unterschiedlichen Adressen, erscheinen Name und Vorname bei Ausführung des obigen Befehls trotzdem nur einmal im Suchergebnis. DISTINCT kann die Ausführungszeit drastisch erhöhen (→ *2.2.9*).

Auch komplexe Operationen wie Joins können mit SQL realisiert werden. Dazu sei ein etwas umfangreicheres Beispiel gegeben:

```
CREATE TABLE Adressen
(
  Nummer INT NOT NULL,
  Vorname VARCHAR(50) NOT NULL,
  Name VARCHAR(50) NOT NULL,
  Strasse VARCHAR(50),
  Hausnummer INT,
  Geburtstag DATE
)

INSERT INTO Adressen VALUES (1,'Petra','Mustermann','Schlossallee',1,'01.01.1980')
INSERT INTO Adressen VALUES (2,'Peter','Mustermann','Schlossallee',1,'01.01.1978')
INSERT INTO Adressen VALUES (3,'Udo','Meier','Winkelgasse',25,'24.03.1981')
INSERT INTO Adressen VALUES (4,'Uta','Müller','Am Park',30,'12.05.1975')

CREATE TABLE Schulden
(
  Person INT NOT NULL
  Betrag FLOAT NOT NULL
)

INSERT INTO Schulden VALUES (3,49.95)
INSERT INTO Schulden VALUES (1,23.57)
```

Die neu eingeführte Tabelle Schulden enthält neben einer fortlaufenden Nummer die Nummer einer Adresse und einen Rechnungsbetrag. Mit folgendem SQL-Statement wird nun die Adresse aller Personen gebildet, deren Schulden (etwa bei einer Bank) mehr als 40 Euro betragen:

```
SELECT (Vorname,Name,Strasse,Hausnummer) FROM
  Adressen JOIN Schulden ON Schulden.Person=Adressen.Nummer
  WHERE Schulden.Betrag > 40.00
```

Die Tabelle, auf die sich eine SELECT-Anweisung bezieht, kann also auch durch ein JOIN in Echtzeit generiert werden. Hinter ON wird eine Bedingung angegeben, bei deren Erfüllung die Tupel verbunden werden **[Mat03]**.

2.4 Implementierung (physikalische Ebene)

Durch die Client-Server-Architektur (→ *2.3*) und die Verwendung von SQL (→ *2.3.1*) kann der Datenbank-Client von der eigentlichen Implementierung abstrahieren. Dennoch hat die interne Struktur des Datenbanksystems große Auswirkungen auf die Performanz.

Die Berechnung eines Joins gehört zu den wichtigsten Operationen innerhalb relationaler Datenbanken. Dadurch ist die effiziente Realisierung von Joins besonders kritisch für die Effizienz eines Datenbanksystems. Zwei bekannte Algorithmen sind Merge-Join und Nested-Loop-Join **[Saa05]**:

- Der Merge-Join oder »Verbund durch Mischen« zweier Relationen *r* und *s* ist besonders dann effizient, wenn eine oder beide Relationen sortiert nach den zu verbindenden Attributen vorliegen. Seien *x* diese Attribute, also $x = r \cap s$. Der Merge-Join geht dann wie folgt vor: *r* und *s* werden nach *x* sortiert, falls diese nicht schon sortiert sind. Dann werden *r* und *s* gemischt, d.h. beide Relationen werden sequenziell durchlaufen und passende Paare in das Ergebnis aufgenommen.
- Der Nested-Loop-Join realisiert eine doppelte Schleife über beide Relationen.

Die Laufzeit der beiden Algorithmen hängt maßgeblich von der Zeit ab, die der Zugriff auf ein bestimmtes Tupel erfordert, und damit von der Art der internen Datenspeicherung. Natürlich könnten die Tupel in willkürlicher Reihenfolge in einer Liste angeordnet werden, aber dies würde keinen Vorteil bei Anfrageoperationen erwirken. Darum werden Tupel oft in Abhängigkeit vom Wert eines Attributs, des »Primärschlüssels«, abgespeichert, um so einen schnellen Zugriff zumindest über dieses Attribut zu ermöglichen. Verbreitet sind zwei Alternativen **[Saa05]**:

- Die Tupel können geordnet abgelegt werden. Das Einfügen und Suchen erfolgt dann mit logarithmischem Aufwand, Verbundoperationen über den Primärschlüssel werden beschleunigt.
- Eine gestreute Speicherung mittels einer Hashfunktion führt bei direktem Zugriff über den Primärschlüssel zu konstantem Aufwand, ermöglicht aber kein Durchlaufen in Sortierreihenfolge.

Beide Verfahren erreichen einen schnellen Zugriff über den Primärschlüssel. Strukturen, die den Zugriff über die Werte von Attributen unterstützten, werden als *Index* bezeichnet. Die Speicherung von Tupeln mittels Index erfordert einen hohen Aufwand beim Einfügen und Ändern von Tupeln, da zum Beispiel die sequenzielle Ordnung bewahrt bleiben muss. Dies ist insbesondere bei sehr großen Relationen mit einer Reorganisation der Speicherung verbunden. Einige Datenbanksysteme verzichten daher im Normalfall auf einen Index, so dass neue Tupel einfach am Ende einer Datei angehängt werden können **[Saa05]**.

Nach diesen Vorbemerkungen kann nun der Aufwand der wichtigsten relationalen Operationen abgeschätzt werden. Seien *n* und *m* die Anzahl der Tupel der Relationen **[Saa05]**:

- Die Komplexität der Projektion (→ *2.2.4*) reicht von $O(n)$ (vorliegender Zugriffspfad oder Projektion auf Primärschlüssel) bis $O(n \log n)$ (Duplikateliminierung durch Sortieren).
- Der Aufwand für die Selektion (→ *2.2.5*) reicht von konstantem Aufwand $O(1)$ einer Hash-basierten Zugriffsstruktur bis hin zu $O(n)$, falls ein sequenzieller Durchlauf notwendig ist. In der Regel kann man von $O(\log n)$ als Resultat baumbasierter Zugriffspfade ausgehen.
- Der Aufwand einer Join-Operation (→ *2.2.7*) reicht von $O(n+m)$ bei sortiert vorliegenden Tabellen bis zu $O(n*m)$ bei geschachtelten Schleifen.

2.4.1 Zugriffspfade

Die obige Aufwandsabschätzung zeigt, dass der Zeitbedarf von Operationen ganz erheblich von der internen Organisation der Relation abhängt. Die Strukturen einer gespeicherten Relation lassen sich in zwei Kategorien einteilen: statisch und dynamisch **[Saa05]**.

Das oben erwähnte Hashset impliziert eine statische Speicherung. Zwar ist es in $O(1)$ möglich, neue Tupel anzulegen und zu löschen. Allerdings muss vorher die Gesamtgröße festgelegt werden; wird sie erreicht, ist eine in der Regel eine vollständige und damit zeitaufwändige Neuorganisation der Datei erfordertlich **[Saa05]**.

Diesen Nachteil haben dynamische Organisationsformen nicht. Der einfachste dynamische Aufbau ist eine Heap-Datei. Wie der Name andeutet, werden die Tupel völlig unsortiert »auf einem Haufen gestapelt«. Die Reihenfolge der Tupel ist die zeitliche Reihenfolge der Aufnahme in die Relation, denn neue Tupel werden einfach am Ende der Datei angefügt. Das Suchen eines Tupels erfordert leider ein Durchsuchen der Gesamtdatei und somit linearen Aufwand. Die Löschoperation basiert auf dem Suchen, da das zu löschende Tupel erst gefunden werden muss. Danach wird es als ungültig markiert; alternativ kann bei uniform großen Tupeln das letzte Tupel an die gelöschte Stelle verschoben werden, so dass sich die Dateigröße reduziert **[Saa05]**.

Zusammenfassend kann gesagt werden, dass die Neuaufnahme von Daten mit der Komplexität $O(1)$ zwar konkurrenzlos günstig ist, das Suchen von Daten mit $O(n)$ bei n zu durchsuchenden Tupeln aber den maximalen Aufwand bedeutet **[Saa05]**.

Eine sequenziell geordnete Datei legt Tupel im Gegensatz zur Heap-Datei sortiert ab. Das Einfügen neuer Tupel wird durch das Einhalten der Sortierung natürlich komplizierter. Ist das Tupel zwischen zwei existierenden Tupeln einzusortieren, so werden die nachfolgenden Tupel verschoben. Der Aufwand der Löschoperation bleibt $O(1)$, wenn das Tupel als ungültig markiert wird. Andernfalls ist die Komplexität $O(n)$ **[Saa05]**.

Alle nachfolgenden Organisationsformen werden gegenüber den bisherigen direkten Verfahren:

- die Suche erheblich beschleunigen
- mehr Speicherplatz belegen (u.A. durch Hilfsstrukturen wie Indexdateien)
- mehr Zeitbedarf beim Anlegen und Löschen von Tuplen erfordern

Wird eine Relation anhand des Primärschlüssels in einer sortierten Datei gespeichert, so kann schnell auf einzelne Tupel zugegriffen werden, wenn der Primärschlüssel bekannt ist. Oft ist aber auch ein schneller Zugriff bezüglich anderer Attribute gewünscht. Dazu müssen neben der Relation ergänzende Indexe abgespeichert werden. Bei einer Heap-Datei oder einem Hashset ist sogar für den Primärschlüssel ein Index erforderlich, wenn ein besonders schnelles Auffinden eines Tupels gewünscht wird, da die Tupel nicht sortiert vorliegen. Ein Index über den Primärschlüssel heißt Primärindex, ergänzende Indexe werden Sekundärindex genannt **[Saa05]**.

Zur Indexierung stehen nun verschiedene Datenstrukturen zur Verfügung, die teilweise tief in die Dateiorganisation eingreifen. Es sei nochmal darauf hingewiesen, dass der Aufwand für Änderungen an der Relation mit der Anzahl der Indexe ansteigt.

2.4.1.1 Invertierte Listen

Ein Zugriffspfad in Form einer invertierten Liste besitzt zu jedem Tupel der Hauptdatei einen Eintrag (*w*, *s*) in der Indexdatei. *w* ist dabei der Schlüsselwert des Tupels der Hauptdatei, *s* das zugeordnete Offset innerhalb der Hauptdatei. Da ein Schlüsselwert, außer bei einem Primärschlüssel, mehrfach vorkommen kann, werden für ein *w* entweder mehrere Einträge in der Indexdatei angelegt, wobei die Sortierung der Indexdatei zunächst nach Sekundärschlüsselwerten und dann nach Offset erfolgt, oder für ein *w* wird eine Liste von Adressen innerhalb der Hauptdatei angegeben **[Saa05]**:

Hauptdatei

4711	Anton	Müller
5588	Peter	Mustermann
6834	Peter	Schulze
7754	John	Doe
8832	Antje	Schmitz
9912	Ehrenfried	Höchst
9999	Anton	Mustermann

Indexdatei, Typ 1

Antje	8832
Anton	4711
Anton	9999
Ehrenfried	9912
John	7754
Peter	5588
Peter	6834

Indexdatei, Typ 2

Antje	8832
Anton	4711, 9999
Ehrenfried	9912
John	7754
Peter	5588, 6834

Abb. 2.4-1: invertierte Listen für eine Heap-Datei

Das Suchen innerhalb der invertierten Liste ist in $O(\log n)$ möglich, so dass Suchanfragen bezüglich eines bestimmten Attributs in derselben Zeit verarbeitet werden können. Das Anlegen eines neuen Tupels benötigt die Suche über die Hauptdatei, das anschließende Einfügen in die Hauptdatei und das Anpassen der Indexdateien. Eine Löschoperation wird ebenfalls über eine Suche und das Löschen in der Hauptdatei ausgeführt. Der Indexeintrag muss danach als ungültig markiert oder entfernt werden **[Saa05]**.

Ein Problem invertierter Listen ist die schlechte Skalierbarkeit. Das bedeutet, dass bei großen Datenmengen Leistungseinbußen hinzunehmen sind. Da eine invertierte Liste immer sortiert sein muss, ist das Einfügen oder Löschen von Tupeln problematisch, genau wie bei sortierten Hauptdateien. Da sie sich schlecht in der Größe ändern lassen, werden invertierte Listen »statisch« genannt. Dynamische Indexstrukturen ermöglichen sowohl einen schnellen Zugriff auf Tupel der Hauptdatei als auch eine effiziente Größenanpassung.

2.4.1.2 B-Bäume

Ausgangspunkt dynamischer Baumverfahren im Datenbankbereich ist ein ausgeglichener, balancierter Suchbaum. Das bedeutet, dass die Daten im Baum sortiert anhand eines Attributs abgespeichert werden. Befindet sich an einem Knoten im Baum der Wert w, so werden die Werte $w' < w$ im linken Teilbaum, die Werte $w' > w$ im rechten Teilbaum von w gespeichert. Ein Suchbaum ist balanciert, wenn das Einfügen immer einen Baum ergibt, in dem alle Pfade von der Wurzel zu den Blättern des Baumes gleich lang sind. Das Problem ist dabei, Algorithmen zum Einfügen und Löschen von Elementen zu finden, die diese Eigenschaft bewahren. Als Datenstruktur für den Hauptspeicher sind binäre Suchbäume beliebt, beispielsweise AVL-Bäume **[Saa05]**.

Im Datenbankbereich werden die Knoten der Suchbäume nun auf die Blockstruktur des Datenträgers (→ *3*) zugeschnitten. Ist ein Block z.B. 1024 Byte groß, so sollen auch entsprechend viele Informationen, also mehrere Indexeinträge, in einem Knoten untergebracht werden. Der Suchbaum ist dadurch nicht binär (zwei Nachfolger für einen Knoten), sondern verzweigt breiter **[Saa05]**:

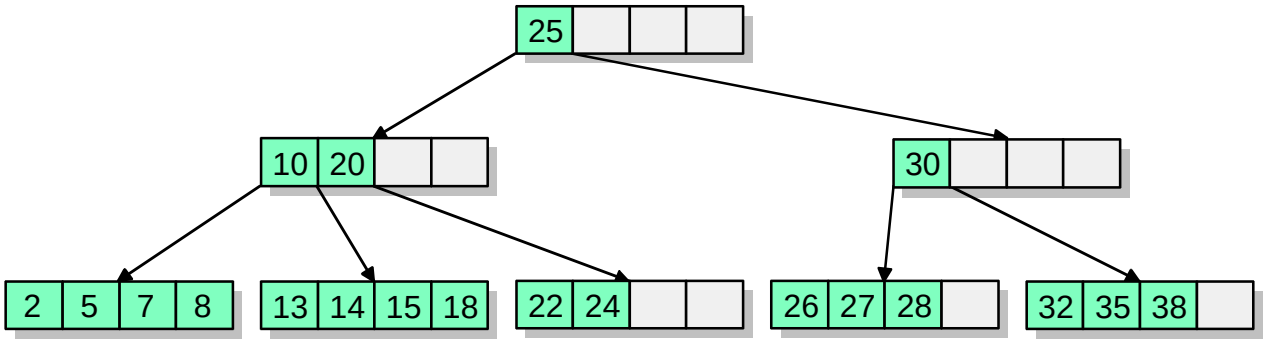


Abb. 2.4-2: B-Baum **[Saa05]**

Im Prinzip ist der B-Baum ein dynamischer, balancierter Indexbaum, bei dem jeder Indexeintrag auf ein Tupel der Hauptdatei zeigt. Ein solcher B-Baum wäre völlig ausgeglichen, wenn alle Wege von der Wurzel zu den Blättern gleich lang sind, und jeder Knoten gleich viele Indexeinträge aufweist. Da ein solches vollständiges Ausgleichen nach Änderungen zu ineffizient bezüglich der Laufzeit wird, wurde folgendes Kriterium für den B-Baum eingeführt: jeder Knoten außer der Wurzel enthält zwischen m und $2m$ Daten. Die Konstante m wird auch »Ordnung« genannt. Falls in der Hauptdatei n Tupel gespeichert werden, ist der Zugriff auf den richtigen Knoten in $\log_m(n)$ möglich. Weitere Vorteile des B-Baumes sind **[Saa05]**:

- Durch das Balancierungskriterium wird eine Eigenschaft nahe an der vollständigen Balance erreicht (das erste Kriterium wird vollständig erfüllt, das zweite zumindest näherungsweise).
- Das Kriterium garantiert mindestens 50% Speicherplatzausnutzung.
- Es gibt einfache, schnelle Algorithmen zum Suchen, Einfügen und Löschen von Tupeln, die jeweils eine Komplexität von $O(\log_m(n))$ aufweisen. Insbesondere wird beim Einfügen von Elementen der »Überlauf« von vollen Knoten zur Wurzel propagiert, so dass der Baum an der Wurzel wächst; dadurch bleiben alle Blätter auf derselben Ebene.

Ein B-Baum und seine Varianten wie B^+-Bäume (→ *2.4.1.3*) kann nicht nur als Index eingesetzt werden, sondern auch direkt Tupel aufnehmen. Davon machen nicht nur Datenbanksysteme, sondern auch Dateisysteme wie NTFS (→ *3.3*) Gebrauch.

2.4.1.3 B^+-Bäume

Ein Nachteil von B-Bäumen liegt darin, dass sie nicht effizient in Sortierreihenfolge durchlaufen werden können; gerade diese Operation eines Index-Scans spielt aber z.B. bei Joins (→ *2.2.7*) eine Rolle. Daher ist die in der Praxis am häufigsten eingesetzte Variante ein B^+-Baum, der sogar effizientere Änderungsoperationen und eine Verringerung der Baumhöhe ermöglicht **[Saa05]**.

Mit B^+-Bäumen wird der Nachteil eines ineffizienten Durchlaufs behoben. Ein B^+-Baum ist ein »hohler Baum«, da die inneren Knoten keine Informationen tragen und nur zum Verweis auf die Blattseiten dienen. Jeder Zugriff benötigt nun gleichmäßig $O(\log_m(n))$ Schritte, da immer bis zu den Blättern gesucht werden muss. Die Tupel mit den Werten $w' < w$ befinden sich also im linken Teilbaum des Wertes w, die Tupel mit den Werten $w' \geq w$ rechts davon **[Saa05]**.

In der Praxis wird ein B^+-Baum häufig als Indexstruktur eingesetzt, weil er die Hauptdatei in seine Struktur integriert. Da das Ausnutzen einer sortierten Hauptdatei letztlich zu einer Verkettung der Blätter führt, ist das vollständige Durchlaufen in Sortierreihenfolge unproblematisch. Die Erhaltung der Sortierung innerhalb der Hauptdatei ist durch die Einbindung in einen B^+-Baum in $O(\log_m(n))$ möglich.

Während sich das Verhalten beim Einfügen eines neuen Elements kaum ändert, wird ein Löschvorgang gegenüber einem B-Baum weitaus effizienter durchführbar sein. Im Normalfall hat ein Löschen nur Auswirkungen auf der Blattebene. Ein Verschieben eines Elements von der Blattebene durch eine im Inneren des Baumes vorzunehmende Löschung findet nicht statt, da Werte im Inneren des B^+-Baumes stehen bleiben können, auch wenn das letzte Tupel mit diesem Wert gelöscht wird. Lediglich bei Unterlauf-Situationen muss der B^+-Baum ausgeglichen werden, um die üblichen Eigenschaften zu erhalten **[Saa05]**.

Bisher wurde nur der Fall betrachtet, dass Tupel schrittweise und jeweils einzeln in einen B- oder B^+-Baum eingefügt werden. Es kommt jedoch auch häufig vor, dass bei einer vorliegenden Menge von Tupeln eine Indexstruktur aufgebaut werden muss. Dieses Einlesen von Massendaten nennt

man auch »bulk loading«. Der weiter oben beschriebene Algorithmus wird dafür durch die immer am Wurzelknoten startende Operation nicht effizient sein. Für das Einlesen von Massendaten eignet sich eher die Strategie, die vorliegenden Indexeinträge zunächst zu sortieren und dann sortiert in den leeren Baum einzufügen. Der Vorteil ist dabei, dass der Baum immer nur nach rechts und oben wächst. Alle Einfügungen konzentrieren sich auf den rechten Pfad des Baumes (von der Wurzel bis zum am weitesten rechts stehenden Blatt). Da es wahrscheinlich ist, dass sich diese Knoten auf diesem Pfad immer in einem *Cache* befinden, kann man eine effizientere Bearbeitung erwarten. Weiterhin können die zu belegenden Blöcke im Voraus bestimmt und auf einmal bereitgestellt werden **[Saa05]**.

2.4.1.4 Statisches Hashen

Die klassischen Hashverfahren sind für die Anwendung auf Hauptspeicher-Arrays konzipiert worden. Für den Einsatz als Speicher- oder Indexstruktur für Datenbanken müssen einige Modifikationen vorgenommen werden **[Saa05]**.

Das Prinzip von Hashfunktionen in Datenbanken besteht darin, Schlüsselwerte auf sogenannte »buckets« abzubilden; darunter versteht man einen Speicherbereich, der aus einem oder mehreren Blöcken besteht. Den Buckets können Überlaufblöcke in einer geeigneten Organisation (z.B. verkettet) zugeordnet sein **[Saa05]**:

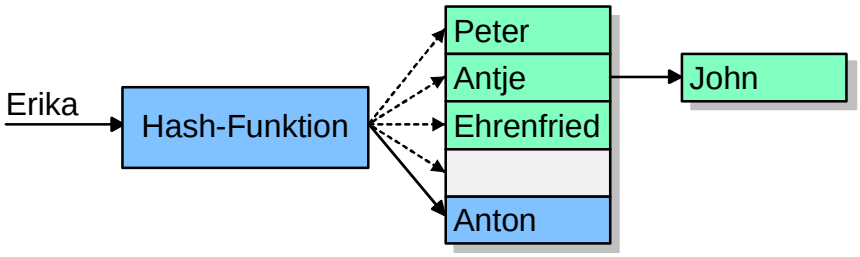


Abb. 2.4-3: Hashset **[Saa05]**

Beim Einsatz als Hauptdatei enthält ein Hashset die Tupel einer Relation, als Indexdatei nur Verweise auf die eigentlichen Tupel, die dann z.B. als Heap-Datei organisiert sind. Bei ausreichendem Speicherplatz und einer Hashfunktion, die die gegebenen Werte gleichmäßig streut, treten praktisch keine Überlaufseiten auf. In diesem Fall garantiert ein Hashset konstante Zugriffskosten **[Saa05]**.

Problematisch ist allerdings, dass die Leistungsfähigkeit von Hashverfahren stark von der Wahl der Hashfunktion abhängig ist. Eine schlechte Wahl kann fast alle Tupel demselben Block zuordnen, so dass die Datenorganisation zu einer verketteten Liste von Überlaufblöcken entartet. Ein Beispiel soll dies verdeutlichen: der Index bestehe aus 100 Buckets, indexiert werden sollen Kundendatensätze anhand einer fortlaufenden Kundennummer **[Saa05]**:

- Werden die ersten beiden Stellen der Kundennummer als Hashwert ausgewählt, werden die Tupel in wenigen Buckets quasi sequenziell abgespeichert.
- Die letzten beiden Stellen der Kundennummer hingegen verteilen die Tupel gleichmäßig auf alle Buckets.

Da die Hashfunktion die Werte möglichst gut streuen soll, ist ein sortiertes Ausgeben aller Tupel nicht effizient möglich. Trotz dieses Nachteils werden statische Hashverfahren als Ergänzung der B^+-Bäume (→ *2.4.1.3*) in kommerziellen Datenbanksystemen wie Oracle eingesetzt **[Saa05]**.

2.4.1.5 Dynamisches Hashen

Die einfache Übertragung des statischen Hashsets auf das Datenbank-Umfeld führt zu einigen Problemen, die den Einsatz trotz unschlagbarer Zeitkomplexität einschränken. Eines der Hauptprobleme ist die Anpassung der Hashfunktion an wachsende und schrumpfende Datenmengen **[Saa05]**.

Die Grundversion der Hashverfahren zeichnet sich durch mangelnde Dynamik betreffend der Speicherorganisation aus: ein zu großer Speicherbereich lässt zwar Luft für das Wachstum der Datenmengen, führt aber zu schlechterer Speicherauslastung. Ein gut ausgelasteter Speicherbereich führt bei Wachstum der Datenmengen sehr schnell zu schlechterer Performanz. In diesem Fall bleibt somit nur die Vergrößerung des Speicherbereichs und damit die Änderung der Hashfunktion übrig. Falls das geschieht, müssen bei den klassischen Verfahren alle Tupel neu »gehasht« werden. Eine Alternative zur kompletten Reorganisation bieten die dynamischen Hashverfahren **[Saa05]**.

Für dynamische Hashverfahren sind Hashfunktionen wünschenswert, die eine feste Berechnungsvorschrift haben, aber deren Bildbereich dynamisch vergrößert werden kann. Ein Ansatz, der diese Forderung erfüllt, benutzt Bitstrings als Hashwerte. Für die Adressierung im Bildbereich werden jeweils die ersten d Ziffern benutzt. Erhöht man d auf d+1, verdoppelt sich der Bildbereich **[Saa05]**.

Das naive Verdoppeln des Bildbereichs beim Auftreten von Kollisionen ist allerdings ineffizient, da auch Blöcke geteilt werden, die nicht voll sind. Ein verbesserter Ansatz ist das Spiral-Hashen: dabei werden die Werte der Hashfunktion $h(k)$ auf dem Intervall 0,0 bis 1,0 exponentiell verteilt, um die ursprüngliche Gleichverteilung derart zu verformen, dass am Anfang des Intervalls die Trefferdichte doppelt so groß ist wie am Ende. Für die Transformation wird die Funktion $exp(n)$ benutzt, die wie folgt definiert ist **[Saa05]**:

$$exp(n) = 2^n - 1$$
$$exphash(k) = exp(h(k)) = 2^{h(k)} - 1$$

Abb. 2.4-4: veränderte Hashfunktion beim Spiral-Hashen **[Saa05]**

Bei einer Analyse stellt man fest, dass die ersten 10 Prozent des abgebildeten Bereichs (Intervall von 0,0 bis 0,1) bezüglich der Breite etwa auf die Hälfte des Abbildungsbereichs der letzten 10 Prozent (Intervall von 0,9 bis 1,0) abgebildet werden. Als Ergebnis sind die Treffer bei Gleichverteilung von $h(k)$ am Beginn des Zielbereichs doppelt so dicht angeordnet wie am Ende **[Saa05]**:

n	$2^n - 1$
0,0	0,0000000
0,1	0,0717735
0,2	0,1486984
0,3	0,2311444
0,4	0,3195079
0,5	0,4142136
0,6	0,5157166
0,7	0,6245048
0,8	0,7411011
0,9	0,8660660
1,0	1,0000000

Abb. 2.4-5: exponentielle Verteilung von Werten durch exphash(k) **[Saa05]**

Zur Erweiterung des Zielbereichs wird der erste Block am vorderen Teil des Hashsets entfernt und durch zwei Blöcke am Ende ersetzt. Die gespeicherten Werte werden wie folgt auf diese beiden Blöcke verteilt: waren im Ursprungsblock Werte mit dem Präfix *x* abgelegt, so werden die Werte mit Präfix *x*0 auf die erste und die mit *x*1 auf den zweiten angehängten Block verteilt. Durch diese Vorgehensweise erklärt sich auch der Begriff »Spirale« – der Speicherbereich der inneren Seite wird verdoppelt, liegt also »weiter außen« auf einer Spirale. Ein Spiralspeicher kann natürlich auch schrumpfen, sobald eine festgelegte Auslastung unterschritten wird. Hierbei werden zwei Blöcke am Ende zu einem Block am Anfang zusammengefasst **[Saa05]**.

Das Spiral-Hashen kann nur effizient sein, wenn im Normalfall jeweils der innerste bzw. erste Block überläuft. Dies wird durch *exphash*(*k*) erreicht. In realen Anwendungen müssen allerdings auch hier Überlaufblöcke oder andere Kollisionsstrategien eingesetzt werden **[Saa05]**.

2.4.1.6 Multidimensionales Hashen

Multidimensionale Zugriffsverfahren bilden die zu organisierenden Tupel auf einen mehrdimensionalen Datenraum ab. Die Anzahl der indexierten Attribute bestimmt dabei die Anzahl der Dimensionen des Raumes **[Saa05]**.

Multidimensionales Hashen baut auf den dynamischen Hashverfahren (→ *2.4.1.5*) auf. Hashwerte sind also Bitfolgen, von denen jeweils ein Anfangsstück als aktueller Hashwert dient; es wird je ein Bitstring pro beteiligtem Attribut berechnet. Die Anfangsstücke werden nun rotiert und zu einem gemeinsamen Hashwert $h_i(k)$ zusammengesetzt **[Saa05]**:

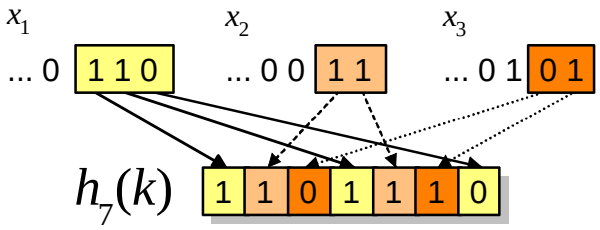


Abb. 2.4-6: Komposition der Hashfunktion $h_7(k)$ bei multidimensionalem Hashen **[Saa05]**

2.4.1.7 kd- und kdB-Bäume

Ein weiteres Verfahren zum multidimensionalen Indexieren sind kd-Bäume. Beim in → *2.4.1.6* dargestellten multidimensionalen Hashen wird der Hashwert $h_i(k)$ gebildet, indem zyklisch durch die Dimensionen des Suchraums (x_1, x_2 und x_3) rotiert wird und jeweils ein Bit angehängt wird. Auf ähnliche Weise wird ein binärer Baum zum kd-Baum: die Ebenen innerer Knoten rotieren zyklisch durch alle Dimensionen **[Saa05]**. Als Beispiel dient ein zweidimensionaler Ausschnitt des RGB-Farbraums, bei dem alle Grünwerte **00h** sind, also nur blaue, rote und violette Farbtöne vorkommen:

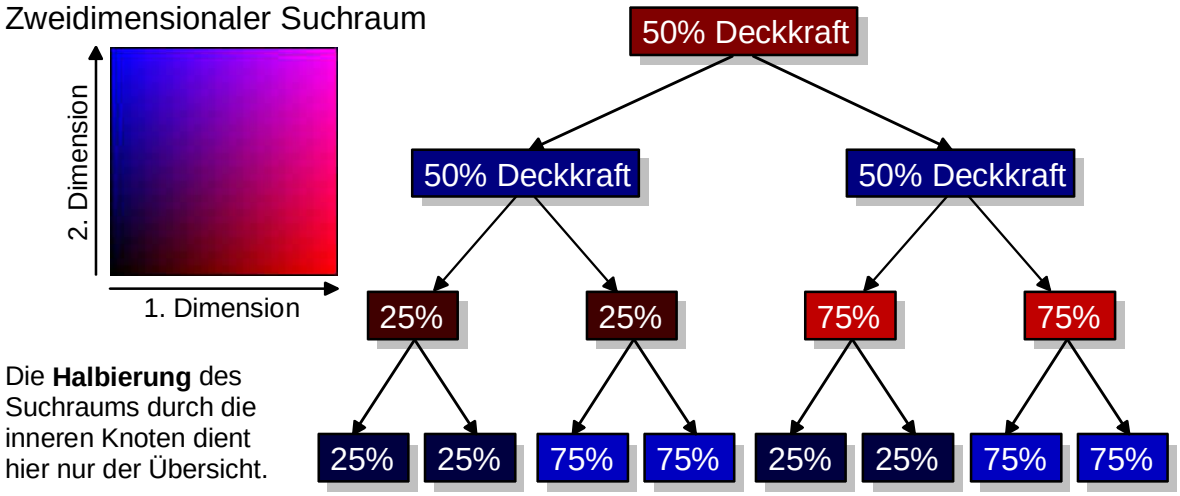


Abb. 2.4-7: Ausschnitt eines kd-Baums über 2 Dimensionen (rot und blau)

In → *Abb. 2.4-7* wird auf der ersten Ebene des kd-Baums die erste Dimension behandelt. Alle Blätter, die über den linken Ast erreichbar sind, haben weniger als 50% rote Farbdeckung, alle über den rechten Ast erreichbaren Farben haben mindestens 50% Rotanteil. Auf der zweiten Ebene wird nun unabhängig von der ersten Dimension über den Farbwert der 2. Dimension (blau) entschieden. Der

jeweils links Ast führt zu Farben, die weniger als 50% blauer Farbabdeckung aufweisen, die jeweils rechten Zweige zu Farben mit mindestens 50% Blauanteil – unabhängig vom jeweiligen Rotwert.

Analog zu B-Bäumen werden kd-Bäume durch kdB-Bäume verbessert, indem die Blockstruktur von Datenträgern ausgenutzt wird (ein Knoten also mehr als zwei Nachfolger hat). Außerdem wird durch dieselben Algorithmen wie bei B-Bäumen sichergestellt, dass ein kdB-Baum immer balanciert ist.

kd- und kdB-Bäume haben den Nachteil, dass zum Auffinden eines Tupels alle indexierten Attribute bekannt sein sollten. Ist dies für ein Attribut (etwa den Blauanteil in → *Abb. 2.4-7*) nicht der Fall, so müssen auf den Ebenen, die sich darauf beziehen, **alle** Nachfolgeknoten bearbeitet werden. Solche »partial match«-Anfragen werden also nicht effizient unterstützt **[Saa05]**.

2.4.1.8 R- und R⁺-Bäume

Eine fortgeschrittene Technik zur multidimensionalen Indexierung sind R- bzw. R⁺-Bäume. R-Bäume wurden 1984 von Guttman eingeführt **[Gut84]** und unterteilen den Suchraum sukzessiv in einzelne Regionen; daher auch der Name der Datenstruktur. Folgendes Beispiel zeigt einen zweidimensionalen Suchraum und den zugehörigen R-Baum:

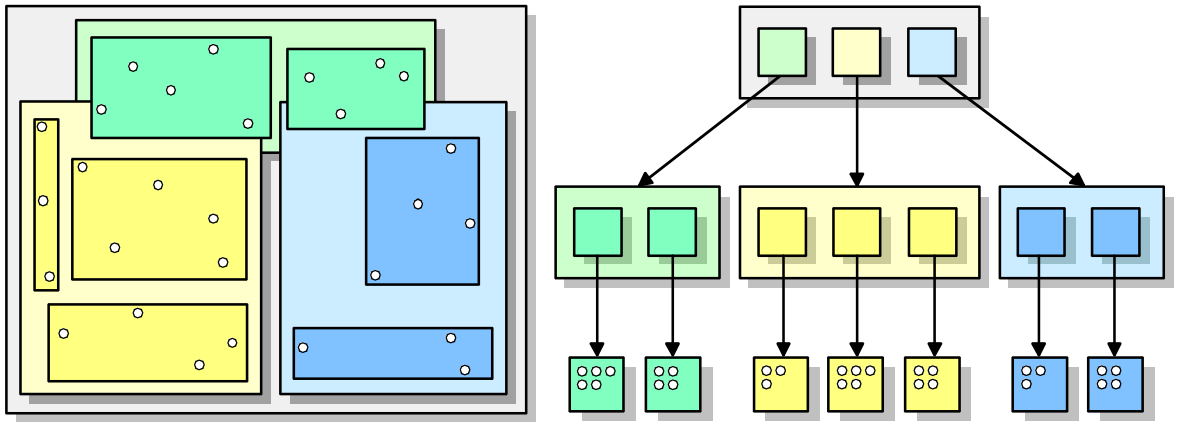


Abb. 2.4-8: R-Baum **[IDF00]**

Die Wurzel entspricht einem Rechteck, das alle Objekte umfasst. Die Aufteilung in Regionen erfolgt im obigen Beispiel in nicht-disjunkte Rechtecke; jedes Objekt wird schließlich eindeutig einem Blatt des R-Baums zugeordnet. Sie stellen die letzte Stufe der Regionenaufteilung dar **[Saa05]**.

In der Regel wird gefordert, dass die umschließenden Regionen minimal sind, es sich also im Beispiel um minimal umschließende Rechtecke handeln muss. Zwar funktioniert ein R-Baum auch ohne diese Eigenschaft, allerdings arbeiten einige Algorithmen wie die Suche nach dem nächsten Nachbarn nur unter diese Annahme korrekt **[Saa05]**.

Ein R-Baum ist analog zum B-Baum (→ *2.4.1.2*) immer ausbalanciert; die Algorithmen zum Einfügen und Löschen basieren ebenfalls auf dem Teilen und Verschmelzen von Knoten und der Aufwärtspropagation eines Überlaufs zur Wurzel. Die Suche im R-Baum erfolgt durch Schnittbildung eines Suchrechtecks mit den Rechtecken der Regionen: ist der Schnitt nicht leer, muss der betroffene Teilbaum weiter verfolgt werden. Daraus ergibt sich, dass ein R-Baum die Daten umso besser indexiert, je besser die Aufteilung beim Teilen von Knoten optimiert wird **[Saa05]**.

R^+-Bäume wurden von Sellis et al. in **[Sel87]** vorgestellt. Im Gegensatz zu R-Bäumen ist in R^+-Bäumen die Aufteilung in Regionen disjunkt. Jedem gespeicherten Punkt des Suchraums ist somit eindeutig maximal ein Blatt zugeordnet. In **jeder** Baumebene ist einem Punkt ebenfalls maximal ein Rechteck zugeordnet, so dass es einen eindeutigen Pfad von der Wurzel des R^+-Baums zum speichernden Blatt gibt.

Weitere multidimensionale Indexstrukturen existieren, führen allerdings über den Umfang einer Einführung in die Datenbanktechnik hinaus. Sie bauen oft auf dem kd-, kdB- oder R-Baum auf, wie zum Beispiel R*- **[Bec90]** oder X-Bäume **[Ber96]**.

2.4.1.9 Partitionierung

Eine weitere Möglichkeit, Zugriffe auf große Relationen zu beschleunigen ist die *Partitionierung*. Dabei wird eine Relation in disjunkte Partitionen aufgeteilt, die auf verschiedenen Festplatten abgespeichert werden. Daraus ergibt sich eine Reihe von Vorteilen **[Saa05]**:

- Wird die Relation anhand eines Attributs aufgeteilt, so können Anfragen mit Restriktion bezüglich dieses Attributs auf die betroffene Partition beschränkt werden. Dadurch muss nicht die gesamte Relation eingelesen werden.
- Werden die Partitionen auf verschiedenen Festplatten abgelegt, können Leseoperationen auf der Relation parallel auf allen Partitionen durchgeführt werden. So wird für das vollständige Lesen einer Relation, die auf n Festplatten verteilt ist, nur $1/n$ der Zeit benötigt, die für das Lesen der gesamten Relation von einer Festplatte notwendig wäre. Schließlich kann auch von der parallelen Bearbeitung weiterer Operationen profitiert werden.

Es gibt nun verschiedene Möglichkeiten, die Tupel einer Relation auf mehrere Festplatten zu verteilen, etwa anhand des Wertebereichs eines Attributs, mit einer Hashfunktion oder durch zyklisches Verteilen der Tupel auf die einzelnen Partitionen **[Saa05]**:

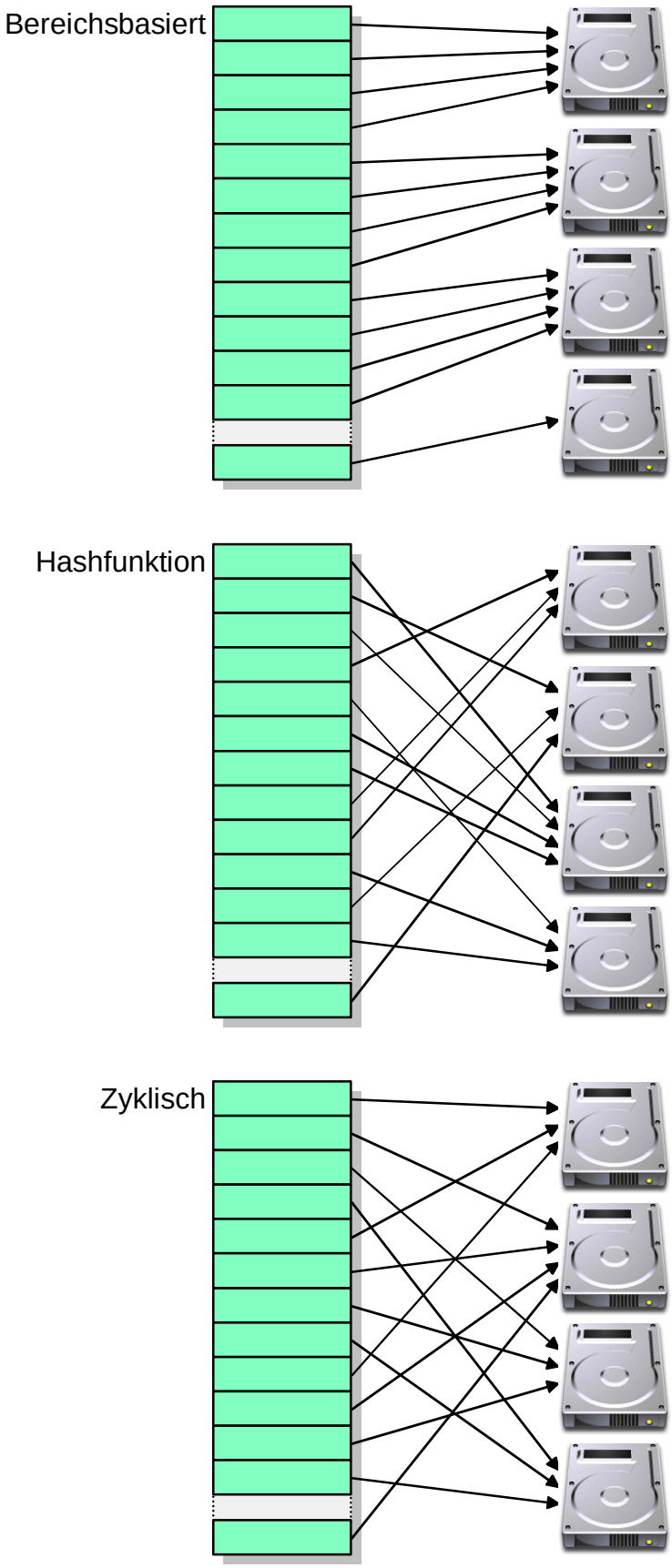


Abb. 2.4-9: Partitionierungsstrategien **[Saa05]**

Ein Vorteil der hashbasierten Partitionierung ist die anwendungsunabhängige gleichmäßige Verteilung der Tupel auf die beteiligten Festplatten. Die Partitionierung anhand eines Wertebereichs kann jedoch bei einer Anfrage gezielt berücksichtigt werden, so dass Operationen auf einzelne Partitionen beschränkt bleiben. Dabei besteht jedoch die Gefahr, dass eine ungleichmäßige Aufteilung die oben genannten Vorteile zunichte macht **[Saa05]**.

2.4.2 Umsetzung in SQL

Kommerzielle Datenbanksysteme implementieren verschiedene Organisationsformen und Zugriffspfade für Relationen, die teilweise weit über die oben erwähnten Datenstrukturen hinausgehen. Bei SQL-basieren Systemen ist es möglich, direkt bei der Erzeugung einer Tabelle deren Organisationsform festzulegen und Indexe zu erzeugen. Folgendes Beispiel zum Anlegen einer partitionierten Tabelle (→ *2.4.1.7*) würde von Oracle-Servern ausgeführt werden können **[Saa05]**:

```
CREATE TABLE Kunden
(
  Nummer INT NOT NULL,
  Vorname VARCHAR(50) NOT NULL,
  Name VARCHAR(50) NOT NULL,
  Strasse VARCHAR(50),
  Hausnummer INT,
  PLZ INT NOT NULL,
  Ort VARCHAR(50),
  PRIMARY KEY (Nummer)
)
PARTITION BY RANGE (PLZ)
(
  PARTITION KundenNord VALUES LESS THAN (50000) TABLESPACE TS_NORD
  PARTITION KundenSued VALUES LESS THAN (99999) TABLESPACE TS_SUED
)
```

Auch Indexe können von einer Client-Applikation mit SQL angelegt werden **[Mat03]**:

```
CREATE INDEX IdxKunden1 on Kunden (Ort)
```

Oracle unterstützt auch für Indexe Partitionierungen, indem ein `PARTITION`-Statement an den SQL-Befehl `CREATE INDEX` angehängt wird **[Saa05]**:

```
CREATE INDEX IdxKunden2 on Kunden (PLZ)
PARTITION BY RANGE (PLZ)
(
  PARTITION KundenNord VALUES LESS THAN (50000) TABLESPACE TS_NORD
  PARTITION KundenSued VALUES LESS THAN (99999) TABLESPACE TS_SUED
)
```

Die konkrete Wahl von Datenstrukturen obliegt dem jeweiligen Datenbanksystem und kann je nach Produkt unterschiedlich sein, wenn SQL-Statements wie `USING HASH`, `USING BTREE` oder `PARTITION BY` nicht benutzt werden **[Saa05]**.

3 Physikalische Dateisysteme

Analog zu Datenbanken (→ *2*) bestehen auch Dateisysteme aus einer logischen Ebene und einer physikalischen Ebene, die die logische Struktur speichert und verwaltet. Während bei Datenbankensystemen die physikalischen Datenstrukturen in der Regel nicht binärkompatibel sind, so dass eine Datenbank in der Praxis nur mit einem bestimmten DBMS eingesetzt werden kann, sind die logischen und physikalischen Ebenen von Dateisystemen standardisiert, so dass unterschiedliche Betriebssysteme und Geräte-Firmwares (etwa in einer Digitalkamera oder einem MP3-Player) problemlos auf dasselbe Dateisystem zugreifen können:

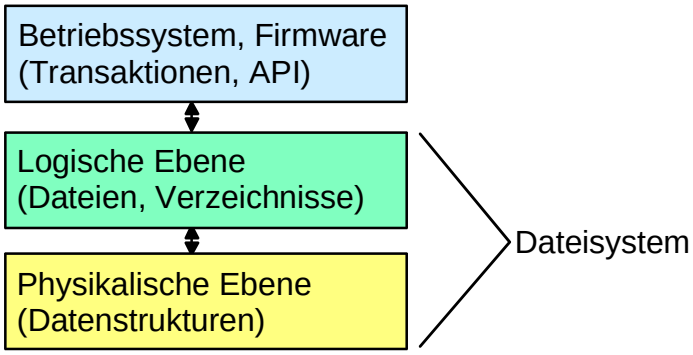


Abb. 3-1: Begriffsdefinition

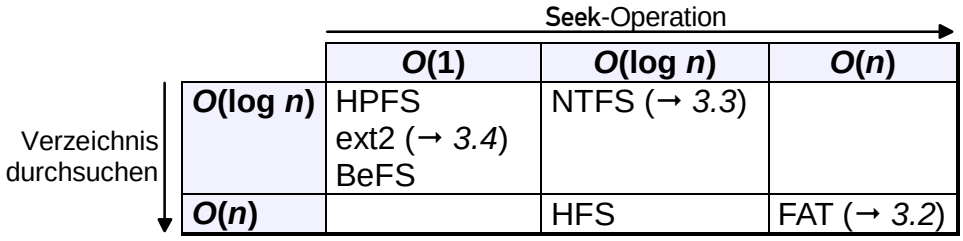
Dateisysteme lassen sich in bestimmte Kategorien einteilen, wobei nur die sog. »Allzweck-Dateisysteme« von weiterem Interesse sind. Sogenannte »Netzwerk-Dateisysteme« wie Samba oder NFS **[Sun89]** sind eigentlich keine Dateisysteme, sondern Netzwerk-Protokolle zum Zugriff auf Dateisysteme, die sich auf einem entfernten Computer befinden.

Auf Hardwareebene arbeiten Speichermedien blockorientiert. Das bedeutet, dass der Gesamtspeicher eines Datenträgers in einzelne, gleich große Blöcke unterteilt ist, die die kleinste Einheit für einen Datentransfer darstellen. Ein häufig benutztes Synonym für Block ist *Sektor*.

Die Hauptaufgabe von physikalischen Dateisystemen besteht darin, mehrere Blöcke des Datenträgers zu einem linearen Speicherbereich zusammenzusetzen, der auf logischer Ebene als Dateikörper für Nutzdaten zur Verfügung steht. Zusätzlich werden auf physikalischer Ebene weitere Informationen abgespeichert, um auf logischer Ebene Verzeichnisse und Dateien zu verwalten. Für den weiteren Verlauf sind Dateisysteme wie ISO9660, die für einmal-beschreibbare Medien entwickelt wurden, ohne Bedeutung; sie sind wesentlich einfacher aufgebaut, da Dateien in der Regel linear gespeichert werden und keine Fragmentierung auftreten kann.

Dateisysteme unterscheiden sich in den verwendeten Datenstrukturen und Algorithmen. Für die besonders wichtigen Operationen **SEEK**, also das Bewegen des Dateizeigers zu einer bestimmten Position innerhalb des Dateikörpers, und das Suchen eines bestimmten Verzeichniseintrags anhand des

Dateinamens ergeben sich dadurch unterschiedliche Laufzeiten. Allzweck-Dateisysteme lassen sich anhand dieser Laufzeiten in einer zweidimensionalen Tabelle anordnen:



	***O*(1)**	***O*(log *n*)**	***O*(*n*)**
***O*(log *n*)**	HPFS ext2 (→ *3.4*) BeFS	NTFS (→ *3.3*)	
***O*(*n*)**		HFS	FAT (→ *3.2*)

Abb. 3-2: Übersicht über physikalische Allzweck-Dateisysteme

In diesem Kapitel werden die physikalischen Dateisysteme FAT (→ *3.2*) und ext2 (→ *3.4*) detailliert vorgestellt, denn sie haben eine große Verbreitung und stellen die Extrempunkte bezüglich der Tabelle in → *Abb. 3-2* dar. Vor allem dem FAT-Dateisystem fällt dabei eine große Bedeutung zu, da es von fast allen Betriebssystemen benutzt werden kann und durch die Verwendung auf den meisten Flash-Speichermedien wie Memorysticks (→ *5.3*) oder Compact Flash im Zuge der digitalen Fotografie vermutlich eine größere Verbreitung erlangt hat als in der DOS-Ära. Zusätzlich wird noch kurz auf NTFS (→ *3.3*) eingegangen, da NTFS seit Microsoft Windows XP der Standard für dieses weit verbreitete Betriebssystem ist.

3.1 Partitionierung

Heutige Datenträger, insbesondere Festplatten, sind sehr groß, so dass auf demselben Rechner oft mehrere Betriebssysteme mit jeweils eigenen Dateisystemen parallel installiert werden können; auch beim Einsatz eines einzigen Betriebssystems ist das Aufteilen der Festplatte sinnvoll, um z.B. Daten und Programme zu trennen und in unterschiedlichen Dateisystemen unterzubringen. Daher existiert für PCs ein Standard zum *Partitionieren*, also Unterteilen eines Datenträgers in mehrere Bereiche; Disketten werden traditionell nicht partitioniert, obwohl dies technisch möglich wäre. Um die Datenstruktur für Partitionsgrenzen verstehen zu können, ist zunächst auf die Adressierung üblicher Festplatten einzugehen.

3.1.1 Adressierung

Um einen Sektor lesen oder beschreiben zu können, muss dem Laufwerk die Position mitgeteilt werden. Hier existieren im PC-Bereich zwei Modelle: das ältere *CHS* und das modernere *LBA*.

3.1.1.1 CHS

CHS basiert auf der Laufwerksgeometrie von Festplatten und steht für »Cylinder Head Sector«. Bei Festplatten sind mehrere rotierende Scheiben übereinander montiert und haben oben und unten einen Abtastkopf; pro Scheibe hat eine Festplatte also in der Regel 2 Leseköpfe. Die einzelnen Sektoren einer Scheibenoberfläche sind bei diesem Modell auf konzentrischen Kreisspuren angeordnet, die zusammen einen Zylinder bilden. Innerhalb eines Zylinders sind die Sektoren nummeriert, so dass durch die CHS-Angabe ein Sektor eindeutig adressiert werden kann. Die Spezifikation wurde einmal erweitert und gestattet folgende Werte:

	Bits	Wertebereich
Zylinder	10 Bit	0-1023
Köpfe	8 Bit	0-15 oder 0-254
Sektoren pro Spur	6 Bit	1-63

Abb. 3.1-1: Wertebereich von CHS

Die Gesamtgröße eines Laufwerks ergibt sich aus *Sektorgröße* ∗ *Sektoren* ∗ *Köpfe* ∗ *Zylinder*. Mit dem CHS-Modell lassen sich somit bei der üblichen Sektorgröße von 512 Byte maximal 528 MB (1024 Zylinder, 16 Köpfe) oder 8,4 GB (1024 Zylinder, 255 Köpfe) adressieren. Moderne Festplatten arbeiten aber intern nicht mehr mit der CHS-Geometrie, da sich auf den Spuren am Plattenrand mehr Sektoren unterbringen lassen als auf inneren Spuren, was von CHS nicht abgebildet werden kann.

3.1.1.2 LBA

LBA steht für »Logical Block Address«. Dabei werden einfach alle Sektoren beginnend mit 0 durchnummeriert. Diese Nummern werden an das Laufwerk übertragen und bei Festplatten intern auf die tatsächliche Geometrie umgerechnet. Für die LBA-Adresse stehen auf Hardwareseite 24, auf Softwareseite 32 Bit zur Verfügung; das entspricht 120 GB bzw. 2 TB. Eine Erweiterung des hardwareseitigen Adressraums auf 48 Bit ist definiert und wird großflächig eingestzt.

3.1.1.3 Zusammenfassung

Das CHS-System basiert auf der Laufwerksgeometrie, da ältere Controller diese Daten direkt an die Festplatte übermitteln, analog zu Diskettenlaufwerken. Wie erwähnt lassen sich am Rand einer Plattenscheibe mehr Sektoren unterbringen als im Inneren, so dass die Abhängigkeit von der Geometrie die Entwicklung von Festplatten mit größerer Kapazität behindert hat. Um abwärtskompatibel zu bleiben, benutzen moderne Festplatten im Zusammenspiel mit alten Controllern eine fiktive Geometrie mit 16 oder 255 Köpfen, um zu älteren Controllern kompatibel zu sein.

Bei Flash-Speichern existiert keine Laufwerksgeometrie mit Zylindern, Köpfen und Sektoren. Daher wurde das LBA-System eingeführt, das von Festplatten auf die tatsächliche Geometrie abgebildet und von Flash-Speichern sogar nativ unterstützt wird.

3.1.2 Master Boot Record

Die Datenstruktur, die die Partitionen festlegt, wird *Partitionssektor* oder *Master Boot Record* genannt. Sie kann mit komfortablen Programmen erstellt werden:

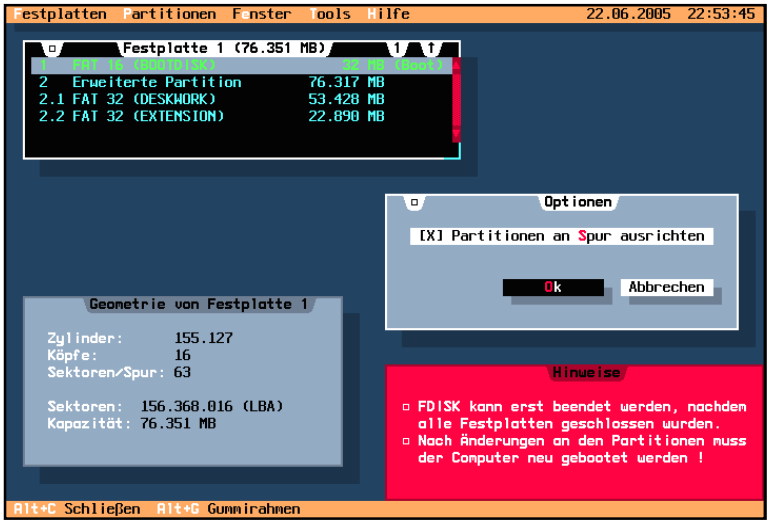


*Abb. 3.1-2: DW-DOS **FDISK***

Der Partitionssektor befindet sich an der CHS-Adresse Zylinder 0, Kopf 0 und Sektor 1, was bei jeder Festplatte mit LBA 0 zusammenfällt. Er hat eine Größe von 512 Byte und hat folgenden Aufbau:

```
type
  SecPos=record
    Kopf: Byte;
    SecZyl: Word;
end;
  PartEntry=record
    Status: Byte;
    StartSec: SecPos;
    PartTyp: Byte;
    EndSec: SecPos;
    SecOfs,SecAnz: LongInt;
end;
  PartSec=record
    BootCode: array[0..$1B9] of Byte;
    SerialNo: LongInt;
    Flags: Word;
    PartTable: array[1..4] of PartEntry;
    IDCode: Word;
end;
```

Die Datenstruktur **SecPos** kann eine Sektorposition im CHS-Format speichern, indem die 10 Bits für die Zylinderangabe und die 6 Bit breite Sektornummer zusammen in einem Word gespeichert werden. Der Partitionssektor enthält einen maximal 446 Byte großen Programmcode zum Booten der Festplatte, dann 4 Einträge für Partitionen und einen **IDCode**, der immer den Wert **AA55h** haben muss und den Partitionssektor als gültig ausweist.

Jeder Eintrag in **PartTable** (Partitionstabelle) beschreibt die Grenzen der jeweiligen *Partition* einmal im CHS-Format durch die Angabe von Anfangs- und Endsektor, und einmal im LBA-Format durch Angabe des Startsektors (**SecOfs**) und der Anzahl der Sektoren (**SecAnz**). Dadurch können partitionierte Festplatten mit beiden Systemen adressiert werden. Ist Bit 7 von **Status** gesetzt, wird die Partition als bootfähig markiert; der Programmcode im Master Boot Record hat die Aufgabe, das Betriebssystem von der ersten bootfähigen Partition zu starten oder eine Fehlermeldung auszugeben. Die letzte Variable, **PartTyp**, beschreibt das Dateisystem, das auf der Partition installiert ist:

	Dateisystem
00h	Frei
01h, 11h	FAT12
02h, 12h	XENIX root
03h, 13h	XENIX user
04h, 06h, 0Eh, 14h, 16h, 1Eh	FAT16
05h, 0Fh, 15h, 1Fh	Erweiterte Partition

	Dateisystem
07h, 17h, 42h	NTFS
08h, 18h	AIX
09h, 19h	AIX boot
0Bh, 0Ch, 1Bh, 1Ch	FAT32
82h	Linux swap
83h	Linux native
EEh	GPT, → *3.1.5*
EFh	EFI, → *3.1.5*

Bit 4 (**10h**) gesetzt: Partition ist versteckt

Abb. 3.1-3: von IBM, Microsoft und für Linux definierte Partitionstypen **[Mic07]**

Die Codes in der obigen Abbildung erscheinen zunächst unübersichtlich, folgen aber einer bestimmten Systematik. Die Tabelle ist jedoch bei weitem nicht vollständig, zumal einige Nummern von mehreren, teilweise antiken Dateisystemen benutzt werden; → *Abb. 3.1-3* enthält nur die Codes, die von IBM, Microsoft und für Linux benutzt werden. Der Code **83h** ist für Datenpartitionen von Linux reserviert; das Dateisystem in dieser Partitionen wird dadurch allerdings nicht bestimmt. Es kann sich also zum Beispiel um ext2 (→ *3.4*), ext3, ReiserFS, XFS, JFS oder andere handeln.

Ist bei einem von IBM oder Microsoft benutzten Code Bit 4 (**10h**) gesetzt, so ist die Partition unsichtbar. Diese Funktion wird von einigen Bootmanagern benutzt, um Partitionen vor jedem Zugriff durch das gerade aktive Betriebssystem zu verstecken. Daher existiert für FAT12 sowohl der Code **01h** (normal) als auch **11h** (unsichtbar). Bei FAT16, FAT32 und erweiterten Partitionen (→ *3.1.3*) gibt es jedoch auch für die sichtbaren Partitionen zwei Codes; sie zeigen an, ob die Partition innerhalb des von CHS adressierbaren Bereichs liegt (**04h**, **05h**, **06h**, **0Bh**) oder nicht (**0Ch**, **0Eh**, **0Fh**).

Zusammenfassend ist der Partitionssektor also eine Datenstruktur, die im ersten Sektor eines Datenträgers gespeichert ist und alle angelegten Partitionen mit ihren Dateisystemen referenziert:

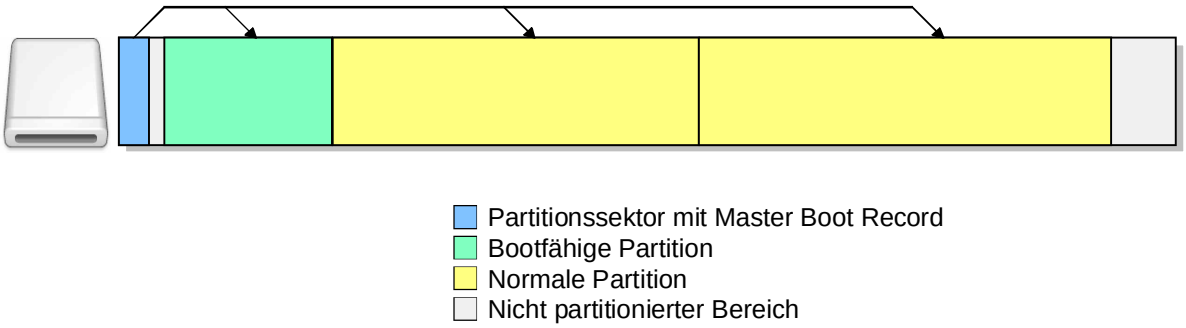


Abb. 3.1-4: Partitionierung einer Festplatte mit 3 primären Partitionen

3.1.2.1 Master Boot Record unter DW-DOS und Windows NT/2000/XP

Die in der Überschrift genannten Betriebssysteme erweitern die Definition des Master Boot Records, indem ab Offset **1B8h** eine 32 Bit lange Seriennummer untergebracht wird. Ab Offset **0DAh** befindet sich eine Datenstruktur, die den Zeitpunkt des letzten Partitionierens festhält:

```
type
  Timestamp=record
    ID: Word;
    PhysNo,Sec,Min,Hour: Byte;
end;
```

PhysNo enthält dabei die physikalische Nummer der Festplatte zum Zeitpunkt des Partitionierens, beginnend mit **80h**. **ID** muss immer den Wert 0 annehmen.

Rot hervorgehoben: **Timestamp** (ab **0DAh**), **SerialNo** (ab **1B8h**) und **IDCode** (ab **1FEh**)

Abb. 3.1-5: Master Boot Record von DW-DOS

Abb. 3.1-6: Interpretation von →Abb. 3.1-5

3.1.3 Erweiterte Partitionen

Mit dem bisher vorgestellten Prinzip lässt sich eine Festplatte in maximal vier Partitionen aufteilen, da die Struktur der Partitionstabelle nur Platz für vier Einträge bietet. Die Partitionscodes **05h** und **0Fh** (→ *Abb. 3.1-3*) sind jedoch für erweiterte Partitionen reserviert. Diese Partitionen enthalten kein Dateisystem, sondern weitere Partitionen, denen dann ein neuer Partitionssektor vorangestellt ist. Es handelt sich also um eine Art »Festplatte in der Partition«. Mit MS-DOS wurde festgelegt, dass jede erweiterte Partition nur eine normale Partition und eine neue erweiterte Partition enthalten kann, so dass eine verkettete Liste entsteht:

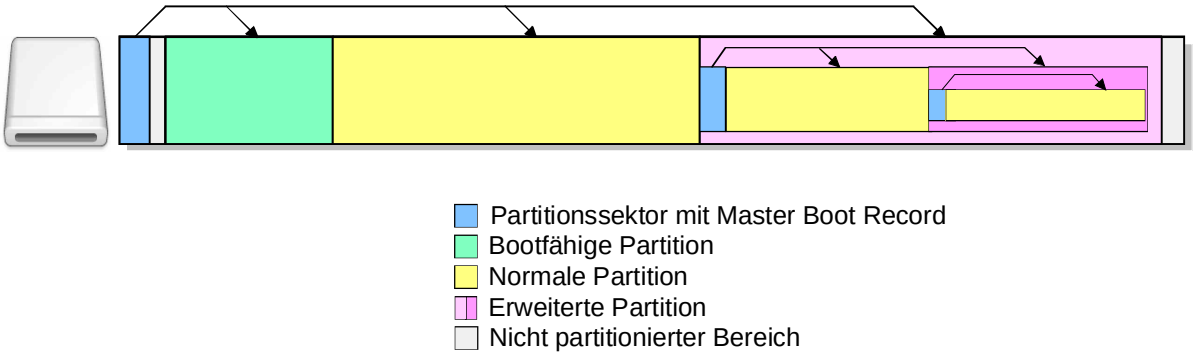


Abb. 3.1-7: Partitionierung einer Festplatte mit erweiterter Partition

Die Positionsangaben in einem Partitionssektor, der sich in einer erweiterten Partition befindet, beziehen sich nicht auf den Anfang des Datenträgers, sondern auf den Anfang der erweiterten Partition. Beginnt diese beispielsweise im LBA 50000 und die erste Datenpartition im LBA 50001, so enthält der Partitionssektor der erweiterten Partition im Feld **SecOfs** den Wert 1 und nicht 50001. Partitionen innerhalb einer erweiterten Partition sind normalerweise nicht bootfähig, da der Platz im Master Boot Record nicht ausreichend ist, um Programmcode zum Scannen der erweiterten Partitionen unterzubringen.

3.1.4 Dynamische Laufwerke

Im Server-Bereich ist es üblich, mehrere Festplatten zusammenzuschalten und als ein großes Laufwerk zu verwenden; diese Technik wird *RAID* genannt (Redundant Array of Independent Disks). Abhängig von der Zielsetzung gibt es u.A. verschiedene Modi; die wichtigsten sind **[Tan02]**:

- »Spanned«: das Dateisystem wird über mehrere Festplatten aufgespannt, so dass der erste Teil auf der ersten Festplatte gespeichert wird, der zweite Abschnitt auf der zweiten Festplatte usw.
- »Striped«: auch RAID-Level 0 genannt; die Blöcke des Dateisystems werden zyklisch auf die Festplatten verteilt, so dass sich die Zugriffsgeschwindigkeit erhöht.
- »Mirrored«: beide Festplatten enthalten dieselben Daten (RAID-Level 1).

Als Server-Betriebssystem sollte Windows NT RAID-Fähigkeiten aufweisen, musste aber gleichzeitig aus Kompatibilitätsgründen das Partitionsschema von DOS übernehmen. Dadurch ergeben sich aber Nachteile:

- Änderungen an den Partitionen erfordern einen Neustart des Systems.
- Die Informationen über die RAID-Aufteilung wird in der *Registry* und nicht in der Partitionstabelle gespeichert, was die Migration von Festplatten erschwert.
- Aufgrund der 32-Bit-Zahlenwerte für LBA-Adressen im Master Boot Record (→ *3.1.2*) kann eine Festplatte nur bis maximal 2 TB genutzt werden, selbst wenn die Hardware 48 Bit unterstützt.

Mit Windows 2000 wurde dann der Logical Disc Manager (LDM) eingeführt; mit ihm können die Grenzen von Windows NT überwunden werden. Dazu wird die RAID-Information direkt auf den beteiligten Festplatten gespeichert; außerdem kann die Partitionierung nun ohne Neustart verändert werden. Festplatten, die nach diesem neuen Schema partitioniert sind, werden *dynamisch* genannt; Datenträger mit traditioneller Struktur heißen *basic*.

Dynamisch verwaltete Datenträger sind gegenwärtig nur im Serverbetrieb verbreitet, während Workstations und Flash-Speicher fast ausnahmslos vom Basic-Typ sind. Trotzdem wird der LDM im Folgenden kurz beschrieben, da er vor allem die Begrenzung des Master Boot Records hinsichtlich der maximalen Festplattengröße von 2 TB aufhebt. Dasselbe gilt für die zukunftsweisende Partitionierung durch GUIDs (→ *3.1.5*).

Der Logical Disk Manager verwaltet in den letzten 1024 KB jeder eingebauten Festplatte eine Datenbank, die alle Informationen über die verwalteten Festplatten enthält; dadurch ist die Datenbank robuster gegenüber physikalischen Fehlern als die Registry, die nur auf einer einzigen Festplatte gespeichert wird. Die klassische Partitionstabelle ist nur noch als Platzhalter vorhanden; der Sektor danach trägt die Bezeichnung PRIVHEAD (private header) und verweist auf die LDM-Datenbank **[LDM05]**:

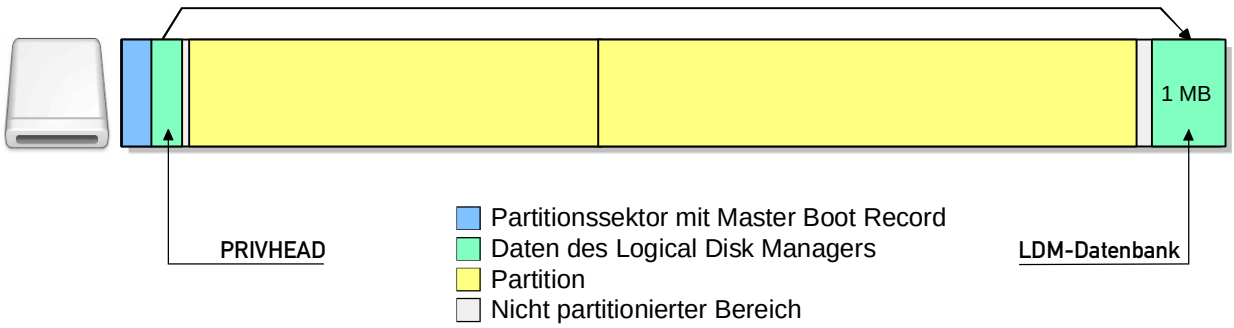


Abb. 3.1-8: Aufbau der Festplatte mit Logical Disk Manager **[LDM05]**

Der letzte Sektor der LDM-Datenbank (und damit der letzte Sektor des gesamten Datenträgers) enthält eine Backup-Kopie von **PRIVHEAD**. Es gilt nun nicht mehr, dass jede Partition ein Dateisystem enthält. Vielmehr wird eine vierstufige Zuordnungshierarchie benutzt **[LDM05]**:

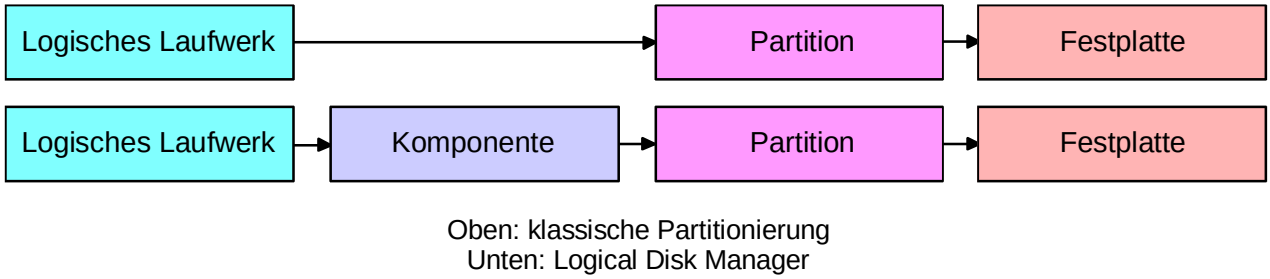
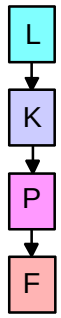
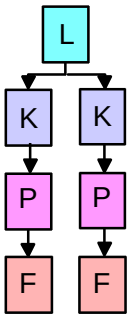


Abb. 3.1-9: Hierarchie zwischen logischem Laufwerk und physikalischem Datenträger **[LDM05]**

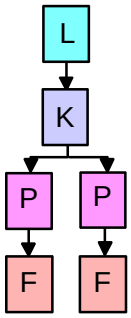
Bei der klassischen Partitionierung ist einem logischen Laufwerk immer genau eine Partition auf einer physikalischen Festplatte zugeordnet. Bei Verwendung des LDM bestehen logische Laufwerke aus einer oder mehreren Komponenten; das rechts dargestellte Layout entspricht also weitgehend dem klassischen Ansatz und ist die einfachste denkbare dynamische Konfiguration **[LDM05]**.



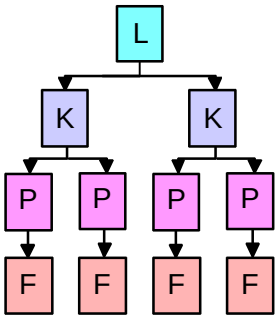
Dieser Entwurf verdoppelt das komplette Dateisystem auf zwei Komponenten, die jeweils eine eigene Partition auf unterschiedlichen Festplatten in Anspruch nehmen. Dadurch wird ein vollständig gespiegeltes Laufwerk realisiert (RAID-Level 1) **[LDM05]**.



Bei der rechts gezeigten Anordnung besteht das logische Laufwerk aus einer einzigen Komponente, allerdings setzt sich diese aus zwei Partitionen auf zwei unterschiedlichen Festplatten zusammen. Abhängig davon, ob die Komponente die Blöcke abwechselnd auf die Partitionen verteilt oder die zweite Partition an die erste anhängt, handelt es sich um ein »striped«- oder »spanned«-Laufwerk. Wenn die Komponente die Daten mit Parity auf weiteren Festplatten speichert, wird RAID-Level 5 realisiert **[LDM05]**.



Bei dieser Mischform basiert das logische Dateisystem auf zwei identischen Komponenten, die ihrerseits »striped« oder »spanned« sind. Durch diese Anordnung kann also ein »mirrored striped« oder »mirrored spanned« realisiert werden (RAID-Level 0+1) **[LDM05]**. »Mirrored RAID 5« wäre zwar auch möglich, aber völlig sinnlos, da das RAID-Level 5 bereits redundant ist und daher keiner Spiegelung bedarf.



3.1.5 GUID Partition Table (GPT)

GUID ist die Abkürzung für »Globally Unique Identifier« und bezeichnet eine 128 Bit lange, global eindeutige Bitfolge, die oft aus der (bereits weltweit eindeutigen) MAC-Adresse einer Netzwerkkarte und einem Zeitstempel gebildet wird **[RFC4122]**. Durch den großen Zahlenraum von 2^{128} ist die Wahrscheinlichkeit, dass zwei GUIDs identisch sind, $2^{-128} = 2{,}9387 * 10^{-39}$. GUIDs spielen bei einem neuartigen Partitionierungsschema eine Rolle, das GUID Partition Table (GPT) genannt wird.

Der Master Boot Record (→ *3.1.2*) ist gegenwärtig ein sehr weit verbreitetes Partitionierungsschema und eng an das Vorhandensein eines BIOS angelehnt. Die Größe einer Festplatte ist durch die Angabe von 32 Bit großen Blocknummern (bei 512 Byte Sektorgröße) auf 2 Terabyte begrenzt. Dies erscheint heute nicht mehr zeitgemäß, das gleiche gilt für die Begrenzung auf 4 Partitionen und den damit verbundenen Workaround der erweiterten Partition (→ *3.1.3*).

Im Rahmen des »Extensible Firmware Interface« **[Int07]**, das in Zukunft voraussichtlich das veraltete BIOS ersetzen wird, wird ein neuartiges Partitionierungsschema eingeführt. Ähnlich wie dynamische Partitionen werden die Unzulänglichkeiten des Master Boot Record behoben, wobei die Verbreitung im Rahmen eines plattformübergreifenden Standards erfolgt, so dass es sich nicht um eine proprietäre Technologie handelt.

Am Anfang einer Festplatte, die eine GPT enthält, befindet sich zunächst wie gewohnt ein Master Boot Record. Dieser »Protective MBR« markiert für ältere Betriebssysteme die gesamte Festplatte als belegt, und zwar mit den Partitionstypen **EEh** und **EFh** (→ *Abb. 3.1-3*). An den Master Boot Record schließen sich die neuen Datenstrukturen der GUID-Partitionierung an, immer beginnend mit LBA 1. Der erste Block (LBA 1) enthält eine Headerstruktur, daran schließen sich in den Sektoren 2

bis 33 Informationen für bis zu 128 Partitionen an. Am Ende der Festplatte wird die gesamte GPT-Struktur zur Sicherung wiederholt:

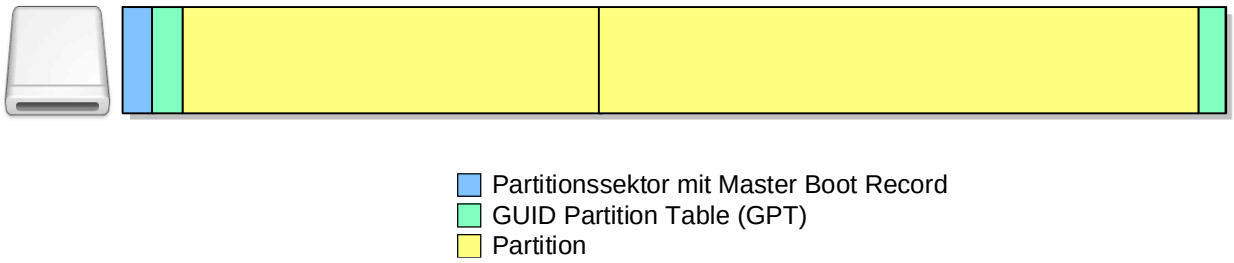


Abb. 3.1-10: Aufbau der Festplatte mit GUID Partition Table **[Mic07]**

Jede GPT-Partition wird durch eine GUID für den Typ des Dateisystems und eine GUID als eindeutige Nummer der Partition identifiziert. Da Anfangs- und Endsektor einer Partition als 64 Bit angegeben wird, kann ein Datenträger bei 512 Byte Sektorgröße $2^{64} * 512 = 8192$ Exabyte umfassen.

3.1.6 ZFS

Die herkömmliche Partitionierung mit Master Boot Record (→ *3.1.2*) oder GPT (→ *3.1.5*) gibt auf einem Datenträger einen Rahmen vor, innerhalb dessen Dateisysteme angelegt werden. Das Dateisystem ist dabei vom Partitionsschema abhängig und kann selbst keinen Einfluss darauf nehmen. Vor allem wird die Ausdehnung eines Dateisystems dadurch auf eine einzige Festplatte beschränkt. Der oben vorgestellte Logical Disc Manager (→ *3.1.4*) führt die »Komponente« als Zwischenschicht ein, die mehrere Partitionen auf unterschiedlichen Festplatten zu einer logischen Einheit zusammenfasst, innerhalb der dann ein Dateisystem angelegt wird. Dieses Dateisystem bleibt dennoch auf die starren Grenzen der Komponente beschränkt – insbesondere kann eine Komponente nicht vergrößert oder verkleinert werden, da kein herkömmliches Dateisystem diese Operationen unterstützt.

Das Dateisystem ZFS von Sun unterscheidet sich von anderen Systemen in genau diesem Punkt: es integriert Partitionierungsverfahren und Dateisystem. Da beide Schichten auf einander abgestimmt sind, ist es unter ZFS problemlos möglich, eine Partition (dort »storage pool« genannt) durch Hinzufügen weiterer Festplatten während des laufenden Betriebs zu vergrößern **[Sun04]**.

3.2 FAT

Das FAT-Dateisystem hat seinen Namen von der Dateizuordnungstabelle erhalten, der *File Allocation Table*, die die Belegung aller Datenbereiche und ihre Verkettung speichert. Das Akronym »FAT« wird schon seit langem sowohl für diese Tabelle als auch für das Dateisystem verwendet.

Aufgrund des hohen Alters und der weiten Verbreitung von FAT sind im WWW viele Quellen über die internen Strukturen des FAT-Dateisystems zu finden, die jedoch oft ungenau sind und sich teilweise widersprechen. Die folgenden Angaben wurden durch intensives Reverse Engineering verifiziert; darüber hinaus liegt ein funktionstüchtiger Treiber vor, der FAT-Dateisysteme liest und korrekt beschreibt (d.h. keine Fehlermeldungen durch Microsoft **SCANDISK**).

Es gibt gegenwärtig drei Varianten des FAT-Dateisystems: FAT12, FAT16 und FAT32. Die Zahl bezieht sich dabei auf die Größe der Einträge in der Dateizuordnungstabelle. Bis auf die Größe dieser Einträge sind FAT12 und FAT16 identisch, während bei FAT32 noch zusätzliche Veränderungen vorgenommen wurden. Daher werden die beiden älteren Varianten zuerst erläutert.

3.2.1 FAT12 und FAT16

Der erste Sektor einer FAT-Partition wird vom *Bootsektor* belegt. Er hat eine feste Struktur und enthält wichtige Informationen zum Dateisystem. Daran schließt sich die FAT an; aufgrund ihrer Bedeutung folgt bei allen Datenträgern außer RAM-Laufwerken noch mindestens eine Kopie. Bei Schreibzugriffen werden alle Kopien aktualisiert, für Lesezugriffe wird in der Regel nur die erste Kopie benutzt. Spezielle Wartungsprogramme wie Microsoft **SCANDISK** prüfen, ob alle FAT-Kopien identisch sind und können eine beschädigte FAT durch eine ihrer Kopien ersetzen. Direkt an die FAT schließt sich ein Bereich für die Einträge im Hauptverzeichnis des Datenträgers an, danach befindet sich der Datenbereich:

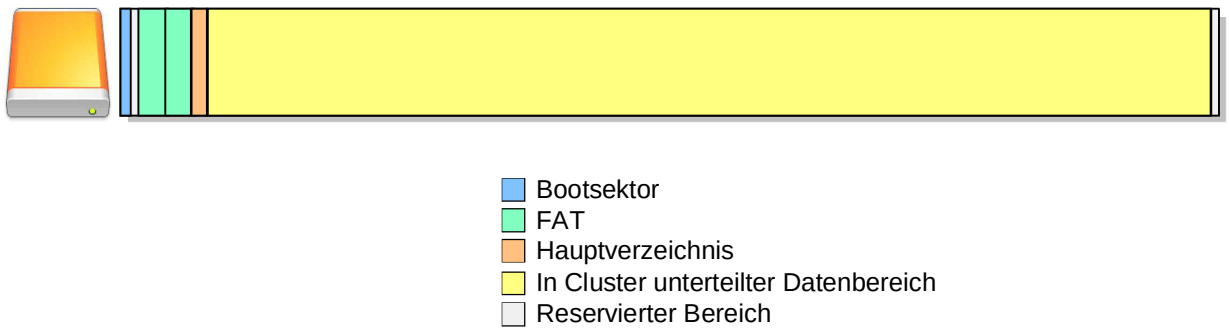


Abb. 3.2-1: Layout eines FAT12- oder FAT16-Dateisystems

Der Datenbereich eines FAT-Dateisystems enthält alle Dateien und Unterverzeichnisse. Der Speicherplatz des Datenbereichs wird allerdings nicht sektorweise verwaltet, sondern in größeren Einheiten: *Clustern*. Ein Cluster besteht immer aus 2^n Sektoren, wobei $n \in [0,6]$ sein muss. Die kleinste Clustergröße ist also 512 Byte (1 Sektor), die maximale 32768 Byte (64 Sektoren). Für Dateien können immer nur ganze Cluster belegt werden, so dass im worst case für eine Datei 32767 Byte Speicherplatz verschwendet werden. Diese Problematik wird in Abschnitt → *3.2.6* näher untersucht.

3.2.1.1 Aufbau der FAT

Wenn eine Datei mehrere Cluster belegt, so müssen diese nicht direkt aufeinander folgen, sondern können wahllos über den Datenbereich verteilt sein, so dass kein Cluster ungenutzt bleibt. Ein häufiger Wechsel der Position ist bei mechanischen Festplatten jedoch langsam, so dass es spezielle Optimierungsprogramme gibt, die die belegten Cluster des Dateisystems möglichst optimal anordnen und so den Zugriff beschleunigen.

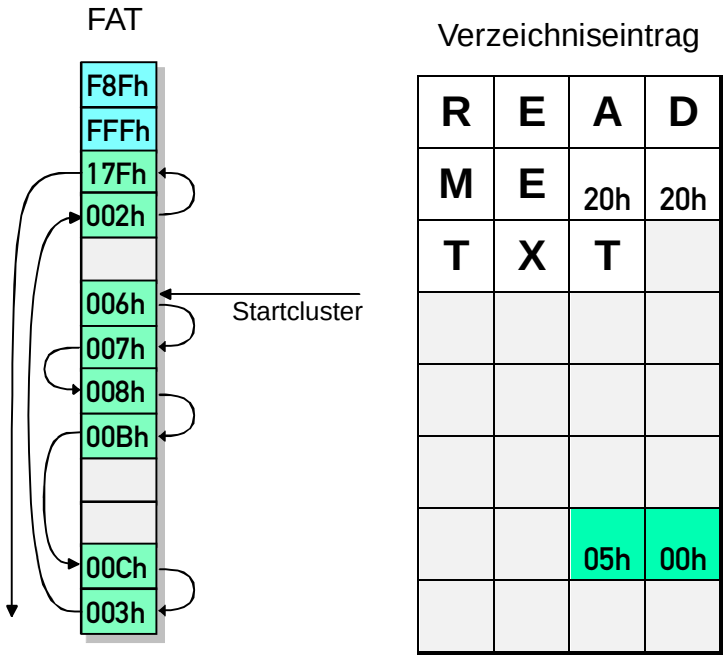
Die FAT enthält einen Eintrag für jeden Cluster des Datenbereichs, der entweder 12 oder 16 Bit breit ist. Jeder dieser Einträge bestimmt für einen Cluster den **nächsten Cluster**. Ist der Startcluster einer Datei bekannt, kann durch einen Lesezugriff auf die FAT die Nummer des zweiten Clusters bestimmt werden, und so fort; es handelt sich also um eine einfach verkettete Liste. Ein Seek innerhalb eines Dateikörpers ist dadurch in linearer Zeit möglich. Reservierte Werte werden u.A. für nicht belegte Cluster und den letzten Cluster einer Datei verwendet **[Tis94]**:

	Bedeutung
000h	Freier Cluster
001h	Reserviert
FF0h - FF6h	Reserviert
FF7h	Fehlerhafter Cluster
FF8h - FFFh	Letzter Cluster
Andere Werte	Nächster Cluster

Abb. 3.2-2: FAT-Einträge bei FAT12

	Bedeutung
0000h	Freier Cluster
0001h	Reserviert
FFF0h - FFF6h	Reserviert
FFF7h	Fehlerhafter Cluster
FFF8h - FFFFh	Letzter Cluster
Andere Werte	Nächster Cluster

Abb. 3.2-3: FAT-Einträge bei FAT16



Der Startcluster einer Datei ergibt sich aus seinem Verzeichniseintrag (→ *3.2.1.3*)

Abb. 3.2-4: Clusterverkettung durch FAT-Einträge **[Tis94]**

Die ersten beiden FAT-Einträge beziehen sich nicht auf Cluster, sondern enthalten Statusinformationen über das Dateisystem, die u.A. beim Start von Windows ausgelesen werden und eine Überprüfung der Partition durch **SCANDISK** zur Folge haben können:

	11	10	9	8	7	6	5	4	3	2	1	0
Eintrag 0	1	1	1	1	Media Descriptor							
Eintrag 1	S	P	1	1	1	1	1	1	1	1	1	1

Media Descriptor: gibt die Formatierung einer Diskette an, → *Abb. 3.2-6*
S: Dateisystem inkonsistent (Absturz während der letzten Sitzung) wenn **0**
P: physikalischer Fehler während der letzten Sitzung wenn **0**

Die Bits **S** und **P** sind immer die höchstwertigen Bits eines FAT-Eintrags (wichtig bei FAT16 und FAT32)

Abb. 3.2-5: Bedeutung der ersten beiden FAT-Einträge bei FAT12

	Datenträgertyp	Kapazität
F0h	3½"-Diskette, 2 Seiten, 80 Spuren, 18 Sektoren/Spur	1440 KB
	3½"-Diskette, 2 Seiten, 80 Spuren, 21 Sektoren/Spur	1700 KB
	3½"-Diskette, 2 Seiten, 83 Spuren, 21 Sektoren/Spur	1760 KB
	3½"-Diskette, 2 Seiten, 80 Spuren, 36 Sektoren/Spur	2880 KB
F8h	Festplatte, RAM-Laufwerk, Flash-Speicher	
F9h	5¼"-Diskette, 2 Seiten, 80 Spuren, 15 Sektoren/Spur	1200 KB
	3½"-Diskette, 2 Seiten, 80 Spuren, 9 Sektoren/Spur	720 KB
	3½"-Diskette, 2 Seiten, 80 Spuren, 10 Sektoren/Spur	800 KB
FAh	5¼"-Diskette, 1 Seite, 80 Spuren, 8 Sektoren/Spur	320 KB
	3½"-Diskette, 1 Seite, 80 Spuren, 8 Sektoren/Spur	320 KB
FBh	5¼"-Diskette, 2 Seiten, 80 Spuren, 8 Sektoren/Spur	640 KB
	3½"-Diskette, 2 Seiten, 80 Spuren, 8 Sektoren/Spur	640 KB
FCh	5¼"-Diskette, 1 Seite, 40 Spuren, 9 Sektoren/Spur	180 KB
FDh	5¼"-Diskette, 2 Seiten, 40 Spuren, 9 Sektoren/Spur	360 KB
	5¼"-Diskette, 2 Seiten, 40 Spuren, 10 Sektoren/Spur	400 KB
FEh	5¼"-Diskette, 1 Seite, 40 Spuren, 8 Sektoren/Spur	160 KB
FFh	5¼"-Diskette, 2 Seiten, 40 Spuren, 8 Sektoren/Spur	320 KB

Rot hinterlegte Einträge sind heute nicht mehr in Gebrauch

Abb. 3.2-6: Media Descriptor **[Tis94]**

Der erste Cluster im Datenbereich hat somit die Nummer 2. Da für die Clusternummer 12 bzw. 16 Bit zur Verfügung stehen und bestimmte Werte reserviert sind, ergibt sich eine maximale Clusteranzahl von 4078 bzw. 65518; aus den höchstens 64 Sektoren pro Cluster (32 KB) ergibt sich eine Maximalgröße von 128 MB für FAT12 und 2 GB für FAT16. Diese Beschränkungen waren 1995 der Grund für die Einführung von FAT32 (→ *3.2.2*).

Beim FAT12-Dateisystem kommt es übrigens vor, dass sich ein FAT-Eintrag über zwei Sektoren erstreckt, so dass eine Implementierung besonders kompliziert ist. Dafür ist die gesamte FAT höchstens 12 Sektoren groß, so dass sie problemlos im Arbeitsspeicher vorgehalten werden kann.

3.2.1.2 Bootsektor für FAT12 und FAT16

Der Bootsektor befindet sich wie bereits erwähnt immer im ersten Sektor einer Partition und enthält wichtige Informationen zum Aufbau des Dateisystems und zur Größe seiner Komponenten. Er hat für FAT12 und FAT16 folgende Struktur:

```
type
  FAT1216BootSector=record
    Jump: array[1..3] of Byte;
    Vendor: array[1..8] of Char;
    BYTEProSektor: Word;
    SektorenProCluster: Byte;
    ReservierteSektoren: Word;
    FATAnz: Byte;
    ROOTEntries,VolumeSize: Word;
    MediaType: Byte;
    SektorenProFAT,SektorenProSpur,Heads: Word;
    Offset,VolumeSizeBig: LongInt;
    PartitionNumber: Word;
    Signature: Byte;
    SerialNumber: LongInt;
    VolumeID: array[1..11] of Char;
    Name: array[1..8] of Char;
end;
```

Im Anschluss daran ist im Bootsektor noch Platz für Programmcode, den sogenannten Bootloader. Der Programmcode im Master Boot Record (→ *3.1.2*) lädt den Bootsektor der ersten bootfähigen Partition in den Arbeitsspeicher und führt ihn aus. Aufgabe des Bootloaders ist es, im Dateisystem den Betriebssystem-Kernel zu finden und ihn zu starten; aufgrund dieses Vorgangs hat der Bootsektor auch seinen Namen erhalten **[Tis94]**.

Um im Bootsektor Daten unterbringen zu können, sind am Anfang 3 Byte für einen Sprungbefehl reserviert (**JMP**), so dass die Daten beim Booten nicht als Binärcode behandelt und ausgeführt, sondern übersprungen werden. Direkt danach ist im Feld **Vendor** Platz für einen 8 Zeichen langen String, der den OEM-Code des Formatierungsprogramms enthält, also etwa **MSWIN4.1** oder **DESKWORK**. Die nachfolgenden Felder beschreiben dann die Struktur des Dateisystems.

BYTEProSektor sollte immer 512 sein, zusammen mit **SektorenProCluster** wird dadurch die Clustergröße bestimmt. **ReservierteSektoren** enthält die Anzahl der Sektoren, die sich zwischen dem Partitionsanfang und dem Anfang der ersten FAT befinden. Dieser Wert muss also mindestens 1 sein, da zwischen Partitionsanfang und FAT mindestens der Bootsektor abgespeichert wird (→ *Abb. 3.2-1*). Aber auch größere Werte sind denkbar, um weitere wichtige Sektoren abzuspeichern. Hiervon wird beispielsweise bei FAT32 (→ *3.2.2*) Gebrauch gemacht; der Standardwert beträgt dort 32.

FATAnz ist die Anzahl der FAT-Kopien (→ *3.2.1*) innerhalb des Dateisystems; Standardwert ist 2. Ausnahme sind RAM-Laufwerke: sie haben in der Regel nur eine einzige FAT, da dort keine physikalischen Fehler zu befürchten sind.

ROOTSize enthält die Anzahl der Einträge im Hauptverzeichnis; mehr dazu in Abschnitt → *3.2.1.3*. In **VolumeSize** wird die Anzahl der Sektoren innerhalb der Partition gespeichert; reicht dieses 16 Bit große Feld nicht aus, wird es auf 0 gesetzt und die Variable **VolumeSizeBig** verwendet. **Offset** ist die Anzahl der Sektoren zwischen LBA 0 (also Anfang des physikalischen Datenträgers) und Anfang der Partition. Dieser Wert muss selbstverändlich mit dem entsprechenden Eintrag innerhalb der Partitionstabelle (→ *3.1.2*) übereinstimmen. **MediaType** enthält **immer** einen Media Descriptor gemäß → *Abb. 3.2-6*, obwohl dieser nur bei Disketten von Bedeutung ist. Er sollte mit dem Media Descriptor im ersten FAT-Eintrag (→ *Abb. 3.2-5*) identisch sein.

In **SektorenProFAT** wird die Größe einer FAT gespeichert. Mit diesen Angaben können nun die LBA-Nummern aller wichtigen Datenstrukturen des Dateisystems errechnet werden:

Start der FAT = **Offset** + **ReservierteSektoren**
Start des Hauptverzeichnis = Start der FAT + **SektorenProFAT**∗**FATAnz**
Start des Datenbereichs = Start des Hauptverzeichnis + **ROOTEntries**÷16 (→ *3.2.1.3*)

Abb. 3.2-7: Positionen der Datenstrukturen innerhalb des FAT-Dateisystems

Außerdem enthält der Bootsektor noch die Anzahl der Sektoren pro Spur (**SektorenProSpur**) und die Anzahl der Köpfe (**Heads**), die natürlich mit der Geometrie des Laufwerks übereinstimmen muss. Das Feld **PartitionNumber** wird beim Mounten der Partition im Arbeitsspeicher ausgefüllt und hat im abgespeicherten Bootsektor bei Festplatten den Wert **80h**, bei Disketten den Wert **00h**.

Mit DOS 5.0 wurde der Bootsektor erweitert; dazu muss **Signature** den Wert **29h** haben. Dann enthalten die Felder **SerialNumber** und **VolumeID** die Seriennummer und den Datenträgernamen, wie sie unter DOS beim **DIR**-Befehl erscheinen:

*Abb. 3.2-8: Beispielausgabe des **DIR**-Befehls*

Name enthält entweder die Zeichenkette '**FAT12** ' oder '**FAT16** ', aufgefüllt mit Leerzeichen. Die letzten beiden Byte des Bootsektors (also ab Offset **1FEh**) müssen genau wie der Partitionssektor die Signatur **AA55h** tragen.

3.2.1.3 Hauptverzeichnis für FAT12 und FAT16

Direkt im Anschluss an die FAT-Kopien befindet sich das Hauptverzeichnis. Da jeder Eintrag in einem Verzeichnis 32 Byte lang ist, können pro Standardsektor mit 512 Byte Größe 16 Einträge gespeichert werden. Zusammen mit dem Feld **ROOTSize** aus dem Bootsektor (→ *3.2.1.2*) kann somit auch die Länge des Hauptverzeichnisses in Sektoren berechnet werden. Da die Größe des Hauptverzeichnisses durch das Formatierungsprogramm festgelegt wird, kann dort auch nur eine bestimmte Zahl an Dateien untergebracht werden. Dieser Umstand wurde erst mit der Einführung von FAT32 (→ *3.2.2*) behoben. Jeder Verzeichniseintrag weist folgendes Format auf:

```
type
  DirEntry=record
    Name: array[1..8] of Char;
    Ext: array[1..3] of Char;
    Attr: Byte;
    Reserved: array[1..10] of Byte;
    Zeit: Word;
    Datum: Word;
    FirstCluster: Word;
    Size: LongInt;
end;
```

	Bedeutung
01h	Read only
02h	Versteckt
04h	Systemdatei
08h	Volume-ID
10h	Verzeichnis
20h	Zu archivieren

Abb. 3.2-9: Dateiattribute

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Stunde					Minute						Sekunden÷2				
Jahr-1980								Monat			Tag				

Abb. 3.2-10: Codierung von Datum und Uhrzeit

R	**E**	**A**	**D**
M	**E**	20h	20h
T	**X**	**T**	Attr
		Zeit	
Datum		Cluster	
Dateigröße			

Abb. 3.2-11: Schema

Ein Eintrag im Hauptverzeichnis besteht aus einem 8 Zeichen langen Namen und einer 3 Zeichen langen Erweiterung, die ggf. mit Leerzeichen aufgefüllt werden. Das erste Zeichen des Dateinamens kann eine besondere Bedeutung haben:

	Bedeutung
00h	Dieser Eintrag und alle Nachfolger sind noch nie belegt gewesen (Urzustand)
05h	Das erste Zeichen hat den Code **E5h** (σ)
E5h	Dieser Eintrag ist ungültig; die Datei bzw. das Unterverzeichnis wurde gelöscht

Abb. 3.2-12: Markierung gelöschter Dateien

Am Ende des Dateieintrags wird das Datum und die Uhrzeit der letzten Änderung gespeichert (→ *Abb. 3.2-10*); sind diese 32 Bit alle **0**, so ist die Dateizeit ungültig und wird nicht angezeigt. Danach befindet sich die Nummer des ersten Clusters der Datei im Feld **FirstCluster** und die Größe in **Size**. Die maximale Dateigröße beträgt 2 GB.

Im Feld **Attr** werden die Attribute einer Datei gespeichert (→ *Abb. 3.2-9*). Versteckte Dateien und Systemdateien werden von der Standardshell **COMMAND.COM** nicht angezeigt. Das Archiv-Attribut wird bei jedem Schreibzugriff auf eine Datei gesetzt und wird von diversen Backup-Programmen benutzt. Der Datenträger-Name aus dem Bootsektor wird in der Regel noch zusätzlich als Verzeichniseintrag mit dem Attribut **VOLUME_ID** im Hauptverzeichnis gespeichert. Neben diesen Attributen existiert noch das Attribut **DEVICE**, das von MS-DOS aber nur intern verwendet wird:

```
ATTR_READ_ONLY  equ  1h
ATTR_HIDDEN     equ  2h
ATTR_SYSTEM     equ  4h
ATTR_VOLUME_ID  equ  8h
ATTR_DIRECTORY  equ 10h
ATTR_ARCHIVE    equ 20h
ATTR_DEVICE     equ 40h  ; This is a VERY special bit.
                         ; NO directory entry on a disk EVER
                         ; has this bit set. It is set non-zero
                         ; when a device is found by GETPATH
```

*Abb. 3.2-13: Auszug aus der Datei **DIRENT.INC***

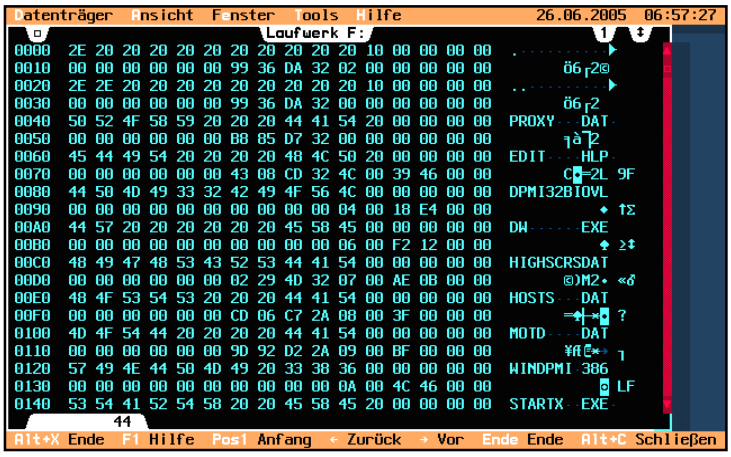


Abb. 3.2-14: Verzeichniseinträge

3.2.1.4 Unterverzeichnisse

Ist das Directory-Attribut eines Verzeichniseintrags gesetzt, wird durch diesen Eintrag keine Datei, sondern ein Unterverzeichnis beschrieben. Der Verzeichniseintrag für ein Unterverzeichnis enthält immer die Dateigröße 0 **[Tis94]**.

Das Feld, das normalerweise auf den ersten Cluster der Datei zeigt, gibt hier den Cluster an, der die Verzeichniseinträge des Unterverzeichnisses beinhaltet. Unterverzeichnisse werden genau wie Dateien in beliebigen Clustern gespeichert, die nicht unbedingt direkt aneinander anschließen. Verkettet werden die verschiedenen Cluster eines Unterverzeichnisses genau wie die Cluster einer Datei über die Einträge der FAT (→ *3.2.1.1*) **[Tis94]**.

Gespeichert werden darin die gleichen 32-Byte-Einträge, die auch im Hauptverzeichnis zu finden sind. Im Gegensatz zum Hauptverzeichnis werden Verzeichnisse allerdings dynamisch verwaltet: wird ein Verzeichnis erzeugt, reserviert DOS zunächst nur einen Cluster und legt darin leere Verzeichniseinträge an (→ *Abb. 3.2-12*). Sind alle Einträge gefüllt, wird beim nächsten Erzeugen einer Datei oder eines Unterverzeichnisses ein neuer Cluster für die Aufnahme weiterer Einträge belegt und mit dem ersten Cluster über die FAT verbunden **[Tis94]**.

Andererseits gibt DOS Verzeichnis-Cluster auch wieder frei, wenn sie leer geworden sind, weil alle darin befindlichen Dateien gelöscht wurden. Die Anzahl der Einträge in Unterverzeichnissen ist dadurch im Gegensatz zum Hauptverzeichnis nicht limitiert, sondern wird angepasst **[Tis94]**.

Bei der Erstellung eines Verzeichnisses werden in ihm direkt zwei Einträge mit den Namen '.' und '..' angelegt. Sie können nicht gelöscht werden und verschwinden erst bei der Auflösung des gesamten Unterverzeichnisses mit Hilfe des RD-Befehls **[Tis94]**:

```
ROOT@LocalHost|A:\SUBDIR>dir

Die Bezeichnung von Laufwerk A: ist DISKETTE
Seriennummer: 191D-07D7
Suchmaske: A:\SUBDIR\*.*

.            [Verzeichnis]  -D----  25.06.2005 22:41:48
..           [Verzeichnis]  -D----  25.06.2005 22:41:48
DIR      BMP         51.702  A-----  25.06.2005 18:45:44

              51.702 Byte in 1 Datei und 2 Verzeichnissen
            1.405.440 Byte frei

ROOT@LocalHost|A:\SUBDIR>
```

Abb. 3.2-15: Unterverzeichnis

Der erste der beiden Einträge verweist auf das aktuelle Unterverzeichnis, sein Startcluster enthält die Nummer des ersten Clusters des aktuellen Unterverzeichnisses. Der zweite Eintrag verweist auf das übergeordnete Verzeichnis; handelt es sich dabei um das Hauptverzeichnis, so wird als Startcluster 0, sonst der Startcluster des übergeordneten Verzeichnisses angegeben **[Tis94]**. Stimmen diese beiden Einträge nicht genau, so wird das Unterverzeichnis von den meisten DOS-Versionen als ungültig abgewiesen; ein CD-Befehl scheitert:

```
ROOT@LocalHost|I:\>dir

Die Bezeichnung von Laufwerk I: ist MEMORYSTICK
Suchmaske: I:\*.*

SUBDIR       [Verzeichnis]  -D----  25.06.2005 22:50:18
MEMSTICK IND              0  ----HR
MSTK_PRO IND              0  ----HR
MEMORYST ICK  [Bezeichnung]  A-I---

                  0 Byte in 3 Dateien und 1 Verzeichnis
          246.841.344 Byte frei

ROOT@LocalHost|I:\>cd subdir
Ungültiges Verzeichnis !

ROOT@LocalHost|I:\>
```

Abb. 3.2-16: ungültiges Unterverzeichnis

Da Verzeichnisse als lineare Liste von Verzeichniseinträgen gespeichert werden, ist das Auffinden eines Eintrags nur mittels Durchsuchen der Liste und damit in linearer Zeit möglich.

3.2.1.5 Inkompatible Erweiterungen

Bereits vor der Einführung von FAT32 (→ *3.2.2*) bereitete die Maximalgröße für FAT16-Partitionen von 2 GB Probleme. Daher gestatten beispielsweise DR-DOS und neuere Windows-Versionen das Erstellen von FAT16-Partitionen mit einer Größe von bis zu 4 GB, indem 128 Sektoren zu einem Cluster von 64 KB Größe zusammengefasst werden. Die so formatierten Dateisysteme können allerdings von den meisten anderen Betriebssystemen nicht gelesen werden, so dass von einer Verwendung insbesondere bei Flash-Speichern abzusehen ist **[Mue03]**.

3.2.2 FAT32

Mit FAT32 wurden einige Änderungen am Dateisystem vorgenommen, die sich auch im Bootsektor widerspiegeln; Einträge, die gegenüber den Bootsektoren älterer FAT-Versionen kompatibel sind, werden hellgrau (binärkompatibel) und dunkelgrau (gleiche Bedeutung wie bei FAT12 und FAT16, aber an ein anderes Offset innerhalb des Bootsektors verschoben) dargestellt:

```
type
  FAT32BootSector=record
    Jump: array[1..3] of Byte;
    Vendor: array[1..8] of Char;
    ByteProSektor: Word;
    SektorenProCluster: Byte;
    ReservierteSektoren: Word;
    FATAnz: Byte;
    ROOTEntries,VolumeSize: Word;
    MediaType: Byte;
    SektorenProFAT,SektorenProSpur,Heads: Word;
    Offset,VolumeSizeBig: LongInt;
    SektorenProFATBig: LongInt;
    Flags,Version: Word;
    RootCluster: LongInt;
    InfoSector,Backup: Word;
    Reserved: array[1..12] of Byte;
    Partitionnumber: Word;
    Signature: Byte;
    SerialNumber: LongInt;
    VolumeID: array[1..11] of Char;
    Name: array[1..8] of Char;
end;
```

Die wichtigsten Änderungen bei FAT32 betreffen die FAT: die Breite der Einträge wurde von 16 Bit auf 32 Bit ausgedehnt. Gegenwärtig sind allerdings die oberen 4 Bit reserviert, so dass das Dateisystem eigentlich FAT28 heißen müsste; es kann also 2^{28} Cluster verwalten. Bei einer maximalen Clustergröße von 32 KB ergibt das eine Maximalgröße von 8,8 TB. Gegenwärtig können aufgrund der Beschränkungen im Master Boot Record (→ *3.1.2*) allerdings nur Partitionen bis maximal 2 TB angelegt werden. Es ist denkbar, dass zukünftig die oberen 4 Bit freigegeben werden, um Partitionen bis 128 TB Größe zu ermöglichen.

Die 32-Bit-Einträge in der FAT haben analog zu den älteren Versionen die folgende Bedeutung:

	Bedeutung
0000000h	Freier Cluster
0000001h	Reserviert
FFFFFF0h - FFFFFF6h	Reserviert
FFFFFF7h	Fehlerhafter Cluster
FFFFFF8h - FFFFFFFh	Letzter Cluster
Andere Werte	Nächster Cluster

Abb. 3.2-17: FAT-Einträge bei FAT32

Da statt **SektorenProFAT** (16 Bit) nun das Feld **SektorenProFATBig** (32 Bit) benutzt wird, kann eine FAT-Kopie deutlich mehr als 65535 Sektoren belegen. Die viel größere Clusteranzahl ermöglicht nicht nur größere Dateisysteme, sondern hat auch bei Partitionen bis 2 GB deutliche Vorteile gegenüber FAT16, da die Cluster entsprechend kleiner sein können und so weniger Speicherplatz verschwendet wird (→ *3.2.6*).

Da Clusternummern nun eine Größe von 32 Bit haben, muss auch die Datenstruktur der Verzeichniseinträge entsprechend erweitert werden; binärkompatible Einträge sind hellgrau dargestellt:

```
type
  DirEntry=record
    Name: array[1..8] of Char;
    Ext: array[1..3] of Char;
    Attr: Byte;
    Reserved: array[1..8] of Byte;
    FirstClusterHi: Word;
    Zeit: Word;
    Datum: Word;
    FirstClusterLo: Word;
    Size: LongInt;
end;
```

R	E	A	D
M	E	20h	20h
T	X	T	Attr
Cluster		Zeit	
Datum		Cluster	
Dateigröße			

Abb. 3.2-18: Schema

Die Nummer des Startclusters wird also auf 2 Words aufgeteilt, indem der reservierte Bereich entsprechend verkürzt wird. Zudem sind nun Dateien bis zu einer Größe von 4 GB möglich, da das Dateisystem selbst im Gegensatz zu FAT16 diese Größe erreichen kann.

Die enorme Anzahl von maximal 2^{28} Clustern bringt aber weitere Probleme mit sich: so sind z.B. im Bootsektor keine Informationen zum freien Speicherplatz zu finden. Diese werden bei FAT12 und FAT16 beim Mounten errechnet, indem jeder Eintrag in der FAT begutachtet wird. Bei maximal 65518 Clustern, deren FAT-Einträge höchstens 256 Sektoren belegen, ist das kein Problem. Beim FAT32-Dateisystem kann die FAT allerdings so groß werden, dass eine Berechnung des freien Speicherplatzes zu lange dauert. Dasselbe Problem ergibt sich bei der Suche nach freien Clustern.

Aus diesem Grund wird bei FAT32-Dateisystemen hinter dem Bootsektor, der aus Sicherheitsgründen möglichst nicht verändert werden soll, der »File System Info Sector« abgespeichert, um variable Daten aufzunehmen. Er wird im Bootsektor durch das Feld **InfoSector** referenziert; die Angabe enthält das Offset in Sektoren, gemessen vom Partitionsstart **[Mue03]**:

```
type
  FSInfoSector=record
    LeadingSignature: LongInt;
    Reserved1: array[1..480] of Byte;
    StructureSignature: LongInt;
    FreeCount: LongInt;
    NextFree: LongInt;
    Reserved2: array[1..12] of Byte;
    TrailingSignature: LongInt;
end;
```

Die **LeadingSignature** muss **52526141h** sein, die **StructureSignature 72724161h** und die **TrailingSignature 000055AAh**. Beinhalten die Angaben in **FreeCount** und **NextFree** den Wert -1, so sind sie ungültig und neu zu berechnen. Die Präsenz eines solchen Infosektors ist allerdings optional; ist er nicht vorhanden, enthält die Variable **InfoSector** im Bootsektor den Wert **FFFFh [Mue03]**.

Bootsektor und Infosektor liegen bei FAT32-Dateisystemen in der Regel doppelt vor, um im Schadensfall die defekten Sektoren durch eine Kopie ersetzen zu können. Das Feld **Backup** im Bootsektor enthält das Offset von Bootsektor und ggf. anschließendem Infosektor, gemessen in Sektoren vom Partitionsanfang. Die Angabe ist nur gültig, wenn sie ungleich **FFFFh** ist **[Mue03]**.

Eine weitere Neuerung von FAT32 besteht darin, dass es kein klassisches Hauptverzeichnis mehr gibt; statt dessen wird es wie normale Unterverzeichnisse (→ *3.2.1.4*) in verketteten Clustern innerhalb des Datenbereichs abgelegt. Dadurch ist die Anzahl der Dateien im Hauptverzeichnis nicht mehr limitiert. Die Konsequenz daraus ist, dass das Feld **ROOTEntries** im Bootsektor den Wert 0 haben muss; die Nummer des Startclusters findet sich im neu definierten Eintrag **RootCluster**. Das Layout eines FAT32-Laufwerks sieht somit etwas einfacher aus:

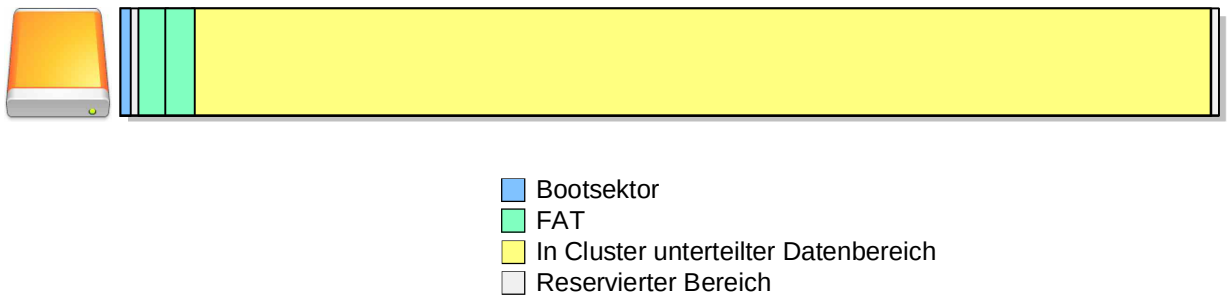


Abb. 3.2-19: Layout eines FAT32-Dateisystems ohne statisches Hauptverzeichnis

Windows 98 verweigert übrigens den Zugriff auf alle FAT32-Medien, die kleiner als 512 MB sind, selbst wenn diese von anderen Betriebssystemen oder Geräten korrekt formatiert wurden.

3.2.3 Lange Dateinamen

Die Dateinamen für das FAT-Dateisystem sind auf 8 Zeichen und eine 3 Zeichen lange Erweiterung begrenzt (→ *3.2.1.3*). Mit der zunehmenden Popularität von Microsoft Windows wurde das FAT-Dateisystem ergänzt, um flexiblere Dateinamen zu ermöglichen.

3.2.3.1 Windows NT

Die ersten Erweiterungen am FAT-Dateisystem implementiert das Server-Betriebssystem Windows NT. Um Kompatibilität zu Unix-Systemen herzustellen, müssen auch Dateien gespeichert werden, deren Namen nur aus Kleinbuchstaben besteht. Das bisher unbenutzte Byte, das sich im Verzeichniseintrag an Offset 12 befindet (→ *Abb. 3.2-20*), wird daher von Windows NT verwendet. Ist Bit 3 gesetzt (**08h**), so sind die 8 Buchstaben des Dateinamens in Kleinbuchstaben darzustellen, obwohl sie auf dem Datenträger als kompatible Großbuchstaben gespeichert sind. Bit 4 (**10h**) hat dieselbe Bedeutung für die Dateiendung. Die übrigen Bits dieses »NT-Bytes« sind undefiniert und sollen mit **0** initalisiert werden.

R	E	A	D
M	E	20h	20h
T	X	T	Attr
NT			
Cluster		Zeit	
Datum		Cluster	
Dateigröße			

Abb. 3.2-20: NT-Byte

3.2.3.2 VFAT

Seit der Einführung von Windows 95 ist das Speichern von Dateien mit bis zu 255 Zeichen langen Namen möglich, und zwar mit allen Varianten des FAT-Dateisystems. Diese Technik wird »VFAT« genannt; der Begriff ist jedoch irreführend, da an der FAT keinerlei Änderungen vorgenommen werden. Vielmehr wird für eine Datei mit langem Namen ein kurzes Alias gebildet:

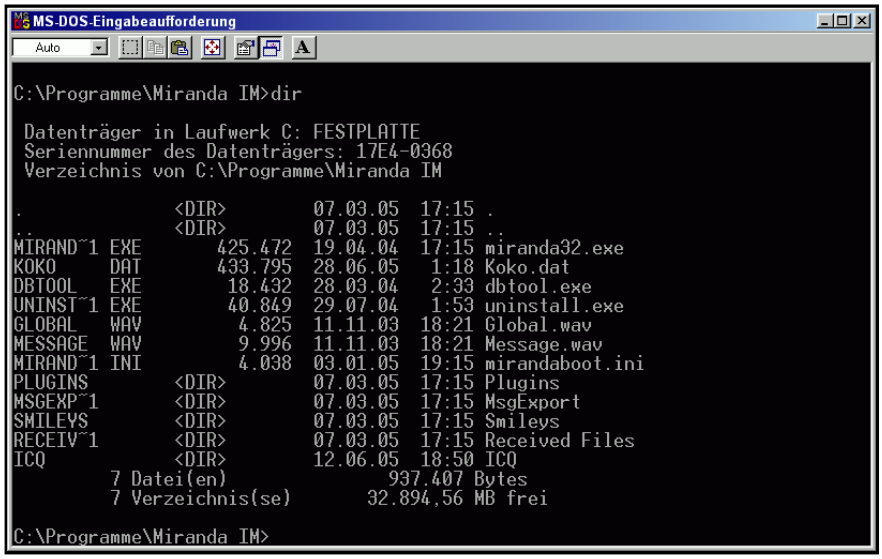


Abb. 3.2-21: Alias für lange Dateinamen

Lange Dateinamen werden nun gespeichert, indem für die entsprechenden Dateien mehrere Directory-Einträge belegt werden. Die zusätzlichen Einträge erhalten alle das Attribut **0Fh** (Volume-ID, Systemdatei, Versteckt und Read only), denn diese Kombination kommt bei keiner Datei ohne langen Dateinamen vor, so dass die zusätzlichen Verzeichniseinträge von DOS-Programmen ignoriert werden. Die so markierten Verzeichniseinträge haben nicht das bisher vorgestellte Format, sondern enthalten 13 Unicode-Zeichen, die je 16 Bit groß sind. Außerdem enthält der Verzeichniseintrag eine fortlaufende Nummer zur Ordnung der Einträge. Die Bytes, die im Normalfall den Startcluster enthalten, werden zur Sicherheit mit **0** initialisiert **[Mue03]**.

3.2.4 FAT32+

Eine weitere Ergänzung, die »FAT32+« genannt wird, wurde Anfang 2006 von Kuhnt, Georgiev und Davis definiert **[Kuh06]**. FAT32+ gestattet Dateien, die größer als 4 GB sind und wird seit 2006 von DESKWORK, DR-DOS, DW-DOS und Linux unterstützt. Übliche FAT-Versionen stellen in den Verzeichniseinträgen (→ *Abb. 3.2-11*) 32 Bit für die Dateigröße zur Verfügung, was Dateien mit einer Länge von 2^{32}-1 Byte, also knapp 4 GB, ermöglicht. Wenn auf inkompatible Erweiterungen (→ *3.2.1.5*) verzichtet wird, können größere Dateien nur unter FAT32 (→ *3.2.2*) auftreten, daher der Name »FAT32+«.

FAT32+ stellt 38 Bit für die Speicherung der Dateilänge bereit, was Dateigrößen bis 256 GB ermöglicht (2^{38}-1 Byte). Die zusätzlichen 6 Bit werden dabei auf die reservierten Bits 0-2 und 5-7 des NT-Bytes (→ *3.2.3.1*) verteilt, da diese von keinem Betriebssystem einschließlich Windows NT verwendet werden.

Aus dieser Vorgehensweise entstehen allerdings einige Probleme bezüglich der Kompatibilität **[Kuh06]**. Wird eine FAT32+-Partition von einem Betriebssystem gelesen oder beschrieben, das FAT32+ nicht beherrscht, so kann die Dateigröße nicht korrekt berechnet werden, denn an der üblichen Stelle befinden sich nur die niederwertigen 32 Bit. Aus diesem Grund wird im Bootsektor (→ *3.2.2*) die Version des Dateisystems mit 0.1 statt wie bisher mit 0.0 angegeben, um ein Mounten zu verhindern.

3.2.5 Transaction-Safe FAT

Die Datenstrukturen des FAT-Dateisystems, vor allem Teile der FAT, werden bei Bedarf in den Arbeitsspeicher eingelesen und dort verändert. Zu bestimmten Zeitpunkten, etwa dem Schließen einer Datei nach einem Schreibzugriff, werden die veränderten Sektoren dann auf den Datenträger zurückgeschrieben. Stürzt das System ab, bevor alle Sektoren gespeichert wurden, wird das FAT-Dateisystem beschädigt; typische Schäden, die dann z.B. von Microsoft **SCANDISK** behoben werden müssen, sind *verlorene* Cluster, *querverbundene* Dateien, mehrere Dateien mit gleichem Na-

men, Dateien ohne Directory-Eintrag und so weiter. Ein Verfahren zur Vermeidung von inkonsistenten Dateisystemen ist das *Journaling* **[Dat05]**.

3.2.5.1 Journaling

Journaling-Dateisysteme bezeichnen jede Dateioperation wie Anlegen, Schreiben, Löschen oder Umbenennen als »Transaktion«. Durch das Design eines Journaling-Dateisystems kann eine Transaktion immer nur ganz oder gar nicht gespeichert werden; Transaktionen sind also unabhängig von ihrer Größe *atomar*, obwohl eigentlich nur das Beschreiben eines einzigen Sektors atomar ist. Das wird dadurch erreicht, dass ein Bereich des Datenträgers als Journal reserviert wird. Auf Sektor- oder Blockebene wird dort jede Transaktion protokolliert **[Gia99]**:

1. Anlegen eines neuen Journal-Eintrags
2. Jeder Cluster oder Sektor, der zu den internen Datenstrukturen des Dateisystems gehört (also keine Nutzdaten einer Datei enthält), werden bei einem Schreibzugriff nicht direkt auf den Datenträger, sondern zunächst ins Journal geschrieben
3. Schließen des Journal-Eintrags
4. Kopieren aller Cluster aus dem Journal-Eintrag an die entsprechende Stelle des Dateisystems
5. Löschen des Journal-Eintrags

Alle Schreiboperationen werden also zunächst außerhalb der »echten« Datenstrukturen gesammelt. Stürzt nun das System während einer beliebigen Phase ab, so bleibt das Dateisystem konsistent. Tritt ein Absturz während der Schritte 1 bis 3 auf, so ist der Journal-Eintrag ungültig und wird beim nächsten Mounten ignoriert; an Sektoren außerhalb des Journals wurden noch gar keine Änderungen vorgenommen. Werden hingegen Phasen 4 oder 5 unterbrochen, so findet das Betriebssystem beim nächsten Mounten einen intakten Journal-Eintrag vor und fährt einfach mit Schritt 4 fort; ggf. werden dabei Sektoren doppelt aus dem Journal übertragen **[Gia99]**.

3.2.5.2 Funktionsweise von TFAT

TFAT (Transaction-Safe FAT) setzt voraus, dass das Dateisystem mit mindestens zwei FAT-Kopien formatiert wurde (→ *3.2.1.2*); die zweite Kopie wird als Journal betrachtet. Bei Schreibzugriffen auf die FAT wird zunächst nur die als Journal verwendete Kopie verändert, erst danach wird die jeweils andere aktualisiert. Der Vorteil liegt darin, dass das Dateisystem auf dem Datenträger kompatibel zu FAT bleibt, dafür die Verzeichniseinträge aber keinem Journaling unterliegen. Ein Datenverlust ist bei TFAT also nach wie vor möglich, wenn auch wesentlich unwahrscheinlicher **[Dat05]**. TFAT-Dateisysteme arbeiten bei Schreibzugriffen langsamer als übliche FAT-Partitionen, Lesezugriffe sind etwa gleich schnell **[MSD05a]**.

3.2.6 Performanz des FAT-Dateisystems

Das FAT-Dateisystem wird bei fast allen Flash-Speichermedien (→ *5.3*) verwendet und ist daher vermutlich in größeren Stückzahlen im Einsatz als zu Zeiten von MS-DOS. Daher ist es wichtig, die Performanz zu untersuchen.

3.2.6.1 Clusterverkettung

Die FAT verwaltet die von einer Datei belegten Cluster als verkettete Liste. Wird die Datei also sequentiell gelesen, muss nach dem Lesen eines Clusters der Folgecluster aus der FAT gelesen werden. Bei einer **SEEK**-Operation ist es erforderlich, vom Startcluster ausgehend die komplette Verkettung bis zum Zielcluster zu ermitteln (→ *3.2.1.1*), was nur in $O(n)$ möglich ist.

Um die Geschwindigkeit von Operationen auf dem FAT-Dateisystem zu maximieren, sollte die Clustergröße möglichst groß sein, also 32 KB betragen (64 KB große Cluster können nicht von allen Betriebssystemen gelesen werden, → *3.2.1.5*).

3.2.6.2 Verschwendeter Speicher

Ein Nachteil von großen Clustern ist der verschwendete Speicher im letzten Cluster einer Datei. Bei einer angenommenen Clustergröße von 32 KB würde eine Datei mit einer Größe von 1 KB eine Verschwendung von 31 KB verursachen. Dies ist einer der größten Kritikpunkte am FAT-Dateisystem; deshalb gibt es eine Vielzahl an Programmen, die die Größe des verschwendeten Speichers (engl. »Slack«) berechnen:

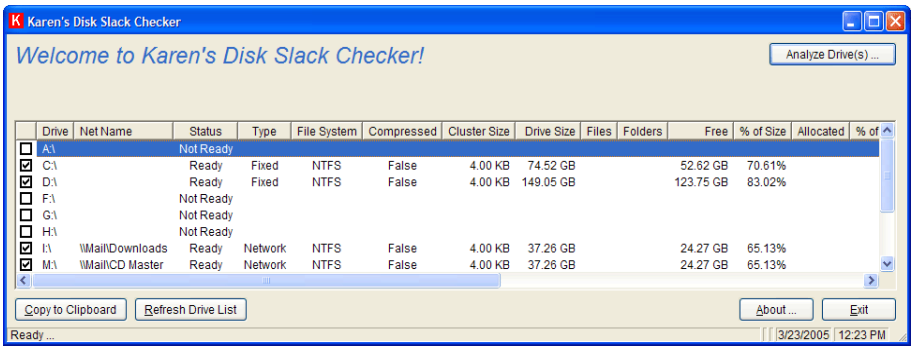


Abb. 3.2-22: Karen's Disk Slack Checker **[KSC05]**

Norton Partition Magic **[NPM05]** und Stat 2000 **[Sta05]** bieten die Möglichkeit, den Inhalt einer FAT-Partition zu analysieren und die Speicherverschwendung bei verschiedenen Clustergrößen zu berechnen. Der Benutzer kann mit Norton Partition Magic **[NPM05]** eine neue Clustergröße auswählen; das Programm manipuliert dann das Dateisystem ohne Datenverlust, so dass dieses fortan die neue Clustergröße verwendet:

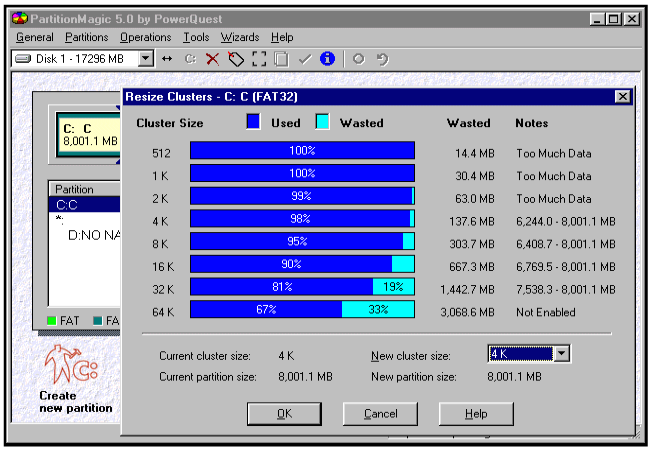


Abb. 3.2-23: Norton Partition Magic **[NPM05]**

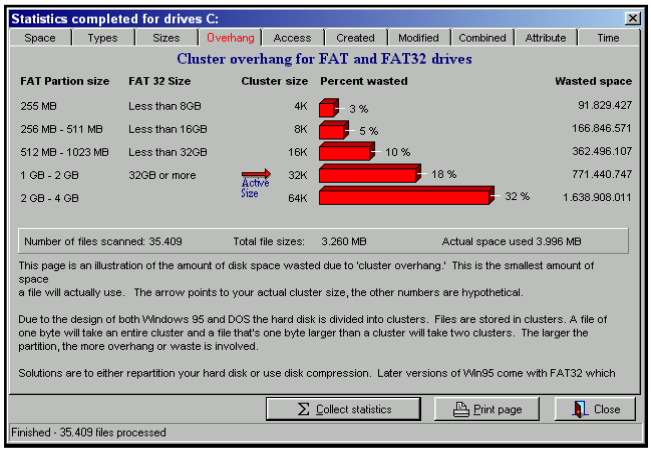


Abb. 3.2-24: Stat 2000 **[Sta05]**

Die Diagramme in → *Abb. 3.2-23* und → *Abb. 3.2-24* zeigen kein proportionales Wachstum des verschwendeten Speicherplatzes bei einer Verdoppelung der Clustergröße: 19% des belegten Speicherplatzes bei 32 KB großen Clustern werden ungenutzt belegt, bei 64 KB sind es aber nicht etwa 38%, sondern nur 33% (→ *Abb. 3.2-23*). Walker schlägt vor, bei fehlender Kenntnis der tatsächlichen Dateigrößen den ungenutzen Speicherplatz wie folgt zu berechnen **[Wal97]**:

$$\textit{Verschwendeter Speicher} = \textit{Dateianzahl} * \textit{Clustergröße} * 0{,}5 * \textit{Hilfsfaktor}$$

Partitionsgröße	Clustergröße	Hilfsfaktor
>128 bis 256 MB	4096 (4 KB)	1,2
>256 bis 512 MB	8192 (8 KB)	1,6
>512 bis 1024 MB	16384 (16 KB)	1,8
>1024 bis 2048 MB	32768 (32 KB)	1,9

Abb. 3.2-25: Slack-Berechnung nach **[Wal97]**

Die Einführung eines willkürlichen Hilfsfaktors (»*fudge factor* is a constant that adjusts for things like file size distribution«) ist sehr unbefriedigend – Grund genug, den Zusammenhang zwischen Clustergröße und verschwendetem Speicher näher zu untersuchen.

Sei n die Clustergröße in Byte, s die Größe einer beliebigen Datei. Dann ergibt sich für den Slack die Größe $s \bmod n$, also höchstens n-1 Byte, denn bei einer Speicherplatzverschwendung n Byte wäre der letzte Cluster vollständig unbenutzt und somit nicht belegt. Wird nun die Clustergröße verdoppelt, so sind zwei Fälle denkbar:

1. $s \bmod (2n) = s \bmod n$

   Das bedeutet, dass der verschwendete Speicherplatz unverändert bleibt. Als Beispiel sei eine Datei der Größe s = 1000 Byte betrachtet; die Clustergröße n betrage 512 Byte. Die Datei belegt also zwei Cluster zu je einem Sektor, der letzte Cluster enthält 24 ungenutzte Byte. Wird die Clustergröße nun verdoppelt, so belegt die Datei einen Cluster mit zwei Sektoren; am Ende befinden sich immer noch 24 Byte Speicherplatz, die nicht belegt sind.

2. $s \bmod (2n) = (s \bmod n) + n$
   In diesem Fall tritt mindestens eine Verdopplung des verschwendeten Speicherplatz auf. Als Beispiel für Fall 2 sei eine Datei der Größe s = 1500 Byte betrachtet; die Clustergröße n betrage 512 Byte. Die Datei belegt dann drei Cluster zu je einem Sektor, der letzte Cluster enthält 36 ungenutzte Byte. Wird die Clustergröße nun verdoppelt, so belegt die Datei zwei Cluster mit jeweils zwei Sektoren; am Ende befinden sich jetzt also 548 Byte Slack. Das ist mehr als 15 mal so viel.

Bei allen Dateien mit $s \leq n$ tritt bei einer Verdopplung der Clustergröße Fall 2 auf. Hat s etwa dieselbe Größenordnung wie n, wächst der verschwendete Speicher bei dieser Stufe relativ zum alten Slack also sehr stark (maximal Faktor n+1). Ist hingegen $s << n$, so fällt die Dateigröße s kaum ins Gewicht, so dass sich der Slack annähernd verdoppelt. Folgerichtig konvergiert der Hilfsfaktor in **[Wal97]** gegen 2, um zusammen mit dem festen Faktor 0,5 ein proportionales Wachstum abzubilden. Bei großen Dateien mit s >> n sind beide Fälle etwa gleich wahrscheinlich; solche Dateien tragen also nicht sehr stark zu einem Wachstum des verschwendeten Speichers bei. Gemittelt über alle Dateien wächst bei ihnen der Slack von Stufe zu Stufe nur um durchschnittlich ½n – sie gehen mit dem konstanten Faktor 0,5 in **[Wal97]** ein.

Der verschwendete Speicherplatz hängt also tatsächlich von der Verteilung der Dateigrößen ab; die von Walker in **[Wal97]** vorgeschlagenen Hilfsfaktoren orientieren sich an einer als »typisch« angenommenen Verteilung. Über die Verteilung von Dateigrößen wird viel diskutiert, vor allem bezüglich des Datenverkehrs im WWW. Es wird schnell offensichtlich, dass es mehr kleine Dateien gibt als große Dateien; die Verteilung muss also endlastig (engl. »long-tailed«) sein **[Wik05b]**. Basierend auf der von Irlam 1993 durchgeführten »Unix File Size Survey« **[Irl93]** werden verschiedene Modelle wie die Pareto- **[Wik05c]** oder lognormale **[Wik05d]** Verteilung erörtert **[Dow01]**.

Zur Visualisierung des verschwendeten Speichers bei verschiedenen Clustergrößen wurde zunächst ein Testprogramm (→ *C*) erstellt, das die Dateien einer FAT-Partition analysiert und die Speicherverschwendung bei verschiedenen Clustergrößen berechnet, das genaue Zahlen ausgibt:

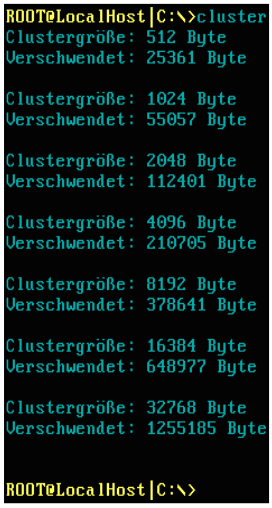


*Abb. 3.2-26: Ausgabe von **CLUSTER.EXE** (→ C)*

Mit dem Testprogramm wurden sechs FAT-Partitionen untersucht:

1. FAT12 enthält eine DOS-Installation mit Treibern
2. FAT16 mit minimaler DOS-Installation und einigen Anwendungsprogrammen
3. FAT32 mit dem hier vorgestellten Datenbank-Dateisystem
4. FAT32 mit Windows 3.11 und großen Videodateien
5. FAT32 mit Windows 98, vielen Applikationen und wenigen Benutzerdateien
6. FAT32 mit Windows 98, diversen Anwendungen und Benutzerdateien

Folgende Ergebnisse wurden geliefert:

	512 Byte	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB
1	24961	54657	112001	210305	374145	644481	1250689
2	111199	247391	464479	802399	1420895	2444895	4247135
3	466835	951699	1858963	3646867	7132563	12899731	24155539
4	733413	1452773	2973413	5584613	10651365	19523301	33400549
5	7514869	15060725	29788917	55991029	104233717	186006261	327072501
6	5864606	11569822	23084702	45719198	80314014	176285726	335429278

Angaben in Byte

Abb. 3.2-27: Testergebnisse

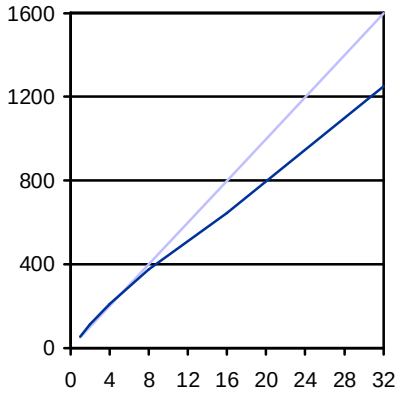


Abb. 3.2-28: Partition 1

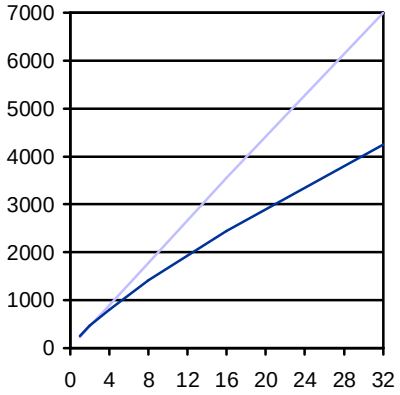


Abb. 3.2-29: Partition 2

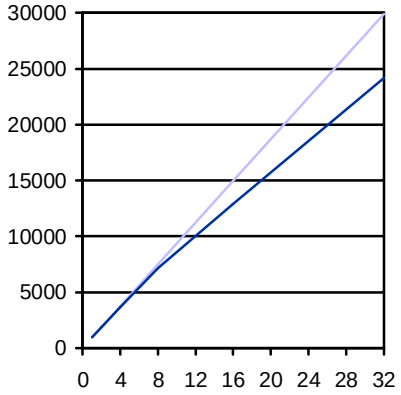


Abb. 3.2-30: Partition 3

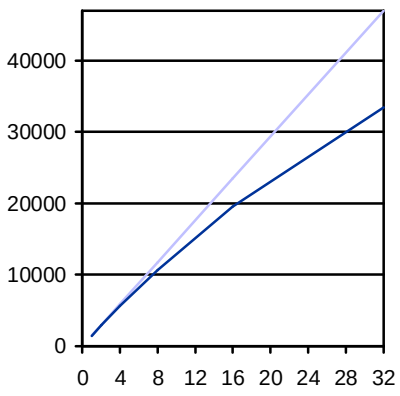


Abb. 3.2-31: Partition 4

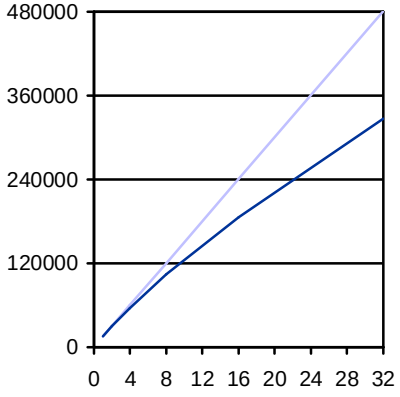


Abb. 3.2-32: Partition 5

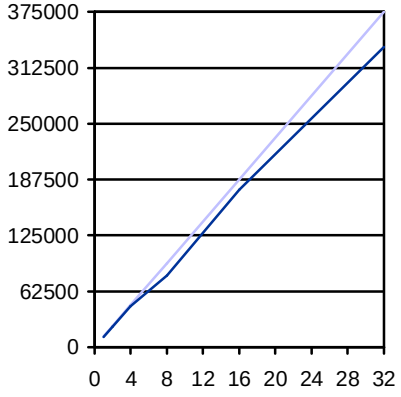


Abb. 3.2-33: Partition 6

In den Diagrammen zeigt die dunkelblaue Linie den tatsächlich verschwendeten Speicherplatz an, während die hellblaue Linie von einem linearen Wachstum ausgeht. Es zeigt sich in allen betrachteten Fällen, dass der nicht genutzte Speicher wie vorhergesagt zunächst stärker wächst als die Clustergröße (durch die gewählte Skalierung im Diagramm nicht sichtbar), danach aber teilweise deutlich schwächer. Kleine Dateien verschwenden also überproportional viel Speicher, was sich offensichtlich mit zunehmender Clustergröße relativiert. Da in → *3.2.6.1* die Geschwindigkeitsvorteile großer Cluster aufgezeigt wurden, ist als Folge davon besonders stark darauf zu achten, dass möglichst nur große Dateien im physikalischen Dateisystem abgelegt werden.

3.2.6.3 Lange Dateinamen

Selbstverständlich sind Dateinamen mit 8 Zeichen Länge unbefriedigend. Es ist dennoch nicht ratsam, VFAT zu implementieren, da sich dort lange Dateinamen auf viele Verzeichniseinträge verteilen und diese blockieren (→ *3.2.3*). Das ist vor allem bei FAT12- und FAT16-Partitionen ungünstig, da das Hauptverzeichnis dort nur eine bestimmte Anzahl an Einträgen aufnehmen kann (→ *3.2.1.3*).

3.3 NTFS

NTFS ist das Akronym für »New Technology File System« und wird von Microsoft als Nachfolger von FAT vermarktet. Es wurde mit Windows NT eingeführt und kann ohne Zusatztreiber nicht von DOS oder Windows 9x gelesen werden. Die maximale Partitionsgröße beträgt 2^{64} Byte, also 16 EB **[Sol98]**; bei Laufwerken mit üblicher Partitionstabelle (→ *3.1.2*) ist die maximale Größe allerdings auf 2 TB begrenzt. Der schematische Aufbau einer NTFS-Partition ist einfacher als bei FAT (→ *3.2*), allerdings sind die einzelnen Datenstrukturen von NTFS deutlich komplexer:

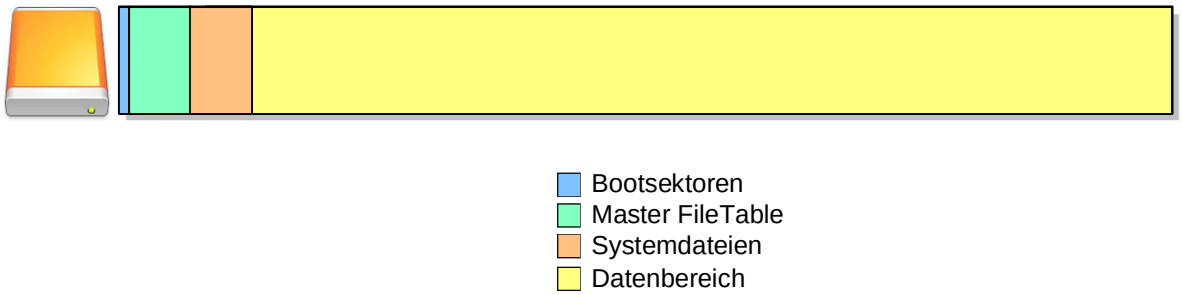


Abb. 3.3-1: Aufbau einer NTFS-Partition

3.3.1 Bootsektoren

Da das NTFS-Dateisystem komplizierte Datenstrukturen besitzt, sind die ersten 16 Sektoren für den Bootbereich reserviert. Er bietet genügend Platz, um Programmcode zum Finden und Laden der Systemdateien unterzubringen. Im ersten Sektor werden Informationen zum Dateisystem abgelegt **[Mue03]**; er hat große Ähnlicheit mit dem FAT-Bootsektor (→ *3.2.1.2*). Identische Felder sind hellgrau dargestellt:

```
type
  RealLargeNumber=record
    LoDWORD,HiDWORD: LongInt;
end;
  NTFSBootSector=record
    Jump: array[1..3] of Byte;
    Vendor: array[1..8] of Char;
    ByteProSektor: Word;
    SektorenProCluster: Byte;
    ReservierteSektoren: Word;
    Reserved1: array[1..5] of Byte;
    MediaType: Byte;
    Reserved2: array[1..2] of Byte;
    SektorenProSpur,Heads: Word;
    Offset: LongInt;
    Reserved3: array[1..8] of Byte;
    TotalSectors64,MFTCluster,MIRCluster: RealLargeNumber;
    MFTClusterPerEntry: ShortInt;
    Reserved4: array[1..3] of Byte;
    ClusterPerIndex: ShortInt;
    Reserved5: array[1..3] of Byte;
    SerialNo: RealLargeNumber;
    Reserved6: array[1..4] of Byte;
end;
```

Viele Variablen befinden sich an derselben Stelle wie im FAT-Bootsektor, da sie für die meisten Dateisysteme als *Boot Parameter Block* normiert sind. Einige Einträge mussten allerdings nach hinten verlagert werden, da sie nun 64 Bit lang sind, wie zum Beispiel die Gesamtzahl der Sektoren in **TotalSectors64**.

Genau wie beim FAT-Dateisystem ist der NTFS-Datenbereich (→ *3.3.4*) in Cluster unterteilt. Im Bootsektor finden sich der Startcluster des Hauptverzeichnisses, das hier *Master File Table* (MFT) genannt wird. Direkt im Anschluss befindet sich der Startcluster der MFT-Kopie, da selbige aus Sicherheitsgründen doppelt abgespeichert wird (**MFTCluster**, **MIRCluster**). Außerdem wird angegeben, aus wie vielen Clustern ein MFT-Eintrag (**MFTClusterPerEntry**) und ein Block eines Unterverzeichnis-Index (**ClusterPerIndex**) bestehen. Die Clusterverkettung wird in einem B^+-Baum (→ *2.4.1.3*) gespeichert, so dass ein Seek in $O(\log n)$ möglich ist.

3.3.2 Master File Table

Die MFT enthält Einträge für alle Dateien und Unterverzeichnisse des Dateisystems. Die ersten 16 Einträge sind allerdings für besondere Systemdateien (→ *3.3.3*) reserviert:

	Name	Systemdatei	Zweck
0	**$Mft**	MFT	Enthält die Master File Table
1	**$MftMirr**	Gespiegelte MFT	Sicherheitskopie der Master File Table
2	**$LogFile**	Logdatei	Transaktions-Journal zur Datenrettung
3	**$Volume**	Volume	Enthält u.A. Volume-Label und -Version
4	**$AttrDef**	Attribute	Beschreibung aller definierten Dateiattribute
5	**$**	Hauptverzeichnis	Enthält den Dateindex des Hauptverzeichnisses
6	**$Bitmap**	Belegte Cluster	Tabelle aller allokierten Cluster
7	**$Boot**	Bootsektoren	Enthält die 16 Bootsektoren der Partition
8	**$BadClus**	Unbrauchbare Cluster	Tabelle aller physikalisch unlesbaren Cluster
9	**$Secure**	Sicherheitsdatei	Sicherheits-Beschreibungen aller Dateien
10	**$Upcase**	Großbuchstaben	Tabelle mit Unicode-Großbuchstaben
11	**$Expand**	Erweiterungen	
12-15			Reserviert

Abb. 3.3-2: Einträge für Systemdateien in der MFT **[Mue03]**

Unter NTFS können Dateien aus mehreren Datenströmen, sog. *Forks*, bestehen. Typische Forks enthalten die Attribute der Datei (Dateiname, Größe, Dateizeit) oder den Dateiinhalt. Alle Forks sind Bestandteil der MFT. Passt ein Fork allerdings nicht in den MFT-Eintrag, was bei größeren Dateien zum Beispiel auf den Inhalt zutrifft, so wird der entsprechende Fork durch eine Reihe von Clustern aus dem Datenbereich indiziert. Damit ersetzt die MFT zusammen mit den Dateien **$Bitmap** und **$BadClus** die klassische FAT:

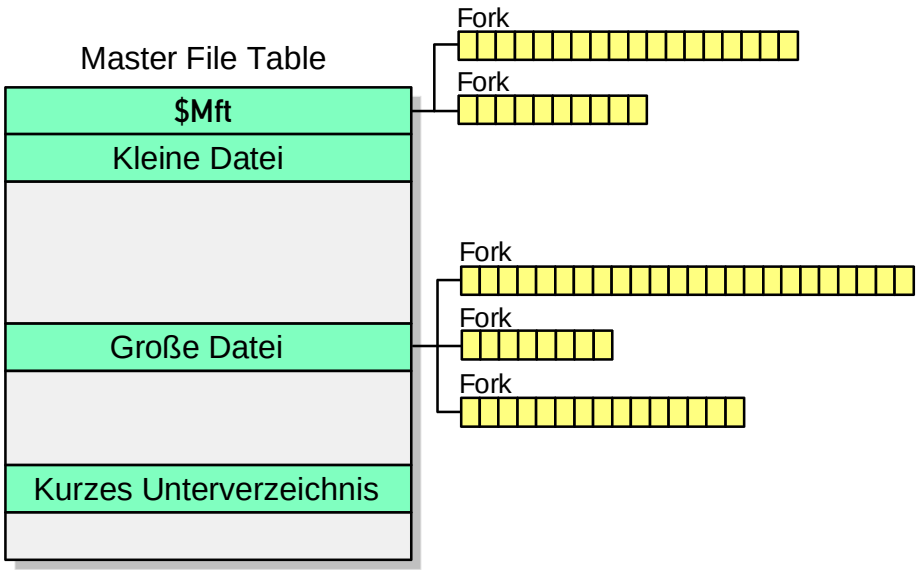


Abb. 3.3-3: Forks

3.3.3 Systemdateien

Direkt an die MFT schließt sich ein reservierter Bereich für Systemdateien wie **$LogFile** (4 MB), **$Bitmap** oder **$BadClus** an. Erst danach befindet sich der Datenbereich, dessen Cluster beliebigen Dateien zugewiesen werden können.

3.3.4 Datenbereich

Der Datenbereich ist in gleich große Cluster unterteilt und enthält alle Dateiforks, die nicht in der MFT Platz finden. Windows NT verwendet standardmäßig folgende Clustergrößen:

	Clustergröße
Bis 512 MB	1 Sektor, 512 Byte
513 bis 1024 MB	2 Sektoren, 1024 Byte
1025 bis 2048 MB	4 Sektoren, 2048 Byte
Ab 2049 MB	8 Sektoren, 4096 Byte

Abb. 3.3-4: Default-Clustergröße bei NTFS **[Mue03]**

3.4 ext2

Das ext2-Dateisystem ist eines der Standard-Dateisysteme von Linux und wurde ursprünglich 1993 entwickelt. Es existieren auch Implementierungen für NetBSD, FreeBSD, OpenBSD, Microsoft Windows, OS/2 und RISC OS **[Wik05a]**.

ext2 teilt viele seiner Eigenschaften mit traditionellen Unix-Dateisystemen, wie z.B. das Konzept der *I-Nodes* (→ *3.4.3*). Wenn gewünscht, ist es um Fähigkeiten wie Zugriffskontrolle (*ACLs*), *Fragmente*, Wiederherstellung gelöschter Daten und Kompression erweiterbar. Die meisten der genannten Funktionen sind nicht standardmäßig implementiert, sondern existieren nur als Patches. Weiterhin gibt es einen Versionsmechanismus, der es erlaubt, neue Funktionen abwärtskompatibel hinzuzufügen, wie das bei der Journaling-Erweiterung ext3 geschehen ist **[Wik05a]**.

3.4.1 Clustergruppen

Der Datenbereich eines ext2-Dateisystems wird in Cluster aufgeteilt. Diese haben eine feste Größe von 1 KB, 2 KB oder 4 KB; die Clustergröße wird bei der Erstellung des Dateisystems festgelegt. Die Cluster werden in Clustergruppen zusammengefasst, um die Fragmentierung zu reduzieren und den Zugriff auf große Mengen aufeinander folgender Daten zu beschleunigen, indem die nötigen Kopfbewegungen der Festplatte minimiert werden. Jede Clustergruppe beginnt mit einer Kopie des Bootsektors (→ *3.4.2*). Dahinter befindet sich eine Deskriptortabelle, die Informationen über jede

Clustergruppe enthält. Zwei Cluster in der Nähe des Gruppenstarts sind für zwei Bitmaps reserviert, die die Belegung der Cluster und I-Nodes anzeigen. Die auf die Bitmaps folgenden Cluster fungieren als I-Node-Tabelle für die Gruppe; die übrigen sind als Datenblöcke nutzbar **[Wik05a]**:

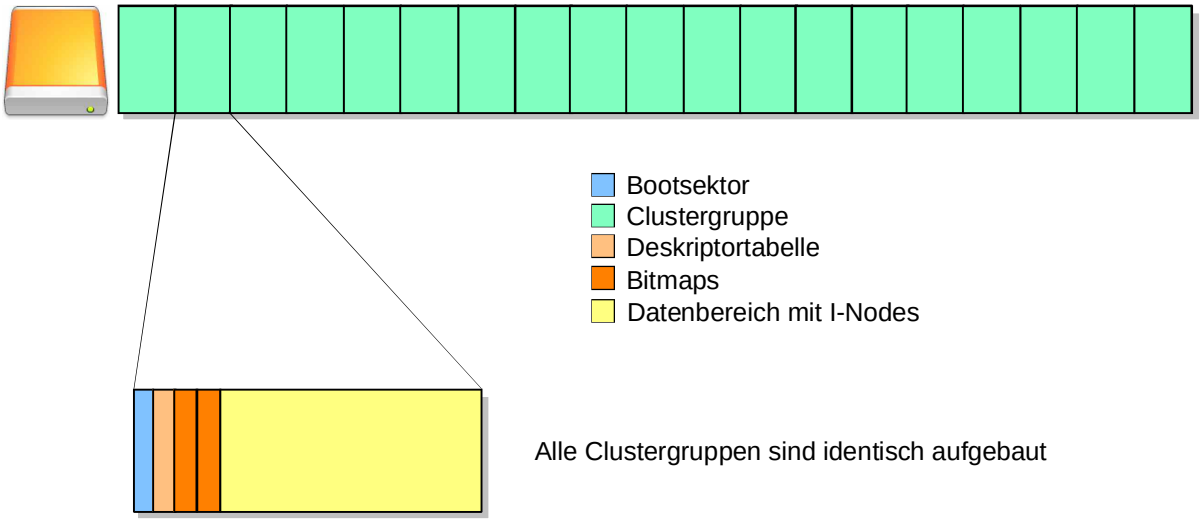


Abb. 3.4-1: Layout des ext2-Dateisystems

3.4.2 Bootsektor

Der Bootsektor enthält wie bei den anderen Dateisystemen alle Informationen über die Konfiguration, wie z.B. die Anzahl der Blöcke und I-Nodes im Dateisystem, welche davon unbenutzt und wie viele I-Nodes und Cluster in jeder Clustergruppe vorhanden sind **[Wik05a]**.

3.4.3 I-Nodes

I-Nodes sind ein fundamentales Konzept des ext2-Dateisystems; jedes Objekt im Dateisystem wird durch einen I-Node repräsentiert. Ein I-Node enthält Zeiger auf die Cluster, in denen die Daten des Objekts abgelegt sind, und außerdem alle Attribute mit Ausnahme seines Namens. Dazu gehören Zugriffsrechte, Besitzer, Gruppe, Flags, Größe, die Anzahl der benutzten Blöcke, Zugriffszeitpunkt, Änderungszeitpunkt, Löschzeitpunkt, Anzahl der Links, Fragmente, erweiterte Attribute und eventuelle ACLs **[Wik05a]**.

Es gibt einige ungenutzte und überladene Variablen in einem I-Node: ein Bereich ist für die Verzeichnis-ACL reserviert, wenn der I-Node ein Verzeichnis repräsentiert, andernfalls enthält dieses Feld die oberen 32 Bit der Dateigröße, wenn der I-Node zu einer regulären Datei gehört. Dadurch sind Dateien möglich, die größer als 2 GB sind **[Wik05a]**.

Es gibt im I-Node Zeiger auf die ersten 12 Blöcke, welche direkt die Daten der Datei enthalten. Außerdem gibt es einen Zeiger auf einen indirekten Block (der wiederum Zeiger auf den nächsten Satz von direkten Blöcken der Datei enthält) und einen Zeiger auf einen doppelt indirekten Block (der Zeiger auf weitere indirekte Blöcke enthält) **[Wik05a]**:

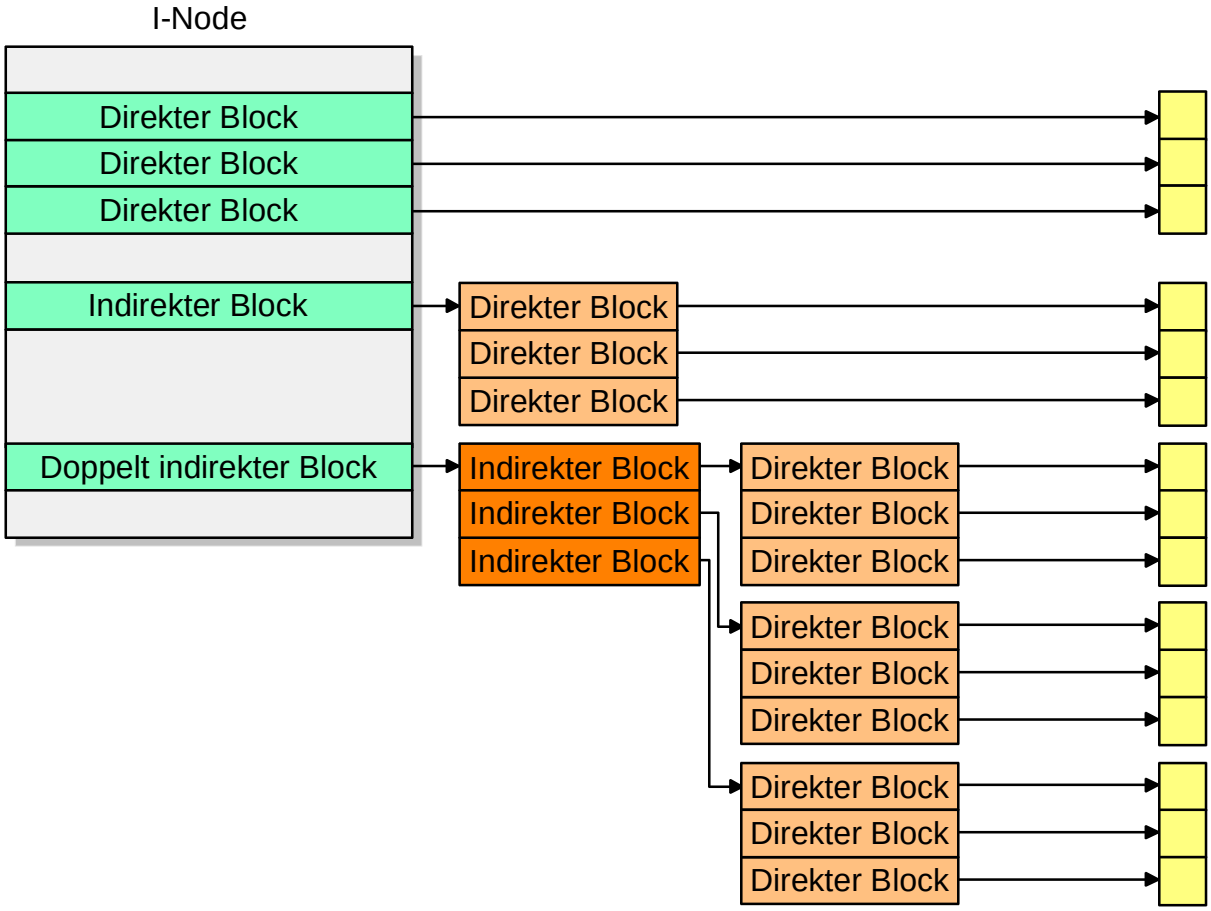


Abb. 3.4-2: Blockadressierung

Das **Flags**-Feld enthält einige ext2-spezifische Flags, die nicht durch **CHMOD** beeinflusst werden können; sie erlauben einer Datei besonderes Verhalten. So gibt es Flags für sicheres Löschen, Unlöschbarkeit, Kompression, synchrone Updates, Schreibschutz, indizierte Verzeichnisse, Journaling (→ *3.2.5.1*) und Einiges mehr. Nicht alle Flags lassen sich derzeit sinnvoll verwenden **[Wik05a]**.

3.4.4 Kompatibilität

Ext2 verfügt über einen ausgreiften Kompatibilitätsmechanismus, der es erlaubt, Dateisysteme mit einem Kernel zu verwenden, dessen **ext2fs**-Treiber von einigen verwendeten Funktionen nichts weiß. Es gibt dabei drei Felder von je 32 Bit Länge: eines für kompatible Eigenschaften (**COMPAT**), eines für nur lesekompatible Features (**RO_COMPAT**) und eines für inkompatible Eigenschaften (**INCOMPAT**) **[Wik05a]**.

Ein COMPAT-Flag bedeutet, dass das Dateisystem eine Eigenschaft enthält, aber das Datenformat auf dem Datenträger 100% kompatibel zu älteren Formaten ist, so dass ein Kernel, der diese Funktion nicht kennt, im Dateisystem lesen und schreiben könnte, ohne es inkonsistent zu machen. Bestes Beispiel für ein COMPAT-Flag ist die Funktion HAS_JOURNAL eines ext3-Dateisystems: ein Kernel ohne ext3-Unterstützung kann ein solches Dateisystem problemlos als ext2 mounten und dann unter Umgehung des Journals darauf schreiben, ohne etwas zu beschädigen **[Wik05a]**.

Ein RO_COMPAT-Flag zeigt an, dass das Datenformat des Dateisystems beim Lesen 100% kompatibel zu älteren Formaten ist. Ein Kernel ohne Kenntnis der in Frage stehenden Funktion könnte jedoch das Dateisystem korrumpieren, wenn er darauf schreibt, daher wird das Mounten einer solchen Partition verweigert. Ein Beispiel für eine lesekompatible Eigenschaft ist SPARSE_SUPER, ein Dateisystemlayout, bei dem weniger Kopien des Bootsektors als normal üblich auf dem Datenträger abgelegt werden. Ein alter Kernel kann problemlos von einer solchen Festplatte lesen, wenn er jedoch einen Schreibversuch unternehmen würde, würden seine Schreibroutinen irreführende Fehlermeldungen ausgeben **[Wik05a]**.

Ein INCOMPAT-Flag zeigt an, dass sich das Datenformat so geändert hat, dass Kernel ohne diese Eigenschaft das Dateisystem noch nicht einmal lesen oder auch nur mounten könnten. Als Beispiel für eine solche inkompatible Zusatzfunktion kann die optionale Kompression dienen: ein Kernel, der die Daten nicht dekomprimiert, würde nur unsinnige Bytes lesen. Auch ein inkonsistentes ext3-Dateisystem ist so lange inkompatibel, bis ein ext3-fähiger Kernel das Journal abgespielt und so die Inkonsistenzen beseitigt hat. Danach kann das ext3-System auch wieder als ext2 gemountet werden **[Wik05a]**.

Das Hinzufügen neuer Eigenschaften zum ext2-Dateisystem erfordert auch immer eine Aktualisierung der zugehörigen Prüfungswerkzeuge, da diese alle Dateisystemeigenschaften kennen müssen, um eine zuverlässige Feststellung und Behebung von Inkonsistenzen zu ermöglichen **[Wik05a]**.

3.4.5 Performanz des ext2-Dateisystems

Das ext2-Dateisystem ist aufgrund der eingesetzten Datenstrukturen auf physikalischer Ebene sehr performant und bildet damit den Gegenpunkt zum FAT-Dateisystem (→ *3.2*). Verzeichniseinträge werden als B^+-Baum (→ *2.4.1.3*) gespeichert, so dass ein Eintrag in $O(\log n)$ gefunden werden kann. Der I-Node (→ *3.4.3*), der zu jeder Datei gehört, gestattet einen Seek in $O(1)$, da nach spätestens drei Zugriffen die Position des Clusters feststeht.

4 Metadaten in Multimedia-Dateiformaten

Metadaten sind Informationen, die nicht zu den eigentlichen Nutzdaten gehören, sondern diese beschreiben. Metadaten können implizit sein oder explizit abgespeichert werden. Klassische explizite Metadaten sind der Dateiname und das Datum der letzten Änderung einer Datei. Explizit gespeicherte Metadaten sind für die Benutzung einer Datei nicht zwingend erforderlich: eine Bilddatei kann beispielsweise auch ohne Änderungsdatum angezeigt werden, und sogar auf einen Dateinamen kann theoretisch verzichtet werden. Implizite Metadaten sind im Gegensatz dazu für den Betrieb notwendig. Die meisten Bildformate enthalten beispielsweise die Größe des Bildes, also Länge und Höhe in Pixeln, und Audiodateien müssen die Samplerate und Bittiefe des Klangs abspeichern. Fehlen diese Informationen, kann die Datei nicht benutzt werden.

Diverse Multimedia-Dateiformate enthalten heute weitreichende Metadaten, wie zum Beispiel Titel, Copyright oder das verwendete Aufnahmegerät; die Spezifikationen für Metadaten in MP3- (→ *4.1*), JPG- (→ *4.2*) und AVI-Dateien (→ *4.3*) werden im Folgenden vorgestellt.

4.1 ID3

ID3 ist ein Standard zum Speichern von Metadaten innerhalb einer MP3-Datei. ID3 ist die Abkürzung für »Identify MP3«. Eine MP3-Datei besteht aus einer langen Folge von Frames, die neben Formatangaben die eigentlichen Audiodaten enthalten:

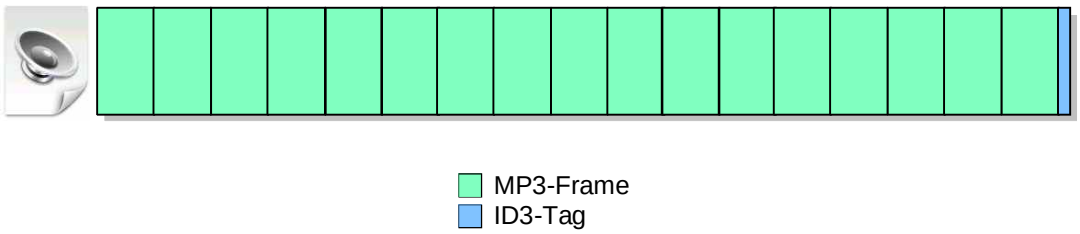


Abb. 4.1-1: Aufbau einer MP3-Datei mit ID3-Tag **[ID305]**

4.1.1 Aufbau eines MP3-Frames

Eine MP3-Datei besitzt keinen Header, vielmehr werden Informationen wie die verwendete Samplerate am Anfang eines jeden Frames gespeichert. So ist es möglich, einen fortwährenden Datenstrom zu übertragen; die Abspielsoftware muss nur den Anfang eines Frames suchen und findet dort alle benötigten Angaben. Der Frameheader ist 4 Byte lang und hat folgenden Aufbau:

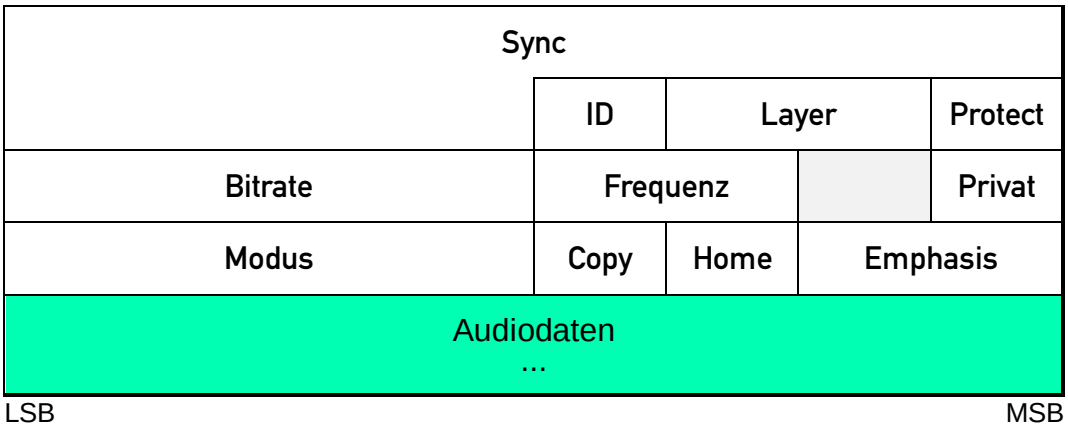


Abb. 4.1-2: Format eines MP3-Frames **[ID305]**

Besonders wichtig sind die 12 **Sync**-Bits am Anfang: sie sind alle **1**. An keiner anderen Stelle innerhalb einer MP3-Datei kommen hintereinander 12 gesetzte Bits vor, so dass auf diese Weise der Anfang eines Frames identifiziert werden kann und sich eine Abspielsoftware mit dem Datenstrom synchronisiert.

4.1.2 ID3, Version 1.0

Das Fehlen eines Headers am Anfang von MP3-Dateien lässt es zunächst problematisch erscheinen, Zusatzinformationen unterzubringen. Da aber jedes MP3-Frame eine durch die Samplefrequenz und Bitrate festgelegte Länge hat und jedes Frame durch 12 gesetzte Bits eingeleitet wird, lassen sich hinter einem Frame beliebige Informationen unterbringen. Das Byte **FFh**, mit dem die 12 Synchronisations-Bits eingeleitet werden, darf allerdings niemals innerhalb der Metadaten vorkommen, da ein Decoder sonst irrtümlich den Beginn eines neuen Frames erkennen könnte.

ID3 besteht in der Version 1.0 aus einer 128 Byte großen Datenstruktur, die zum einfachen Auffinden immer am Ende einer MP3-Datei abgespeichert wird:

```
type
  ID3Tag=record
    IDCode: array[1..3] of Char;
    Titel,Kuenstler,Album: array[1..30] of Char;
    Jahr: array[1..4] of Char;
    Kommentar: array[1..30] of Char;
    Genre: Byte;
end;
```

IDCode besteht aus der Zeichenfolge '**TAG**'; findet ein Decoder diese Zeichen 128 Byte vor Dateiende, werden die folgenden 125 Byte als gültiger ID3-Tag interpretiert. Die Attribute **Titel**, **Kuenstler**, **Album** und **Kommentar** enthalten bis zu 30 Zeichen langen Text, der ggf. mit **#0** aufgefüllt wird; **Jahr** speichert das Erscheinungsjahr im Klartext. **Genre** enthält eine Nummer, die das Musikgenre wie Klassik oder Rock codiert **[ID305]**. Diese Angabe wird aber nur selten genutzt, da einige Codes doppelt belegt sind oder manche Programme nicht alle möglichen Angaben zuordnen.

4.1.3 ID3, Version 1.1

Viele MP3-Dateien werden durch Kompression von Audio-CDs erzeugt; daher entstand der Wunsch, die Nummer des Tracks auf der Ursprungs-CD mit abzuspeichern. ID3 trägt dem in der Version 1.1 Rechnung, indem das Kommentar-Feld um 1 Byte verkürzt und so Platz für die Tracknummer geschaffen wird:

```
type
  ID3Tag=record
    Titel,Kuenstler,Album: array[1..30] of Char;
    Jahr: array[1..4] of Char;
    Kommentar: array[1..29] of Char;
    Track,Genre: Byte;
end;
```

Track ist nur gültig, wenn **Kommentar[29]=#0** ist, andernfalls wird das Byte als letztes Zeichen des Kommentarstrings interpretiert **[ID305]**.

4.2 Exif

Schon früh entstand der Wunsch, bei Fotos diverse Metadaten zu archivieren, wie dieses Formular eines Fotografen aus den 1960er-Jahren zeigt:

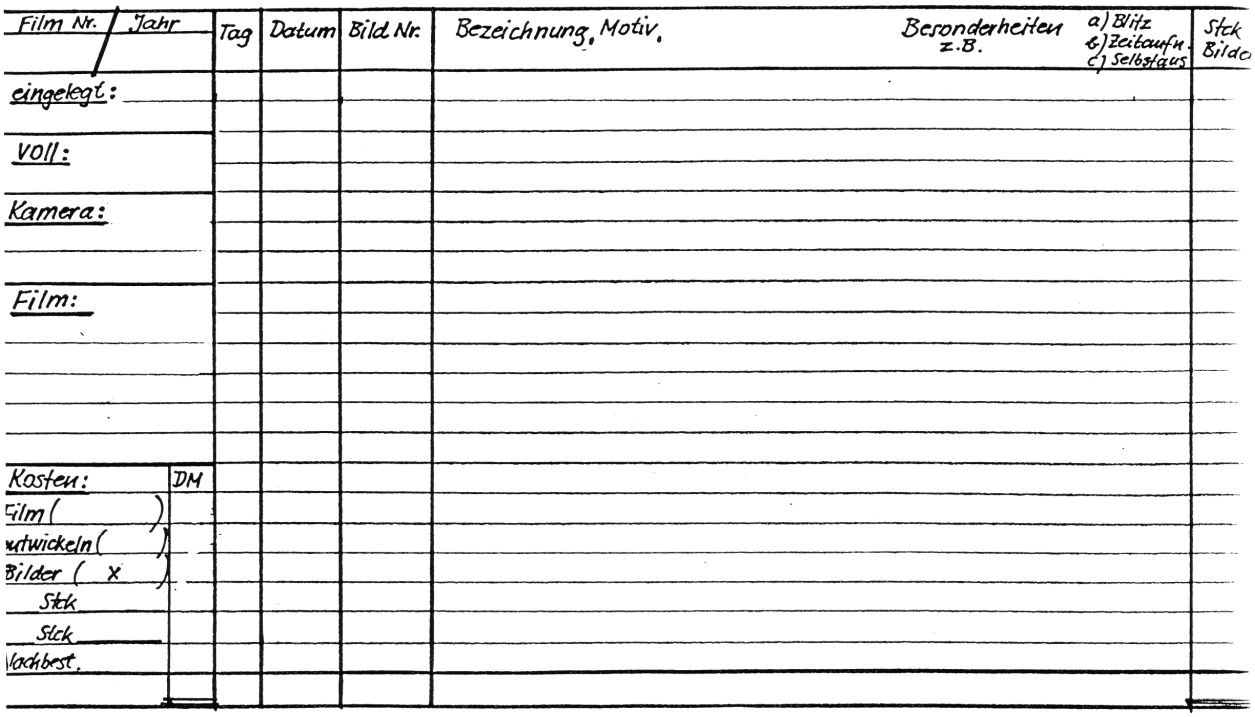


Abb. 4.2-1: Metadaten bei Fotos

Fast alle Digitalkameras speichern Bilder als JPEG-Datei ab, die um Metadaten erweitert sind. Exif steht für »Exchangable Image File« und ist ein Standard für Zusatzinformationen in Bilddateien **[EXI05]**, der von der Japan Electronics and Information Technlogy Industries Association **[JEI05]** entwickelt wurde.

4.2.1 JPEG File Interchange Format (JFIF)

Das JPEG-Format verwendet die Diskrete Cosinus-Transformation zur Kompression von Bilddaten **[Kol03]**. Neben dem eigentlichen JPEG-Datenstrom müssen allerdings auch weitere Daten wie Bildauflösung und Kompressionstabellen mit abgespeichert werden. Daher wurde das JFIF-Format definiert **[JPE92]**, das alle Informationen in einer einzigen Datei zusammenfasst. Ein neuer Teil der Datei wird mit einem Marker eingeleitet, der aus dem Byte **FFh** besteht, gefolgt von einem ID-Code und einem Word, das die Länge der Daten im Big-Endian-Format angibt:

```
type
  Marker=record
    Start,IDCode: Byte;
    Size: Word;
end;
```

Marker mit einem **IDCode** kleiner 192 sind ungültig, ab **C0h** sind unter anderem folgende Marker definiert worden:

	Bedeutung
C0h	Codierung als »Baseline-Sequential«, enthält Formatangaben (Länge, Höhe, ...)
C2h	Codierung als »Progressive JPEG«, enthält Formatangaben (Länge, Höhe, ...)
C4h	Huffman-Tabelle
D8h	»Start of image« (**SOI**)
D9h	»End of image« (**EOI**)
DAh	»Start of scan« (**SOS**)
DBh	Quantisierungs-Tabelle
E0h-EFh	Applikations-Marker **APP0** bis **APP15** (**APP0**: JFIF-Signatur; **APP1**: Exif; **APP7**: Siemens Thumbnail; **APP14**: Adobe)
FEh	JPEG-Kommentar

Abb. 4.2-2: wichtige JFIF-Marker **[Wik05e]**

Vor allem die Marker **APP0** bis **APP15** enthalten interessante Informationen. Diverse Programme, wie z.B. Photostudio für Windows **[Haw03]**, können diese Daten ausführlich darstellen:

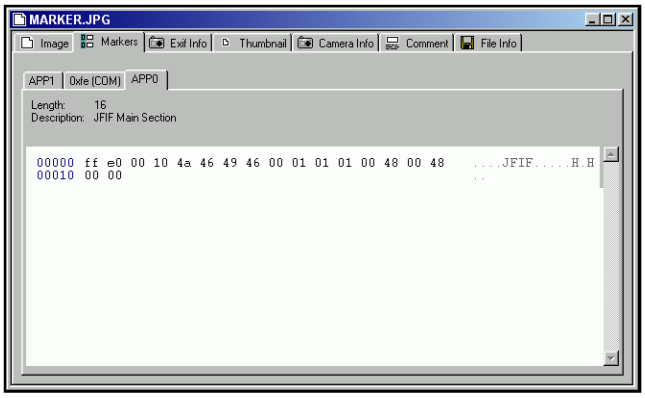


*Abb. 4.2-3: Hexdump von **APP0***

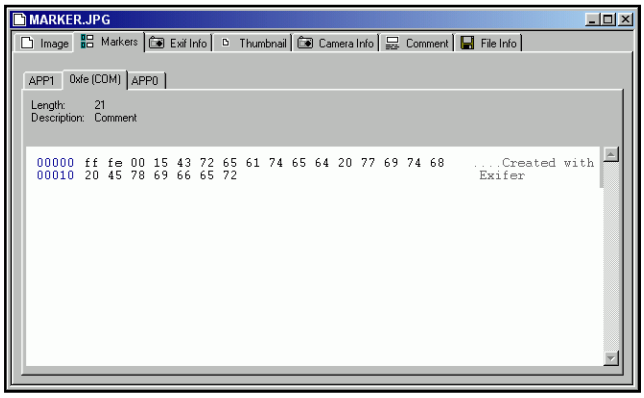


Abb. 4.2-4: Hexdump des JPEG-Kommentars

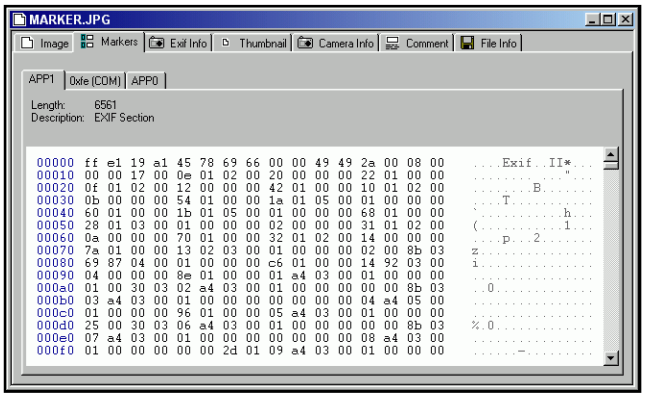


*Abb. 4.2-5: Hexdump von **APP1***

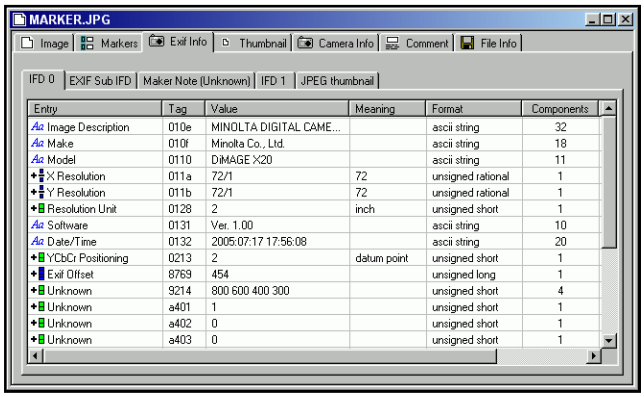


*Abb. 4.2-6: Exif-Daten in **APP1***

APP0 enthält den 5 Byte langen nullterminierten String '**JFIF**'+**#0**, daran schließen sich Angaben wie die JFIF-Version, die Bilddichte in DPI oder ein Vorschaubild an **[JPE92]**. **APP1** enthält die Exif-Daten und wird im folgenden Abschnitt (→ *4.2.2*) näher vorgestellt. **APP7** ist in JPG-Dateien enthalten, die mit Siemens-Mobiltelefonen aufgenommen wurden; sie werden in Abschnitt → *4.2.3* betrachtet. Ebenfalls häufig anzutreffen ist der **APP14**-Marker; er zeigt an, dass die Datei mit Adobe Photoshop erzeugt wurde und enthält diverse Programmeinstellungen.

Für einige Anwendungsfälle sind bestimmte Marker unerwünscht. So sind zum Beispiel bei der Verwendung von kleinen JPEG-Bildern für das Layout einer WWW-Seite eingebettete Vorschaubilder störend, da sie die Dateigröße mehr als verdoppeln können. Durch das Löschen unerwünschter Marker konnte ein JPEG-Bild mit 60×69 Pixeln von 18 KB auf 4 KB verkürzt werden, da die Datei mit Adobe Photoshop erzeugt wurde, das zusätzlich zum Vorschaubild noch alle Einstellungen der Arbeitsoberfläche ins Bild eingebettet hat. Der Quellcode eines Übersetzungsmoduls, das diese Aufgabe übernimmt, ist zusammen mit einigen Erläuterungen in → *D* zu finden.

4.2.2 Exif-Marker (APP1)

Die Exif-Informationen werden vom **APP1**-Marker eingeleitet. Daran schließt sich eine Datenstruktur an, die sich am TIFF-Bildformat orientiert; ein Verzeichnis enthält Zeiger auf die eigentlichen Daten und nachfolgende Verzeichnisse:

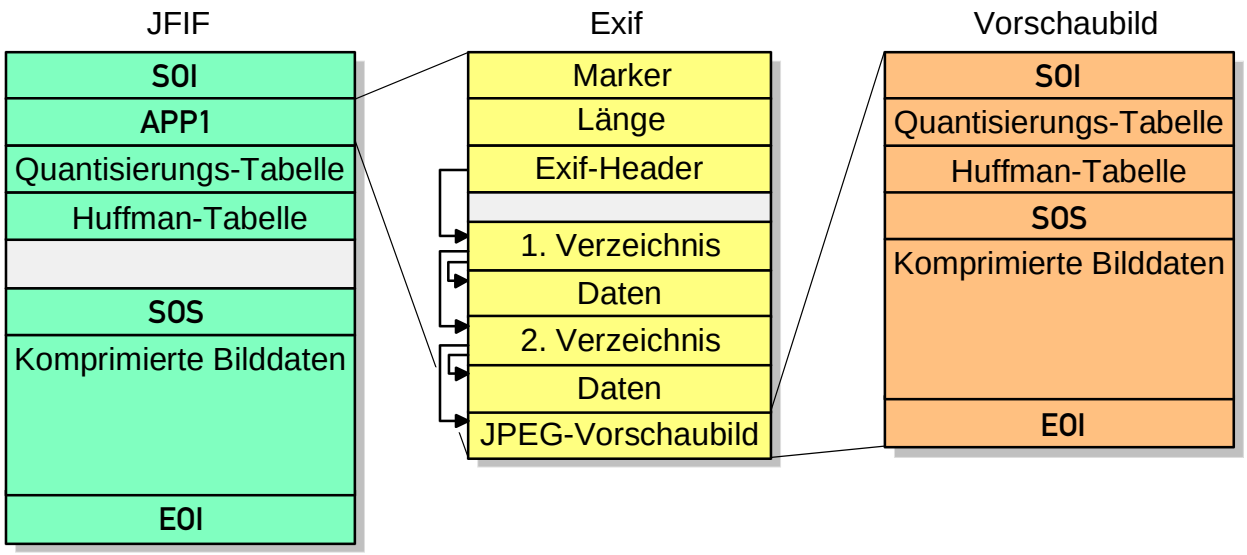


Abb. 4.2-6: Aufbau einer JPG-Datei mit Exif-Marker **[JEI02]**

Der Exif-Header weist folgende Struktur auf:

```
type
  ExifHeader=record
    ID: array[1..6] of Char;
    Order: array[1..2] of Char;
    TagMark: Word;
    IFDOffset: LongInt;
end;
```

Das Feld **ID** kennzeichnet einen gültigen Exif-Header, wenn es den String '**Exif**'+#0#0 enthält. Alle folgenden Zahlenangaben können entweder als Big- oder Little-Endian abgespeichert worden sein. Dies wird durch das Feld **Order** angezeigt: der String '**II**' steht für Little Endian (Intel-Prozessoren), '**MM**' für Big Endian (Motorola-CPUs). **IFDOffset** enthält die Position des ersten Verzeichnisses, gemessen in Byte vom Anfang des Exif-Headers.

Ein Datenverzeichnis gruppiert zusammengehörige Elemente und hat einen einfachen Aufbau; ein Word enthält die Anzahl der Einträge, daran schließen sich entsprechend viele Exif-Tags an:

```
type
  ExifTag=record
    TagID,Typ: Word;
    Count,Offset: LongInt;
end;
```

Jedes Element kann durch die Variable **TagID** identifiziert werden, außerdem hat jeder Eintrag einen festen Datentyp, der in **Typ** definiert wird. Folgende Elemente sind in den meisten JPG-Dateien mit Exif enthalten; eine vollständige Liste findet sich in **[JEI02]**:

	Datentyp
1	Byte
2	Char – die Zeichenkette wird mit #0 beendet
3	Word
4	LongInt
5	Bruch, der aus 2 LongInt gebildet wird

Abb. 4.2-7: häufige Exif-Datentypen **[JEI02]**

	Inhalt	Datentyp
270	Bildtitel	Char
271	Gerätehersteller	Char
272	Gerätemodell	Char
305	Firmware	Char
306 36867	Aufnahmezeit	Char
33432	Copyright	Char

	Inhalt	Datentyp
33434	Belichtungszeit	Bruch
33437	Blende	Bruch
34665	Nächstes Verzeichnis	
34855	ISO	Word
37385	Blitz	Word
37386	Focus	Bruch
41495	Focusmessung	Word

Abb. 4.2-8: wichtige Exif-Elemente **[JEI02]**

Manche Exif-Elemente weisen eine besondere Formatierung auf: so hat zwar die Aufnahmezeit den Datentyp **Char**, ist also formal ein Freitextfeld; dennoch hat der dort abgespeicherte String immer das Format '**YYYY:MM:DD HH:MM:SS**'. Beim »Blitz«-Feld gibt das niederwertigste Bit an, ob der Blitz ausgelöst wurde; die anderen Bits speichern den verwendeten Blitz-Modus wie »Red-Eye-Reduction«. Eine vollständige Auflistung aller Tags findet sich in **[JEI02]**.

Ein besonders mächtiges, aber zur Zeit noch wenig verbreitetes Exif-Tag ist in der Lage, GPS-Koordinaten aufzunehmen. Hochwertige Kameras mit einem GPS-Empfänger speichern so die Aufnahmeposition innerhalb der JPEG-Datei ab:

Abb. 4.2-9: Kamera mit GPS-Empfänger (Produktfoto Sony)

Danach können die Fotos mit einer geeigneten Software, die auf die gespeicherten GPS-Koordinaten zugreifen kann, innerhalb einer Karte an ihren Aufnahmeorten dargestellt werden:

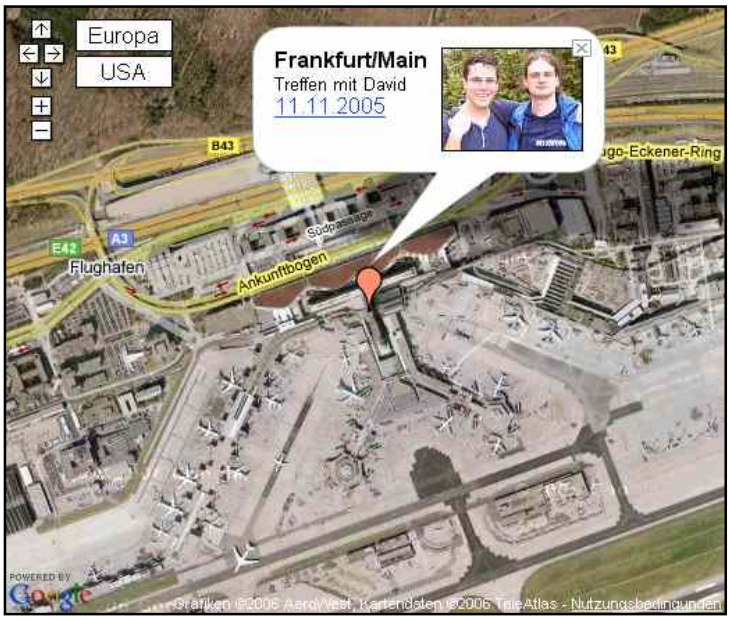


Abb. 4.2-10: Bild mit Geodaten, dargestellt von Google Maps **[DES06]**

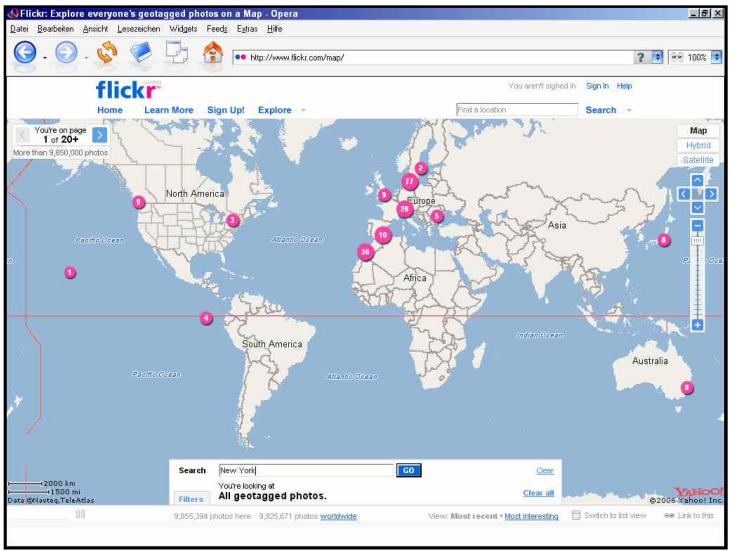


Abb. 4.2-11: mit Geodaten versehene Fotoalben beim Bilderdienst »Flickr« **[Fli07]**

4.2.3 Siemens S65 Mobiltelefon

Das Siemens S65 ist ein modernes Multimedia-Telefon, das über eine Kamera und ein leistungsstarkes Dateisystem verfügt, um aufgenommene Fotos als JPEG-Bild abzuspeichern. Das S65 ist bezüglich der Software und des Dateisystems identisch mit dem Siemens CX65 (modische Version mit Blinklichtern), dem M65 (Outdoor-Version) und dem SP65 (Business-Handy ohne Kamera).

Das Siemens S65 legt aufgenommene Bilder wie die meisten anderen Kamerahandys auch als JFIF-kompatible (→ *4.2.1*) **JPG**-Datei ab. Die Dateien des Siemens S65 werden um den **APP7**-Marker (→ *4.2.1*) erweitert, der von der Zeichenkette '**<SIEMENS THUMBNAIL>**' eingeleitet wird und ein Vorschaubild enthält:

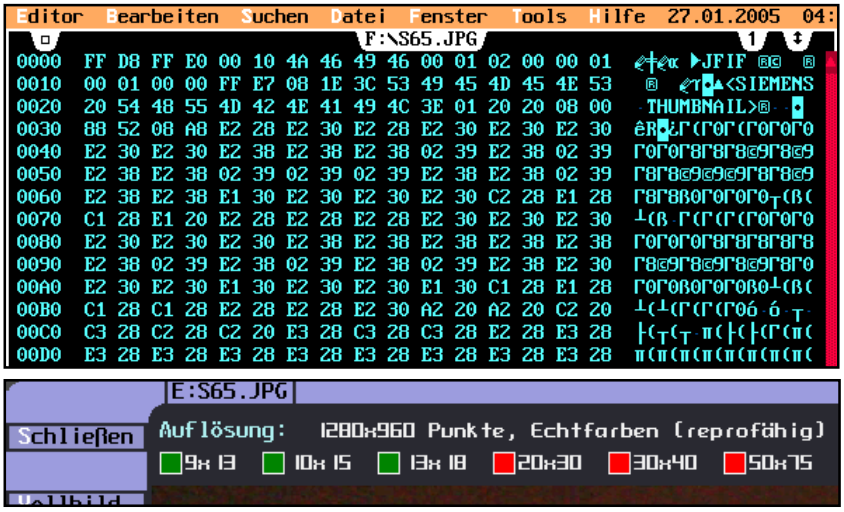


Abb. 4.2-11: JPEG-Bild vom Siemens S65 (Original)

Da Vorschaubilder die Dateigröße aufblähen (→ *4.2.1*), kann der Übersetzer in → *D* sie entfernen. Der String '**<SIEMENS THUMBNAIL>**' im **APP7**-Marker beweist jedoch, dass das Bild von einem Siemens Mobiltelefon aufgenommen wurde. Aus diesem Grund fügt der Übersetzer mit der Prozedur **CreateExif** einen normgerechten Exif-Marker (→ *4.2.2*) ein, der '**Siemens**' als Hersteller und '**Mobile Phone**' als Gerät enthält:

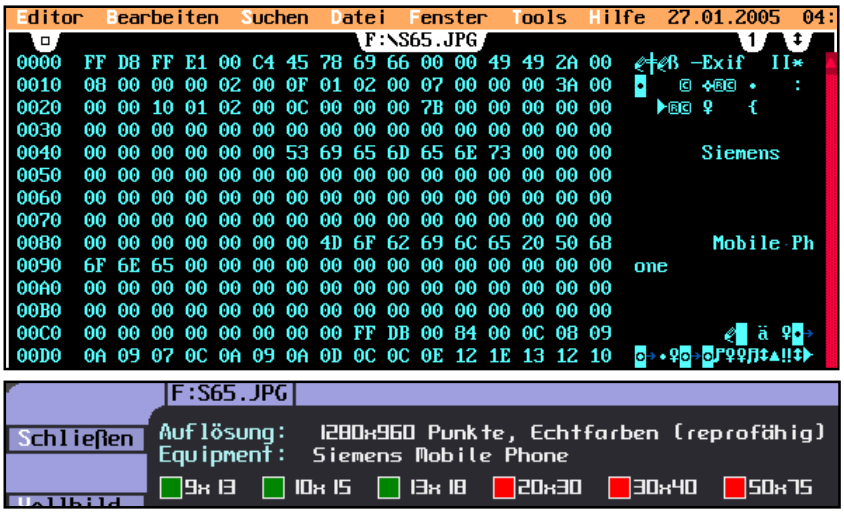


Abb. 4.2-12: JPEG-Bild vom Siemens S65 (übersetzt)

Bei Mobiltelefonen von Nokia (und damit auch beim Siemens SX1, das von Nokia entwickelt wurde) stellt sich die Situation etwas komplizierter dar. Die von diesen Geräten erzeugten Bilddateien enthalten weder ein Vorschaubild noch einen Exif-Marker; statt dessen findet man Metadaten inner-

halb des JPEG-Kommentars (→ *4.2.1*) vor. Der Kommentar-String enthält den Namen des Herstellers, gefolgt von der Gerätebezeichnung, dem Aufnahmedatum und der Aufnahmezeit. Die einzelnen Angaben innerhalb der Zeichenkette werden durch einen Zeilenumbruch (**#$0A**) voneinander getrennt:

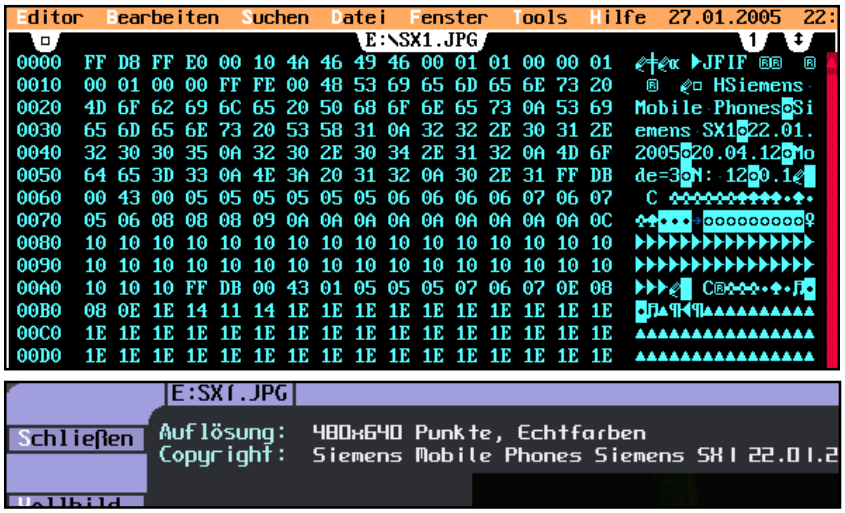


Abb. 4.2-13: JPEG-Bild vom Siemens SX1 (Original)

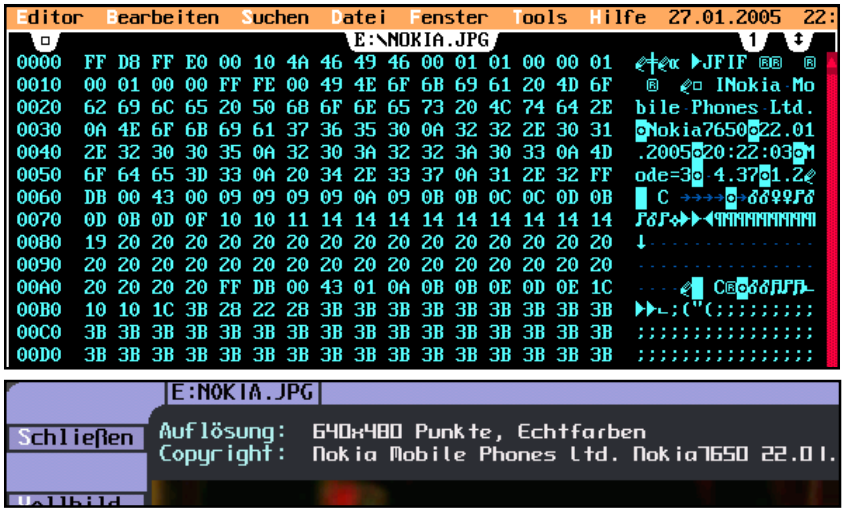


Abb. 4.2-14: JPEG-Bild vom Nokia 7650 (Original)

Der Übersetzer in → *D* erkennt diese JPEG-Kommentare, wenn sie (zur Sicherheit durch **UppCase**()) in Großbuchstaben umgewandelt) mit der Zeichenkette '**SIEMENS MOBILE PHONES**' bzw. '**NOKIA MOBILE PHONES**' beginnen. In diesem Fall ruft **TranslateJPG** die Prozedur **HandlePhones** auf, die die Elemente des Kommentarstrings trennt. Die einzelnen Metadaten werden dann an die Prozedur **CreateExif** übergeben, die daraus einen normalen Exif-Marker erzeugt und ausgibt:

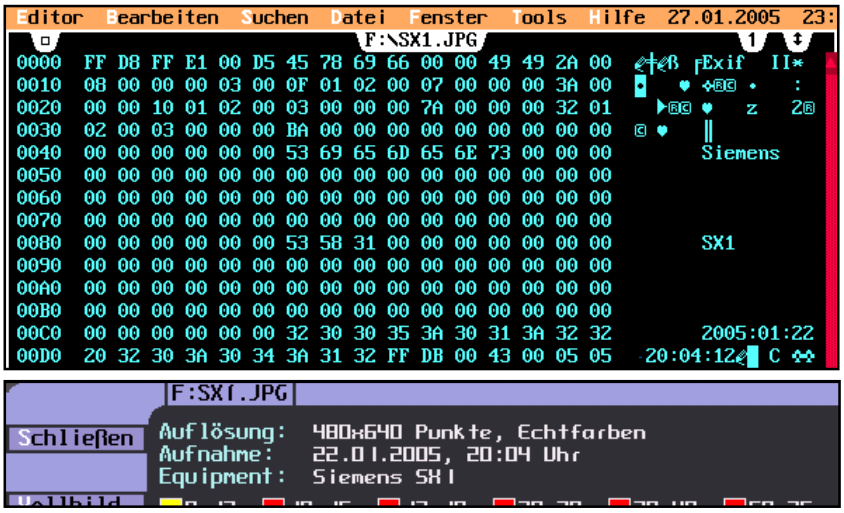


Abb. 4.2-15: JPEG-Bild vom Siemens SX1 (übersetzt)

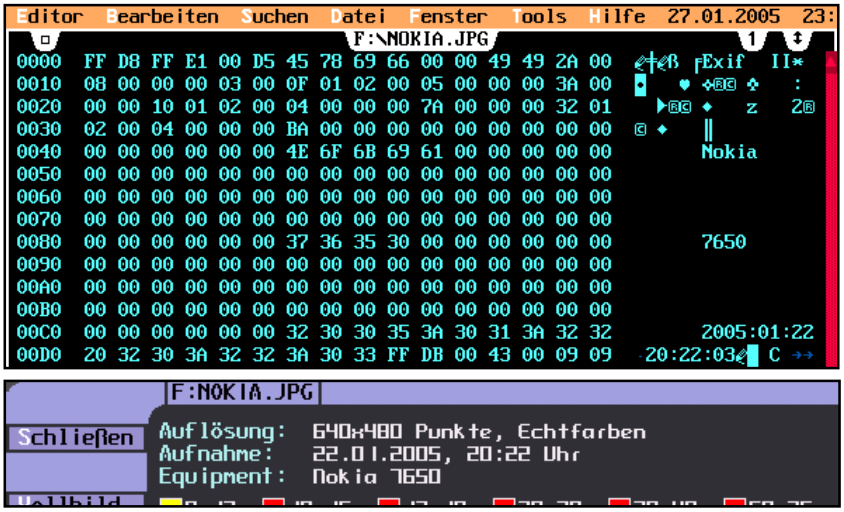


Abb. 4.2-16: JPEG-Bild vom Nokia 7650 (übersetzt)

4.3 Open DML

AVI (Audio Video Interleave) ist ein von Microsoft definiertes Containerformat für Videos. In einer AVI-Datei können mehrere Video- und Audiotracks vorhanden sein; diese werden mit verschiedenen Verfahren kodiert. Das Komprimieren und Entpacken wird dabei von Zusatzmodulen durchgeführt, sog. Codecs (Encoder/Decoder). Codecs können bei Bedarf installiert werden, um neue und bessere Videoformate innerhalb eines AVI-Containers zu unterstützten. Wichtige AVI-Codecs sind Cinepak und DivX **[Kol03]**.

Ein besonderer Nachteil von AVI-Dateien liegt in der Größenbeschränkung auf 2 GB. Als AVI-Dateien mit Windows 3.10 eingeführt wurden, war FAT16 (→ *3.2.1*) das eingesetzte physikalische Dateisystem. Es kann insgesamt nur 2 GB groß werden, so dass die Größenbeschränkung für AVIs hier keine Rolle spielte. Ein Programm, das konform zu Open DML ist, kann mehrere Segmente mit

je zwei Gigabyte Größe innerhalb einer einzigen AVI-Datei korrekt verarbeiten, wenn z.B. NTFS (→ *3.3*) als physikalisches Dateisystem benutzt wird **[DML96]**.

Open DML (Digital Media Library) wurde 1996 von der Open DML File Format Workgroup unter Federführung des Grafikkartenherstellers Matrox entwickelt. Interessanter als die Aufhebung der Größenbeschränkung sind allerdings die in Open DML spezifizierten Metadaten.

4.3.1 Aufbau einer AVI-Datei

Eine AVI-Datei besteht, ähnlich wie JFIF-Dateien (→ *4.2.1*), aus mehreren Segmenten, hier »Chunk« genannt. Ein Chunk kann weitere Unterchunks enthalten und wird von folgender Struktur eingeleitet, an die sich die Nutzdaten anschließen:

```
type
  ChunkHeader=record
    FourCC: array[1..4] of Char;
    Size: LongInt;
end;
```

Das Feld **FourCC** enthält einen 4 Buchstaben großen Code, der den Typ des Chunks beschreibt. Der Chunktyp **LIST** enthält Unterchunks; als Nutzdaten besteht er aus einem weiteren **FourCC**-Feld, das den Typ der Liste angibt, gefolgt von weiteren Chunks **[MSD05b]**.

Jeder Track einer AVI-Datei wird mit 0 beginnend durchnummeriert. Innerhalb des **LIST**-Chunks **movi** befinden sich die einzelnen Videoframes. Jedes Frame besteht aus einem weiteren **LIST**-Chunk vom Typ **rec**. Jeder **rec**-Chunk enthält einen Unterchunk für jeden Track, also meistens für ein Videoframe und für Audiodaten. Die Namen dieser Datenchunks fangen mit 2 Ziffern an, die die Tracknummer kennzeichnen. Daran schließen sich 2 Buchstaben an, die den Typ des Chunks kennzeichenen **[MSD05b]**:

	Inhalt
db	Unkomprimiertes Videobild (Bitmap)
dc	Komprimiertes Videobild
YV, 32, id, iv	Wird von einigen Codecs wie YUV oder Intel Indeo erzeugt; wie **dc**
pc	Palettenwechsel für Codecs mit 256 Farben
wb	Audiodaten

Abb. 4.3-1: Chunks mit Nutzdaten in AVI-Dateien

Daraus ergibt sich folgender Aufbau einer typischen AVI-Datei mit Audio- und Videotrack:

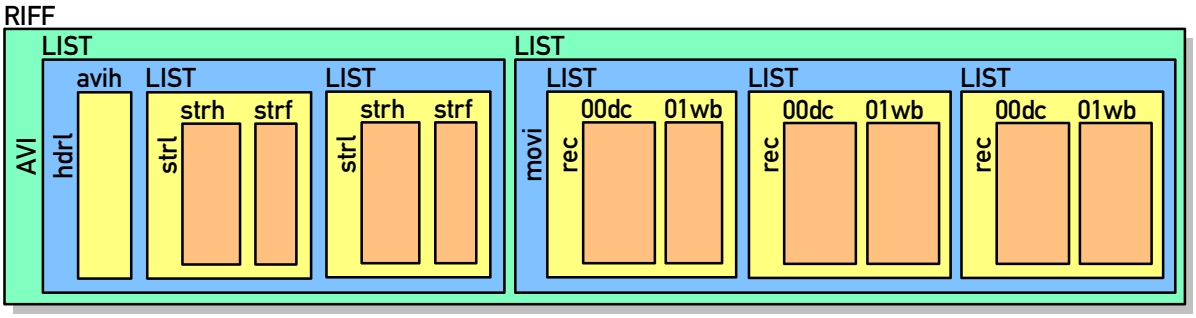


Abb. 4.3-2: schematischer Aufbau einer AVI-Datei **[MSD05b]**

4.3.2 Metadaten

In eine AVI-Datei können leicht Metadaten eingefügt werden; dazu definiert **[DML96]** einen neuen LIST-Typ namens INFO. Die Elemente des INFO-Chunks können mit einem Programm wie abcAVI **[Sor05]** eingesehen und bearbeitet werden:

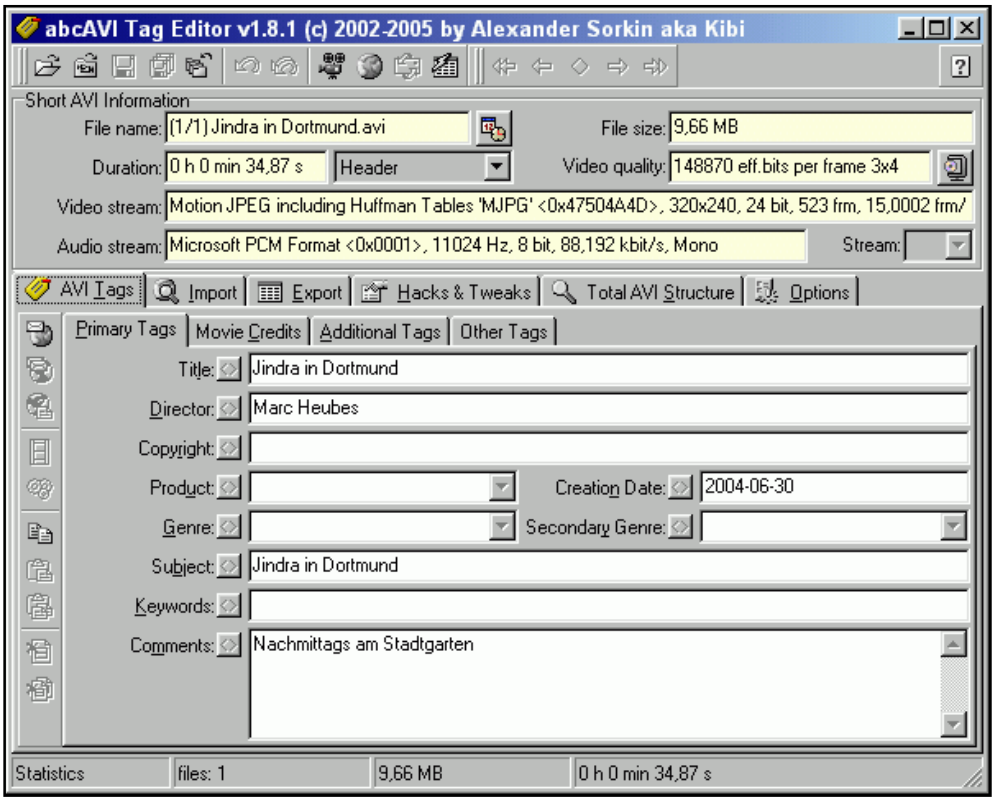


Abb. 4.3-3: abcAVI **[Sor05]**

Alle Unterchunks von INFO enthalten einen nullterminierten String, das FourCC-Feld des Chunks gibt die Bedeutung an. Eine vollständige Aufzählung findet sich in **[DML96]**; folgende Angaben sind besonders wichtig und decken sich teilweise mit bestimmten Exif-Tags in JPEG-Bildern (→ *Abb. 4.2-8*):

	Bedeutung	Entsprechende Exif-Tags aus → *Abb. 4.1-11*
IART	Künstler (Artist)	
ICMT	Kommentar (Comments)	Kommentar (JFIF-Marker **FFFEh**)
ICDR	Aufnahmedatum (Creation date)	306, 36867
ISBJ	Thema bzw. Titel (Subject)	270
ISFT	Software	271, 272, 305

Abb. 4.3-4: wichtige Chunks mit Metadaten in AVI-Dateien **[DML96]**

5 Semantische und virtuelle Dateisysteme

Dieses Kapitel stellt verschiedene Beispiele bereits existierender semantischer und virtueller Dateisysteme vor. Ein physikalisches Dateisystem (→ *3*) behandelt alle Dateien gleich, insbesondere werden sie mit uniformen Attributen wie Dateiname und Dateizeit (→ *3.2.1.3*) ausgestattet. Programme greifen über die Funktionen des Betriebssystems direkt auf alle Dateien und Verzeichnisse zu. Eine Dateiliste, die mit **DIR** bzw. **LS** erzeugt wird, zeigt die tatsächlich vorhandenen Dateien:

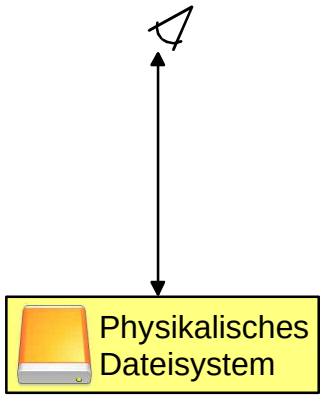


Abb. 5-1: physikalisches Dateisystem

Genau dies ändert sich durch ein semantisches Dateisystem. Dabei handelt es sich um ein Software-Modul, das zwischen physikalischem Dateisystem und Anwendungsschicht angesiedelt ist und die Sicht auf das Dateisystem für einige oder alle Dateien und Verzeichnisse verändert, beispielsweise durch das Einblenden von *virtuellen Dateien* (→ *5.4.2*):

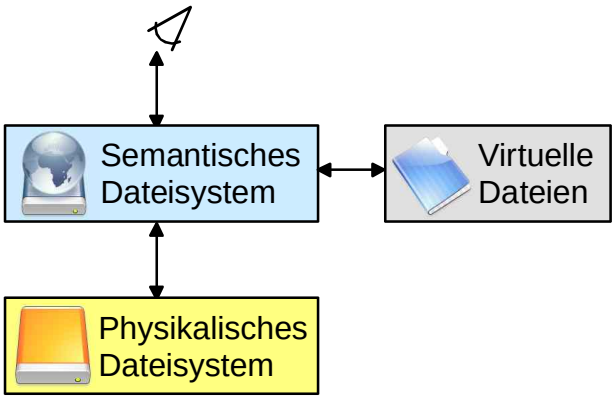


Abb. 5-2: semantisches Dateisystem

Eine weitere interessante Anwendung für semantische Dateisysteme stellt bei bestimmten Dateitypen der Zugriff auf die darin enthaltenen Metadaten dar (→ *4*).

Virtuelle Dateisysteme sind für Spezialanwendungen wie **/proc** (→ *5.5*) gedacht; sie basieren auf keinem physikalischen Dateisystem mehr, sondern nutzen das Datei- und Verzeichniskonzept zur Präsentation andersartiger Daten wie Systeminformationen:

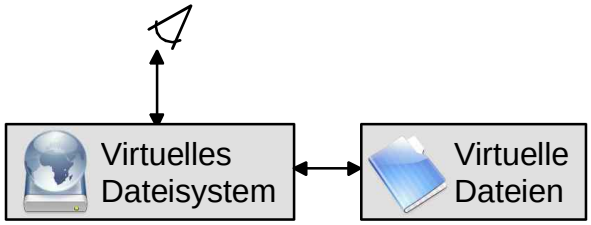


Abb. 5-3: virtuelles Dateisystem

Die folgenden Abschnitte präsentieren diverse Ausprägungen semantischer Dateisysteme, die von einfachen Vorgängen wie dem Zusammenfassen von Papierkorb-Ordnern (→ *5.1*) bis hin zu komplexen Regelwerken für das ganze physikalische Dateisystem (→ *5.2*, → *5.3*, → *5.4*) reichen.

5.1 Zusammengesetzter Papierkorb von Mac OS X

Das Betriebssystem Mac OS X löscht Dateien nicht sofort, sondern verschiebt sie bis zur endgültigen Vernichtung in den sog. »Papierkorb«. Da Mac OS X mehrere Benutzer unterstützt, wurde der Papierkorb als verstecktes Unterverzeichnis .**Trash** des persönlichen Ordners ~ realisiert (~/.**Trash**, links in → *Abb. 5.1-1*). Werden Dateien von externen Speichermedien wie USB- oder Firewire-Festplatten gelöscht (rechts in → *Abb. 5.1-1*), so werden diese aus Performance-Gründen nicht in den Papierkorb des Benutzers (also auf einen anderen Datenträger) **kopiert**, sondern nur auf demselben Speichermedium **verschoben** (was durch Verschieben der Verzeichniseinträge sehr effizient möglich ist). Da auch hier die Benutzeraccounts berücksichtigt werden müssen, legt Mac OS auf externen Datenträgern den unsichtbaren Ordner /.**Trashes** an, mit Unterverzeichnissen für jeden Account.

Mac OS X verwendet die Technik eines semantischen Dateisystems, um die physikalischen Papierkorb-Verzeichnisse für den Benutzer zu verstecken. Diese Ordner werden allerdings sichtbar, sobald der Datenträger von einem anderen Betriebssystem wie z.B. Microsoft Windows gemountet wird **[Dav05]**. Unter Mac OS X hat der Benutzer keinen Zugriff auf einen einzelnen Papierkorb-Ordner, sondern sieht nur ein einziges Symbol im *Dock* (unten in → *Abb. 5.1-1*), das alle **Trash**-Ordner zusammenfasst. Enthält nur ein einziges Verzeichnis Dateien, wird das Dock-Symbol als gefüllt dargestellt. Wird dieser globale Papierkorb geleert, wird der Inhalt aller zum Account gehörenden physikalischen **Trash**-Verzeichnisse gelöscht **[Bec06]**:

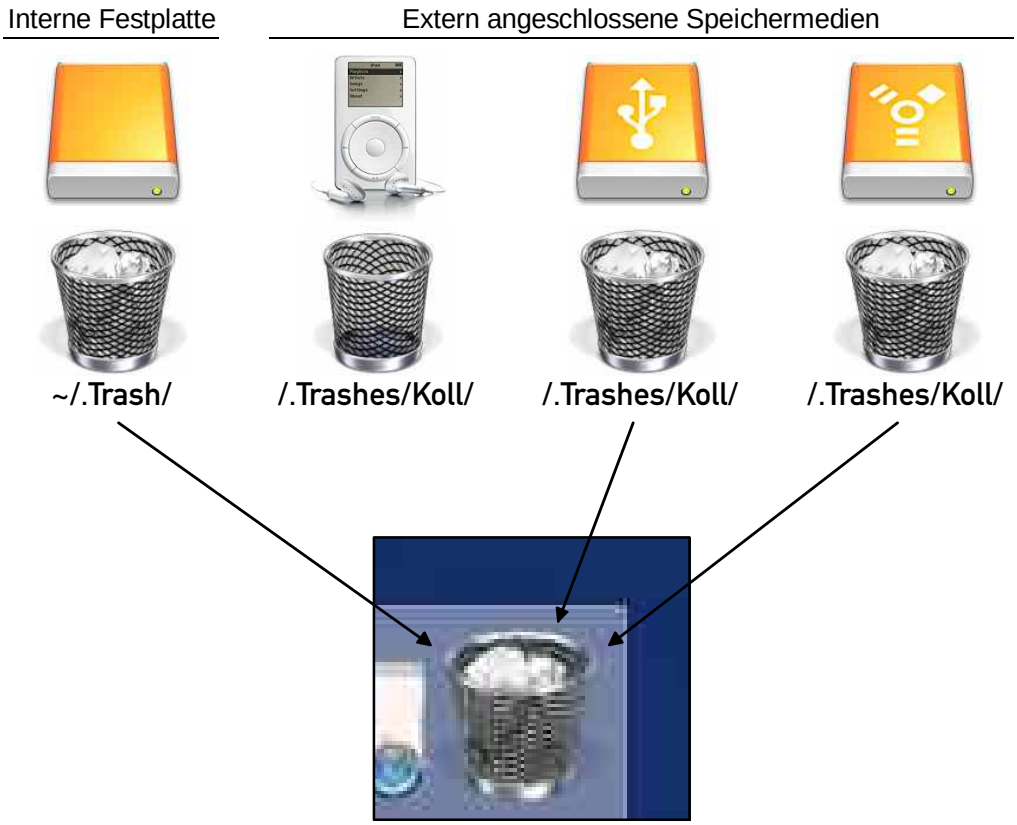


Abb. 5.1-1: zusammengesetzter Papierkorb für den Account »Koll«

5.2 Dateisystem für Digitalkameras

Der Exif-Standard (→ *4.2*) definiert zusätzliche Metadaten für JPEG-Bilder. Nahezu alle digitalen Fotokameras speichern Zusatzinformationen nach diesem Standard innerhalb der **JPG**-Dateien ab. Beim Betrachten von bereits aufgenommenen Fotos ist es wünschenswert, dass die Digitalkamera auch auf die vorhandenen Exif-Metadaten zugreifen kann, um etwa zusammen mit dem Bild Informationen zur Aufnahmezeit oder Belichtung anzuzeigen. Außerdem soll die Kamera nur auf Fotos und Videos, nicht aber auf andere Dateitypen zugreifen.

Um zu möglichst vielen Geräten und Betriebssystemen kompatibel zu sein, müssen Speichermedien für Digitalkameras im FAT-Dateisystem (→ *3.2*) formatiert werden. **[JEI98]** definiert ergänzende Regeln für das physikalische Dateisystem, insbesondere den Namen **DCIM** für das Unterverzeichnis, in dem Digitalkameras ihre Fotos abzuspeichern haben. Außerhalb dieser Verzeichnisstruktur können so auch andere Dateitypen abgespeichert werden, ohne die Funktion der Kamera zu beeinträchtigen. Das semantische Dateisystem beschränkt sich also ausschließlich auf den **DCIM**-Ordner.

In **DCIM** selbst dürfen jedoch keine Dateien abgelegt werden, sondern nur weitere Unterverzeichnisse. Die Namen dieser Unterverzeichnisse bestehen aus genau 3 Ziffern, gefolgt von 5 **Groß**buchstaben; erlaubt sind Zahlen von 100 bis 999, so dass 900 Unterverzeichnisse möglich sind **[JEI98]**:

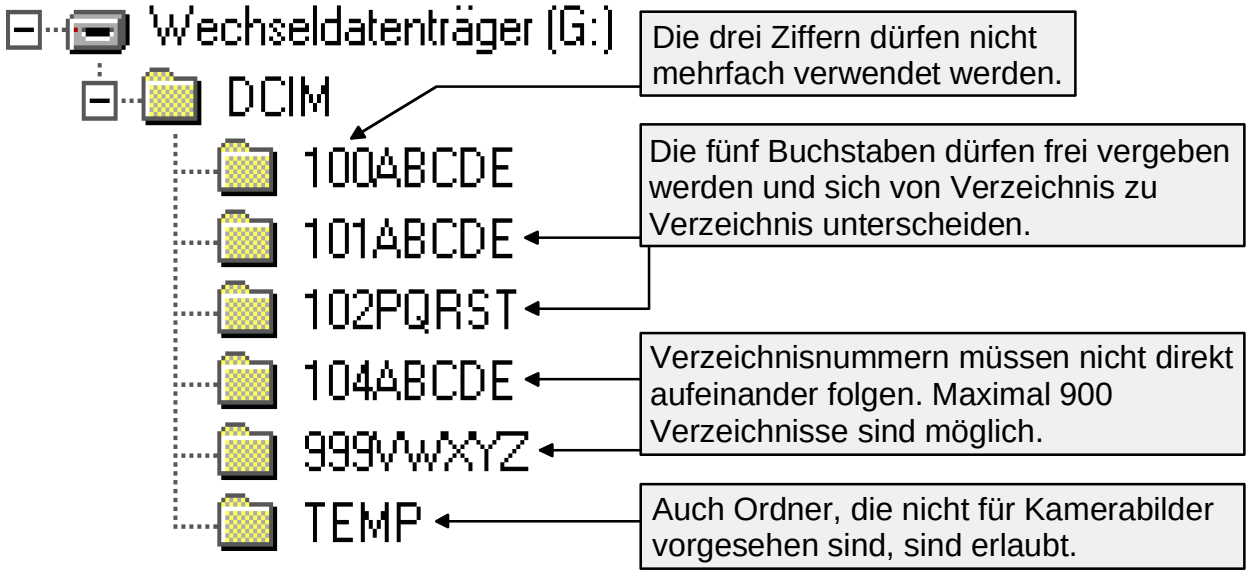


Abb. 5.2-1: Verzeichnisstruktur nach **[JEI98]**

Verzeichnisnamen wie **101abcde**, **103A** oder **IM08ABCD** sind also nicht erlaubt. Unterverzeichnisse dürfen jedoch mit dem »Read only«-Attribut (→ *3.2.1.3*) versehen werden, um das Löschen von Dateien innerhalb des Verzeichnisses zu verhinden **[JEI98]**. Diese Funktion wird lediglich von der Kamera-Firmware unterstützt, da unter DOS das »Read only«-Attribut eines Verzeichnisses nur das Löschen des Ordners (nicht jedoch von enthaltenen Dateien) verhindert, sofern diese nicht ebenfalls »Read only« sind. Hier werden die logischen Strukturen des Dateisystems also unterschiedlich interpretiert (→ *Abb. 3-1*).

Dateinamen innerhalb der **DCIM**-Unterverzeichnisse bestehen aus genau 4 **Groß**buchstaben, gefolgt von 4 Ziffern, einem Punkt und einer Namenserweiterung mit 3 Buchstaben, wie z.B. **JPG**, **MPG** oder **AVI**. **ABCDE0005.JPG**, **abcdefg0006.JPG**, **ABCD00~1.JPG** oder **ABCDEFGH.JPG** sind demnach ungültige Dateinamen. Innerhalb eines Verzeichnisses darf es keine Dateien mit gleichen Ziffernkomponenten geben, selbst wenn sie sich durch die Dateierweiterung unterscheiden. Aus den Nummern von Verzeichnis und Datei können so eindeutige Bezeichner einer Datei wie **100-0897** gebildet werden; Buchstaben und Dateiextension werden dabei entfernt **[JEI98]**.

5.3 Sony Memorysticks

Sony Memorysticks sind Flash-Speicherkarten, die in vielen Geräten (darunter auch Digitalkameras) eingesetzt werden können. Das semantische Dateisystem dieser Datenträger, das eine Ergänzung des Dateisystems von Digitalkameras darstellt (→ *5.2*), vereinfacht die Benutzung in mobilen Geräten und garantiert gleichzeitig, dass sich die Dateien diverser Gerätetypen nicht gegenseitig stören oder überschreiben. Dies wird besonders deutlich, wenn ein Memorystick mit einem Sony-PDA benutzt wird (→ *5.3.3*).

5.3.1 Hardware

Zunächst ist zu beachten, dass die meisten Memorystick-Typen in zwei Größen angeboten werden: neben dem länglichen Standard-Memorystick werden viele Varianten auch als verkürzte Memorystick Duo und Memorystick Micro angeboten, die in besonders kleinen Geräten wie Mobiltelefonen Verwendung findet. Mit einem Adapter können die kleinen Memorystick Duo und Memorystick Micro jederzeit auf Standardgröße gebracht werden, um in anderen Geräten oder Kartenlesern betrieben zu werden:

Abb. 5.3-1: Adapter für Memorysticks

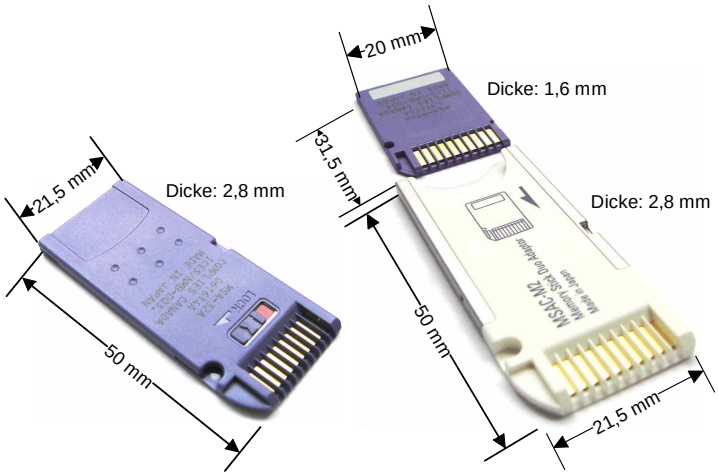


Abb. 5.3-2: Maße **[Mem05b]**

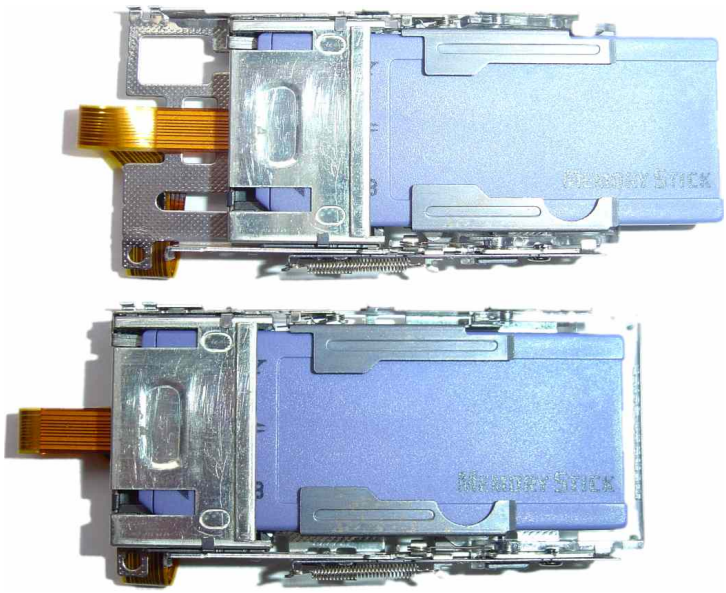


Abb. 5.3-3: internes Memorystick-Laufwerk

Die zehn Kontakte eines Memorysticks dienen der Steuerung und Datenübertragung und sind mit einem Controller verbunden, der seinerseits den Speicher ansteuert. Durch den Controller ist es möglich, Daten seriell zu übertragen und erst nach Abschluss des Transfers einen Sektor im Flash-Speicher zu löschen und neu zu beschreiben. Dieses Design wird sogar beim Öffnen des Gehäuses deutlich:

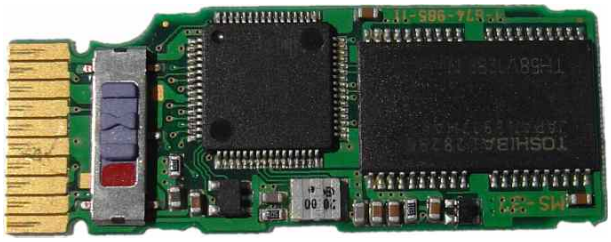


Abb. 5.3-4: geöffneter Memorystick (16 MB)

Aufgrund verschiedener Anwendungsgebiete und durch den technischen Fortschritt gibt es eine Vielzahl unterschiedlicher Memorysticks, die sich in vier Familien einteilen lassen:

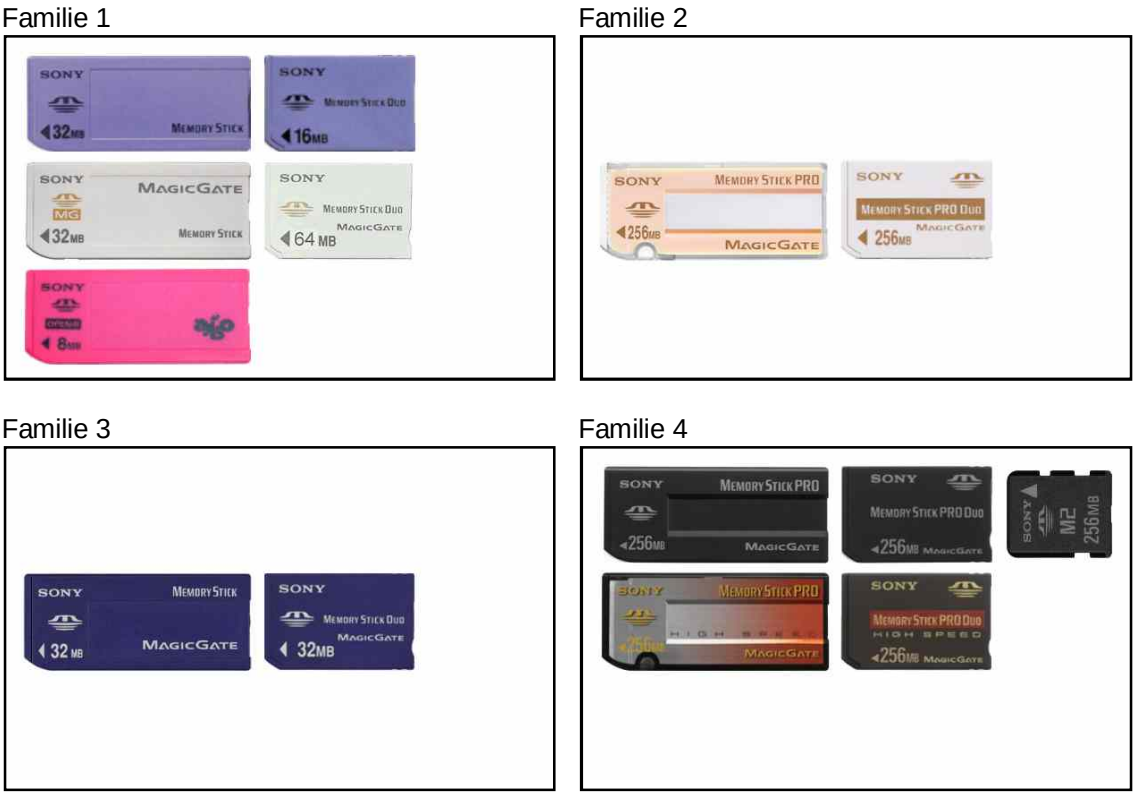


Abb. 5.3-5: Memorystick-Familien

5.3.1.1 Familie 1 (alte Memorysticks)

Die Memorysticks dieser Familie waren die erste Variante, die im Handel erhältlich war; es gibt sie in einer Kapazität von 4 MB bis 128 MB. Diese Grenze hat ihre Ursache in der Verwendung von FAT12 (→ *3.2.1*) als physikalisches Dateisystem, denn viele ältere Sony-Geräte beherrschen FAT16 (→ *3.2.1*) oder FAT32 (→ *3.2.2*) noch nicht. Die Kontakte sind wie folgt belegt:

	Belegung
1	VSS
2	BS
3	
4	DIO0
5	
6	INS
7	
8	CLK
9	VCC
10	VSS

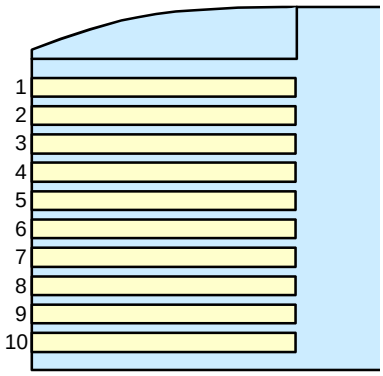


Abb. 5.3-6: Pinbelegung bei Standard-Memorysticks **[Mem05b]**

Die Kontakte **VSS** und **VCC** versorgen den Memorystick mit Strom; Pin 1 und 10 (**VSS**) sind elektrisch verbunden. Im Laufwerk sind die Pins für **VSS**, **VCC** und **INS** weiter vorne, so dass durch den früheren Kontakt mit **INS** das Einschieben eines Sticks erkannt werden kann. Die drei unbelegten Pins sind hochohmig. **DIO0** (Data I/O 0), **BS** (Bit Select) und **CLK** (Clock) übertragen Befehle und Daten zum Controller im Inneren des Sticks:

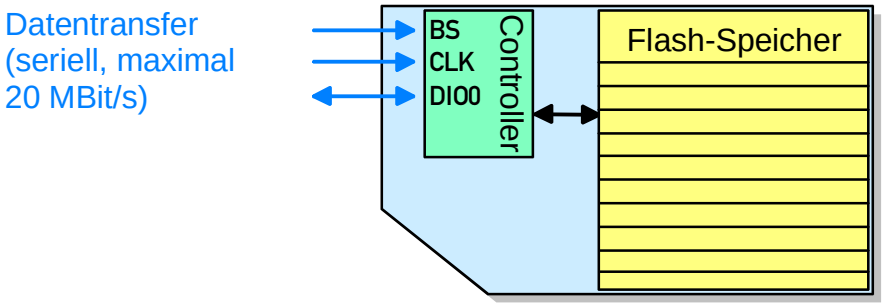


Abb. 5.3-7: serielle Datenübertragung zum Controller **[Mem05a]**

Die hellblauen Memorysticks stellen das Grundmodell dar; die weißen Memorysticks erweitern die Grundausführung mit dem *MagicGate*-Kopierschutz. MagicGate ist eine hardwarebasierte Verschlüsselung, bei der bestimmte Dateien codiert abgespeichert werden. Der Schlüssel wird dabei dem Memorystick entnommen, nachdem sich die Software per Challenge/Response autorisiert hat **[Hen00]**. Bestimmte Geräte können die codierten Dateien dann wieder entschlüsseln – der Anwender hingegen nicht, so dass ein Tauschen von Musikdateien unterbunden wird. Andere Dateien, wie etwa aufgenommene Bilder, werden aber weiterhin unverschlüsselt abgespeichert:

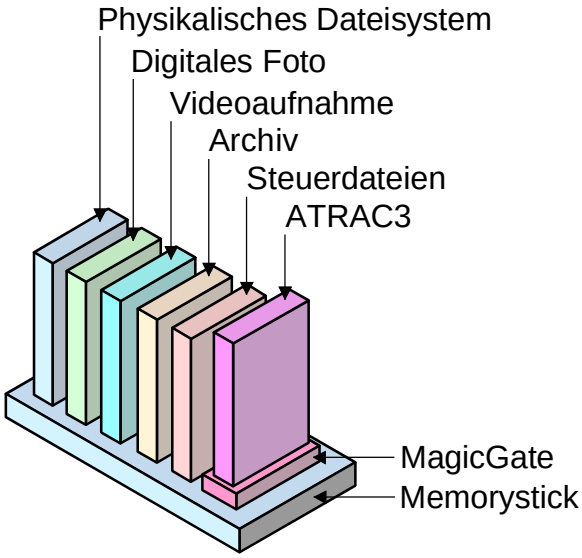


Abb. 5.3-8: selektive Verschlüsselung durch MagicGate **[Mem05b]**

Die violetten Memorysticks gibt es nur mit 8 MB oder 16 MB Speicherkapazität. Sie sind schwer erhältlich und nicht für den normalen Einsatz gedacht, obwohl sie natürlich auch in einer Digitalkamera funktionieren. Vielmehr wird die MagicGate-Verschlüsselung benutzt, um das Auslesen der Steuersoftware für Aibo-Hunderoboter zu verhindern. Diese Memorysticks dürfen niemals formatiert werden, da die enthaltene Firmware sonst zerstört wird.

5.3.1.2 Familie 2 (alte Memorystick PRO)

Um die Begrenzung auf 128 MB Speicher abzuschaffen, wurde eine neue Art Memorysticks entwickelt: der Memorystick PRO. Er ist ab einer Größe von 256 MB erhältlich und benutzt auch schnelleren Flash-Speicher als seine Vorgänger. Geräte, die einen Memorystick PRO benutzen können, müssen somit auch das FAT16- und FAT32-Dateisystem mounten können. Der verbesserte Flash-Speicher kann auch schneller angesprochen werden als bei den herkömmlichen Memorysticks, da die vorher nicht belegten Pins nun als Datenleitungen verwendet werden:

	Belegung
1	VSS
2	BS
3	DIO1
4	DIO0
5	DIO2
6	INS
7	DIO3
8	CLK
9	VCC
10	VSS

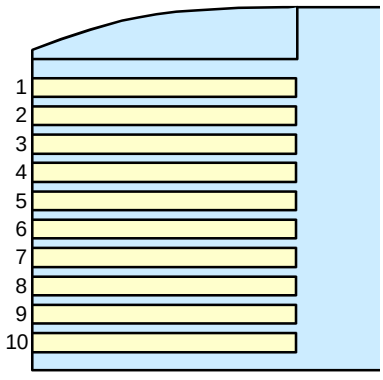


Abb. 5.3-9: Pinbelegung bei Memorystick Pro **[Mem05b]**

Die vier Datenleitungen werden parallel verwendet, so dass ein Byte mit zwei anstatt acht Taktzyklen übertragen werden kann:

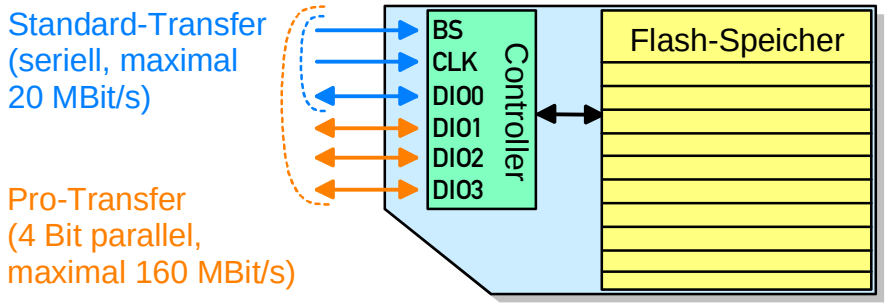


Abb. 5.3-10: parallele Datenübertragung zum Controller **[Mem05a]**

Durch die veränderte Adressierung und Ansteuerung ist der Memorystick PRO nicht mehr zu älteren Geräten kompatibel. Vor allem PDAs können aber durch ein Firmware-Update, das neue Hardware- und Dateisystem-Treiber enthält, auf einen Memorystick PRO zugreifen. Außerdem bieten alle Memorystick PRO serienmäßig die MagicGate-Verschlüsselung.

5.3.1.3 Familie 3 (neue Memorysticks)

Die Standard-Memorysticks der neuen Generation sind dunkelblau und stellen eine Mischform der ersten beiden Familien dar. Sie können in allen, also auch älteren, Geräten betrieben werden und sind standardmäßig mit der MagicGate-Verschlüsselung ausgestattet.

Die Memorysticks dieser Familie sind durchweg mit dem schnellen Flash-Speicher der Memorystick PRO ausgestattet. Die Ansteuerung kann dabei auf zwei Arten erfolgen: mit nur einer Datenleitung, um kompatibel zum ursprünglichen Memorystick zu sein, und mit 4 Datenleitungen in den Geräten, in denen ein Memorystick PRO betrieben werden kann. Nur die Kapazität bleibt auf 128 MB beschränkt.

5.3.1.4 Familie 4 (neue Memorystick PRO)

Die neue Generation der Memorystick PRO gibt es als normale Variante und zusätzlich als »High-speed«-Memorystick. Es werden jedoch robustere Speicherchips eingebaut, die den Betrieb zwischen -25°C und +85°C erlauben. Die Highspeed-Memorysticks haben eine garantierte Schreibgeschwindigkeit von 15 MBit/s und eigenen sich damit vor allem für hochauflösende Filmaufnahmen.

5.3.2 Physikalisches Dateisystem

Memorysticks werden wie Festplatten partitioniert (→ *3.1*) und können mit jedem beliebigen Dateisystem formatiert werden; damit die Speichermedien aber in anderen Geräten als einem PC eingesetzt werden können, müssen sie eine primäre FAT-Partition enthalten. Für viele Größen sind mehrere Varianten denkbar: ein Memorystick mit 64 MB könnte sowohl mit einem FAT12- als auch einem FAT16-Dateisystem formatiert werden. Bei beiden Varianten wären unterschiedliche Clustergrößen möglich (→ *3.2.1*).

Verschiedene Geräte halten sich allerdings nicht an die Vorgaben eines bereits formatieren Memorysticks: wird beispielsweise ein Medium mit 32 MB Speicherkapazität korrekt mit einem FAT16-Dateisystem initialisiert, geht eine Digitalkamera vom Typ Sony DSC-P92 trotzdem davon aus, dass es sich um eine FAT12-Partition handelt, so dass bei einer Aufnahme das Dateisystem zerstört wird. Werden beim Formatieren andere Clustergrößen gewählt als vom Gerät erwartet, wird das Medium mit der Fehlermeldung »Format error« zurückgewiesen. Von diesen Problemen wird unter **[Lin05]** berichtet.

Die folgende Tabelle zeigt die Parameter, die Sony-Geräte für Standard-Memorysticks erwarten; die Partitionen müssen als bootfähig markiert werden:

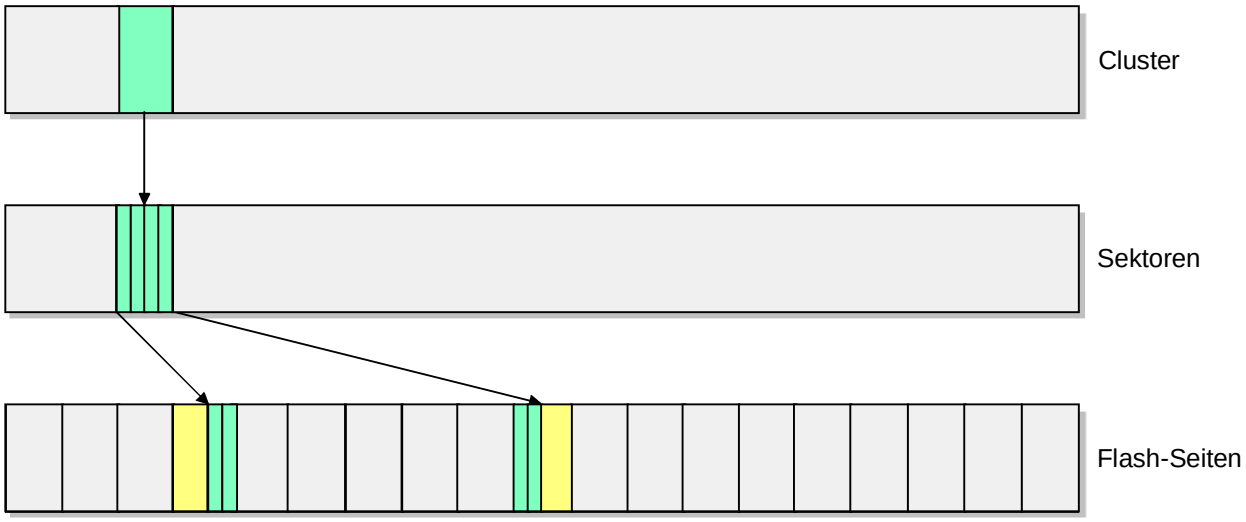
	Dateisystem	Clustergröße
4 MB	FAT12	16 Sektoren (8 KB)
8 MB	FAT12	16 Sektoren (8 KB)
16 MB	FAT12	32 Sektoren (16 KB)
32 MB	FAT12	32 Sektoren (16 KB)
64 MB	FAT12	64 Sektoren (32 KB)
128 MB	FAT16	32 Sektoren (16 KB)

Abb. 5.3-12: Parameter für Standard-Memorysticks

Ein Memorystick PRO darf flexibler formatiert werden, sowohl mit FAT16 als auch FAT32. Bei Verwendung des FAT32-Dateisystems sind allerdings die für die jeweilige Datenträgergröße empfohlenen Clustergrößen einzuhalten, da der Memorystick sonst unter Windows XP nicht mit allen USB-Kartenlesern fehlerfrei funktioniert.

5.3.2.1 Struktur des Flash-Speichers

Unabhängig von Dateisystem und Clustergröße ist beim Formatieren darauf zu achten, dass die Clustergrenzen mit den Grenzen physikalischer Flash-Seiten zusammenfallen. Ist dies nicht gewährleistet, so verteilt sich ein Cluster unter Umständen auf zwei Flash-Seiten, was die effektive Zugriffsgeschwindigkeit halbiert:



Für den Zugriff auf einen Cluster des Dateisystems (grün, oben und Mitte) müssen zwei Seiten des Flash-Speichers gelöscht und beschrieben werden, da die Grenzen nicht übereinstimmen.

Abb. 5.3-13: Clustergrenzen und physikalische Seiten des Flash-Speichers

5.3.3 Semantisches Dateisystem

Damit ein Memorystick als solcher erkannt wird, muss im Hauptverzeichnis die Datei **MEMSTICK.IND** als versteckte und schreibgeschützte Datei angelegt werden. Ein Memorystick PRO muss zusätzlich die Datei **MSTK_PRO.IND** enthalten:

```
ROOT@LocalHost|I:\>dir

Laufwerk I: hat keine Bezeichnung.
Suchmaske: I:\*.*

MEMSTICK IND              0  -----HR
MSTK_PRO IND              0  -----HR

                     0 Byte in 2 Dateien und 0 Verzeichnissen
           246.874.112 Byte frei

ROOT@LocalHost|I:\>
```

*Abb. 5.3-14: **MEMSTICK.IND** und **MSTK_PRO.IND***

Sobald ein formatierter Memorystick in eine Digitalkamera eingelegt und ein Bild aufgenommen wird, legt das Gerät eine Verzeichnisstruktur nach **[JEI98]** an – also den **DCIM**-Ordner mit Unterverzeichnissen. Jeder Sony-PDA verfügt über einen Memorystick-Schacht; auf den Geräten sind diverse Programme vorinstalliert, die einen Zugriff auf die Speicherkarte ermöglichen:

Abb. 5.3-15: Memorystick-Software

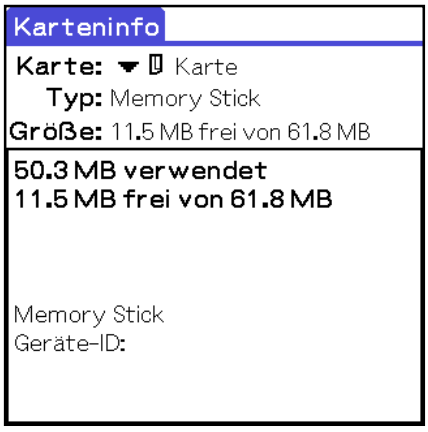


Abb. 5.3-16: Clié Karteninfo

Sony hat **[JEI98]** als Grundlage für das semantische Memorystick-Dateisystem genommen und erweitert. Neben dem **DCIM**-Verzeichnis, das von fast jeder Digitalkamera auf ihrem Datenträger erzeugt wird (→ *5.2*), legen diverse Sony-Geräte weitere Verzeichnisse für ihre Dateien an. Musikgeräte benutzen das Verzeichnis **HIFI** für ATRAC3-Dateien und den Ordner **CONTROL** zum Speichern von Einstellungen. Das Verzeichnis **PALM** wird vom Betriebssystem des Clié-PDA angelegt (Palm OS) und enthält in Unterverzeichnissen alle Programmdateien des PDAs:

```
ROOT@LocalHost|G:\>dir

Laufwerk G: hat keine Bezeichnung.
Suchmaske: G:\*.*

CONTROL       [Verzeichnis]  --D----  04.01.2002 18:01:20
DCIM          [Verzeichnis]  -AD----  12.07.2005 13:59:12
MISC          [Verzeichnis]  -AD----  12.07.2005 13:59:12
HIFI          [Verzeichnis]  --D----  08.07.2001 07:31:48
PALM          [Verzeichnis]  --D----  12.07.2005 18:25:54
MEMSTICK IND              0  -----HR  15.01.2000 00:38:08

                      0 Byte in 1 Datei und 5 Verzeichnissen
            11.763.712 Byte frei

ROOT@LocalHost|G:\>
```

Abb. 5.3-18: Hauptverzeichnis eines viel benutzten Memorysticks

Zum Datenaustausch zwischen dem internen Speicher des PDAs und einem eingelegten Memorystick dient das Programm MS Gate; es verweigert allerdings jeden Zugriff auf Dateien außerhalb des **PALM**-Verzeichnisses:

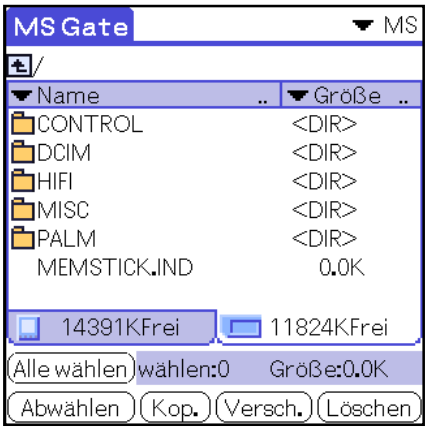


Abb. 5.3-19: Hauptverzeichnis

*Abb. 5.3-20: Unterverzeichnis **DCIM***

So können nur Programme, die **JPG**-Dateien lesen können und auch die Verzeichnisstruktur aus **[JEI98]** kennen, auf den **DCIM**-Ordner zugreifen, wie hier die Software »SplashPhoto«:

*Abb. 5.3-21: Zugriff auf Bilder im Verzeichnis **DCIM***

→ *Abb. 5.3-22* zeigt den Ordner **\PALM\PROGRAMS\MSFILES**, in den MS Gate sämtliche Systemdatenbanken und Dateien überträgt. Alle Dateien innerhalb des Ordners werden nach Typ sortiert und tragen lange Dateinamen; bestimmte Dateigruppen sind schreibgeschützt und können nicht gelöscht werden.

MS Backup bietet fünf Speicherplätze für Sicherheitskopien an (→ *Abb. 5.3-23*). MS Gate zeigt, dass die Dateien des ersten Backups das physikalische Verzeichnis **\PALM\PROGAMS\MSBackup\0** belegen:

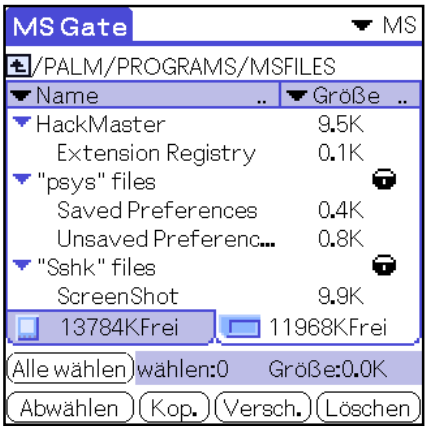


Abb. 5.3-22: Benutzerdateien

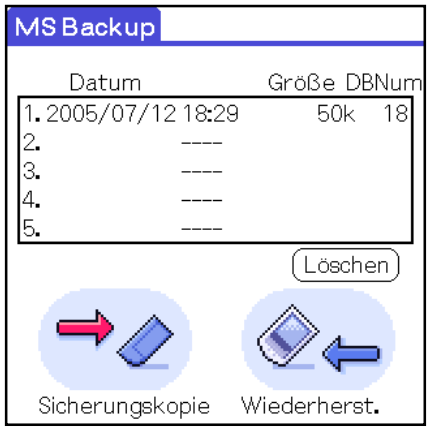


Abb. 5.3-23: MS Backup

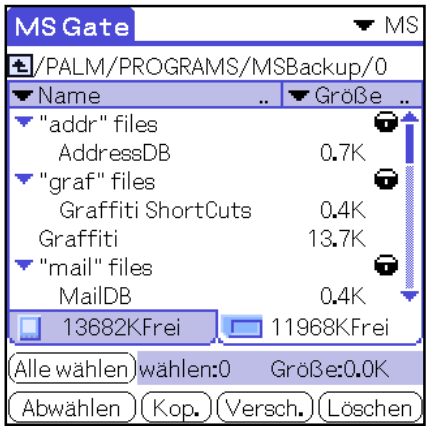


Abb. 5.3-24: Backup

Auf physikalischer Ebene haben alle Dateien auf dem Memorystick einen kurzen Dateinamen:

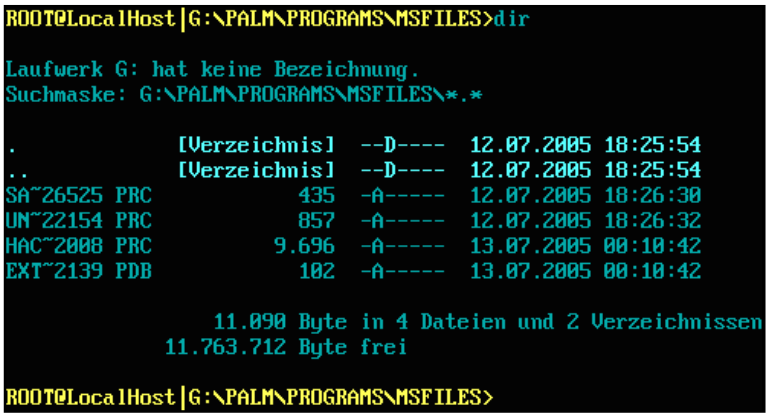


Abb. 5.3-25: Clié-Dateien auf physikalischer Ebene

Betrachtet man nun den Dateiinhalt, so stellt man fest, dass jede Datei mit einem nullterminierten String anfängt, der den eigentlich Dateinamen enthält:

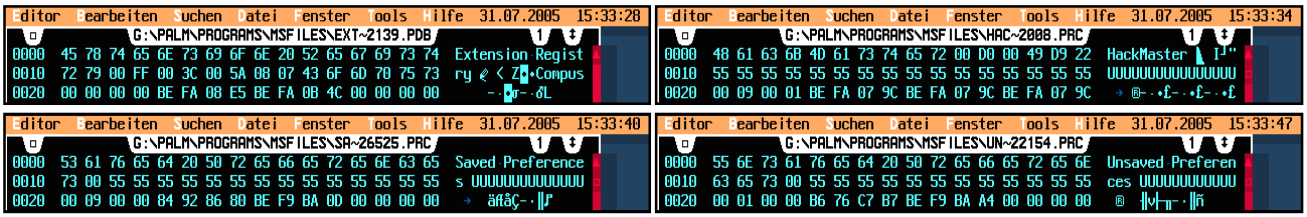


Abb. 5.3-26: lange Dateinamen als nullterminierter String am Dateianfang

Auf ähnliche Weise werden auch die Sicherheitskopien verwaltet. Für jedes Backup wird im Verzeichnis **\PALM\PROGRAMS\MSBACKUP** ein Unterverzeichnis angelegt; der Zeitpunkt, zu dem dieses Verzeichnis angelegt wurde, wird von MS Backup als Datum der Sicherheitskopie angezeigt (→ *Abb. 5.3-23*). Die Datei **BK~22545** ist versteckt und enthält Zusatzinformationen zu jedem Backup:

```
ROOT@LocalHost|G:\PALM\PROGRAMS\MSBACKUP>dir

Laufwerk G: hat keine Bezeichnung.
Suchmaske: G:\PALM\PROGRAMS\MSBACKUP\*.*

.            [Verzeichnis]  --D----  12.07.2005 18:29:12
..           [Verzeichnis]  --D----  12.07.2005 18:29:12
0            [Verzeichnis]  --D----  12.07.2005 18:29:12
BK~22545              132  -A---HR  14.07.2005 19:43:32

                  132 Byte in 1 Datei und 3 Verzeichnissen
              11.763.712 Byte frei

ROOT@LocalHost|G:\PALM\PROGRAMS\MSBACKUP>
```

Abb. 5.3-27: das Verzeichnis ***MSBACKUP***

Etwas komplexer stellt sich der Inhalt des HIFI-Ordners dar, in dem ATRAC3-Dateien zur Musikwiedergabe abgespeichert sind. Die Dateien mit der Endung **.MSA** (Memorystick Audio) enthalten ein Musikstück, die versteckte Datei **MGCRL.MSF** (MagicGate Control) ist zum Decodieren erforderlich:

```
ROOT@LocalHost|G:\HIFI>dir

Laufwerk G: hat keine Bezeichnung.
Suchmaske: G:\HIFI\*.*

.            [Verzeichnis]  --D----  08.07.2001 07:31:48
..           [Verzeichnis]  --D----  08.07.2001 07:31:48
PBLIST   MSF         16.384  -A-----  31.07.2005 16:30:24
MGCRL    MSF          1.024  -----HR  08.07.2001 07:34:14
A3D00039 MSA      1.835.008  -A-----  29.12.2002 14:48:12
A3D0003A MSA      1.703.936  -A-----  29.12.2002 14:48:20
A3D0003B MSA      1.818.624  -A-----  29.12.2002 14:48:26
A3D0003C MSA      2.949.120  -A-----  29.12.2002 14:48:38
A3D0003D MSA      1.900.544  -A-----  29.12.2002 14:48:46
A3D0003E MSA      3.555.328  -A-----  29.12.2002 14:48:58
A3D0003F MSA      3.293.184  -A-----  29.12.2002 14:49:10
A3D00040 MSA      1.835.008  -A-----  29.12.2002 14:49:16
A3D00041 MSA      3.178.496  -A-----  29.12.2002 14:49:28
A3D00042 MSA      1.933.312  -A-----  29.12.2002 14:49:36
A3D00043 MSA      4.243.456  -A-----  29.12.2002 14:49:52
A3D00044 MSA      1.835.008  -A-----  29.12.2002 14:50:00
A3D00045 MSA      3.620.864  -A-----  29.12.2002 14:50:12
A3D00046 MSA      1.949.696  -A-----  29.12.2002 14:50:22
A3D00047 MSA      3.686.400  -A-----  29.12.2002 14:50:34
A3D00048 MSA      3.751.936  -A-----  29.12.2002 14:52:50

             43.107.328 Byte in 18 Dateien und 2 Verzeichnissen
             11.763.712 Byte frei

ROOT@LocalHost|G:\HIFI>
```

Abb. 5.3-28: das Verzeichnis ***HIFI***

Auch im **HIFI**-Ordner befindet sich der eigentliche Dateiname innerhalb der Audiodatei und nicht im physikalischen Dateisystem. Solche Dateien haben also zwei Namen; einen physikalischen Dateinamen und eine Bezeichnung, die dem Benutzer angezeigt wird:

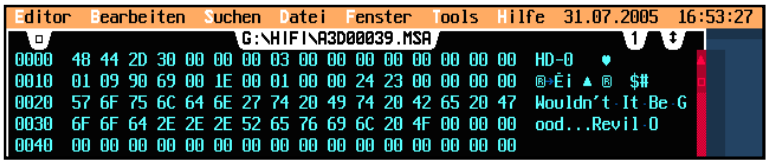


Abb. 5.3-29: Anfang einer ATRAC3-Datei

Die Reihenfolge der Wiedergabe ist zusammen mit anderen Informationen in der Datei **PBLIST.MSF** (Playback List) zu finden. Die hexadezimalen Zahlen wie **39h**, **3Ah** usw. beziehen sich dabei auf den physikalischen Dateinamen (**A3D00039.MSA**, **A3D0003A.MSA**):

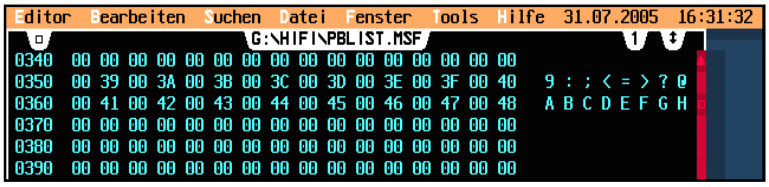


Abb. 5.3-30: Abspielliste

5.4 Siemens S65 Mobiltelefon

Das Siemens S65 ist ein modernes Multimedia-Handy, das über eine Kamera, mehrstimmige Klingeltöne, austauschbare Designs und Spiele verfügt. Diese Features setzen ein leistungsstarkes Dateisystem voraus. Das S65 ist bezüglich der Software und des Dateisystems fast identisch mit dem Siemens CX65 (modische Version mit Blinklichtern), dem M65 (Outdoor-Version) und dem SP65 (Business-Handy ohne Kamera).

5.4.1 Semantisches Dateisystem

Der Siemens S65 verfügt über einen internen Flash-Speicher von 16 MB, wobei etwa 6 MB von der Firmware belegt werden. Die Speicherkapazität kann mit MMC-Karten erweitert werden:

Abb. 5.4-1: Speicherkarte im S65

Abb. 5.4-2: Speicherkarte und PCMCIA-Adapter

5.4.1.1 Externe Speicherkarte

Das S65 kann aufgenommene Bilder direkt auf der Speicherkarte ablegen; sie können dann über einen Adapter (→ *Abb. 5.4-2*) auf einen Computer kopiert werden. Das Mobiltelefon legt Bilder und Videos jedoch nicht in einer zu **[JEI98]** kompatiblen Verzeichnisstruktur ab (→ *5.2*), sondern erzeugt beim Mounten der Speicherkarte sofort die Verzeichnisse **VIDEOS** und **PICTURES**:

```
ROOT@LocalHost|I:\>dir

Laufwerk I: hat keine Bezeichnung.
Seriennummer: 1234/5678
Suchmaske: I:\*.*

Es wurden keine Dateien gefunden, auf die die Beschreibung zutrifft !

           130.088.960 Byte frei

ROOT@LocalHost|I:\>
```

Abb. 5.4-3: frisch formatierte Speicherkarte

```
ROOT@LocalHost|I:\>dir

Laufwerk I: hat keine Bezeichnung.
Seriennummer: 1234/5678
Suchmaske: I:\*.*

VIDEOS        [Verzeichnis]  --D----  13.07.2005 21:07:42
PICTURES      [Verzeichnis]  --D----  13.07.2005 21:07:44

                    0 Byte in 0 Dateien und 2 Verzeichnissen
            130.056.192 Byte frei

ROOT@LocalHost|I:\>
```

Abb. 5.4-4: Speicherkarte nach dem Mounten

5.4.1.2 Interner Speicher

Wesentlich interessanter als die externe Speicherkarte ist der interne Flash-Speicher des Mobiltelefons. Er hat folgende Verzeichnisstruktur:

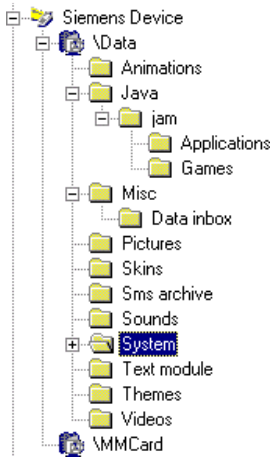


Abb. 5.4-5: physikalische Verzeichnisstruktur (normal)

Interessant daran ist, dass diese Verzeichnishierarchie auf dem Mobiltelefon anders dargestellt wird; es erscheinen deutsche Namen. **\Data\Pictures** wird zu **Bilder**, **\Data\Themes** zu **Themen**, **\Data\Skins** wird als **Farbschemata** angezeigt:

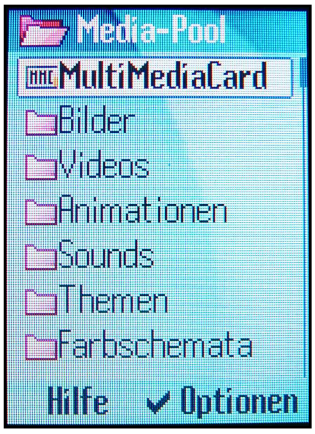


Abb. 5.4-6: Darstellung der Verzeichnisstruktur

Darüber hinaus werden die Verzeichnisse auch anders angeordnet: **Anwendungen** und **Spiele** führen direkt in die Unterverzeichnisse **\Data\Java\jam\Applications** bzw. **\Data\Java\jam\Games**, obwohl **Applications** und **Games** nicht direkt von **\Data** abzweigen. Das Mobiltelefon zeigt also nicht das physikalische Dateisystem an, sondern eine veränderte Darstellung.

5.4.2 Virtuelle Dateien

Der in → *Abb. 5.4-5* gezeigte Verzeichnisbaum ist nicht vollständig; viele Verzeichnisse sind vor dem Zugriff des Anwenders verborgen. Das Programm Open Disc **[OPE05]** ist jedoch in der Lage, ein S65 zu entsperren, wenn es an einer seriellen RS232-Schnittstelle angeschlossen ist:

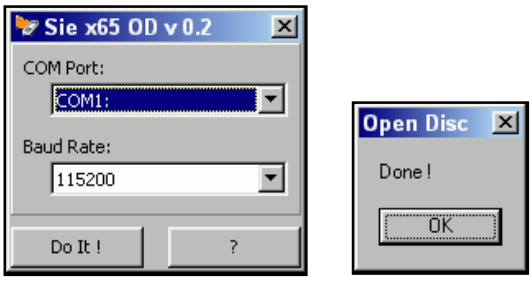


Abb. 5.4-7: Entsperren des internen Speichers

Die freigeschalteten Verzeichnisse enthalten vor allem Konfigurationsdateien und Teile der Systemsoftware. Von den Standarddateien in **\Config\Default** sind vier Kopien in den Ordnern **\Config\Customer1** bis **\Config\Customer4** vorhanden. Diese Verzeichnisse stammen vom Netzbetreiber, die Dateien im aktiven **Customer**-Ordner ersetzen gleichnamige Standarddateien aus **\Config\Default**:

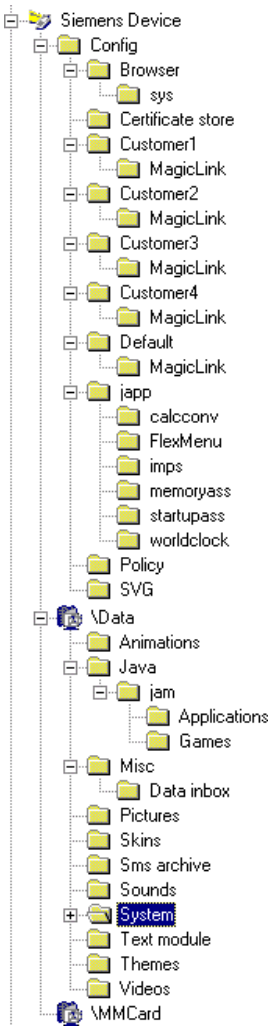


Abb. 5.4-8: Verzeichnisstruktur (entsperrt)

In den Unterverzeichnissen **MagicLink** befindet sich nur die Datei **MagicLinks.xml**. Die Firmware-Variante »BRD-Handel« (also die Referenzversion von Siemens) enthält folgende **MagicLinks.xml**:

```
<?xml version="1.0"?>
<!-- DOCTYPE bml SYSTEM "bml.dtd" -->
<bml>
 <sml application="BROWSER"
    location="JAVA_JAM_GAMES"
    title="Spiele laden"
    url="http://wap.siemens-mobile.com/games/" />
 <sml application="BROWSER"
    location="RINGINGTONE"
    title="Neue Klingeltöne"
    url="http://wap.siemens-mobile.com/sounds/" />
 <sml application="BROWSER"
    location="PICTURES"
    title="Neue Grafiken"
    url="http://wap.siemens-mobile.com/graphics/" />
</bml>
```

Diese XML-Datei definiert *virtuelle Dateien*; damit sind Objekte gemeint, die durch das semantische Dateisystem wie normale Dateien angezeigt werden, aber kein Pendant innerhalb des physikalischen Dateisystems haben:

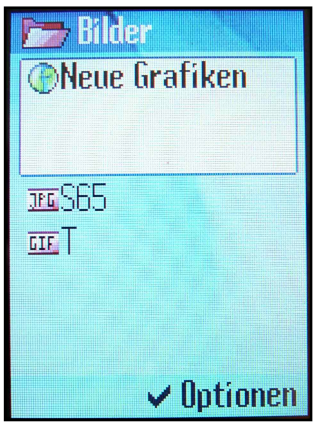


Abb. 5.4-9: virtuelle Datei aus ***MagicLinks.xml***

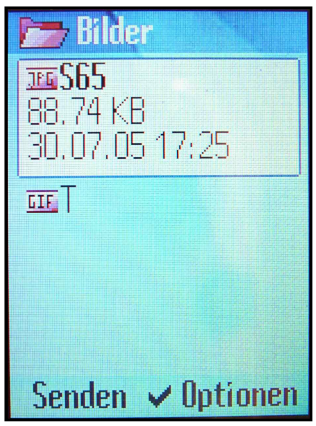


Abb. 5.4-10: bei leerer ***MagicLinks.xml***

Werden die virtuellen Dateien ausgewählt, öffnet sich der WAP-Browser des Mobiltelefons und führt den Anwender auf Seiten mit kostenpflichtigen Grafiken, Klingeltönen und Spielen.

5.5 /proc

DOS und ältere Windows-Versionen weisen jeder Partition und jedem Laufwerk einen Buchstaben zu. `A:` und `B:` sind für zwei Diskettenlaufwerke reserviert, so dass die erste Festplattenpartition (→ *3.1*) mit `C:` bezeichnet wird. `D:` wäre eine zweite Partition oder ein CDROM-Laufwerk und so fort. Vollständige Pfadnamen fangen immer mit einem Laufwerksbuchstaben an (`C:\TEST.TXT`), so dass die Dateisysteme vollständig von einander getrennt sind.

Unix und seine Derivate wie Linux, BeOS, Mac OS X oder Solaris gehen hier anders vor; sie kennen nur eine einzige Verzeichnishierarchie, die Root-Dateisystem genannt wird. An bestimmten Punkten, den *Mountpoints*, werden die Dateisysteme der einzelnen Laufwerke eingehängt:

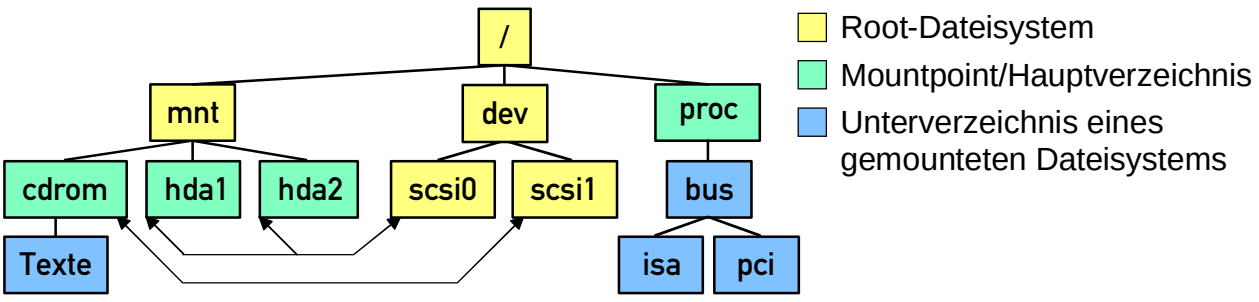


Abb. 5.5-1: Mountpoints

Die gemounteten Dateisysteme sind normalerweise intern mit einem symbolischen Gerät im Verzeichnis **/dev** verbunden, das das physikalische Dateisystem enthält. In → *Abb. 5.5-1* ist der Mountpoint **/mnt/cdrom** mit dem Gerät **/dev/scsi1** verbunden, dem physikalischen CDROM-Laufwerk. Das Verzeichnis Texte ist Teil der CD, **/mnt/cdrom** ist sowohl Mountpoint als auch das Hauptverzeichnis der eingelegten CD. Wenn kein Gerät gemountet ist, existiert ein leerer Mountpoint.

Eine Besonderheit ist das Dateisystem, das unter **/proc** gemountet ist – es besteht ausschließlich aus virtuellen Dateien, die Informationen über den Zustand des Linux-Kernels enthalten und bei Bedarf generiert werden **[Kil84]**:

```
root@rubin:/>ls -l /proc
-r--r--r--   1 root     root            0 2005-07-31 19:13 apm
dr-xr-xr-x   4 root     root            0 2005-07-31 19:13 asound
-r--r--r--   1 root     root            0 2005-07-31 19:13 buddyinfo
dr-xr-xr-x   8 root     root            0 2005-07-31 19:06 bus
-r--r--r--   1 root     root            0 2005-07-31 19:13 cmdline
-r--r--r--   1 root     root            0 2005-07-31 19:13 cpuinfo
-r--r--r--   1 root     root            0 2005-07-31 19:13 crypto
-r--r--r--   1 root     root            0 2005-07-31 19:13 devices
-r--r--r--   1 root     root            0 2005-07-31 19:13 diskstats
-r--r--r--   1 root     root            0 2005-07-31 19:13 dma
dr-xr-xr-x   2 root     root            0 2005-07-31 19:13 dri
dr-xr-xr-x   2 root     root            0 2005-07-31 19:13 driver
-r--r--r--   1 root     root            0 2005-07-31 19:13 execdomains
-r--r--r--   1 root     root            0 2005-07-31 19:13 fb
-r--r--r--   1 root     root            0 2005-07-31 19:13 filesystems
dr-xr-xr-x   3 root     root            0 2005-07-31 19:13 fs
dr-xr-xr-x   4 root     root            0 2005-07-31 19:13 ide
-r--r--r--   1 root     root            0 2005-07-31 19:13 interrupts
-r--r--r--   1 root     root            0 2005-07-31 19:13 iomem
-r--r--r--   1 root     root            0 2005-07-31 19:13 ioports
dr-xr-xr-x  18 root     root            0 2005-07-31 19:13 irq
-r--r--r--   1 root     root            0 2005-07-31 19:13 kallsyms
-r--------   1 root     root    805244928 2005-07-31 19:13 kcore
-r--------   1 root     root            0 2005-07-31 19:13 kmsg
-r--r--r--   1 root     root            0 2005-07-31 19:13 loadavg
-r--r--r--   1 root     root            0 2005-07-31 19:13 locks
-r--r--r--   1 root     root            0 2005-07-31 19:13 meminfo
-r--r--r--   1 root     root            0 2005-07-31 19:13 misc
-r--r--r--   1 root     root            0 2005-07-31 19:13 modules
lrwxrwxrwx   1 root     root           11 2005-07-31 19:13 mounts -> self/mounts
-rw-r--r--   1 root     root            0 2005-07-31 19:06 mtrr
dr-xr-xr-x   3 root     root            0 2005-07-31 19:13 net
-r--r--r--   1 root     root            0 2005-07-31 19:13 partitions
-r--r--r--   1 root     root            0 2005-07-31 19:13 pci
dr-xr-xr-x   2 root     root            0 2005-07-31 19:13 scsi
lrwxrwxrwx   1 root     root           64 2005-07-31 21:06 self -> 3323
-rw-r--r--   1 root     root            0 2005-07-31 19:13 slabinfo
-r--r--r--   1 root     root            0 2005-07-31 19:13 stat
-r--r--r--   1 root     root            0 2005-07-31 19:13 swaps
dr-xr-xr-x   9 root     root            0 2005-07-31 19:13 sys
--w-------   1 root     root            0 2005-07-31 19:13 sysrq-trigger
dr-xr-xr-x   2 root     root            0 2005-07-31 19:13 sysvipc
dr-xr-xr-x   4 root     root            0 2005-07-31 19:13 tty
-r--r--r--   1 root     root            0 2005-07-31 19:13 uptime
-r--r--r--   1 root     root            0 2005-07-31 19:13 version
-r--r--r--   1 root     root            0 2005-07-31 19:13 vmstat
```

Abb. 5.5-2: /proc

```
root@rubin:/>cat /proc/interrupts
           CPU0
  0:    1679183          XT-PIC  timer
  1:       2095          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  7:          1          XT-PIC  parport0
 10:          3          XT-PIC  bttv0, via82cxxx
 11:          0          XT-PIC  uhci_hcd, uhci_hcd
 14:        274          XT-PIC  ide0
 15:       7790          XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

Abb. 5.5-3: /proc/interrupts

```
root@rubin:/>cat /proc/version
Linux version 2.6.11 (root@Knoppix) (gcc-Version 3.3.5 (Debian 1:3.3.5-12)) #2 S
MP Thu May 26 20:53:11 CEST 2005
```

Abb. 5.5-4: /proc/version

Anhang A
Glossar

ACL	Access Control List (Liste für Zugriffskontrolle); Mechanismus, um einzelnen Benutzern den Zugriff auf Dateien und Verzeichnisse zu gestatten
API	Application Programming Interface (Programmierschnittstelle)
Atomar	Unteilbare Operation, die somit nur ganz oder gar nicht durchgeführt wird
Basic	Datenträger, die durch einen *Master Boot Record* partitioniert sind
Boot Parameter Block	Datenstruktur im *Bootsektor*, die vor allem Informationen zur Laufwerksgeometrie enthält
Bootsektor	Erster Sektor einer *Partition* (→ *3.2.1.2*)
Cache	Pufferspeicher, der häufig benötigte *Sektoren* im RAM speichert
CHS	Veraltetes Adressierungsschema für Datenträger (→ *3.1.1.1*)
Cluster	Zusammengefasste Gruppe von *Sektoren* innerhalb des FAT-Dateisystems, die die kleinste Zuordnungseinheit bilden (→ *3.2*)
DCIM	Abkürzung für Digital Camera Images (→ *5.2*)
Dock	Schnellstartleiste von Mac OS X; enthält auch den »Papierkorb« (→ *5.1*)
Dynamisch	Datenträger, die einen leeren *Master Boot Record* enthalten und durch ein neueres Verfahren dynamisch partitioniert sind (→ *3.1.4*)
File Allocation Table	Datenstruktur innerhalb eines FAT-Dateisystems (→ *3.2*), die Informationen über die Belegung und Verkettung von *Clustern* enthält (→ *3.2.1.1*)
Fork (Dateien)	Ein Fork ist eine unabhängige Datenspur einer Datei; unter MacOS bestehen Dateien aus einem »Data Fork« und einem »Resource Fork«.

Fragment	Einige Dateisysteme, z.B. ext2 (→ *3.4*), können Dateien in Fragmente unterteilen, die kleiner als ein *Cluster* sind; ein Cluster wird somit auf mehrere Dateien verteilt.
GUID	Globally Unique Identifier (→ *3.1.5*)
I-Node	»Information Node«, enthält in vielen Unix-Dateisystemen alle Informationen zu einer Datei, einschließlich der von ihr belegten *Sektoren*
Index	Ein Index unterstützt den Zugriff auf Tupel einer *Relation* durch den Wert eines Attributs (→ *2.4*)
Journaling	Dateisysteme mit Journaling werden durch Systemabstürze nicht beschädigt (→ *3.2.5.1*).
LBA	Moderneres Adressierungsschema für Datenträger (→ *3.1.1.2*)
MagicGate	Hardwarebasierte Verschlüsselung zum Copyright-Schutz
Master Boot Record	512 Byte große Datenstruktur am Anfang eines Datenträgers, die Informationen über die *Partitionen* enthält (→ *3.1.2*)
Master File Table	Die MFT enthält Einträge für alle Dateien und Unterverzeichnisse einer NTFS-Partition (→ *3.3.2*).
Memorysticks	Im Rahmen dieses Dokuments sind damit die Flash-Speicher der Firma Sony gemeint (→ *5.3*), nicht aber generische USB-Datenträger.
Mountpoint	Spezielles Verzeichnis der Unix-Verzeichnishierarchie, das mit dem Hauptverzeichnis eines Dateisystems zusammenfällt; alle untergeordneten Verzeichnisse gehören nicht mehr zum Root-Dateisystem.
Nullmarke	Leeres Attribut innerhalb eines Tupels einer *Relation* (→ *2.2.7*)
Partition	Abgegrenzter Bereich eines Datenträgers – der Begriff wird nicht im mathematischen Sinne verwendet, da es Bereiche eines Datenträgers geben kann, die keiner Partition zugeordnet sind (→ *3.1*)
Partitionieren	1. Zugriffsbeschleunigung für *Relationen* (→ *2.4.1.7*) 2. Einteilen eines Datenträgers in *Partitionen* (→ *3.1*)
Partitionssektor	Synonym für *Master Boot Record*

Querverbunden	Zwei oder mehr Dateien sind querverbunden, wenn sie an einer Position ihrer jeweiligen Clusterkette denselben *Cluster* (und damit auch alle Nachfolger) benutzen; Änderungen an einer querverbundenen Datei wirken sich auf unvorhersehbare Weise auf alle verbundenen Dateien aus.
RAID	Redundant Array of Independent Disks; Technik, mehrere Festplatten zu einem großen Laufwerk zusammenzuschalten
Registry	Konfigurationsdatei von Microsoft Windows; wird auch zum Speichern von Applikationseinstellungen benutzt
Relation	Element einer relationalen Datenbank (→ *2.3*), für das die Relationenalgebra gilt (→ *2.2*)
Sektor	Kleinste Einheit eines Mediums zum Datentransfer, meistens 512 Byte
SQL	Structured Query Language, Abfragesprache für relationale Datenbanken (→ *2.3.1*)
Tabelle	*Relation*, in der identische Datensätze mehrfach vorkommen (→ *2.1*)
Verloren	Ein verlorener *Cluster* ist in der *FAT* als belegt markiert, aber keiner Clusterkette einer Datei zugeordnet.
Virtuelle Datei	»Datei«, die nicht Teil eines physikalischen Dateisystems ist (→ *5.4.2*)
Virtuelles Dateisystem	Besteht nur aus *virtuellen Dateien* (→ *5.5*)

Anhang B
Literaturverweise

[Bec90] Beckmann, N. et al.
The R*-tree: An Efficient and Robust Access Method for Points and Rectangles
Proceedings of the ACM SIGMOD conference, Atlantic City 1990

[Bec06] Bechtel, U.
Mac OS X 10.4 Tiger (Intel Edition)
Addison-Wesley Verlag, 1. Auflage 2006

[Ber96] Berchtold, S. et al.
The X-tree: An Index Structure for High-Dimensional Data
Proceedings of the 22nd VLDB Conference, Mumbai 1996
http://www.vldb.org/conf/1996/P028.PDF
Stand: 01.11.2006

[Cod70] Codd, E.
A Relational Model of Data for Large Shared Data Banks
Communications of the ACM, Volume 13, Juni 1970
http://portal.acm.org/citation.cfm?id=362685
Stand: 11.10.2005

[Dat05] Datalight Inc.
Reliable File Systems for Windows CE
http://www.datalight.com/assets/resources/RelFS4CE.pdf
Stand: 19.07.2005

[Dav05] Davisson, G.
Mac OS X Hidden Files & Directories
http://www.westwind.com/reference/OS-X/invisibles.html
Stand: 05.10.2006

[DES06] DESKWORK Programmierwochenenden
http://www.deskwork.de/TEAM/WE.HTM
Stand: 03.10.2006

[DML96] Matrox Open DML File Format Workgroup
Open DML AVI File Format Extensions
Version 1.02, Februar 1996
http://the-labs.com/Video/odmlff2-avidef.pdf
Stand: 07.12.2007

[Dow01] Downey, A.
The structural cause of file size distributions
Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems table of contents, Cambridge 2001
http://portal.acm.org/citation.cfm?id=378824
Stand: 19.07.2005

[EXI05] http://www.exif.org/
Stand: 19.05.2005

[Fli07] http://www.flickr.com/
Stand: 14.01.2007

[Gia99] Giampaolo, D.
Practical File System Design with the Be File System
Morgan Kaufmann Publishers, 1. Auflage 1999

[Gut84] Guttman, A
R-Trees: A Dynamic Index Structure for Spatial Searching
Proceedings of the ACM SIGMOD Conference, Boston 1984
http://www.sai.msu.su/~megera/postgres/gist/papers/gutman-rtree.pdf
Stand: 01.11.2006

[Haw03] Hawkins, J.
Photostudio 2.62
http://www.stuffware.co.uk/photostudio/
Stand: 22.07.2005

[Hen00] Hendrich, N.
Skript »Digitale Audioverarbeitung«
Universität Hamburg, Wintersemester 2000
http://tams-www.informatik.uni-hamburg.de/ lehre/ws2000/vl-audioverarbeitung/13-watermarks.pdf
Stand: 10.07.2005

[ID305] Nilsson, M.
http://www.id3.org/
Stand: 14.07.2005

[IDF00] Institute of Information Systems, ETH Zürich
ID-Forum 23.11.2000
http://www.id.ethz.ch/events/idforum/ archiv/24112000/Informationssuche.pdf
Stand: 01.11.2006

[Int07] Intel Corporation
Extensible Firmware Interface
http://www.intel.com/technology/efi/
Stand: 09.06.2007

[Irl93] Irlam, G.
Unix File Size Survey
http://www.base.com/gordoni/ufs93.html
Stand: 19.07.2005

[JEI98] Japan Electronics and Information Technology Industries Association
Design rule for Camera File system
Version 1.0, JEITA-49-2 1998, Dezember 1998
http://www.exif.org/dcf.PDF
Stand: 22.07.2005

[JEI02] Japan Electronics and Information Technology Industries Association
Exchangable image file format for digital still cameras: Exif
Version 2.2, JEITA CP-3451, April 2002
http://www.exif.org/Exif2-2.PDF
Stand: 22.07.2005

[JEI05] Japan Electronics and Information Technology Industries Association
http://www.jeita.or.jp/english/
Stand: 22.07.2005

[JPE92] Joint Photographic Experts Group
JPEG File Interchange Format
Version 1.02, September 1992
http://www.jpeg.org/public/jfif.pdf
Stand: 22.07.2005

[Kil84] Killian, T. J.
Processes as files
USENIX Association 1984 Summer Conference Proceedings, Salt Lake City 1984

[Kol03] Koll, K.
Analyse und Bewertung verschiedener wichtiger Videoformate
Universität Dortmund, 12.07.2003
http://www.deskwork.de/INFOS/DIPL-ARB.PDF
Stand: 14.02.2007

[KSC05] Karen's Disk Slack Checker
http://www.karenware.com/powertools/ptslack.asp
Stand: 01.07.2005

[Kuh06] Kuhnt, U. et al.
FAT32+
Draft release (revision 2)
http://www.fdos.org/kernel/fatplus.txt
Stand: 04.01.2006

[LDM05] LDM Documentation
Linux NTFS project
http://linux-ntfs.sourceforge.net/ldm/overview/history.html
Stand: 27.06.2005

[Lin05] Linux-USB device overview
http://www.qbik.ch/usb/devices/showdev.php?id=182
Stand: 11.07.2005

[Mat03] Matthiessen, G. und Unterstein, M.
Relationale Datenbanken und SQL
Addison-Wesley Verlag, 3. Auflage 2003

[Mem05a] http://www.memorystick.com/
Stand: 14.07.2005

[Mem05b] http://www.memorystick.org/
Stand: 14.07.2005

[Mic07] Microsoft Corporation
How basic discs and volumes work
http://technet2.microsoft.com/windowsserver/en/library/bdeda920-1f08-4683-9ffb-7b4b50df0b5a1033.mspx
Stand: 09.06.2007

[MSD05a] Microsoft Developer Network
Platform Builder for Microsoft Windows CE 5.0 - Transaction-Safe FAT File System
http://msdn.microsoft.com/library/en-us/wcedata5/html/wce50contransactionsafefatfilesystem.asp
Stand: 17.07.2005

[MSD05b] Microsoft Developer Network
AVI RIFF File Reference
http://msdn.microsoft.com/archive/en-us/directx9_c/directx/htm/avirifffilereference.asp
Stand: 26.07.2005

[Mue03] Mueller, S.
Upgrading and repairing PCs
Que Publishing, 15. Auflage 2003
http://www.quepublishing.com/content/downloads/upgrading/fifteenth_edition/book_samples/sc24_0789729741.pdf
Stand: 26.06.2005

[NPM05] Norton Partition Magic
http://www.powerquest.com/partitionmagic/
Stand: 01.07.2005

[OPE05] Open Disc
http://www.deskwork.de/DOWNLOAD/S65/OPENDISC.ZIP
Stand: 30.07.2005

[RFC4122] RFC 4122
A Universally Unique Identifier (UUID) URN Namespace
http://www.ietf.org/rfc/rfc4122.txt
Stand: 19.09.2007

[Saa05] Saake, G. et al.
Datenbanken: Implementierungstechniken
mitp-Verlag, 2. Auflage 2005

[Sel87] Sellis, T.K. et al.
The R+-Tree: A Dynamic Index for Multi-Dimensional Objects
Proceedings of the 13th VLDB Conference, Brighton 1987

[Sol98] Solomon, D.
Inside Windows NT
Microsoft Press, 2. Auflage 1998

[Sor05] Sorkin, A.
abcAVI
http://abcavi.kibi.ru/download.htm
Stand: 29.07.2005

[Sta05] Stats 2000 Disk Statistics Program
http://www.contactplus.com/products/freestuff/stats.htm
Stand: 19.07.2005

[Sun89] Sun Microsystems
NFS: Network file system protocol specification
Request for Comments 1094, Version 2, März 1989

[Sun04] Sun Microsystems
ZFS: Das ultimative Dateisystem
http://de.sun.com/homepage/feature/2004/zfs/
Stand: 11.01.2007

[Tan02] Tanenbaum, A.S.
Moderne Betriebssysteme
Pearson Education, 2. Auflage 2002

[Tis94] Tischer, M.
PC Intern 4
Data Becker Verlag, 1. Auflage 1994

[Wal97] Walker, W.
Missing disk space - how to calculate it
http://www.nvdi.com/whertra/w950511.htm
Stand: 20.01.1997

[Wik05a] Wikipedia
Ext2
http://de.wikipedia.org/wiki/Ext2
Stand: 04.07.2005

[Wik05b] Wikipedia
Long-tail traffic
http://en.wikipedia.org/wiki/Long-tail_traffic
Stand: 19.07.2005

[Wik05c] Wikipedia
Pareto distribution
http://en.wikipedia.org/wiki/Pareto_distribution
Stand: 19.07.2005

[Wik05d] Wikipedia
Lognormal distribution
http://en.wikipedia.org/wiki/Lognormal_distribution
Stand:19.07.2005

[Wik05e] Wikipedia
JFIF
http://de.wikipedia.org/wiki/JFIF
Stand: 22.07.2005

[Wik05f] Wikipedia
Relationale Algebra
http://de.wikipedia.org/wiki/Relationale_Algebra
Stand: 11.10.2005

[Wik05g] Wikipedia
SQL
http://en.wikipedia.org/wiki/SQL
Stand: 13.10.2005

[Wik05h] Wikipedia
Relationale Datenbank
http://de.wikipedia.org/wiki/Relationale_Datenbank
Stand: 11.10.2005

[Wik05i] Wikipedia
List of SQL database management systems
http://en.wikipedia.org/wiki/List_of_SQL_database_management_systems
Stand: 11.10.2005

Anhang C
Testprogramm zur Clustergröße

Um in → *3.2.4* den Zusammenhang zwischen Clustergröße und verschwendetem Speicher grob untersuchen zu können, wurde ein Testprogramm erstellt. Es kann mit Borland/Turbo Pascal sowohl für den Real Mode als auch den Protected Mode compiliert werden.

Das DOS-Programm durchsucht rekursiv den Verzeichnisbaum des Laufwerks, allerdings nur bis zur 8. Ebene. Unter DOS sind maximal 8 Unterverzeichnisse möglich, Windows gestattet jedoch unbegrenzt tiefe Verzeichnisbäume. DOS-Programme dürfen niemals auf Verzeichnisse zugreifen, die auf einer tieferen Ebene liegen, da sonst interne Puffer überlaufen.

Die Größe aller gefundenen Dateien wird in einer verketteten Liste gespeichert. Jedes Listenelement enthält auch die Anzahl der Dateien dieser Größe, so dass bei vielen Dateien gleicher Länge nur ein Listenelement angelegt werden muss. Nach Abschluss der Suche wird zu allen möglichen Clustergrößen zwischen 512 Byte und 32 KB die Speicherverschwendung errechnet, summiert und ausgegeben.

C.1 CLUSTER.PAS

```
program Cluster;
{$M 32768,262144,655360}
uses Dos;

type
  PFZ=^FZ;
  FZ=record
    Size,Anz: LongInt;
    Next: PFZ;
end;

var
  Filesizes: PFZ;

procedure InsertFilesize(S: LongInt);
var
  Iterator: ^PFZ;
  Element: PFZ;
begin
  Iterator:=@Filesizes;
    while Iterator^<>nil do begin
      if Iterator^^.Size=S then begin {Schon Eintrag gleicher Größe gefunden}
        Inc(Iterator^^.Anz,1);
        exit;
      end;
    Iterator:=@Iterator^^.Next;
    end;
```

```
  {Neuer Eintrag am Ende der Liste}
  New(Element);
  Element^.Size:=S;
  Element^.Anz:=1;
  Element^.Next:=nil;
  Iterator^:=Element;
end;

procedure ScanTree(const Pfad: OpenString; Level: Byte);
var SD: SearchRec;
begin
  FindFirst(Pfad+'*.*',AnyFile,SD);
    while DosError=0 do begin
      if (SD.Attr and Directory)<>0 then begin
         if (SD.Name<>'.') and (SD.Name<>'..') and (Level<8) then
           ScanTree(Pfad+SD.Name+'\',Level+1);
      end else
        if SD.Size>0 then InsertFilesize(SD.Size);
    FindNext(SD);
    end;
end;

procedure Auswerten(ClusterSize: Word);
var
  Iterator: PFZ;
  Slack,Wasted: LongInt;
begin
  writeln('Clustergröße: ',ClusterSize,' Byte');
  Wasted:=0;
  Iterator:=FileSizes;
    while Iterator<>nil do begin
    Slack:=Iterator^.Size-ClusterSize*(Iterator^.Size div ClusterSize);
    Inc(Wasted,Iterator^.Anz*Slack);
    Iterator:=Iterator^.Next;
    end;
  writeln('Verschwendet: ',Wasted,' Byte');
  writeln;
end;

var A: Byte;
begin
  Filesizes:=nil;
  ScanTree('C:\',0);
  {Auswerten}
    for A:=0 to 6 do Auswerten(512 shl A);
end.
```

Anhang D
Übersetzer für JPG-Dateien

Der in diesem Kapitel vorgestellte Übersetzer hat die Aufgabe, eine **JPG**-Datei ins lokale Dateisystem zu kopieren und dabei die enthaltenen Metadaten (→ *4.2*) zu normalisieren.

D.1 Struktur eines Translators

An ein Übersetzungsmodul wird die Quelldatei durch das global bekannte Objekt **Stream** übergeben. Das Objekt **Stream** unterstützt u.A. die Operation **SeekTo**, um zu einer Stelle innerhalb der Datei zu springen, und gibt die Dateigröße durch die Funktion **BufSize** zurück. Allen Übersetzern steht die Funktion **BRead** zur Verfügung, die den Speicherbereich einer Variablen mit Bytes aus der Ursprungsdatei auffüllt; zusätzlich werden ggf. automatisch Fehlercodes innerhalb des Übersetzers gesetzt.

Die Ausgabedatei wird durch die globale Dateivariable **OFile** vom Standardtyp **File** bestimmt; sie wird vor Aufruf des Übersetzers angelegt und mit den Standardfunktionen von Borland Pascal beschrieben.

D.2 JPG-Übersetzer

Der vorliegende Übersetzer geht davon aus, dass die Datei **Stream** eine gültige JFIF-Datei ist. Die Hauptfunktion **TranslateJPG** parst die Datei und sucht nach JPEG-Markern, die durch das Byte **$FF** eingeleitet werden. Ist der Typ in **[$C0..$C7,$C9..$CB,$CD..$CF,$DA,$DB,$DD,$E1,$E7,$EE,$FE]** enthalten, wird der Datenblock meistens durch Ausgabe des Markerheaders und Aufrufs der API-Funktion **TransferChunk** direkt kopiert; lediglich die Marker **APP1**, **APP7**, **APP14** (→ *4.2.1*) und der JPEG-Kommentar (Marker **$FE**) werden näher untersucht.

Handelt es sich bei einem **APP1**-Marker wirklich um einen Exif-Marker, befasst sich die Prozedur **HandleExif** näher mit dem Datensegment und entfernt ein eventuell vorhandenes Vorschaubild. Ein **APP7**-Marker wird ignoriert und mit **CreateExif** durch einen Exif-Marker ersetzt (→ *4.2.2*). Wurde ein **APP14**-Marker von Adobe Photoshop angelegt (→ *4.2.1*), so wird er übertragen, andernfalls übersprungen. Ein JPEG-Kommentar wird auf die Strings '**SIEMENS MOBILE PHONES**' bzw. '**NOKIA MOBILE PHONES**' hin untersucht. Wird eine dieser Zeichenketten gefunden, so ist davon auszugehen, dass der Kommentarstring das Format aus (→ *4.2.3*) aufweist; die Prozedur **HandlePhone** parst dann den Kommentar und ruft mit den gefundenen Daten **CreateExif** auf.

```
function TranslateJPG(TranslationParam: Byte): Boolean; far;
label Skip;
var
  Comment: String;
  ID: array[1..5] of Char;
  Siemens: array[1..19] of Char;
  FilePosi,FP: LongInt;
  L,W: Word;
  A,MarkerTyp: Byte;
begin
  TranslateJPG:=False;
    if ValidDiskTranslat(Stream^.BufSize)=False then exit;
{LCARS-Header}
  OutputHeader.Typ:=dtDigiFoto;
  BlockWrite(OFile,OutputHeader,SizeOf(LCARSType));
{JPEG-Marker}
    repeat begin
      repeat begin
        if BRead(A,1)=False then exit;
      end until A=$FF;
      repeat begin
        if BRead(MarkerTyp,1)=False then exit;
      end until MarkerTyp<>$FF;
      if MarkerTyp<$C0 then exit;
      if MarkerTyp in [$D8,$D9] then begin
        BlockWrite(OFile,A,1); BlockWrite(OFile,MarkerTyp,1);
      end else begin
          if BRead(W,2)=False then exit;
       FilePosi:=Stream^.GetPos+Swap(W)-2;
        if MarkerTyp in [$C0..$C7,$C9..$CB,$CD..$CF,$DA,$DB,$DD,$E1,$E7,$EE,$FE] then begin
          case MarkerTyp of
          $E1: begin              {Wirklich Exif ?}
                 if BRead(ID,5)=False then exit;
                 if ID<>'Exif'+#0 then goto Skip;
               HandleExif(Stream^.GetPos-5,Swap(W)-2);
               goto Skip;
             end;
          $E7: begin              {Wirklich Siemens Thumbnail ?}
                 if BRead(Siemens,19)=False then exit;
                 if Siemens<>'<SIEMENS THUMBNAIL>' then goto Skip;
               CreateExif('Siemens','Mobile Phone',0,0,0,0,0,0);
               goto Skip;
             end;
          $EE: begin              {Wirklich Adobe ?}
                 if BRead(ID,5)=False then exit;
                 if ID<>'Adobe' then goto Skip;
               Stream^.SeekTo(Stream^.GetPos-5);
             end;
          $FE: begin              {JPEG Comment vom Siemens SX1 ?}
               L:=Swap(W);
                 if L>255 then L:=255;
               Comment[0]:=Char(L);
               FP:=Stream^.GetPos;
                 if BRead(Comment[1],L)=False then exit;
                 if (UppCase(LeftStr(Comment,21))='SIEMENS MOBILE PHONES') then begin
                    if HandlePhone('Siemens',Comment) then goto Skip;
                 end else
                   if (UppCase(LeftStr(Comment,19))='NOKIA MOBILE PHONES') then begin
                      if HandlePhone('Nokia',Comment) then goto Skip;
                   end;
               Stream^.SeekTo(FP);
             end;
```

```
          end;
        BlockWrite(OFile,A,1); BlockWrite(OFile,MarkerTyp,1); BlockWrite(OFile,W,2);
          if TransferChunk(Swap(W)-2)=False then exit;
      end;
Skip:
      if FilePosi>Stream^.BufSize then exit;
    Stream^.SeekTo(FilePosi);
  end;
 end until MarkerTyp=$DA;
{Rest}
 TranslateJPG:=TransferChunk(0);
end;
```

Die nachfolgende Prozedur parst eine bereits bestehende Exif-Datenstruktur (→ *4.2.2*) und entfernt ggf. das enthaltene Vorschaubild:

```
type
 ExifTag=record
  TagID,Typ: Word;
  Count,Offset: LongInt;
end;
 ExifHeader=record
  ID: array[1..6] of Char;
  Order: array[1..2] of Char;
  TagMark: Word;
  IFDOffset: LongInt;
  Anz: Word;
end;
 ByteArray=record
  case Integer of
  0: (Raw: array[0..65527] of Byte);
  1: (Hd: ExifHeader);
end;
 PByteArray=^ByteArray;

procedure HandleExif(Pos: LongInt; L: Word);
label Nochmal;
var
 Tag: ExifTag;
 P: PByteArray;
 NextIFD: LongInt;
 Anz,A,ID,W: Word;
 Motorola: Boolean;
begin
{Einlesen}
  if L>65528 then exit;
 New(P); Stream^.SeekTo(Pos);
  if BRead(P^,L)=False then exit;
{Parsen}
  with P^ do begin
  Motorola:=(Hd.Order='MM');
   if Motorola=True then Anz:=Swap(Hd.Anz) else Anz:=Hd.Anz;
  W:=16+Anz*12; Move(P^.Raw[W],NextIFD,4);
   if Motorola=True then NextIFD:=SwapLong(NextIFD);
   if NextIFD>0 then begin
      if NextIFD+8<L then L:=NextIFD+8;
    FillChar(Raw[W],4,0);
   end;
  W:=16;
```

```
Nochmal:
    for A:=1 to Anz do begin
    Move(Raw[W],Tag,12);
     if Motorola=True then with Tag do begin
      TagID:=Swap(TagID);
      Typ:=Swap(Typ);
      Count:=SwapLong(Count);
      Offset:=SwapLong(Offset);
     end;
     case Tag.TagID of
     34665: begin
              W:=Tag.Offset+6;
              Move(Raw[W],Anz,2); Inc(W,2);
               if Motorola=True then Anz:=Swap(Anz);
              goto Nochmal;
            end;
     273,288,289,322..325,34858,37500: FillChar(Raw[W],12,0);
     else if (Tag.Offset+6>L) or (Tag.Offset<0) then FillChar(Raw[W],12,0);
     end;
    Inc(W,12);
    end;
  FillChar(Raw[W],4,0);
  end;
{Ausgeben}
 ID:=$E1FF; W:=Swap(L+2);
 BlockWrite(OFile,ID,2); BlockWrite(OFile,W,2); BlockWrite(OFile,P^,L);
 Dispose(P);
end;
```

Diese Prozedur erstellt anhand ihrer Parameter einen neuen Exif-Marker (→ *4.2.2*) und gibt ihn aus:

```
procedure CreateExif(const Vendor,Device: OpenString; Jahr: Word;
    Monat,Tag,Stunde,Minute,Sekunde: Byte);
type
 ExifType=record
  TagEscape,TagID: Byte;
  TagLength: Word;
  Hd: ExifHeader;
  Tags: array[1..4] of ExifTag;
  Vendor,Device: array[1..64] of Byte;
  Aufnahmezeit: array[1..19] of Byte;
end;
var
 Schablone: ExifType;
 St: String[19];
 No,L: Byte;
begin
 FillChar(Schablone,SizeOf(Schablone),0);
 Schablone.TagEscape:=$FF;
 Schablone.TagID:=$E1;
 Schablone.TagLength:=Swap(SizeOf(Schablone)-2);
 Schablone.Hd.ID:='Exif'+#0#0;
 Schablone.Hd.Order:='II';
 Schablone.Hd.TagMark:=$2A;
 Schablone.Hd.IFDOffset:=8;
 Schablone.Hd.Anz:=0;
  if Vendor<>'' then begin
  {Daten}
    L:=length(Vendor);
     if L>64 then L:=64;
    Move(Vendor[1],Schablone.Vendor,L);
```

```
    {Tag}
      Inc(Schablone.Hd.Anz,1);
        with Schablone.Tags[Schablone.Hd.Anz] do begin
        TagID:=271;
        Typ:=2;
        Count:=L;
        Offset:=Ofs(Schablone.Vendor)-Ofs(Schablone)-10;
        end;
    end;
    if Device<>'' then begin
    {Daten}
      L:=length(Device);
        if L>64 then L:=64;
      Move(Device[1],Schablone.Device,L);
    {Tag}
      Inc(Schablone.Hd.Anz,1);
        with Schablone.Tags[Schablone.Hd.Anz] do begin
        TagID:=272;
        Typ:=2;
        Count:=L;
        Offset:=Ofs(Schablone.Device)-Ofs(Schablone)-10;
        end;
    end;
    if Jahr<>0 then begin
    {Daten}
      St:=LeadingZero(Jahr)+':'+LeadingZero(Monat)+':'+LeadingZero(Tag)+#32+
          LeadingZero(Stunde)+':'+LeadingZero(Minute)+':'+LeadingZero(Sekunde);
      Move(St[1],Schablone.Aufnahmezeit,19);
    {Tag}
      Inc(Schablone.Hd.Anz,1);
        with Schablone.Tags[Schablone.Hd.Anz] do begin
        TagID:=306;
        Typ:=2;
        Count:=L;
        Offset:=Ofs(Schablone.Aufnahmezeit)-Ofs(Schablone)-10;
        end;
    end;
  BlockWrite(OFile,Schablone,SizeOf(Schablone));
end;
```

Die folgende Funktion prüft, ob der JPEG-Kommentar das Format aus (→ *4.2.3*) aufweist. Ist das der Fall, werden die Daten geparst und an **CreateExif** übergeben:

```
function HandlePhone(const Vendor: OpenString; Comment: OpenString): Boolean;
var
  Device: String[15];
  Datum,Zeit: String[10];
  Jahr: Word;
  Sekunde,Minute,Stunde,Tag,Monat,Y: Byte;
begin
  HandlePhone:=False;
  Device:=''; Jahr:=0; Monat:=0; Tag:=0; Stunde:=0; Minute:=0; Sekunde:=0;
  Y:=Pos(#10,Comment);
    if Y=0 then exit;
  Comment:=Copy(Comment,Y+1,255);
  Y:=Pos(#10,Comment);
    if Y>0 then begin {Equipment}
      Device:=LeftStr(Comment,Y-1);
        if UppCase(LeftStr(Device,length(Vendor)))=UppCase(Vendor) then
     Device:=LTrim(Copy(Device,length(Vendor)+1,255));
```

```
    Comment:=Copy(Comment,Y+1,255);
     if Pos(#10,Comment)=11 then begin        {Datum}
       Datum:=LeftStr(Comment,10);
        if Datum[6] in ['-','.',':'] then begin        {Deutsch}
          Jahr:=Val(Copy(Datum,7,4));
          Monat:=Val(Copy(Datum,4,2));
          Tag:=Val(LeftStr(Datum,2));
        end else
          if Datum[6]='/' then begin                   {US}
            Jahr:=Val(Copy(Datum,7,4));
            Monat:=Val(LeftStr(Datum,2));
            Tag:=Val(Copy(Datum,4,2));
          end else
            if Datum[5] in ['-','.',':'] then begin    {Japan}
              Jahr:=Val(LeftStr(Datum,4));
              Monat:=Val(Copy(Datum,6,2));
              Tag:=Val(Copy(Datum,9,2));
            end;
       Comment:=Copy(Comment,12,255);
        if Pos(#10,Comment)=9 then begin           {Uhrzeit}
          Stunde:=Val(LeftStr(Comment,2));
          Minute:=Val(Copy(Comment,4,2));
          Sekunde:=Val(Copy(Comment,7,2));
        end;
     end;
   end;
  CreateExif(Vendor,Device,Jahr,Monat,Tag,Stunde,Minute,Sekunde);
  HandlePhone:=True;
end;
```