Eidgenössische Technische Hochschule Zürich

Institut für Geodäsie
und Photogrammetrie

Bericht Nr.

62

P A N D A

PASCAL Netzwerk
Datenbankverwaltungssystem

Version 1 B

André Frank

September 1982

Anmerkung des Herausgebers:

Die Beiträge, die in der Schriftenreihe Berichte des Instituts für Geodäsie und Photogrammetrie erscheinen, dienen vor allem dem Unterricht und der Dokumentation. Sie sind deshalb in erster Linie für Mitarbeiter des Instituts und für Studenten bestimmt. Einzelne Hefte können auch einem weiteren Kreis von Interessenten zur Verfügung gestellt werden. Die Auflage ist auf den besonderen Zweck des Heftes abgestimmt.

VORWORT

Im Lehrbereich Fehlertheorie und Datenverarbeitung des IGP laufen seit einiger Zeit Forschungsarbeiten, deren erstes Ziel die Realisierung eines funktionstüchtigen Gebrauchsmusters eines minimalen Landinformationssystems ist. Dieser Prototyp soll sichtbar und demonstrierbar machen, welche Vorteile ein raumbezogenes, datenbankorientiertes Landinformationssystem mit interaktiver Grafik zu bieten vermag.

Das Datenbanksystem PANDA, PASCAL-Netzwerk-Datenbankverwaltungssystem, ist nach mehrjähriger Arbeit mit dem Datenbanksystem DBMS-10, das wir auf der DEC-10 Anlage des ZIR benützt haben, entstanden; dabei konnte auf den im Umgang mit dem DBMS-10 erworbenen Kenntnissen aufgebaut werden. Wichtig ist dabei, dass PANDA fast ausschliesslich in Standard-PASCAL geschrieben ist und deshalb auch geeignet sein dürfte, auf Kleinsystemen implementiert zu werden, womit ein zweites wichtiges Ziel unserer Entwicklungsarbeit erreicht wäre.

Den in die EDV weniger Eingeweihten mag dieser Bericht zeigen, welche zentrale Rolle ein solches Datenverwaltungssystem für die Datenverwaltung und die Anwendungsprogrammierung spielt und wie flexibel in Zukunft Informationssysteme den Bedürfnissen angepasst werden können.

Diese erste Stufe des Verständnisses, mit der sich viele begnügen können, muss durch diejenigen vertieft werden, die sich für die Programmierung von fachspezifischen Anwendungen (Anwenderprogramme) interessieren. PANDA-Realisierungen entlasten sie von der Datenverwaltung!

Die Systemspezialisten, d.h. diejenigen, die an der Verbesserung der (Datenbank-) Systemprogramme arbeiten möchten, werden noch tiefer in die PANDA-Struktur eindringen müssen.

Die vorliegende Darstellung ist ein erster Wurf. Kommentare, Kritik und Verbesserungsvorschläge nehmen wir z.H. einer nächsten Version dankbar entgegen.

Ich danke Herrn Frank für diese vielleicht 'bahnbrechende' Arbeit.

Prof. R. Conzett

INHALTSVERZEICHNIS

EINLEITUNG

Aus dem Bestreben, dem PASCAL-Programmierer eine Sammlung von Hilfsmitteln für die Strukturierung von Daten in die Hand zu geben, entstand die Programmsammlung

PANDA PASCAL Netzwerk Datenbankverwaltungssystem.

Unvermutet entstand dabei ein fast vollständiges Datenbank-Verwaltungsystem, das sich stark an das CODASYL-DBTG Netzwerk-Modell anlehnt, in einzelnen Punkten aber die Möglichkeiten von PASCAL zu Vereinfachungen ausnützt.

Wichtig scheint mir, dass PANDA zu 99 % in Standard-PASCAL geschrieben ist und somit leicht auf den verschiedensten Computern eingesetzt werden kann. PANDA ist im Rahmen unserer Forschungsarbeiten zur Realisierung von Landinformationssystemen ein wichtiger Schritt zur Untersuchung der Frage nach der geometrisch-technischen Aufgaben angemessenen Datenstruktur und Datenbank. In diesem Punkt ist besonders auf die für Datenbanksysteme unübliche Art der Pufferverwaltung hinzuweisen.

Der vorliegende Bericht zur Dokumentation von PANDA zerfällt in verschiedene Teile:

A. Allgemeine Beschreibung

B. Benützeranleitung für den Anwendungsprogrammierer

C. Interne Programmdokumentation für den Unterhalt und den Weiterausbau

D. Hinweise für mögliche Weiterentwicklungen

Dieser ersten Version des Programmes und auch des Berichtes wird hoffentlich eine zweite verbesserte Form folgen. Kommentare, Kritik und Verbesserungsvorschläge wären für uns deshalb sehr wertvoll.

Den geduldigen ersten Benützern danke ich, besonders zu erwähnen ist Beat Sievers, der mit seiner aufbauenden Kritik zur Verbesserung von PANDA viel beigetragen hat und diesen Bericht redigiert hat.

A. Frank

P A N D A

PASCAL NETZWERK DATENBANKVERWALTUNGSSYSTEM

A. ALLGEMEINE BESCHREIBUNG

1. Ziel

PANDA ist ein Versuch, für die wichtigsten Leistungen eines Datenbanksystems eine sehr einfache Implementierung zu finden, die den speziellen Bedürfnissen der Verwaltung geometrischer Daten angepasst ist. PANDA wurde für die Verwaltung der Daten konzipiert, die für graphische Darstellungen benötigt werden; PANDA kann aber allgemein zur Datenverwaltung auf verschiedenen Computern verwendet werden.

PANDA ist zu 99 % in Standard-PASCAL |Jensen/Wirth| geschrieben und lässt sich daher leicht auf andere Anlagen übertragen.

Der einfache Lösungsansatz erlaubte die Implementierung in wenigen Wochen, in etwa 3500 Zeilen PASCAL-Code auf der DEC-10. Er macht PANDA auch für Mikrocomputer geeignet: Die übersetzten Programme für den 6809 Motorola Mikroprozessor (TSC-PASCAL) beanspruchen nur 15 kBytes.

PANDA wurde gegenüber kommerziellen Datenbankverwaltungssystemen im Hinblick auf die Verarbeitung geometrischer Daten erweitert um |Frank|:

- Zugriff nach Lage im Raum
- Verarbeitung von m : n Beziehungen.

2. Anforderungen an PANDA

Ein Datenbankverwaltungssystem sollte zumindest folgende Funktionen |Codd 82| erfüllen:

- Datenbeschreibung unabhängig vom Benützerprogramm
- Speicherung und Wiederfinden von Datenelementen aufgrund eines oder mehrerer Schlüssel
- Binden und Lösen von Beziehungen zwischen verschiedenen Datenelementen (z.B. Punkt X gehört zur Parzellendefinition 315)
- Zugriff auf Daten über diese Beziehungen
  . Datenstruktur unabhängig von Benützerprogramm
  . Kontrolle von Konsistenzbedingungen
  . Datensicherung
- Transaktionskonzept - entweder sind alle zusammengehörigen Aenderungen durchgeführt oder gar keine; die Daten sind immer in einem konsistenten Zustand
  . Aufzeichnungen über Aenderungen (Journaling)

Es ist nicht leicht, alle diese Anforderungen vollständig zu erfüllen. Kaum eines der heute verfügbaren Datenbanksysteme erreicht dies.

Im Laufe der Implementierung war also schrittweise zu entscheiden, welche Forderungen erfüllt werden können, wobei jeweils zwischen Aufwand (Programmierarbeit, Programmlänge und Laufzeitverhalten) und Ertrag abzuwägen ist.

Das Ziel ist, ein möglichst einfaches System zu erstellen, das nur die notwendigen Dienstleistungen enthält. Der klar modulare Aufbau muss aber den späteren Einbau zusätzlicher Funktionen erlauben. So wird die Forderung, dass ein Benützer nur für die von ihm gebrauchten Funktionen bezahlen soll, optimal erfüllt. Allenfalls müssen auch vorhandene Funktionen in Fällen, in denen sie nicht gebraucht werden, aus PANDA entfernt werden können.

PANDA bringt bei der Anwendung die für Datenbanksysteme typischen Vorteile:

- Programmunabhängige Datenstrukturen (Schema),
- klar definierte Schnittstellen zu den Daten (Data Manipulation Language).

Programme, die auf PANDA aufbauen, sollten sich daher ohne grosse Probleme auch auf andere Datenbanksysteme umstellen lassen.

PANDA ist für uns ein sehr geeignetes Test-System, um Anforderungen der geometrisch-technischen Datenverarbeitung an Datenbanksysteme praktisch abzuklären. Erweiterungen um spezielle Funktionen sind wegen der klaren Struktur sehr einfach und rasch zu bewerkstelligen. Hier können die raumbezogenen Speicherungs- und Zugriffsalgorithmen nochmals, und mit mehr Freiheit als bei der Realisierung mit DBMS-10, studiert werden.

3. Entstehung von PANDA

PANDA entstand aus der Notwendigkeit, für eine beschränkte Anwendung (Manipulation und Darstellung von geometrischen Daten des Vermessungswesens) eine Grundlage für die strukturierte Verwaltung der Daten, zumindest während der Laufzeit der Programme, zu erstellen.

Sofort zeigte sich, dass die üblicherweise in Datenbanksystemen vorhandenen Strukturierungsmethoden eine geeignete Grundlage abgeben, unabhängig davon, ob sie nach dem relationalen oder Netzwerk-Modell organisiert sind. Dies ergab die erste Schicht von PANDA (SETS), die selbständig implementiert und getestet wurde.

Aus verschiedenen Gründen ist die Speicherung der Daten in externe Dateien erwünscht:

- Programme für Aufbau und für Verarbeitung der Datenstruktur lassen sich trennen,
- Tests lassen sich einfacher durchführen, indem die gleichen strukturierten Daten immer wieder als Input benützt werden können,

- der Bedarf an Speicherplatz sinkt, was wichtig ist im Hinblick auf den Einsatz von Kleinsystemen.

Daraus ergab sich, auf der Idee der Objekt-Speicherung von SMALLTALK (Kaehler) basierend, der virtual element manager VEM.

Schliesslich zeigte sich, dass ein Suchen von Daten aufgrund bestimmter Werte erforderlich ist: die Schicht DZ (Direktzugriff) wurde angefügt. Dies kann als Hinweis gewertet werden, dass die von Datenbanksystemen angebotenen Funktionen genau den Bedürfnissen in einer grossen Zahl von Anwendungsprogrammen entspricht.

Ohne es zu beabsichtigen, entstand damit der grösste Teil eines Datenbanksystemes. Im Vergleich beispielsweise zu DEC's DBMS-10 Implementierung des CODASYL-DBTG Vorschlages fehlen vor allem folgende Funktionen:

- Aufzeichnung über Aenderung (Journaling),
- Schnelle Implementierung von Sets mit vom System verwalteter Reihenfolge,
- Sets mit Verbindung zum vorangehenden member (prior pointers),
- Steuerung der physischen Speicherung,
- Automatisches Einschliessen von member-Entitäten in bestimmte Sets,
- Sub-Schema,

die aber dank des streng schichtweisen Aufbaus bei Bedarf ohne weiteres noch eingebaut werden könnten. Bis jetzt wurde darauf verzichtet, um PANDA so klein als möglich zu halten.

Die Konzepte von PANDA entstammen weitgehend dem CODASYL-DBTG Vorschlag. Erstaunlicherweise hat aber die Verwendung von PASCAL als Sprache des Anwenderprogrammierers, zum Teil aber auch durch dessen Anwendung für die Implentierung, verschiedene Konzepte stark beeinflusst (siehe A-8, auch B-3.6, B-3.7).

4. Implementierungsüberlegungen

4.1 Datenmodell

Datenbanksysteme können intern nach zwei verschiedenen Datenmodellen aufgebaut werden: Relational oder Netzwerk. Die Wahl des internen Datenmodelles ist nicht unbedingt entscheidend für die Wahl des Datenmodelles des Benützers - im allgemeinen ist aber die Wahl des gleichen Modelles vorteilhaft.

4.1.1 Relationales Datenmodell

Das Relationale Datenmodell baut auf Tabellen auf, die je die Daten eines Types aufnehmen:

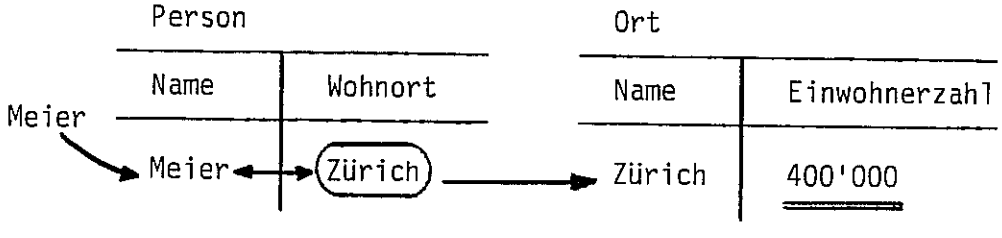
Person

Name	Wohnort
Meier	Zürich
Müller	Uster
Bänziger	Schlieren
Huber	Zürich
Peter	Zürich
Xavier	Uster

Ort

Name	Einwohnerzahl
Zürich	400'000
Uster	20'000
Schlieren	15'000
.	.
.	.
.	.

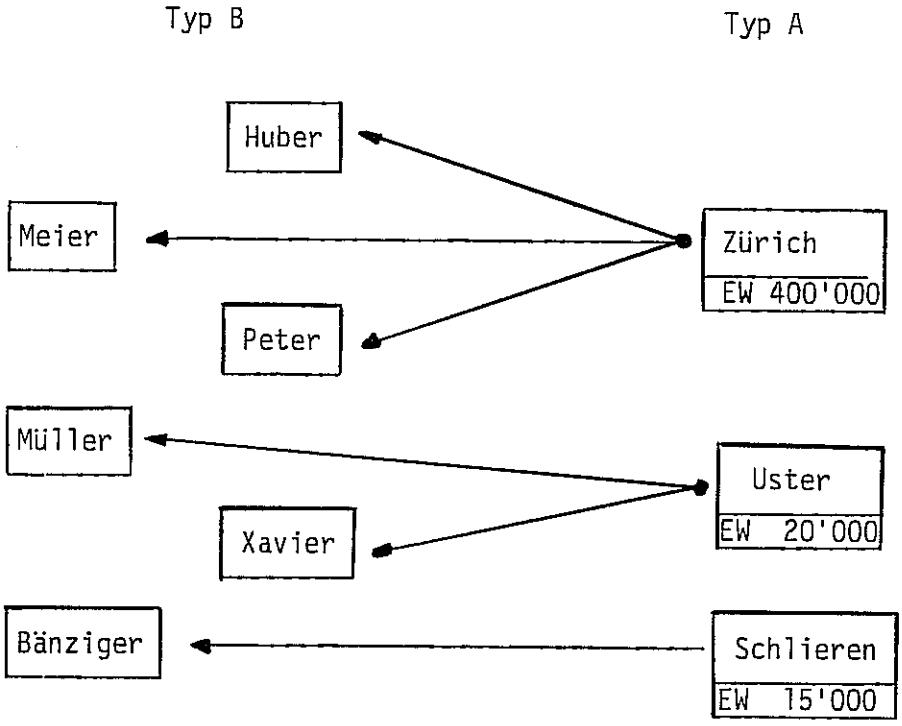
Die Beziehungen zwischen den Daten werden durch die Gleichheit der Datenwerte ausgedrückt, also für die Frage 'Wie viele Einwohner zählt der Wohnort von Herrn Meier' ergibt sich:



4.1.2 Netzwerkmodell

Das Netzwerkmodell baut auf künstlichen (nicht durch Datenwerte geschaffene) Verbindungen zwischen den Entitäten auf. Die künstlichen Verbindungen werden durch Zeiger (engl. pointer) dargestellt.

Dabei wird die 1 : n Beziehung bevorzugt. 1 : n Beziehungen sind solche, bei denen eine Entität vom Typ A (z.B. Ortschaft) mit einer Vielzahl von Entitäten vom Typ B (z.B. Einwohner) verbunden wird, anderseits aber jede Entität vom Typ B mit höchstens einer Entität vom Typ A verbunden ist. Also etwa:



Durch verfolgen des Pfeiles von 'Huber' nach 'Zürich' ergibt sich hier die Antwort auf die obige Frage direkt.

4.1.3 Evaluation

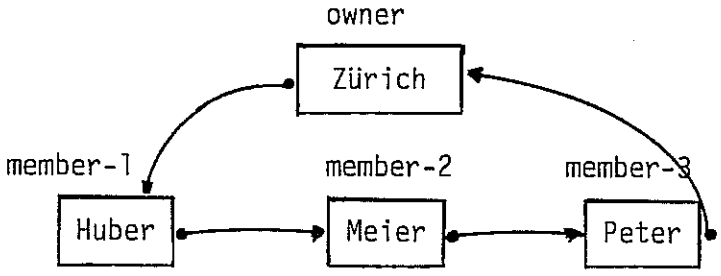
Für den Entscheid zwischen diesen beiden Datenmodellen müssen mögliche Implementierungen überlegt werden.

4.1.3.1 Implementierung des Relationalen Datenmodelles

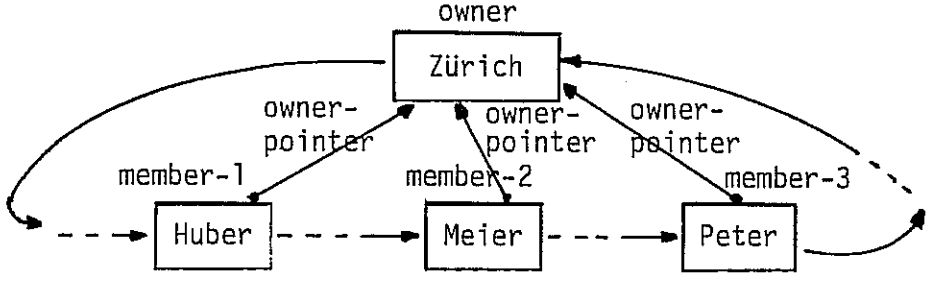
Die einfachste Implementierung erfolgt als Erweiterung des Datentypes 'file' etwa so:
Die Datenelemente werden pro Typ getrennt in Dateien abgelegt. Diese sind meist nach Schlüsselfeldern sortiert und der Zugriff wird mit Hilfe eines B*-Baumes |Denert/Franck S. 185| erleichert. Zugriffe nach anderen Werten (z.B. 'Welche Personen wohnen in Zürich') werden entweder durch invertierte Dateien oder mit sequentiellem Absuchen gelöst.

4.1.3.2 Netzwerk-Implementierung

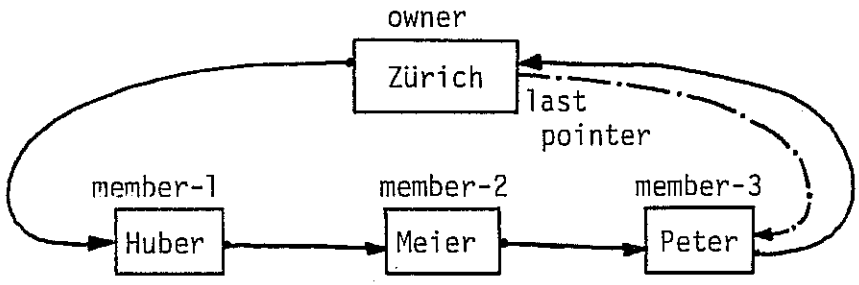
Die Datenorganisation mit Records und Zeigern bietet sich an (Listen-Strukturen Denert/Franck S. 84). Ausgehend vom einmal vorhandenen owner-record wird eine Zeigerkette aufgebaut, die alle member-records verbindet:



Werden die Listen lang und ist der Zugriff vom member (Einwohner) zum owner (Ort) häufig, so werden zusätzliche Zeiger eingeführt (owner pointer).



Muss häufig am Schluss einer Liste eingefügt werden, so erfordert dies jedesmal ein Absuchen der Liste bis zum Schluss. Um diesen Aufwand zu reduzieren, kann ein zusätzlicher Zeiger vom owner auf den letzten member eingeführt werden (last pointer).



Solche Listen können nach bestimmten Kriterien sortiert werden (z.B. Alphabet des Einwohnernamens) und erlauben dann rasche sequentielle Verarbeitung in dieser Reihenfolge.

4.1.3.3 Beurteilung

Das relationale Datenmodell ist mathematisch einfacher zu behandeln und verschiedene Eigenschaften sind heute theoretisch gut erforscht. Diese Ergebnisse können aber i.allg. auch benutzt werden, wenn die Implementierung nach einem andern Modell erfolgt.

Die relationale Implementierung bietet den Vorteil der grösseren physischen Unabhängigkeit der Daten. Daten in Tabellenform können jederzeit reorganisiert werden, ohne dass die Beziehungen zwischen den Datenelementen verlorengehen, weil diese ebenfalls als Daten gespeichert sind. Die Verbindung zwischen zwei Datenelementen, dargestellt durch übereinstimmende Werte, scheint theoretisch einfacher als die Verkettung.

Für die Implementierung ist dies aber nicht entscheidend, sondern hier muss stärker auf das Verhalten während des Betriebes geachtet werden. Dabei spielt weniger die reine (CPU-) Rechenzeit als die Anzahl der zeitraubenden Lese- und Schreiboperationen auf Massenspeicher eine Rolle.

Es scheint generell, dass ein Zugriff auf ein verbundenes Element bei der relationalen Implementierung i. allg. mehr als zwei Lese-Operationen benötigt (zuerst Transformation Wert → Adresse, nach her einlesen des an der betreffenden Stelle gespeicherten Datensatzes). Das Verfolgen einer Zeigerkette braucht pro Schritt höchstens eine Lese-Operation. Relationale Implementierungen speichern meist die einzelnen Tabellen getrennt, gegenüber den Netzwerk-Implementierungen, die Datensätze verschiedener Art nahe beieinander speichern können, so dass durch die Pufferung der Daten zwischen Massenspeicher und Programm zusätzlich physische Leseoperationen eingespart werden können.

Dies ermöglicht, oft gleichzeitig verwendete Datensätze (z.B. Haus und die zugehörigen Eckpunkt-Koordinaten) physisch so zu plazieren, dass mit einer Lese-Operation auf dem Massenspeicher mehrere interessierende Datensätze gefunden werden. Auf diesen Ueberlegungen basiert das System der raumbezogenen Speicherung |Frank 81|.

Schliesslich scheint eine Netzwerkimplementierung auch von Vorteil, wenn zu jeweils einem owner nur wenige member gehören und diese oft alle nacheinander abgearbeitet werden müssen. Dies ist bei der Anwendung für graphische Darstellungen wahrscheinlich der Fall.

Aehnliche Ueberlegungen haben auch in Datenbanksystemen zur möglichen Speicherung von Verweisen geführt. |Astrahan| |Rebsamen|

Aus diesen Gründen wurde eine Implementation nach dem Netzwerk-Datenmodell gewählt.

4.2 Programmiersprache

PANDA ist in PASCAL geschrieben, wobei zusätzliche Routinen für das Schreiben und Lesen von Random Access Files benützt werden. Damit scheint eine leichte Uebertragbarkeit auf andere Anlagen, insbesondere Mikrocomputer gesichert, die im allgemeinen Direktzugriff-Dateien in PASCAL anbieten. Die Uebertragung von der DEC-10 auf das 6809 Kleinsystem unter UniFlex beanspruchte insgesamt einen Tag.

Wir beabsichtigen, PANDA den Regeln, wie sie in |Kernighan / Plaugher| für leicht transportable PASCAL Programme vorgeschlagen sind, anzupassen.

Das strenge Typenkonzept von PASCAL wurde nicht durchbrochen, was schliesslich eine Vielzahl von Vereinfachungen erlaubte, aber eine wesentliche Einschränkung mit sich brachte:
Die vom Benützer gesehenen Record-Types müssen, damit sie von PANDA einheitlich behandelt werden können, alle als variante Teile eines Record-Types erklärt werden. Als Beispiel

```
type element_types  =  (person, ort);
     element        =  record of
                          .
                          .
                    case rt    :  element_types of
                       person  : (name:     alfa;
                                  wohnort:  alfa);
                       ort     : (ortsname: alfa);
                    end;
```

Die Dimensionierung dieses Element-Records erfolgt vom Compiler nach dem Platzbedarf des längsten Varianten-Teils, d.h. alle Records-Types brauchen gleichviel Platz (≙ starke Vereinfachungen), und zwar soviel wie der längste Record-Typ (≙ Platzverschwendung).

Diese Einschränkung bringt grosse Vorteile bei der Speicherplatzverwaltung, indem alle Elemente immer genau gleich viel Platz brauchen und daher die allereinfachsten Verfahren angewendet werden können.

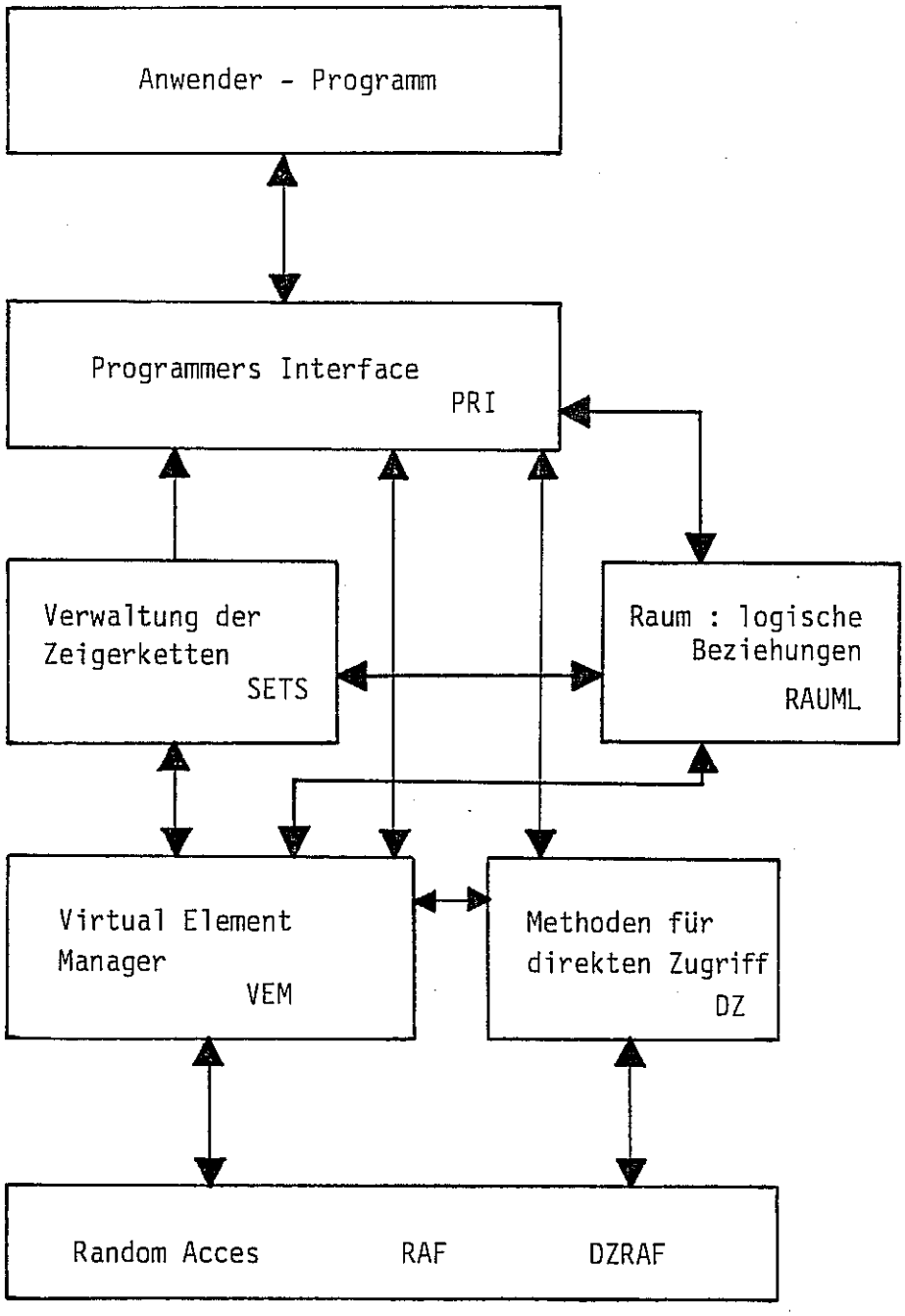
Die Nachteile des grösseren Platzbedarfes dürften bei der geplanten Anwendung weniger schwer wiegen:

- keine sehr grossen Datenmengen
- alle Record-Typen etwa ähnliche Grösse
  (eventl. aufteilen der besonders grossen).

Für die Speicherung auf dem Massenspeicher kann mit wenig Aufwand eine Verdichtung erfolgen, so dass nur die von einem Datensatz belegten Felder gespeichert werden müssen. Dies sollte erst im Zusammenhang mit der 'raumbezogenen Speicherung' (physische Clusterung) erfolgen.

4.3 Schichten von PANDA

Ein Datenbanksystem kann in verschiedene Schichten zerlegt werden. Die Implementierungsarbeit ist besonders einfach, wenn diese Schichten je als abgeschlossene Sammlung von Prozeduren erstellt werden können. Dies wurde konsequent durchgeführt.


Anwender - Programm
Programmers Interface
PRI
Verwaltung der
Zeigerketten
SETS
Raum : logische
Beziehungen
RAUML
Virtual Element
Manager
VEM
Methoden für
direkten Zugriff
DZ
Random Acces
RAF
DZRAF

5. Physische Datenspeicherung (RAF random access file)

Die Daten werden in Dateien mit Datensätzen gleicher Länge gespeichert. Dies erlaubt random access. Dazu werden Routinen verwendet, die nicht in Standard PASCAL enthalten sind, aber in vielen Fällen als Erweiterung anzutreffen sind. Auf der DEC-10 werden die entsprechenden FORTRAN Routinen aufgerufen.

5.1 Pufferverwaltung (VEM Virtual element manager)

Der Hauptspeicher ist nicht gross genug, um den ganzen Datenbestand gleichzeitig aufnehmen zu können, ein Teil ist immer auf dem Massenspeicher vorhanden. Für die Programme der höheren Schichten ist es aber einfacher, wenn sie auf Datensätze immer in der gleichen Art zugreifen können, unabhängig ob diese bereits im Hauptspeicher sind oder zuerst eingelesen werden müssen. Der virtual element manager verwaltet einen Puffer, indem die jeweils benötigten Datensätze aufbewahrt werden, so dass alle höheren Schichten einfach zugreifen können und sich nur der VEM mit der physischen Speicherung befasst.

5.2 Zeigerketten (SETS)

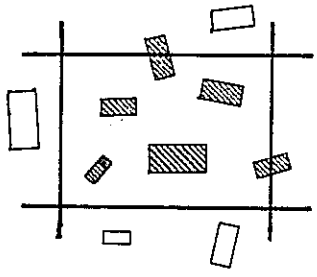
Der Aufbau und die Veränderung der Zeigerketten geschieht in einer eigenen Schicht, so dass der Anwendungsprogrammierer vom Unterhalt dieser Ketten nichts bemerkt.

5.3 Direkter Zugriff (DZ)

Ein eigener Satz von Routinen sorgt dafür, dass auf bestimmte Datensätze aufgrund eines Datenwertes zugegriffen werden kann (z.B. finde Datensatz mit Wert 'Meier' 'Karl').

5.4 Raumbezogener Zugriff (RAUML)

In einem Datenbanksystem, das vor allem Daten geometrischer Objekte verarbeitet, ist eine Zugriffsmethode nach der Lage der Objekte wichtig |Frank 81| |Tamminen 81|. Es muss möglich sein, alle Objekte innerhalb eines bestimmten Gebietes einfach zu finden, was z.B. durch eine Abfrage-Funktion wie



viereck

gib alle elemente vom typ a innerhalb viereck,
wo viereck ein achsparalleles Rechteck bildet, welches das interessierende Gebiet beschreibt.

Ist das interessierende Gebiet anders begrenzt, wird das viereck als umschreibend gewählt und nachher aus den gefundenen Objekten eine zusätzliche Auswahl getroffen.

Zur Unterstützung dieser Zugriffsmethode ist eine (logische) Gliederung des bearbeiteten Koordinatenraumes in Felder notwendig. Zur Beantwortung einer Frage wird dann zuerst entschieden, welche Felder zumindest teilweise innerhalb des viereckes liegen und nur diese müssen auf interessierende Objekte untersucht werden. Dieses Verfahren wird später durch die Steuerung der Placierung der Daten (Modul RAUMP) beschleunigt.

5.5 Benützerschnittstelle (PRI programmers interface)

In dieser Schicht werden alle Dienstleistungen von PANDA von der inneren Form des Aufrufes in die Form, wie sie dem Anwendungsprogrammierer zur Verfügung steht, umgesetzt. Hier wird einerseits für eine einheitliche Form der Aufrufe und ihrer Parameter gesorgt, anderseits könntenhier auch zusätzliche Kontrollen für die Sicherung der Datenkonsistenz eingebaut werden. Anwendungsprogrammierer müssen unbedingt vermeiden, andere Routinen aus PANDA direkt aufzurufen - dies würde sowohl die Datensicherheit und die Datenkonsistenz gefährden, als auch spätere Aenderungen in PANDA ohne Anpassung der Programme erschweren; diese Forderung lässt sich aber in PASCAL nicht programmässig erzwingen.

6. Konzept von PANDA: Datenstruktur und Operationen des Anwenderprogrammierers

6.1 Datensätze

Es können in beliebiger Reihenfolge Datensätze erzeugt, verändert, gelesen oder gelöscht werden. Jeder Datensatz erhält eine eindeutige Bezeichnung ("felp" = file element pointer), die vom System bei der Erzeugung des Datensatzes vergeben wird. Diese Bezeichnung des Datensatzes sollte aber vom Anwendungsprogrammierer höchstens zum Vergleich, ob zwei Datensätze den gleichen meinen (dann ist das Feld 'recnr' bei beiden gleich), darf aber auf keinen Fall als Wert in andern Datenfeldern gespeichert werden.

Der Inhalt eines Datensatzes muss der Deklaration dieser Element-Record-Variante entsprechen (einfache Konsistenzbedingungen über PASCAL Typendeklarationen).

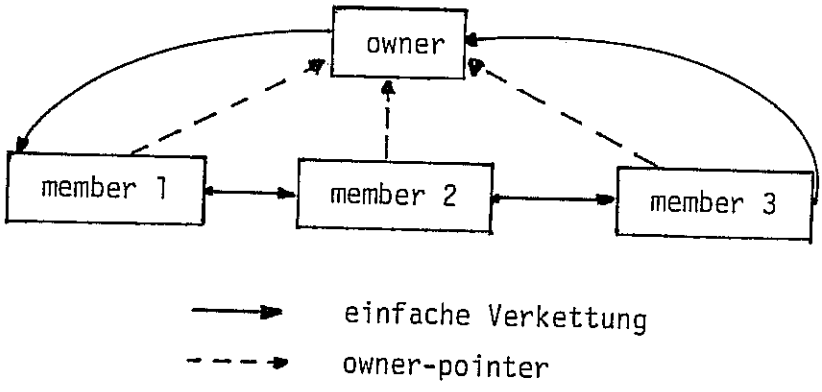
Es können mit Datensätzen die Operationen

- erzeugen
- schreiben/verändern
- lesen
- löschen

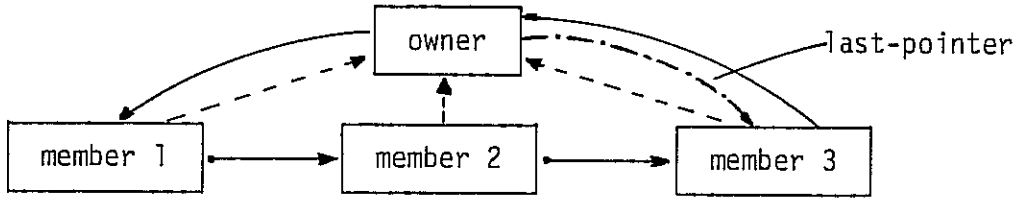
durchgeführt werden.

6.2 Beziehungen zwischen Datensätzen (SETS)

Es werden 1 : n Beziehungen mittels einfach verketteter Listen dargestellt. (Eine Erweiterung auf doppelte Verkettung wäre einfach möglich, wurde aber als im Moment unnötig weggelassen). Zusätzlich können fakultativ direkte Zeiger zum "Owner" eingeführt werden.



Muss in einem solchen Set häufig am Schluss ein member eingefügt werden, kann noch ein zusätzlicher last-pointer vom owner auf den letzten member eingefügt werden.



Datensätze können in Beziehungen

- eingefügt
- an einer bestimmten Stelle eingeführt
- aus Beziehungen herausgelöst

werden. Bevor ein Datenelement gelöscht wird, wird es automatisch aus allen Beziehungen herausgelöst, an denen es teilnimmt.

Ausgehend von einem Datenelement kann

- die Anzahl der "member" in einer Beziehung gezählt werden,
- das nächste Element oder
- der "owner" gefunden werden.

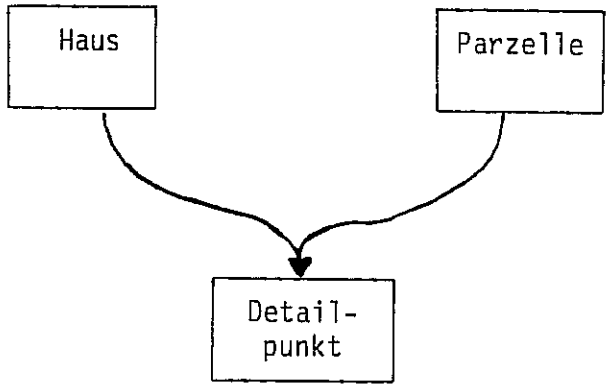
Sortierte Sets

Es kann festgelegt werden, dass die Elemente in einem Set in einer bestimmten Reihenfolge der Werte erscheinen sollen. Beim Einfügen eines neuen Elementes wird der Platz des Elementes von PANDA aufgrund der Datenwerte selbständig bestimmt und auch beim Aendern des Elementes wird überprüft, ob das Element deshalb an eine andere Stelle im Set gehört.

In solchen Sets kann, ausgehend von einem Owner, das Element mit einem bestimmten Wert gefunden werden

Sets mit alternativen Owner-Typen

In PANDA ist, in Abweichung vom CODASYL-Vorschlag, erlaubt, dass ein Set-Typ zwei verschiedene Owner-Record-Typen hat.



Der Set-Typ Umriss hat als Owner-Record-Typ Haus oder Parzelle und als Member-Record-Typ Detailpunkt. Diese Konstruktion ist wesentlich für die Erstellung generalisierter Prozeduren:

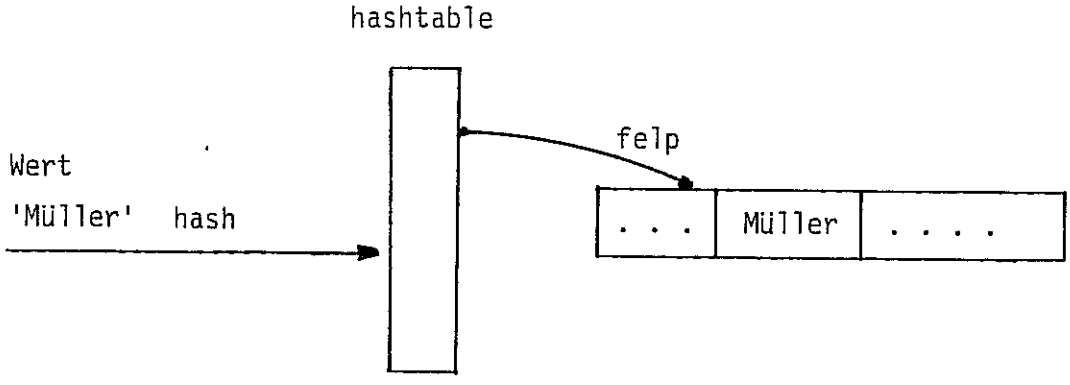
> 'Zeichne-Umriss' sollte ausführbar sein, unabhängig, ob es sich um den Umriss des Hauses oder einer Parzelle handelt.

Auf der Ebene 'Vorkommen' hat jedes Set nur einen Owner (entweder ein Haus-Record oder ein Parzelle-Record).

6.3 Direkter Zugriff auf einen Datensatz (DZ)

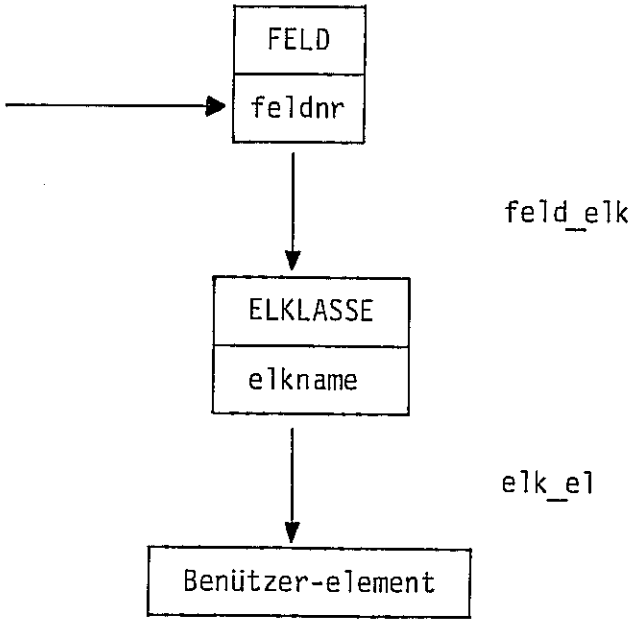
Der direkte Zugriff auf einen Datensatz aufgrund des Wertes eines Feldes (z.B. "gib Person mit Name 'Müller'") wird über eine zusätzliche hashtable |Knuth 3, S. 506| gelöst.

Der gegebene Wert wird nach einem bestimmten Verfahren in eine Adresse umgerechnet (hash-Funktion). An dieser Adresse findet man dann den eindeutigen Bezeichner (felp) des gesuchten Datensatzes.



6.4 Raumbezogener Zugriff (logische Beziehung) RAUML

Die in 5.4 erwähnte Einteilung in Felder geschieht zweistufig:



(Die übrigen Details der Implementierung werden später festgelegt.) |Frank 81|

7. Konzept von PANDA: Speicherverwaltung

7.1 Datei (RAF)

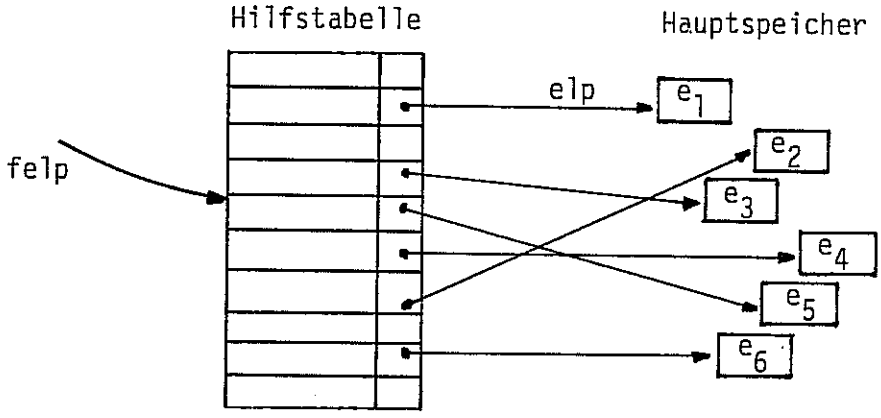
Die Daten werden zur längerfristigen Verwendung in einer Plattenspeicher-(ev. Disketten-)Datei aufbewahrt. Auf diese Datei muss direkt zugegriffen werden können (engl. random access). Dies bietet keine Schwierigkeiten, da alle Datensätze gleich lang sind. Die felp's sind direkt die Nummern der Records in der Datei.

7.2 Hauptspeicher (VEM)

Zur Verarbeitung müssen die Datensätze in den Hauptspeicher und nach Aenderungen von dort wieder zurück in die Datei verbracht werden.

Man muss annehmen, dass nicht genügend Platz im Hauptspeicher zur Verfügung steht, um alle Datensätze der Datei gleichzeitig einzulesen.

Datensätze werden nur in den Hauptspeicher eingelesen, wenn sie gebraucht werden. Neben den Datensätzen selbst muss noch eine Hilfstabelle geführt werden, die angibt, ob und gegebenenfalls wo ein Datensatz im Hauptspeicher steht.



felp = file element pointer
elp = element pointer

In dieser wird eine zusätzliche Angabe gespeichert, die beim Entscheid hilft, welcher Datensatz aus dem Hauptspeicher entfernt werden soll, um Platz für einen neu einzulesenden Datensatz zu machen und gleichzeitig zeigt, ob der Datensatz einfach überschrieben werden kann (nicht verändert = clean) oder ob er zuerst in die Datei herausgeschrieben werden muss (verändert = dirty).

Dieses Verfahren erlaubt, Transaktionen sehr einfach zu behandeln: Während der Transaktion werden nur die Daten im Hauptspeicher verändert - die Datei gibt noch den alten Zustand wieder. Wird die Transaktion abgeschlossen, werden alle veränderten Datensätze auf die Datei geschrieben. Wird die Transaktion hingegen widerrufen, so werden die veränderten Datensätze im Hauptspeicher einfach gelöscht.

Voraussetzung ist dabei, dass alle von einer Transaktion veränderten Datensätze gleichzeitig im Hauptspeicher Platz finden.

Dies entspricht einer besonderen Implementierung der bekannten Methode der Schatten-Seiten (shadow pages) vermischt mit Ideen der 'Main Storage Data base' |Reuter 81|.

7.3 Minimierung von Zugriffen auf den Massenspeicher

Mit diesem Verfahren der Pufferung der benötigten Datensätze im Hauptspeicher sollte sich die Zahl der zeitraubenden Zugriffe auf den Massenspeicher minimalisieren lassen. Insbesondere bei Abfragen, die mehrfach zur Darstellung der gleichen Daten führen, beispielsweise bei der wiederholten graphischen Darstellung während einer Aenderung, kann der grösste Teil der benötigten Daten im Hauptspeicher gehalten werden und muss somit nur einmal eingelesen werden. Diese schnelle Wiederholung von Darstellungen ist für die geplante Anwendung besonders wichtig.

Uebliche DBMS puffern die Daten meist als Datenbank-Seite, so dass neben den momentan interessierenden Datensätzen noch grosse Mengen anderer im Hauptspeicher gehalten werden.

Im ungünstigsten Fall wird durch einen einzigen benötigten Datensatz ein ganzer Seitenpuffer belegt. Da die Seitenpuffergrösse gross gewählt wird (ca. 1 K Bytes), können nur wenige Seiten gleichzeitig im Hauptspeicher verbleiben (ca. 20). Je nach Verteilung der Datensätze auf die Seiten, ergibt dies minimal 20, in sehr günstigen Fällen 300 Datensätze à 20 Bytes (d.h. 30 % verwendbare Datensätze).

Das hier dargestellte Verfahren erlaubt in den gleichen 20 k Bytes die Speicherung der 1000 Datensätze (à 20 Bytes) die zuletzt gebraucht wurden, was etwa einem ganzen Planausschnitt entspricht, wie er auf dem Bildschirm dargestellt werden kann.

Für das Lesen und Schreiben auf dem Massenspeicher wird zusätzlich auf die vom Betriebssystem vorgenommene Pufferung nach Datei-Blöcken abgestellt.

7.4 Direkter Zugriff nach gegebenem Wert (vgl. 6.3)

Die hashtable, die dazu nötig ist, wird als separate Datei erstellt und nachgeführt. Auch dort ist direkter Zugriff erforderlich.

8. Programmer Interface PRI

Die realisierte Schnittstelle des Anwenderprogrammierers zur Datenverwaltung wurde ähnlich wie beim CODASYL-DBTG Vorschlag entworfen. Die CODASYL Data Manipulation Language (DML) ist als Erweiterung von COBOL entworfen, hier mussten die Mittel von PASCAL ohne Erweiterungen ausreichen. Dies bewirkte einige Unterschiede, die im folgenden erklärt sind, wobei bereits erwähnte grundlegende Unterschiede nicht mehr beschrieben werden. Dieser Vergleich basiert auf dem CODASYL-DBTG Vorschlag |CODASYL 1973| und setzt dessen Kenntnis weitgehend voraus.

Der nachfolgende Vergleich wird erschwert durch die unterschiedlichen Implementierungen der verschiedenen Hersteller. Es ist keine vollständige Implementation bekannt. PANDA wird aber erlauben, verschiedene für die Datenkonsistenz wichtige Möglichkeiten, die m.E. nirgends implementiert sind (z.B. ON action CALL procedure), einzubauen und zu testen.

Insgesamt scheint es aber, dass ein Umsetzen der PANDA PRI-Schnittstelle auf CODASYL oder auch umgekehrt keine grossen Schwierigkeiten bereiten dürfte. Das heisst, es sollte durch Zwischenschalten weniger einfacher Routinen ein mit PANDA Befehlen geschriebenes Programm mit einem CODASYL-Datenbanksystem betrieben werden können (und umgekehrt).

Selbstverständlich geht aber bei einer solchen Umsetzung viel Leistung verloren, sie ist also nicht generell empfehlenswert.

Unabhängigkeit von physischen Zugriffspfaden

Die Zugriffsbefehle basieren nur auf den logischen Information tragenden Set-Beziehungen und funktionieren unabhängig davon, ob zur Beschleunigung Hilfsmittel (z.B. owner pointer) eingeführt wurden oder nicht. Hingegen sind natürlich die Set-Beziehungen bedeutungstragend und dürfen nicht weggelassen werden, ohne dass die Programme beeinflusst werden.

Cursor

Für Datenbank-Operationen muss meist die betroffene Entität bezeichnet werden. Nach dem CODASYL-Konzept dienen dazu eine Anzahl von 'current' Feldern, die je auf eine bestimmte Entität weisen. Diese Felder sind dem Programmierer nicht direkt zugänglich. Mit Hilfe der FIND Routinen hingegen kann er die 'current' Entitäten verändern. Die Regeln sind aber kompliziert; zudem wird in der Praxis der Datenbank-Schlüssel einer interessierenden Entität oft in einem separaten Feld gespeichert, um damit diese später wiederzufinden.

Datenaustausch

Nach dem CODASYL-Konzept wird für den Datenaustausch zwischen Datenbank und Anwenderprogramm für jeden Entitäts-Typ ein Datenfeld der entsprechenden Grösse bereitgestellt.

PASCAL enthält (im Gegensatz zu COBOL) das Konzept des Datentypes als allgemeine Beschreibung einer Daten-Art, von der nachher mehrere Exemplare als Felder mit je eigenen Namen vereinbart werden können. Weil nun alle Entitäten Varianten des gleichen Datentypes sind, erlaubt dies eine einfachere Behandlung von Cursor und Datenaustausch.

Die PANDA-Aufrufe verlangen Variablen dieses Datentypes als Parameter. Diese dienen gleichzeitig als Cursor - zeigen also auf eine bestimmte Entität, die von der Aktivität betroffen sein soll - und als Bereich für den Datenaustausch, indem sie die benötigten Daten enthalten.

Es ist Sache des Anwendungsprogrammierers, diese Variablen in genügender Zahl und mit ihrer Funktion entsprechenden Namen zu deklarieren.

Die Schnittstelle zur Datenbank vereinfacht sich damit erheblich, indem für Datenaustausch und Cursor eine einheitliche Lösung benützt wird, und die komplexen Regeln über die Verwendung und Nachführung der 'current' entfallen. Die CODASYL Funktionen FIND (plaziere current) und GET (lies Daten) sind damit in einer einzigen vereinigt.

Zusammenfassend kann hoffentlich festgestellt werden, dass die DML von PANDA die gleichen Funktionen aber in einfacherer Form anbietet.

8.1 CODASYL : FIND

Von den 7 (bei DBMS-10 nur 6) verschiedenen vorgesehenen Möglichkeiten, eine Entität in der Datenbank zu suchen, werden angeboten (vgl. B 3.10).

GIB OWNER	(CODASYL find-2 und find-4)
GIB MIT	(CODASYL find-6)
GIB NEXT	(Teil von CODASYL find-3)
SUCHE	(Teil von CODASYL find-5)
FINDE	(CODASYL find-1 und Teil von find-2)

Im wesentlichen fehlen gewisse Möglichkeiten zu Bewegungen in einem Set (FIND PRIOR, FIND LAST). Das Suchen nach Elementen mit bestimmten Werten ist in sortierten Sets vorhanden, die Implementierung ist aber für grosse Sets noch ungenügend.

In sortierten Sets dürfen in PANDA nie zwei Elemente mit den gleichen Schlüsselwerten vorkommen (CODASYL: duplicates not allowed), was vom System überprüft wird.

Daneben fehlen die Möglichkeiten, Entitäten auf Grund ihrer physischen Speicherung zu suchen. Sollen Programme von der physischen Speicherungsform unabhängig sein, dürfen solche Funktionen nicht benützt werden. Sie werden deshalb in PANDA nicht angeboten.

8.2 CODASYL : GET

Diese Funktion ist in den FIND-Befehlen schon eingeschlossen.

8.3 CODASYL : KEEP und FREE

Diese, übrigens bei DBMS-10 fehlenden Funktionen werden durch den Virtual Element Manager automatisch erfüllt.

8.4 CODASYL : IF MEMBER
IF OWNER
IF EMPTY

Diese Funktionen werden durch GIB NEXT und Test der folgenden Fehlermeldung ersetzt. Da sie aber erfahrungsgemäss selten benützt werden, hat sich eine an sich leicht mögliche selbständige Implementierung bis jetzt nicht aufgedrängt.

8.5 CODASYL : MODIFY, STORE

Diese Funktionen sind in PANDA durch AENDERE ELEMENT und NEUES ELEMENT praktisch gleich gelöst. Es fehlen die automatisch ausgelösten Aktionen für das Einfügen von Elementen in Sets (CODASYL: member is automatic).

8.6 CODASYL : MOVE

Die gewählte Lösung für die 'current' macht diesen Befehl überflüssig.

8.7 CODASYL : INSERT

Im Gegensatz zum CODASYL INSERT muss der Anwendungsprogrammierer den Platz, an dem eine Entität als member in ein Set eingefügt wird, selber bestimmten (Ausnahme: sortierte sets). Beim CODASYL Schema werden hingegen bereits gewisse Festlegungen zum voraus gemacht. Erfahrungsgemäss muss aber auch mit der CODASYL DDL dieser Platz vorgängig des Einfügens bestimmt werden.

8.8 CODASYL : DELETE, REMOVE

Diese für das Lösen von member Entitäten aus Sets und für das Löschen von Entitäten verwendeten Funktionen haben nach dem CODASYL Vorschlag z.T. sehr weitreichende, aber nicht leicht durchschaubare Folgen. PANDA bietet ähnliche Dienste in anderer, m.E. einfacher zu verstehenden Form: LOESE Element aus Set, LOESE SET ausgehend von owner auf, LOESCHE ELEMENT (enthält LOESE und LOESE SET).

9. Datenbeschreibung (Schema)

PANDA ist in PASCAL geschrieben und nicht eine Erweiterung des PASCAL-compilers wie beispielsweise PASCAL/R |Schmidt|, was bestimmte Einschränkungen mit sich bringt. Um den Aufwand gering zu halten, wurde bis jetzt darauf verzichtet, eine umfassende Datenbeschreibungssprache zu implementieren, sondern der Anwendungsprogrammierer muss die benötigten Angaben selber als PASCAL Text formulieren. Damit wurde auch das Problem der Definition einer angemessenen DDL umgangen: eine Implementation der CODASYL-DBTG DDL scheint heute kaum mehr wünschbar, da dort Angaben über logische und physische Strukturierung der Daten ungebührlich vermischt werden. Die neuen Vorschläge |z.B. Thurnherr 80| sind anderseits zu sehr auf das relationale Datenmodell ausgelegt, als dass sie ohne weiteres hätten angewendet werden können.

Es wird hier davon ausgegangen, dass die Datenstruktur nach einer einfachen Methode graphisch dargestellt wird und nachher die einzelnen Teile der Beschreibung aufgrund dieser Graphik erstellt werden können. Damit die Datenbankstruktur in den Programmen verwendet werden kann, muss sie vor der Kompilation in das Anwenderprogramm hineinkopiert werden. Diese Aufgabe übernimmt ein allgemein verwendbarer Precompiler, der in andern Dateien gespeicherten PASCAL-Text in ein Programm einkopiert (INCLUD, siehe Anhang 1.).

Im Detail ergibt sich damit:

Beim Uebersetzen von Anwendungsprogrammen müssen die Datenbeschreibungen als PASCAL Datendeklarationen wie auch die Köpfe der PANDA-Prozeduren im Programm enthalten sein. Anderseits brauchen die PANDA-Prozeduren eine zugängliche Information über die Datenstruktur (Metadaten).

Der Anwendungsprogrammierer schreibt seine Record-Beschreibung als varianten Teil der Typenvereinbarung 'Element' (vgl. B-2). Zusätzlich definiert er den Aufzählungstyp record-types mit allen Record-Namen und den Aufzählungstyp set-types, der alle set-Namen (set = Beziehungen zwischen 'owner' und 'member' records) enthält.

In einer Schema-Tabelle muss er schliesslich für jeden 'record-typ' angeben, von welchem 'Set' dieser berührt wird und ob allenfalls 'owner'-pointer notwendig sind. Schliesslich muss er noch in der 'owner'-list für jedes Set angeben, welcher record-typ 'owner' ist.

In einer anderen Tabelle ist einzutragen, auf welche 'record-typen' direkt zugegriffen werden soll. Für diese müssen die beiden Prozeduren 'dzelhash' und dzgleich' mit speziellen Anweisungen ergänzt werden.

Schliesslich kann mit Hilfe einer Anzahl Konstanten die Datenverwaltung den Anforderungen der konkreten Aufgabe angepasst werden.

Diese in PASCAL zu erstellenden Programmteile werden zusammen mit den übrigen PANDA-Routinen kompiliert und ergeben die Datei SCHEMA, die den Code der Datenverarbeitungsroutinen enthält, der nachher zu den übersetzten Anwenderprogrammen zugeladen wird (im TSC-PASCAL, das keine separate Kompilation erlaubt, müssen alle Routinen auf einmal kompiliert werden). Selbstverständlich sind damit den Anwendungsprogrammen eine Zahl von globalen Variablen von PANDA zugänglich, die auf keinen Fall verändert werden dürfen. PASCAL erlaubt m.E. kein weitergehendes information hiding.

10. Datenkonsistenz

In einem Datenbanksystem sind die Massnahmen, welche die Konsistenz der Daten sicherstellen, besonders wichtig. In PANDA sind folgende Methoden möglich und zum Teil bereits implementiert:

10.1 Statische Wertebereiche

Für jedes Datenfeld kann mit den Mitteln der PASCAL Datendeklaration der Typ und der zulässige Wertebereich festgelegt werden.

10.2 Dynamische Wertebereiche

Unter 'dynamischem Wertebereich' |Thurnherr 80| versteht man Beschränkungen des gültigen Wertebereiches eines Datenfeldes auf Werte, die (dynamisch) durch den übrigen Inhalt der Datenbank festgelegt werden. Beispielsweise darf bei einem 'Haus' das Feld 'Strassenname' nur Werte annehmen, für die eine Strasse mit diesem Namen bereits in der Datenbank existiert.

Dynamische Wertebereiche lassen sich im Netzwerkmodell durch Sets besonders gut festlegen. Es genügt, in obigem Beispiel zu verlangen, dass jedes 'Haus' in einer 'Strasse-Haus' Beziehung stehen muss. (Noch nicht implementiert)

10.3 Eindeutige Schlüssel

Schlüssel für den direkten Zugriff müssen eindeutig sein, d.h. ein bestimmter Wert, darf nur in einer Entität vorkommen. Diese Bedingung wird ständig kontrolliert.

10.4 Komplexere Konsistenzbedingungen

Der schichtweise Aufbau von PANDA gibt klare Ansatzpunkte, wo konsistenzprüfende Routinen eingefügt werden müssen. Daneben ist es auch möglich, höhere Konsistenzbedingungen durch eine darübergelegte Schicht sicherzustellen.

11. Datensicherheit

Hier ist die Transaktion das wichtigste Konzept. Die PANDA-Routinen stellen sicher, dass eine zusammenhängende Anzahl von Veränderungen der Daten nur gesamthaft oder gar nicht durchgeführt werden. Es ist damit ausgeschlossen, dass die Daten durch halb durchgeführte Aenderungen unbrauchbar werden.

Die Transaktion ist aber auch die Einheit der Aufzeichnung von Veränderungen. Solche Aufzeichnungen (Journal) können verwendet werden, um aus Kopien alter Zustände einen verloren gegangenen aktuellen Zustand wiederherzustellen bzw. den aktuellen Zustand auf einen beliebigen Zustand zurückzusetzen. (Nicht implementiert, weil für die geplante Anwendung unwichtig.)

12. Programmierstil

Bei der Programmierung von PANDA wurden verschiedene Ziele berücksichtigt.

Die Prioritäten wurden etwa wie folgt gesetzt:

Schichtenweiser Aufbau

Die Aufteilung der Gesamtaufgabe in klar abgegrenzte Schichten erhielt höchste Priorität, damit ein leicht ausbaubares, speziell für Versuche geeignetes System entsteht.

Standard PASCAL

Es wurde nach Möglichkeit nur die in |Jensen/With 74| beschriebenen Teile von PASCAL verwendet. Im Hinblick auf die Uebertragung auf das 6809 System (TSC-PASCAL) wurde zusätzlich auf Goto und Label und auf geschachtelte Prozedurdeklarationen verzichtet.

Modularisierung

Eindeutig abgrenzbare Funktionen wurden immer in eigenen procedures oder functions eingefasst.

Vermeidung von Zugriffen auf den Massenspeicher

Aus früheren Versuchen ist bekannt, dass die Antwortzeit auf Abfragen vor allem durch die für die Zugriffe auf den Massenspeicher benötigte Zeit bestimmt werden |Fagin 81|. Die Anzahl der notwendigen Zugriffe muss deshalb minimiert werden.

Bewusst wurde auf die Optimierung der Rechenzeit der CPU verzichtet. Durch Auflösung von Prozeduren und Einbau der betreffenden Anweisungen in die aufrufenden Prozeduren liesse sich etwas Zeit sparen - der Verlust an Uebersichtlichkeit der Programme würde dadurch bestimmt nicht wettgemacht.

B. BENÜTZERANLEITUNG

1. Einleitung

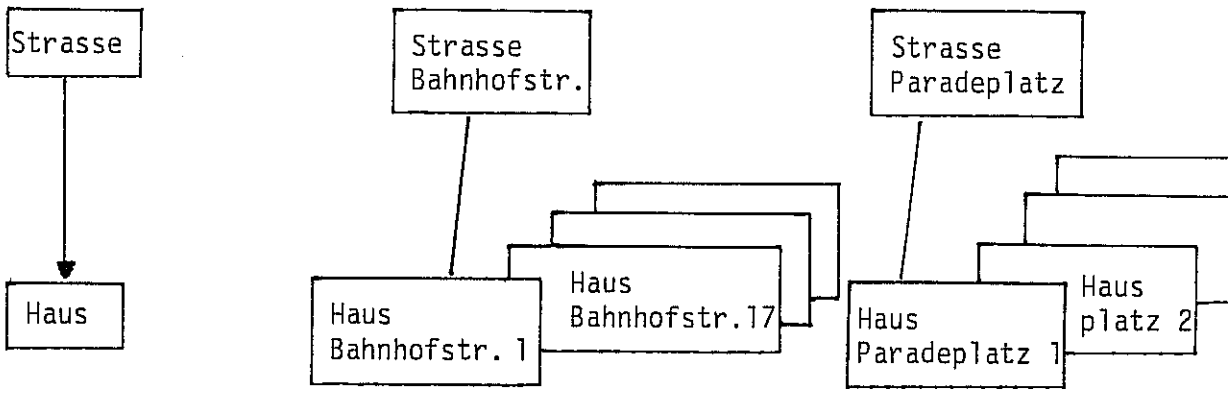
Die Benützeranleitung erklärt, wie die Datenstruktur deklariert wird (ähnlich DDL) und gibt nachher Erklärungen zu den einzelnen, dem Anwendungsprogrammierer zur Verfügung stehenden Routinen (vergleichbar DML).

1.1 Typ-Vorkommen

Im voraus sei aber auf einen Punkt aufmerksam gemacht: Es ist genau zu unterscheiden zwischen den Entitäts-Typen und den Entitäts-Vorkommen bzw. den Set-Typen und den Set-Vorkommen. Zum Entitäts-Typ 'Haus' gehören die Vorkommen 'Haus Bahnhofstr. 1' 'Haus Bahnhofstr. 2' 'Haus Paradeplatz 17' etc. Im Abschnitt 2, der von den Datenstrukturen handelt, wird im allgemeinen unter Entität ein Entitäts-Typ verstanden; auf der Ebene Datenstruktur werden die abstrakten Eigenschaften festgelegt (dass z.B. ein Haus eine Adresse hat). Im Abschnitt 3 hingegen wird vor allem die Behandlung der einzelnen Entitäts-Vorkommen besprochen; Entität bezeichnet dort im allgemeinen den ein bestimmtes Haus darstellenden Datensatz (z.B. 'Haus Bahnhofstr. 15').

Analoges gilt für die Beziehungen zwischen Entitäten. Wir legen abstrakt fest, dass z.B. Häuser an Strassen liegen: Set-Typ Strasse-Haus. Auf der Ebene der Vorkommen wird dann gespeichert, dass das Entitäts-Vorkommen 'Haus Bahnhostr. 15' im Set-Vorkommen eingefügt ist, dessen Owner das Entitäts-Vorkommen 'Strasse Bahnhofstr.' ist (Nebenbei:Set-Vorkommen werden durch das Vorkommen ihres Owners definiert).

Grafisch kann dieser Sachverhalt folgendermassen dargestellt werden:



1.2 Notwendige Dateien

Für die Verwendung von PANDA sind mehrere Dateien notwendig. Der Anwenderprogrammierer legt seine Datenbeschreibung fest, indem er folgende Dateien seiner Aufgabe anpasst:

SETS.CON	(in Anwenderprogramm einschliessen)
SETS1.TYP	(in Anwenderprogramm einschliessen)
SETS3.TYP	(in Anwenderprogramm einschliessen)
SETS.INI	
DZ.INI	

Diese werden zur Uebersetzung der Datenbank-Verwaltungsprozeduren verwendet, die ersten drei müssen auch in die Anwenderprogramme hineinkopiert werden (INCLUD).

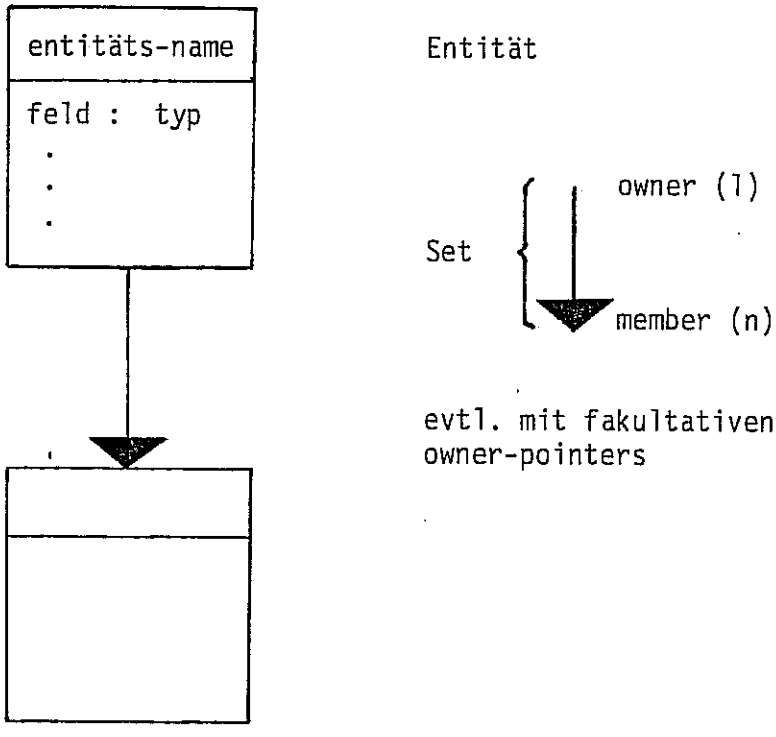
Im Anhang findet sich eine genaue Anweisung zum Vorgehen.

2. Datenstruktur

Die Eingabe der Datenstruktur ist nicht sehr komfortabel und wird vom Compiler und FILOP (siehe B 2.9.1) nur teilweise geprüft. Aufmerksamkeit ist also geboten, insbesondere weil hier entstehende Fehler ev. nur schwer aufzudecken sind (Fehlermeldungen weisen nicht direkt auf hier entstandene Fehler hin!).

2.1 Vorgehen

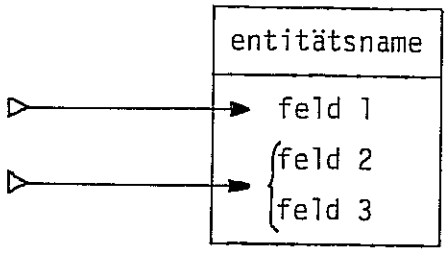
Vor der Beschreibung der Datenstruktur sollte eine grafische Uebersicht über die angestrebte Datenstruktur erstellt werden. Dazu dient folgende Symbolik:



Diese graphische Darstellung sollte unbedingt vor dem weiteren Vorgehen sauber erstellt werden und muss bei Aenderung nachgeführt werden.

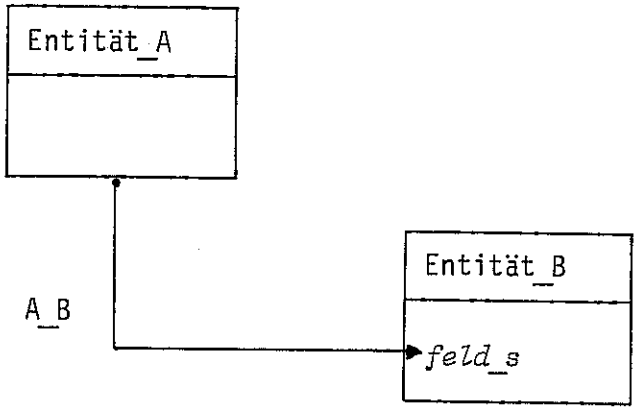
Diese Struktur ist genau zu überprüfen (Normalformen etc. siehe |Zehnder|)und mit den geplanten Zugriffen in den Anwenderprogrammen zu vergleichen.

Direkter Zugriff auf eine Entität bei gegebenem Wert eines, bzw. mehrerer Datenfelder, wird folgendermassen dargestellt:



Es kann entweder mit dem Wert von Feld 1 oder den Werten von Feld 2 und Feld 3 zusammen gesucht werden.

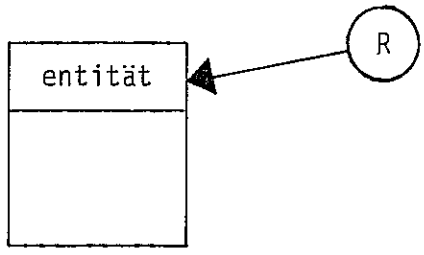
Werden Sets sortiert, so wird eine ähnliche Symbolik verwendet



Im Set A_B kann eine Member-Entität aufgrund des Wertes von Feld_s gesucht werden. Beispielsweise ist ein Vorkommen der *Entität_A* gegeben. Gesucht ist die zugehörige *Entität_B*, die im *feld_s* einen bestimmten, gegebenen Wert enthält (z.B. der Name 'Bahnhofstrasse').

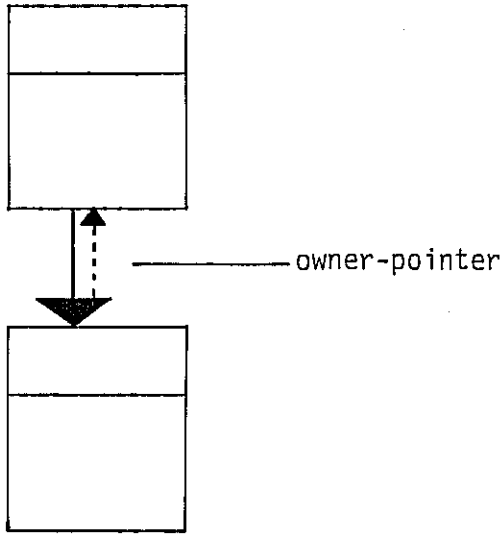
In manchen Fällen mag es praktisch sein, verschiedene Entitäten zu einem Set zusammenzufassen, von dem es nur ein einziges Vorkommen gibt. Zu diesem Zweck kann als owner-record *sys* verwendet werden. Häufig werden die Member in diesen Sets dann geordnet, um rascher zugreifen zu können.

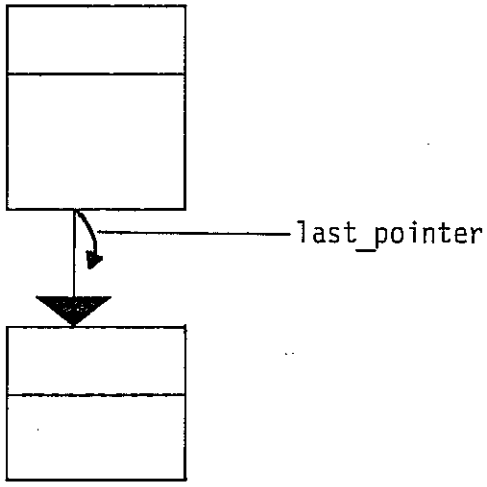
Mit einem separaten Pfeil wird angegeben, auf welche Entitäten raumbezogen zugegriffen werden soll - dies ist ein logischer

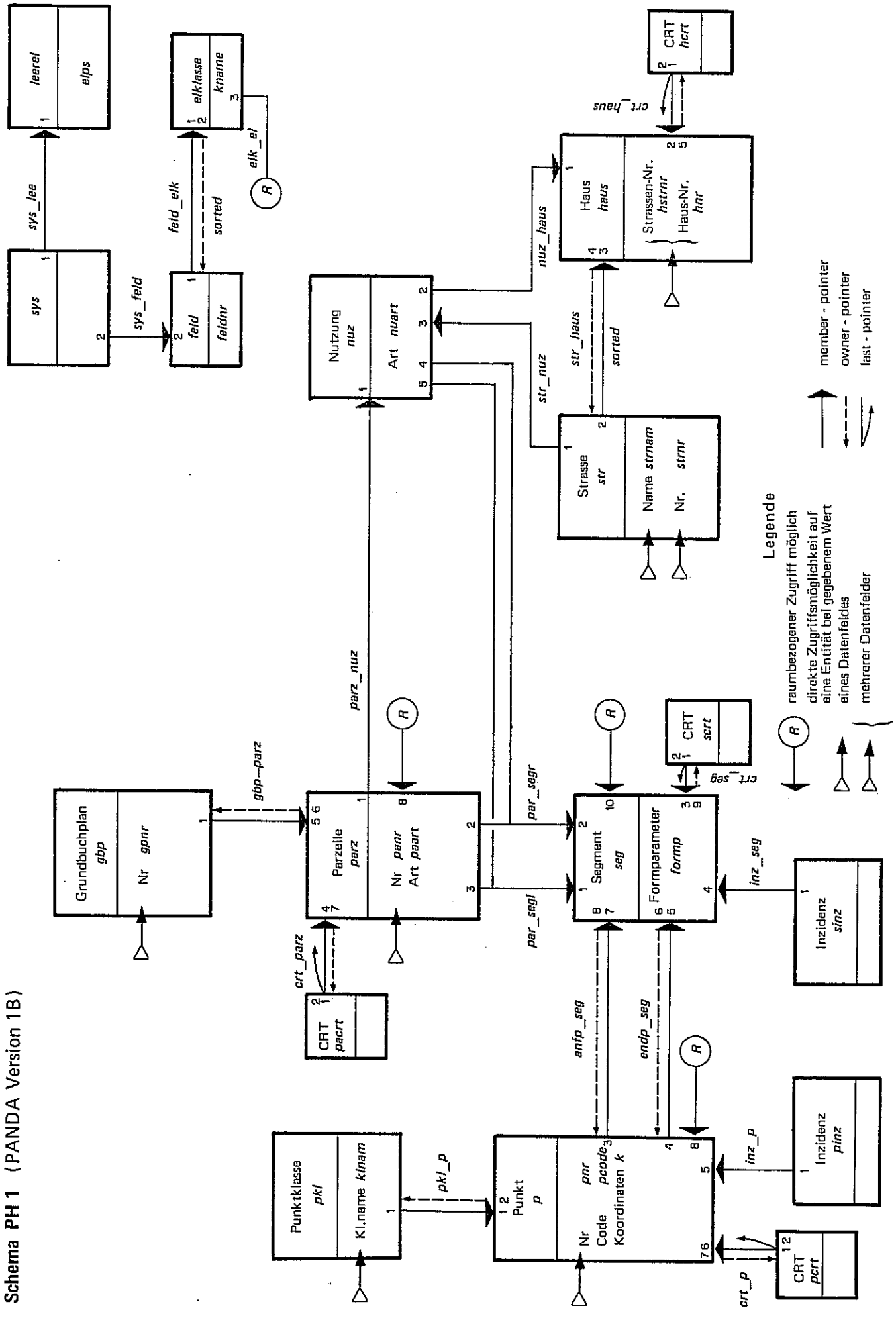


Zugriffspfad.

Zusätzlich müssen nun noch Hinweise auf zusätzliche (physische) Zugriffspfade, die nur zur Beschleunigung der Verarbeitung dienen, eingetragen werden. Das geschieht am besten auf einer Deckfolie mit folgenden Symbolen







Schema PH 1 (PANDA Version 1B)

2.2 SETS 1. TYP

Wenn die Entitätsnamen und die Set-Namen vergeben sind, können die Typen eltypes und settypes (in SETS 1. TYP) beschrieben werden.

```
(* das sind die element_typen, leerel muss am Anfang, sys am Ende stehen    *)
     eltypes = (leerel,
(* benuetzer anpassen : *)
                gbp,parz,seg,pkl,p,pinz,sinz,pcrt,pacrt,scrt,
                nuz,str,haus,hcrt,
(* panda intern *)
                mit, feld, elklasse, sys);

(* set_types sind alle moeglichen Beziehungen zwischen den Elementen:       *)
(* leers muss dabei am Anfang, sys_lee am Ende stehen                       *)
     settypes = (leers,
(* benuetzer anpassen : *)
                 gbp_parz,crt_parz,par_segl,par_segr,crt_seg,
                 inz_seg,pkl_p,anfp_seg,endp_seg,inz_p,crt_p,parz_nuz,nuz_haus,
                 str_nuz,str_haus,crt_haus,
(* panda intern *)
(* fuer rauml *)
                 feld_elk, elk_el, sys_feld,
(* fuer menge *)
                 e_mit, m_mit,
                 sys_lee);

     klassennametyp = (tp,hpp,npp,gp,hp,dp);
 (* Triangulations-, Hauptpolygon-, Nebenpolygon-, Grenz-, Hilfs-, Detail-
    punkte *)

(* Uebrige Typen, die ueberall gebraucht werden  (nicht veraendern !!)      *)
     pointer_index = 1 .. maxpointer;
     elp = ^element;
```

Für *eltypes* (= Element-Typen) muss *leerel* der erste und *sys* der letzte Wert,

für settypes muss *leers* der erste und *sys-lee* der letzte Wert sein- (Set-Namen sollten als <Owner-name> - <Member-name> gebildet werden. Damit diese kürzer als 8 Buchstaben bleiben (PASCAL DEC-10) müssen die Entitäts-Namen als dreibuchstabige Abkürzungen gewählt werden). Die übrigen Typen (*pointer_index* und *elp*) dürfen nicht verändert werden.

2.3 SETS 3. TYP

Sind alle Felder der Entitäten klar, kann auch der variante Teil des Element-Records beschrieben werden (SETS 3. TYP). Dabei müssen die Werte des Tag-Fields (d.h. des Feldes nach dem Wort *case*) genau den Werten der vorher beschriebenen eltypes entsprechen.

Für die Definition dieser Entitäten als Varianten des Element-Records sind wenige Einschränkungen zu beachten:

Alle PASCAL Konstruktionen sind zulässig, eventuell können (ebenfalls in SETS 3. TYP) zusätzliche Record-Definitionen vor der Element-Definition eingefügt werden.

Hingegen müssen Datenfelder, die für den direkten Zugriff verwendet werden sollen, entweder vom Typ integer oder vom Typ *string* sein. String ist ein *packed array* [1.. *stringlength*] *of char* und *stringlength* eine Konstante in SETS.CON.

Auch in SETS3.TYP sind noch andere Definitionen enthalten, die auf keinen Fall verändert werden dürfen.

```
(* nicht veraendern !!                                                    *)
     ft = integer;                                  (*  Fehler-typ       *)
     filename = packed array [1.. fnlength] of char;

     felp = integer;                                (* pointer im random array *)
     string = packed array [ 1.. stringlength] of char;

(* Typen des Benuetzers, die angepasst werden koennen                     *)

     element = RECORD
                                        (* fester Teil des element - Records  *)
               recnr : felp;            (* eindeutige Bezeichnung des Elements*)
               zeiger: array [pointer_index] of felp;
               wo    : viereck;        (* lage des elementes *)
                       (* varianter Teil: vom Benuetzer anzupassen  *)
               CASE eltyp : eltypes of
                 gbp   : (gpnr : integer);
                 parz  : (panr,
                          paart : string);
                 seg   : (formp : integer);
                 pkl   : (klnam : klassennametyp);
                 p     : (pnr,
                          pcode : string;
                          k     : koord);
                 nuz   : (nuart : string);
                 str   : (strnam : string;
                          strnr : integer);
                 haus  : (hstrnr,
                          hnr : integer);
                 sinz  : ();
                 pinz  : ();
                 pacrt : ();
                 scrt  : ();
                 pcrt  : ();
                 hcrt  : ();

                          (* der Rest darf nicht veraendert werden ! *)
                          (* fuer menge *)
                 mit :  ( efelp,        (* felp des verbundenen elementes *)
                          mfelp : felp);    (* felp der menge *)
                          (* fuer rauml *)
                 feld :  (feldnr : integer );
                 elklasse : (kname : eltypes ); (* aufteilung in element-
                                                   klassen *)
               leerel : (elps : array [1 .. recdatalength] of elp);
          END;
```

Für jede Variante ist die Länge zu berechnen, die Länge der längsten Variante ist festzuhalten (Konstante: *recdatalength*) in SETS.CON.

(DEC-10	1	integer	⟶	1	wort
	5	packed char	⟶	1	wort
	1	real	⟶	1	wort)
(SWT	1	integer	⟶	4	bytes
	1	char	⟶	1	byte
	1	real	⟶	8	bytes)

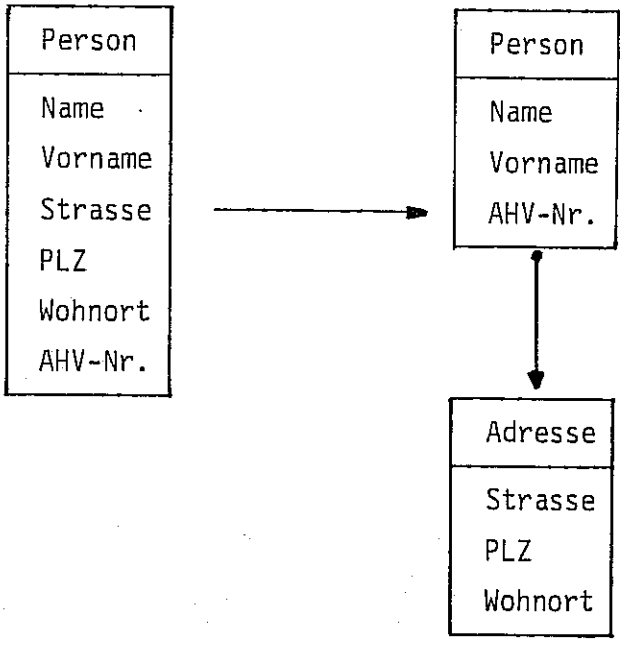
Beispiel (DEC-10)

gbp:	1	wort
parz:	2	string à 20 char =
		2 x 4 wort = 8 wort
seg:	1	wort
pkl:	1	wort
p:	2	string + 3 integer = 2 x 4 + 3 = 11 wort
nuz:	1	string = 4 wort
str:	1	string + 1 integer = 4 + 1 = 5 wort
haus:	2	integer = 2 wort

→ *recdatalength* = 11 .

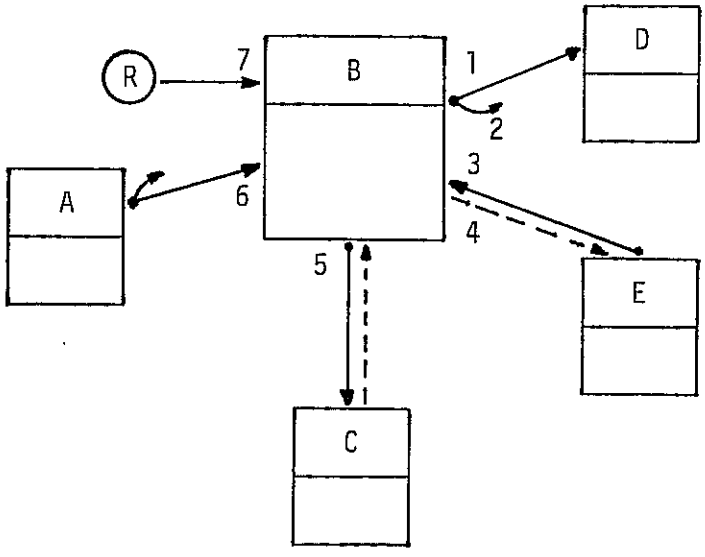
Es empfiehlt sich, wenn ein Record sehr viel länger wird als die andern, dieses aufzuteilen.

Z.B.



2.4 SETS.INI

Nun sind noch die Sets zu definieren (in SETS.INI). Dazu müssen zuerst die Pointer (vgl. A 4.1.3.2) für jedes Record verteilt werden. Dies geschieht, indem pro Entität auf der Zeichnung (am besten im Kreise herum) alle abgehenden und die eintreffenden Pfeile, (sofern nicht owner-pointer oder last-pointer) numeriert werden:



Für den Record-Typ B z.B. wird zugewiesen:

```
                record set    pointer-art
schema      [ B,   B_D,   member ]  : =  1 ;
schema      [ B,   B_D,   lastp  ]  : =  2 ;
schema      [ B,   B_E,   member ]  : =  3 ;
schema      [ B,   B_E,   owner  ]  : =  4 ;
schema      [ B,   B_C,   member ]  : =  5 ;
schema      [ B,   A_B,   member ]  : =  6 ;
schema      [ B, elk_el, member ]  : =  7 ;
```

Schliesslich muss noch für die Sets angegeben werden, welcher Record-Typ "Owner" ist:

```
owner_list  [ A_B ] : = [A] ;
owner_list  [ B_C ] : = [B] ;
owner_list  [ B_D ] : = [B] ;
owner_list  [ E_B ] : = [E] ;
```

Diejenigen Sets, für die eine bestimmte Reihenfolge der Elemente vorgeschrieben werden soll, müssen in die Menge

```
sorted_sets : = [A_B, B_C]
```

eingeschlossen werden.

Diejenigen sets, für die last-pointer notwendig sind, müssen in die Menge lastp-sets eingefügt werden:

```
lastp_sets  : = [B_D] ;
```

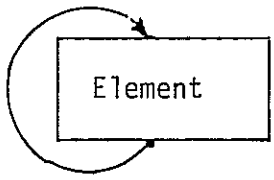
In geordneten Sets darf nur ein Member-Record-Typ vorkommen. Für die Member-Elemente ist in ELKLEI (SETS.INI) anzugeben, wann ein Element als kleiner gilt und deshalb im Set vorangehen soll.

ELPRINT

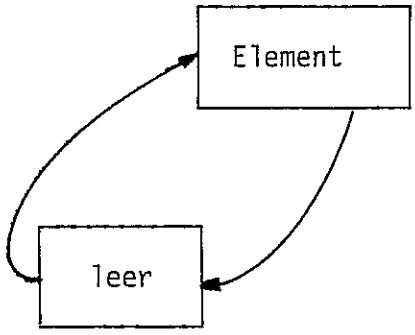
Damit Routinen zum Drucken von Elementen allgemein erstellt werden können, muss in ELPRINT für jeden Element-Typ angegeben werden, wie er gedruckt werden soll.

Gleich wie bei den meisten CODASYL-Implementierungen (nicht aber im neuesten Vorschlag) darf ein Element-Typ nicht gleichzeitig owner-Element-Typ und Member-Element-Typ sein. Tritt dieser Fall auf (z.B. Stückliste, Teil besteht aus Teilen), so muss ein zusätzlicher Element-Typ eingeschoben werden.

Nicht zulässig:



Lösung:



Diese Zuweisungen sind sehr genau zu prüfen; sie werden vom Programm nur teilweise überprüft (FILOP)!

(Die Zuteilung der Pointer-Nummern könnte automatisch erfolgen, erschwert aber eventuelle Aenderungen am Schema bei bereits bestehendem Datenbestand. Kann später ausgebaut werden!)

```
(*.....................................................................
                 .....................................................
S E T S I N I    .....................................................
.............. Initialisieren: schema[record,set,pointer-art] := pointer-nr .
..............  owner_list[set] := owner  ..................
.....................................................................
.....................................................  extern
.....................................................................*)

PROCEDURE setsini;
BEGIN
  schema[sys, sys_lee,member] := 1;
  schema [sys, sys_feld, member] := 2;
(* zusaetzlich fuer alle Elemente ein 'elkl_el'- Set *)
  schema[leerel, sys_lee, member ] := 1;
  owner_list[sys_lee] := [sys];
  owner_list[sys_feld] := [sys];
(* das sind die Sets fuer raumbezogenen Zugriff *)
  schema [feld, feld_elk, member] := 1;
  schema [feld, sys_feld, member] := 2;
  schema [elklasse, feld_elk, member] := 1;
  schema [elklasse, feld_elk, owner] := 2;
  schema [elklasse, elk_el , member] := 3;
  owner_list[feld_elk] := [feld];
  owner_list [elk_el] := [elklasse];
(* zusaetzlich fuer alle Elemente ein 'elkl_el'- Set *)
(* nun die vom Benuetzer definierten Sets *)
  schema[gbp,gbp_parz,member] := 1;
  schema[parz,parz_nuz,member] := 1;
  schema[parz,par_segr,member] := 2;
  schema[parz,par_segl,member] := 3;
  schema[parz,crt_parz,member] := 4;
  schema[parz,gbp_parz,member] := 5;
  schema[parz,gbp_parz,owner] := 6;
  schema[parz,crt_parz,owner] := 7;
  schema[parz,elk_el,member] := 8;
  schema[pacrt,crt_parz,member] := 1;
  schema[pacrt,crt_parz,lastp ] := 2;
  schema[seg,par_segl,member] := 1;
  schema[seg,par_segr,member] := 2;
  schema[seg,crt_seg,member] := 3;
  schema[seg,crt_seg,owner] := 9;
  schema[seg,inz_seg,member] := 4;
  schema[seg,endp_seg,member] := 5;
  schema[seg,endp_seg,owner] := 6;
  schema[seg,anfp_seg,member] := 7;
  schema[seg,anfp_seg,owner] := 8;
  schema[seg,elk_el,member] := 10;
  schema[scrt,crt_seg,member] := 1;
  schema[scrt,crt_seg,lastp ] := 2;
  schema[sinz,inz_seg,member] := 1;
  schema[pkl,pkl_p,member] := 1;
```

```
  schema[p,pkl_p,member] := 1;
  schema[p,pkl_p,owner] := 2;
  schema[p,anfp_seg,member] := 3;
  schema[p,endp_seg,member] := 4;
  schema[p,inz_p,member] := 5;
  schema[p,crt_p,member] := 6;
  schema[p,crt_p,owner] := 7;
  schema[p,elk_el, member] := 8;
  schema[pcrt,crt_p,member] := 1;
  schema[pcrt,crt_p,lastp ] := 2;
  schema[pinz,inz_p,member] := 1;
  schema[nuz,parz_nuz,member] := 1;
  schema[nuz,nuz_haus,member] := 2;
  schema[nuz,str_nuz,member] := 3;
  schema[nuz,par_segr,member] := 4;
  schema[nuz,par_segl,member] := 5;
  schema[str,str_nuz,member] := 1;
  schema[str,str_haus,member] := 2;
  schema[haus,nuz_haus,member] := 1;
  schema[haus,crt_haus,member] := 2;
  schema[haus,crt_haus,owner] := 5;
  schema[haus,str_haus,member] := 3;
  schema[haus,str_haus,owner] := 4;
  schema[hcrt,crt_haus,member] := 1;
  schema[hcrt,crt_haus,lastp] := 2;
  owner_list [gbp_parz] := [gbp];
  owner_list [parz_nuz] := [parz];
  owner_list [par_segr] := [parz,nuz];
  owner_list [par_segl] := [parz,nuz];
  owner_list [crt_parz] := [pacrt];
  owner_list [crt_seg] := [scrt];
  owner_list [inz_seg] := [sinz];
  owner_list [pkl_p] := [pkl];
  owner_list [anfp_seg] := [p];
  owner_list [endp_seg] := [p];
  owner_list [inz_p] := [pinz];
  owner_list [crt_p] := [pcrt];
  owner_list [nuz_haus] := [nuz];
  owner_list [str_nuz] := [str];
  owner_list [str_haus] := [str];
  owner_list [crt_haus] := [hcrt];
  lastp_sets := [crt_parz, crt_p, crt_seg, crt_haus];
  sorted_sets := [feld_elk, str_haus];
(*  bis hier alle Einfuegungen zur Schema-Definition...*)
END;
```

```
(*.....................................................................
                .....................................................
E L K L E I .........................................................
..........  vergleichen von Elementen fuer sortierte Sets ..............
..........  wahr, wenn das erste Element kleiner als das zweite ist .....
.....................................................................extern ...
.....................................................................*)

FUNCTION elklei (VAR a,b : element; s: settypes) : boolean;
BEGIN
  elklei := false;
  IF a.eltyp = b.eltyp
    THEN
      CASE a.eltyp OF
        elklasse : IF s = feld_elk
                     THEN
                       IF a.kname <= b.kname
                         THEN
                           elklei := true;
(* hier einfuegen fuer jeden member-typ in einem sortierten set; *)
        haus : IF s = str_haus
                 THEN
                   IF a.hnr <= b.hnr
                     THEN
                       elklei := true;
(* ende einfuegung *)
        otherwise : elklei := false;
      END;
END;
```

```
(*  ..........................................................
E L P R I N T
     schreibt das element
..........................................................
..............Autor:     A.Frank          1982.....................
..............geaendert: B. Sievers , 8.9.1982 ..................
..........................................................*)

PROCEDURE elprint (e :element );

VAR i : integer;
BEGIN
  WITH e DO
    BEGIN
      writeln (recnr:4,' : record - nummer; element - typ:',eltyp);
      writeln;
      write ('               ');
      FOR i:= 1 to maxpointer DO
        BEGIN
          IF i = 6
            THEN
              BEGIN
                writeln;
                write ('               ');
              END;
          IF zeiger[i] <> 0
            THEN
              write (i:2,': ', zeiger[i]:4,' ',chr(124),' ');
        END;
      writeln;
      writeln;
      CASE eltyp OF
        feld : write ('feld ' , feldnr );
        elklasse : write ('elk ', kname);
(* hier einfuegen *)
        gbp : write ('grundbuchplan ', gpnr);
        parz : write ('parzelle : ', panr, paart);
        seg : write ('segment ', formp);
        p : write ('punkt ', pnr, pcode, k.y, k.x, k.s);
        pkl : write ('punktklasse ', klnam);
(* ende einfuegung *)
        otherwise : writeln;
      END;
      writeln;
    END;
END;
```

2.5 SETS.CON (vgl. auch B 4.1)

Die maximale Zahl der Pointer in einem Element ist der Konstanten *maxpointer* und die Länge der Daten der längsten Entität der Konstanten *datalength* zuzuweisen.

Schliesslich muss für die DEC-10 Version noch die Record-Länge bestimmt werden. Sie ist

reclength = recdatalength + maxpointer + 6

Man beachte, dass *recdatalength* nur die Länge der Daten des Elementes, wie sie vom Benützer in SETS.INI eingefügt wurden, enthält. Um die Länge des Records zu bestimmen (*reclength*), ist dazu der Platz für die Zeiger (*maxpointer*) und sechs weitere von PANDA intern verwendete Felder (*recnr, wo, eltyp*) zu addieren.

```
fnlength = 10;       (*  Laenge des Filenamens                              *)
matchint = maxint;

(* vom benuetzer anzupassen :                                               *)
     stringlength = 20;  (*  maximale Anzahl Buchstaben pro String          *)
     matchstring = '********************';  (*  stringlength Sterne !!      *)
     maxpointer = 10;    (*  maximale Zahl der Pointer pro Element          *)
     recdatalength = 14; (*  Datenfelderlaenge des laengsten Records in Worten

     reclength = 30;     (*  genau maxpointer + recdatalength + 6           *)
     id_pandafile = 100000001;  (* fuer jedes neue schema erhoehen *)
```

Wird *stringlength* verändert, so muss auch die Konstante *matchstring* auf die entsprechende Länge gebracht werden.

id_pandafile

Wird ein neues Schema erstellt, so ist eine *id_pandafile* zu wählen, die möglichst noch nie benützt wurde (max. 10 Stellen DEC-10). Bei allen Aenderungen am Schema, die ein neues Laden der Daten erfordert, wird *id_pandafile* um eins erhöht.

Damit wird sichergestellt, dass mit einem Schema nur Datenfiles bearbeitet werden, die dazu passen.

2.6 DZ.INI

DZ muss auf bestimmte, durch den Anwenderprogrammierer beim Entwurf des Schemas festgelegte Felder zugreifen. Diese Felder können nicht durch Datenwerte bezeichnet werden, sondern der Anwendungsprogrammierer muss diese in den PASCAL-Programm-Text an der richtigen Stelle einfügen (DZ.INI). Die entsprechenden Routinen sind:

DZINIT — wo festgelegt wird, für welche Elementtypen Direktzugriff gewünscht wird.
dzt [eltyp]: = *true*

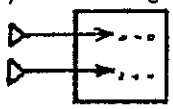
DZGLEICH — hier wird festgelegt, welche Felder verglichen werden müssen, um zu erkennen, dass das Element mit den passenden Werten gefunden wurde

DZELHASH — zeigt, welche Felder bei der Berechnung der Hashfunktion berücksichtigt werden müssen und ob es sich dabei um strings oder um integers handelt.

Werden mehrere Felder für den direkten Zugriff verwendet, so sind zwei Fälle zu unterscheiden:

1. Fall

- die Felder (max. 3) können jedes allein zur eindeutigen Identifizierung dienen.



In dzelhash wird für jedes der Felder ein Hashwert (u, v, w) berechnet

Beispiel:

```
spt: BEGIN
       u: = dzstringhash (spt.id);
       v: = dzinthash (spt.art)
     END
```

In dzgleich wird die Bedingung für Gleichheit wie folgt formuliert:

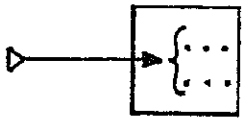
```
IF (feld_1) identisch or match)
and (feld_2) identisch or match)
and ( ... )
```

Die Felder müssen mit den gegebenen Werten übereinstimmen oder den *matchstring* bzw. *matchint* enthalten.

Beispiel:
```
spt: IF ((a.id = b.id) or (a.id = matchstring))
       and ((a.art = b.art) or (a.art = matchint))
     THEN ...
```

2. Fall

- die Felder (max. 3) dienen gemeinsam zur eindeutigen Identifizierung.



In dzelhash wird für jedes der Felder ein Hashwert (u, v, w) berechnet und dieser nachher zu einem einzigen kombiniert *dzhashkomb*)

Beispiel:
```
spt: BEGIN
       u: = dzstringhash (spt.id_1);
       v: = dzstringhash (spt.id_2);
       w: = dzstringhash (spt.id_3);
       dzhashkomb (u, v, w);
     END
```

(*dzhashkomb* ist immer so aufzurufen, auch wenn nur zwei Argumente benützt werden.)

In dzgleich heisst die Bedingung für Gleichheit nun einfach

```
IF (feld_1 identisch)
and (feld_2 identisch)
and ...
```

Alle Felder müssen mit den gegebenen Werten übereinstimmen, damit ein Element bezeichnet ist

Beispiel:
```
spt: IF (a.id_1 = b.id_1)
       and (a.id_2 = b.id_2)
       and (a.id_3 = b.id_3)
     THEN ...
```

Vorläufige Beschränkung: es ist nur ein Weg zum Direktzugriff pro Element erlaubt.

```
(*.....................................................................
..  D Z I N I T  .....................................................
.............  initialisieren der tabelle fuer dz
.............   es wird der filename uebergeben
.............   die extension dzt wird angehaengt
.....................................................extern  ....
.....................................................................*)

PROCEDURE dzinit;                  (*  (n:filename);  in forward deklariert  *)
BEGIN
  dzt[feld] := true;
(* hier einfuegen, auf welche entitaeten direkt zugegriffen werden soll     *)
  dzt[gbp] := true;
  dzt[parz] := true;
  dzt[pkl] := true;
  dzt[p] := true;
  dzt[str] := true;
  dzt[haus] := true;
END;
(*.....................................................................
                .....................................................
D Z E L H A S H  .....................................................
.............  Hashwerte (max 3) fuer ein Element berechnen  ..........
.............  muss angepasst werden !  ...............................
.............  .....................................................
........................................................ intern.......
.....................................................................*)

PROCEDURE dzelhash ;      (* (e : element; VAR u,v,w : dzhash); in forward  *)
BEGIN
  u := 0;
  v := 0;
  w := 0;
  WITH e DO
    CASE eltyp OF
     feld : u := dzinthash(feldnr);
(* hier fuer jede Entitaet mit Direktzugriff einsetzen, welche Felder fuer
                                                  den Zugriff verwendet werden  *)
     gbp  : u := dzinthash (gpnr);
     parz : u := dzstringhash (panr);
     pkl  : u := dzinthash (ord(klnam)+100);
     p    : u := dzstringhash (pnr);
     str  : BEGIN
              u := dzstringhash(strnam);
              v := dzinthash(strnr)
            END;
     haus : BEGIN
              u := dzinthash (hstrnr);
              v := dzinthash (hnr);
              dzhashkomb(u,v,w)
            END;
    END;
END;
```

```
(*.....................................................................
.....................................................................
 D Z G L E I C H .....................................................
.............. vergleichen zweier Elemente, ob sie in ihren ..............
.............. Schluesselwerten uebereinstimmen ..........................
.............. oder das erste don't care ist ..............................
.............. muss ergaenzt werden .....................................
.................................................... intern .......
.....................................................................*)

FUNCTION dzgleich;        (* (a: element; b : element): boolean; in forward  *)
BEGIN
  dzgleich := false;
  IF a.eltyp = b.eltyp      (*  wenn ungleiche element-typen, dann ungleich  *)
    THEN
      CASE a.eltyp OF
       feld : if a.feldnr = b.feldnr then dzgleich := true;

(*  hier fuer alle Records in dzt Gleichheitsbedingung einsetzen            *)
       gbp : IF a.gpnr = b.gpnr
               THEN
                 dzgleich := true;
       parz: IF a.panr = b.panr
               THEN
                 dzgleich := true;
       pkl : IF a.klnam = b.klnam
               THEN
                 dzgleich := true;
       p   : IF a.pnr  = b.pnr
               THEN
                 dzgleich := true;

       str : IF ((a.strnam = b.strnam) or (a.strnam = matchstring ))
                and ( (a.strnr = b.strnr) or (a.strnr = matchint ) )
(* was wenn beide matchstring sind ? *)
               THEN
                 dzgleich := true;
       haus: IF (a.hstrnr = b.hstrnr) and (a.hnr = b.hnr)
               THEN
                 dzgleich := true;
      END;
END;
```

2.7 RAUML.INI

Für jeden Element-Typ, auf den raumbezogen zugegriffen werden soll, müssen die Feld-Klassen in RAUML.INI festgelegt werden.

```
(*.....................................................
R I N I T
     initialisieren der variablen fuer den
     logischen raumbezug
.....................................................*)

PROCEDURE rinit;
BEGIN
  raumbezogen [parz].mink := 3;
  raumbezogen [parz].maxk := 7;
  raumbezogen [seg].mink := 5;
  raumbezogen [seg].maxk :=10;
  raumbezogen [p].mink := 8;
  raumbezogen [p].maxk := 12;
END;
```

2.8 SCHEMA.P ⟹ <schema-name> . REL (DEC-10)

Nun kann SCHEMA.P mit Hilfe von INCLUD (siehe Anhang) in SCHEMA.PAS umgewandelt werden, wobei die Dateien eingeschlossen werden. SCHEMA.PAS wird kompiliert und zusammen mit RAF.REL (von RAF.FOR) und DZRAF.REL (von DZRAF.FOR) zu einer Datei <schema-name>.REL vereinigt (utility FUDGE benützen). Der MIC-Befehl 'do' schema <schema-name>' führt das alles automatisch aus.

Diese Datei <schema-name>.REL enthält die übersetzten Routinen für die Datenverwaltung.

2.9 Hilfprogramme

2.9.1 FILOP zum Eröffnen von Dateien

Gleichzeitig wird in 'do schema <schema-name>' auch das Programm FILOP mit den Werten des aktuellen Schemas übersetzt und damit sofort die leeren Daten- und Hilfsfiles von PANDA erstellt (<schema-name>.DAB und <schema-name>.DZT). Ist dies nicht erwünscht, weil bereits gefüllte Datenfiles existieren, muss 'do schema <schema-name>,f' (f = fast) befohlen werden.

Sollen später zusätzliche Dateien zu einem Schema eröffnet werden, kann wiederum FILOP verwendet werden ('do comp filop' und 'load filop, <schema-name>/lib'),

Beim Laufen von FILOP kann der gewünschte Name der Dateien (ohne extension) eingegeben werden.

Man beachte: FILOP eröffnet neue Dateien nur, wenn keine alten mit dem gewünschten Namen existieren, alte Dateien also vorher löschen.

Läuft FILOP und Dateien existieren bereits, so sind diese nachher beschädigt und können nicht mehr verwendet werden.

FILOP führt gleichzeitig einen Test des neu definierten Schemas durch und meldet angetroffene Fehler. Schema erst benützen, wenn FILOP keine Fehler mehr antrifft!

Die Fehlermeldung 699, 255 weist darauf hin, dass Dateien mit den gewählten Namen schon irgendwo existieren (z.B. im user file directory oder in LIB:).

2.9.2 DRUCK zum Ausdrucken der gespeicherten Daten

DRUCK.P erlaubt die Ausgabe der gespeicherten Daten (inkl. pointer) auf dem Terminal und ist damit ein ideales Hilfsmittel zum Lokalisieren von Fehlern.

DRUCK.P benützt die Ausgaberoutine ELPRINT in SETS.INI.

3. Anwendungsprogrammierung

Dem Anwendungsprogrammierer stehen nur wenige einfache Routinen (vgl. 3.1) zur Manipulation seiner Daten zur Verfügung. Ueber verschiedene der vorher erstellten Dateien werden die zuvor deklarierten Datenstrukturen in sein Programm eingefügt (vgl. 3.2).

Für die Beispiele wird immer das Schema nach Figur in B 2.1 verwendet.

3.1 Einteilung der PANDA-Routinen

Die verschiedenen, dem Anwendungsprogrammierer zur Verfügung stehenden Routinen zerfallen in folgende Gruppen:

- Sitzungs-Definition:
  - initialisieren: procedure PINIT (filename),
  - abschliessen: procedure PSCHLU

- Transaktions-Definition:
  - Ende Transaktion: procedure PTEND
  - Abort Transaktion: procedure PTABO

- Aktionen

  Abfragen

  P GIBO: gib owner: function pgibo (element, owner-element settyp, fehler): boolean;

  P GIBN: gib nächstes member-element: function pgibn (element, settyp, fehler): boolean;

  P GIBM: gib mit Wert: function pgibm (element, owner-element settyp, fehler): boolean;

  P GIBIN: gib elemente innerhalb von Fenster: function pgibin (element, fenster, typen, rsown, fehler): boolean;

P SUCH: suche Entität zu gegebenem Wert:

```
function  psuch  (element, fehler): boolean;
```

Anzahl member im set:

P CARD:

```
function  pcard  (element, settyp, integer,
                  fehler): boolean;
```

P NEU: neue Entitäts-Vorkommen schaffen

```
function  pneu   (element, fehler): boolean;
```

P EIN: einfügen eines Entitätsvorkommens in ein
P EINP: Set-Vorkommen:

a) an beliebiger Stelle:

```
function  pein   (element, owner-element,
                  settyp, fehler): boolean;
```

b) nach der Entität p:

```
function peinp   (element, owner-element,
                  p-element, settyp, fehler):
                  boolean;
```

P MODY: ändere Entität

```
function pmody   (element, fehler): boolean;
```

P LES löse Entitäts-Vorkommen aus einem set-Vorkommen:

```
function ples    (element, settyp, fehler):
                  boolean;
```

P LS aufloesen eines Set-Vorkommens:

```
function pls     (owner-element, settyp,
                  fehler): boolean;
```

P LE löschen eines Entitäts-Vorkommens:

```
function ple     (element, fehler): boolean.
```

- Hilfsroutinen

PFOUT	Fehlermeldungen werden gedruckt
PFNOT	Keine Fehlermeldungen werden gedruckt
PSEE	Ein Element wird aufdatiert und ausgegeben.

3.2 Uebernahme der Datenstruktur in das Anwenderprogramm

Die nach Abschnitt 2 deklarierte Datenstruktur muss in das Anwenderprogramm eingefügt werden. Dies geschieht durch automatisches Hineinkopieren vor dem Compilieren (benütze INCLUD!). Das Anwenderprogramm muss folgende INCLUD-Befehle enthalten:

```
DEC-10
    CONST
          %sets.con
          %rauml.con
    TYPE
          %meter.typ
          %sets 1.typ
          %sets 3.typ    (der diesem Schema entsprechende File
          %rauml.typ      muss für das Kopieren verfügbar sein.)
    VAR
    (* prozeduren *)
          %raf.prh       (wegen einer Eigenheit des DEC-10 compilers
                          beim Einschliessen der FORTRAN library)
          %fdummy.pro
          %pri.prh

SWT 09/TSC-PASCAL:
    CONST
          %panda.con
    TYPE
          %panda.typ
    VAR
          %panda.var

          %panda.pro
```

Damit werden, für den Anwendungsprogrammierer wichtig, die für die Entitäten deklarierten Datenfelder etc. als PASCAL-TYPEN in seinem Programm benützbar.

Die beschriebenen Sets hingegen werden bei der DEC-10 Version nicht direkt im Programm sichtbar, sondern sind nur für die Datenmanipulations-Befehle wesentlich. Dort sind sie über das Laden zusammen mit dem File <schema-name>.REL (vgl. 2.8) verfügbar.

Warnung:
Dem Anwendungsprogrammierer sind prinzipiell verschiedene globale Variablen (besonders im Falle TSC-Pascal) zugänglich. Auf keinen Fall darf er diese durch Zuweisungen im Programm verändern!

3.3 Sitzung (PINIT & PSCHLU)

Bevor mit dem Datenbestand gearbeitet werden kann, müssen die Dateien geöffnet werden. Dazu dient der Befehl
PINIT (<file>)
wobei <file> eine Variable von Typ filename darstellt, deren erste sechs Buchstaben den Namen der Datenbasis angibt. Es ist damit möglich, mit einem Schema mehrere, unterschiedliche Datenbestände zu bearbeiten. Jeder Datenbestand besteht aus dem eigentlichen Datenfile (<name>.DAB) und der Datei für den direkten Zugriff (<name>.DZT). (Beachte 2.9.1 FILOP)

Ein Datenbestand muss immer mit demselben Schema bearbeitet werden und es müssen die beiden Dateien DAB und DZT immer zusammen behandelt werden.

Wird versucht, einen nichtexistierenden Datenbestand zu öffnen, so erfolgen Fehlermeldungen von FORLIB und der Lauf wird abgebrochen, ebenso wenn auf Daten zugegriffen wird, ohne dass der Datenbestand 'offen' ist. (Die Fehlermeldungen bestehen aus zwei Werten, die im DEC-10 FORTRAN-Manual Tabelle H-1 erklärt sind.)

Die Verarbeitung muss unbedingt mit dem Befehl

PSCHLU

abgeschlossen werden, der den Datenbestand schliesst.

Wird ein Programm abgebrochen, so müssen die Files mit dem Monitor command CLOSE geschlossen werden (DEC-10).

3.4 Transaktionen (PTEND & PTABO)

Mehrere Aktionen, Abfragen und Veränderungen, werden zu einer Transaktion zusammengefasst. Der Programmierer kann am Schluss einer Transaktion entscheiden, ob er diese durchführen will, die vorgenommenen Aenderungen also definitiv in den Datenbestand eingetragen werden sollen (PTEND), oder ob er die Aenderungen doch nicht vornehmen will, die Datenbank also wieder den Zustand vor Beginn der Aenderung wiedergeben soll (PTABO). Mit dem Aufruf einer der beiden Befehle beginnt automatisch die nächste Transaktion (die erste beginnt mit PINIT). Beim Abbruch einer Transaktion mit PTABO ist zu beachten, dass nur die Datenbank auf den Zustand zu Beginn der Transaktion zurückgesetzt wird und alle anderen Variablen erhalten bleiben. Insbesondere die Element-Variablen können nun Daten von Datenbank-Elementen enthalten, die in der Datenbank gar nicht (mehr) existieren.

Es wird deshalb empfohlen, PTABO nur am Schluss einer Prozedur zu benützen, so dass diese nachher verlassen wird und die lokalen Variablen, die eventuell falsche Werte enthalten können, verschwinden. Zusätzlich ist dem 'Aufräumen' der variablen Parameter und der globalen Variablen grosse Aufmerksamkeit zu schenken.

Wird ein Programm (z.B. durch einen Hardware-Defekt) unterbrochen, währenddem PTEND abläuft, so können die in den Dateien gespeicherten Daten inkonsistent sein. Dies wird beim nächsten Eröffnen der Datenbank (mit PINIT) entdeckt und das Programm angehalten. Dann muss entweder eine vorsorglich bereitgestellte Kopie des Datenbestandes weiterverwendet werden oder mit Hilfe des Hilfsprogramms RECOVer die Daten durch Entfernen der letzten, unvollständigen Transaktion wieder konsistent gemacht werden (noch nicht implementiert).

Das Abbrechen eines Programmes mit ^C ist so gesichert, dass die Datenkonsistenz erhalten bleibt; ^C kann also bedenkenlos verwendet werden. Die Datenbank enthält jeweils den Zustand, der der letzten noch vollendeten Transaktion entspricht.

Weil Transaktionen auch zur Verbesserung der Verwaltung des Hauptspeicher-Buffers (virtual element manager) verwendet werden, sollte auch bei reinem Abfragen nach Abschluss einiger logisch zusammenhängenden Abfragen PTEND aufgerufen werden.

Eine Transaktion darf nie mehr als maxhash (in VEM.CON) Entitäten berühren.

3.5 Ausgabe von Fehlermeldungen (PFOUT & PFNOT)

Diese beiden Prozeduren steuern die Ausgabe von Fehlermeldungen. Normalerweise wird nach jeder PANDA-Funktion, die nicht korrekt ablaufen konnte, ein erklärender Fehlertext ausgedruckt. Wird dies nicht gewünscht, so kann dies mit dem Aufruf PFNOT unterdrückt werden. Mit PFOUT wird die automatische Ausgabe wieder verlangt.

3.6 Ausgabe von Elementen

Eine im Anwenderprogramm verwendete Element-Variable wird zuerst aufdatiert, so dass sie den Zustand des Hauptspeicher-Buffers (VEM) wiedergibt und anschliessend ausgegeben (interner Aufruf der Prozeduren UPDATE und ELPRINT).

3.7 Abfragen - Veränderungen

Die Aktionen, die einem Anwenderprogrammierer zur Manipulation der Daten zur Verfügung stehen, können in zwei Gruppen geteilt werden:

Abfragen erlauben, die gespeicherten Daten zu lesen. Der Zustand der Daten wird durch Abfragen nicht verändert.

Aenderungen erlauben, neue Daten in die Datenbank einzufügen oder bestehende Daten zu ändern oder zu löschen. Der Zustand der Daten wird durch diese Operationen verändert.

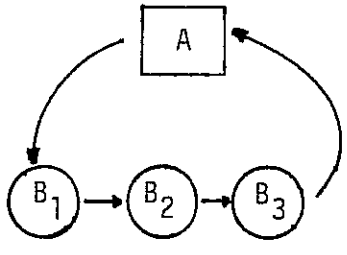
Die PANDA-Routinen garantieren, dass Abfragen nichts am Datenbestand ändern und dass Aenderungen nur Aenderungen an den Daten vornehmen, wenn sie fehlerfrei ablaufen: Zuerst wird getestet, ob die Aenderung durchgeführt werden kann, so dass Fehler vor der ersten Veränderung an den Daten erkannt werden können.

(Ausnahmen: LOESE-SET und LOESCHE-ELEMENT ergeben in Fällen, bei denen die Zeigerketten beschädigt sind, Warnungs-Text! (In diesem Fall wird man besser den Datenbestand nach einer alten Kopie wiederherstellen und überprüfen.)

3.8 Fehler-Behandlung

Bei Datenbankaufrufen können sehr viele Ausnahme-Zustände auftreten, die vom Anwenderprogramm behandelt werden müssen. Sie sind meist Hinweise auf die momentan vorhandenen Daten und nur selten Fehler des Programmierers.

Z.B.



GIB NEXT muss nach dreimaligem Aufruf 'end-of-set' signalisieren.

Die meisten Datenbank-Routinen sind deshalb als Funktionen geschrieben, die *true* sind, wenn die Funktion keinen Fehler bemerkte und sonst *false* . Die genaue Art des Fehlers kann dem Parameter *fehler* und der Dokumentation (Anhang 4.) entnommen werden.

Datenbankfunktionen werden im allgemeinen wie folgt aufgerufen:

```
if not routine (x1, x2 ... )
       then message ('fehler in routine a', fehler)
       else
            (* weitere Verarbeitung *)
```

Sind mehrere Schritte voneinander abhängig, d.h. der nächste soll nur ausgeführt werden, wenn der erste erfolgreich ablief; so kann geschrieben werden:

```
if routine_1 (x1, x2, fehler)
   then
       if routine_2 (x2, x3, fehler)
          then
              if routine_3 (x4, x5, fehler)
                 then
                       (etc.)

if fehler  <  > Ø
   then begin
            message ('fehler in xyz', fehler);
            fehler: = Ø
        end;
```

Beim Aufruf einer Routine wird automatisch der Wert von *fehler* auf Ø gesetzt. Nach mehreren Aufrufen kann also nur der Fehler der letzten ausgeführten Aktion festgestellt werden.

Werden im Programm gewisse schwerwiegende, nicht zu korrigierende Fehlerzustände bemerkt, so wird der Lauf nach einer erklärenden Mitteilung abgebrochen. Dies insbesondere in

PINIT,
PSCHLU und
PTEND,

die normalerweise ohne Fehlermeldung ablaufen.

3.9 Datenaustausch Datenbank - Anwenderprogramm

3.9.1 Variablen-Deklaration

Es ist Aufgabe des Programmierers, Datenfelder für die von der Datenbank gelieferten Daten bereitzustellen (Gegensatz zum CODASYL Konzept). Er deklariert die dazu nötigen Variablen mit

VAR <name 1> , <name 2>, . . . : element;

in seinem Hauptprogramm, bzw. in den Prozeduren und Funktionen. Eine aussagekräftige Wahl der Namen zusammen mit dem nötigen Kommentar ist angezeigt (Hinweis: die verabredeten Entitäts-Typen-Namen dürfen nicht verwendet werden!).

Beispiel:

```
VAR  start_dorf, haus_1, next_haus: element;
```

3.9.2 Datenaustausch

Beim Aufruf der Aktions-Funktionen aus PANDA (vgl. 3.1) werden die Daten der Entitäten in Variablen des Typs *element* übergeben.

Beispiel:

Nach 'Gib-owner (a-element, owner-element, fehler)'

enthält 'owner-element' die Datenwerte
der entsprechenden Entität.

Ebenso werden Input-Parameter in diesen Datenfeldern übergeben.

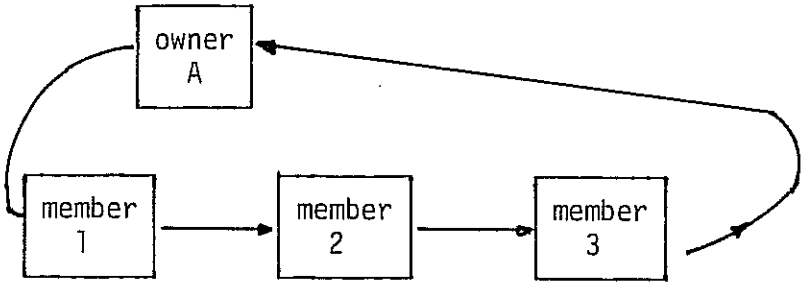
```
Beispiel:   Var   hl          :  element;
            .
            .
            begin
                  hl.eltyp    := haus;
                  hl.strasse  := 'Bahnhofstrasse . . .'
                  hl.nr       := 17

               if not pneu (hl, fehler) then ...
               end;
```

3.10 Cursor

Die Wirkung verschiedener PANDA-Aktionsroutinen ist davon abhängig,welche Entität betroffen wird.

Am deutlichsten ist dies bei 'gib-next (element_1, settyp, fehler)' zu erkennen:



Damit alle Member dieses Sets verarbeitet werden können, muss 'element_1' vor dem ersten Aufruf auf owner A zeigen. Nach dem Aufruf enthält es 'member_1'.

Beim zweiten Aufruf muss es 'member 1' enthalten, es wird 'member 2' zurückgegeben etc.

Das kann wie folgt programmiert werden:

```
          VAR   o, e :  element;
  begin

  (* o enthält owner-entität *)
  e: = o; (* zum Start wird der Variablen e der Wert des Owners
                                                    zugewiesen *)

  WHILE  gib-next (e, <set_1> , fehler)
  DO BEGIN (* e enthält der Reihe nach alle member entitäten*)
          (* verarbeite e  z.B. *)

              write       (e. <feld> );
     END;
  end;
```

gib-next hat solange den Wert *true*, bis die letzte member-Entität gefunden wurde, nachher *false* und der WHILE-loop bricht ab (beachte auch Hinweis im Anhang 2.).

e enthält in diesem Fall den Wert vor dem letzten Aufruf, d.h. die letzte member-Entität im Set.

Eine Element-Variable zeigt also auch auf ein bestimmtes Entitäts-Vorkommen in der Datenbasis (darum wird diese Funktion oft Cursor genannt). Eine gefüllte Element-Variable kann so benützt werden, um eine Stelle bei der Verarbeitung zu markieren und später an dieser Stelle (bzw. relativ zu dieser Stelle) eine weitere Verarbeitung vorzunehmen.

Diese Funktion erfüllen aber nicht etwa die Werte in den Datenfeldern des varianten Teiles des Element-records, sondern nur das Feld *recnr*, (siehe 2.3), das die eindeutige Indentifizierungsnummer des Elements enthält (sg. *felp*).

Das Anwenderprogramm darf den *felp's* (d.h. das *recnr* Feld eines Elements) auf keinen Fall verändern.

(Nebenbei: Die ebenfalls in den Variablen vom Typ *element* enthaltenen Daten der Felder *zeiger*[*i*] sollen ebenfalls nicht verändert werden. Die PANDA-Prozeduren beachten solche Aenderungen prinzipiell nicht, so dass auf diese Weise keine Fehler in die Datenbank hineingetragen werden können.)

3.11 Die einzelnen Aktionen

Die Aktionen werden in der in B-3.1 gewählten Reihenfolge behandelt.

Alle Aktionen sind functions, die den Wert *true* haben, sofern sie ohne Fehler ausgeführt wurden. Tritt ein Fehler auf, so haben sie den Wert *false*, die Art des Fehlers kann aus dem Inhalt der Fehlervariablen erkannt werden und es sind keine Aenderungen in den Daten vorgenommen worden (Ausnahmen: eventuell LOESE SET und LOESCHE ELEMENT).

GIB OWNER

```
function pgibo (element_1, owner_element, set_1,
                fehler): boolean
```

Voraussetzung. element_1 muss eine Entität enthalten, die member im set_1 Set-Typ ist.

Ergebnis: owner_element enthält die Entität, die owner im Set_Vorkommen ist, das durch element_1 und settyp ausgewählt ist.

Fehler:

1	set_1 ist kein deklariertes Set
3	felp von element_1 zeigt nicht auf gültiges element
4, 10	element_1 nicht im bezeichneten Set

GIB NEXT

```
function pgibn (element_1, set_1, fehler): boolean;
```

Voraussetzung: element_1 muss eine Entität enthalten, die im genannten set_1 enthalten ist (als member oder owner)

Ergebnis: element_1 enthält die in der Reihenfolge des Sets nachfolgende Entität. Wenn dabei der owner angetroffen wird, ist GIB NEXT *false* und element_1 enthält immer noch den letzten Member des Set (*fehler* dennoch Ø!).

Fehler

1	set_1 ist kein gültiger Set-Typ
3	element_1 ist nicht von gültigem Element-Typ
4	element_1 ist in keinem Vorkommen von set_1

GIB MIT

function gibm (element_1, owner_1, set_1, fehler): boolean;

Voraussetzungen: element_1 enthält die Datenwerte, die eine Entität dieses Types im sortierten set_1, gegeben durch den Owner, bezeichnet.

Ergebnis: element_1 enthält die gesuchte Entität

Fehler:

24, 26	set_1 ist nicht sortiert
25	keine Entität mit diesen Werten im Set

PGIBIN

function pgibin (element, ~~fenster (: viereck)~~, ~~types (: elset)~~, rsown: rsowntyp, fehler): boolean;

Voraussetzung:
- fenster enthält ein korrektes Fenster im erlaubten Bereich,
- types enthält definierte Element-Typen, die gesucht werden sollen

Ergebnis: solange pgigin wahr ist, enthält *element* je das nächste Element, das das gegebene Fenster berührt

Bemerkung: Die Werte in *rsown* dürfen nicht verändert werden. pgibn = falsch und *fehler* = Ø letztes element wurde gefunden (vgl. gibn)

~~Beim ersten Aufruf muss *rsown.status* = 0 gesetzt werden!~~

Fehler 37 rsown.status nicht im erlaubten Bereich.

SUCHE

function psuch (element_1, fehler): boolean;

Voraussetzung: element_1.eltyp enthält gültigen Element-Typ und in den für die Auswahl dieses Elementes wichtigen Feldern die Werte einer gespeicherten Entität. Sind mehrere Felder alternativ vorgesehen, so müssen die nicht benützten mit den Konstanten *matchstring* oder *matchint* gefüllt werden.

Ergebnis: element_1 enthält die gesuchte Entität

Fehler:

15	dieser Elementtyp ist nicht für den Direktzugriff vorgesehen
16	eine Entität mit diesen Datenwerten ist nicht gespeichert

CARDINALITAET

function pcard (element_1, set_1, integer_1,fehler): boolean;

Voraussetzung: element_1 ist im settyp_1 (als member oder owner)

Ergebnis: integer_1 enthält die Anzahl member im bezeichneten Set

Fehler:

1	set_1 ist kein gültiger Set-Typ
3	element_1 ist nicht von gültigem Element-Typ
4, 10	element_1 ist in keinem Vorkommen von set_1

NEUES ELEMENT

```
function pneu (element_1, fehler): boolean;
```

Voraussetzung: element_1.eltyp enthält gültigen Element-Typ
die Datenfelder sind richtig gefüllt

Ergebnis: diese Entität wird gespeichert

Fehler:

2	kein Platz (sollte nicht auftauchen)
27	element_1 hat Werte für den DZ-Schlüssel, die schon vorkommen (element_1 enthält das gespeicherte Element mit dem gleichen Schlüssel, die vorherigen Werte in element_1 sind überschrieben!)

AENDERE ELEMENT

```
function pmody (element_1, fehler): boolean;
```

Voraussetzung: element_1 enthält bereits gespeicherte Entität, deren Daten verändert werden sollen

Ergebnis: die Daten der Entität entsprechen nachher der in element_1 enthaltenen Werten (es werden alle Felder überschrieben).

Fehler:

9	element_1 hat Werte für den DZ-Schlüssel, die schon vorkommen
17	element_1.recnr zeigt nicht auf Entität vom Typ element_1.eltyp - es ist nicht zulässig, den eltyp zu ändern!
18	element_1.recnr ist nicht im gültigen Bereich

EINFUEGEN ELEMENT

```
function pein (element_1, owner_element, set_1, fehler):
                boolean;

function peinp (element_1, owner_element, position_1,
                set_1, fehler): boolean;

function peinl (element_1, owner-element, set_1, fehler):
                                                boolean;
```

Voraussetzung: element_1 enthält Entität,die Member im set_1 sein kann, aber noch nicht ist

owner_1 enthält Entität, die im set_1 owner sein kann

(Variante 2: position_1 enthält Entität, die bereits im set_1 ist, und zwar im Vorkommen, das durch owner_1 definiert ist)

pein ist auch für sortierte Sets anzuwenden;
die Position wird automatisch bestimmt.

Ergebnis: element_1 wird in set_1

Variante pein: nach dem owner

Variante peinp:nach position _1

Variante peinl:als letztes Element des Sets

eingefügt

Fehler:

1	set_1 ist kein gültiger Set-Typ
3	(irgend ein) Element ist nicht von gültigem Element-Typ
4	position_1 nicht im Set
6	element_1 ist bereits im Set
7, 10	position-1 ist in falschem Vorkommen von set_1
8	owner_1 ist nicht vom owner-typ in set_1
23	im sortierten set_1 ist bereits eine Entität mit diesen Werten vorhanden

LOESE

function ples (element_1, set_1, fehler): boolean;

Voraussetzung: element_1 ist member in einem Vorkommen von set_1

Ergebnis: element_1 ist in keinem Vorkommen von set_1

Fehler:

1	set_1 ist kein deklariertes Set
3	element_1 ist kein gültiger Element-Typ
4	element_1 ist in keinem Vorkommen von set_1
12	Zeiger nicht in Ordnung (sollte nicht vorkommen)
13	element_1 ist owner in set_1

LOESE SET

function pls (element_1, set_1, fehler): boolean;

Voraussetzung: element_1 ist owner in set_1

Ergebnis: alle bisherigen member in set_1 sind aus diesem Set entfernt

Fehler:

1	set_1 ist kein gültiger Set-Typ
3	element_1 ist kein gültiger Element-Typ
8	element_1 ist nicht owner in set_1
12	Zeiger nicht in Ordnung (sollte nicht vorkommen)

LOESCHE ELEMENT

function ple (element_1, fehler): boolean;

Voraussetzung: element_1 ist Entität

Ergebnis: Auf diese Entität kann nicht mehr zugegriffen werden sie ist aus allen Sets entfernt, in denen sie member war und alle Sets, in denen sie owner war, werden aufgelöst.

Fehler:

3	element_1 ist kein gültiger Element-Typ
12	Zeiger nicht in Ordnung (sollte nicht vorkommen)

4. Bedeutung verschiedener Konstanten

Das Verhalten von PANDA kann mit verschiedenen Konstanten dem zu lösenden Problem angepasst werden. Hier wird die Bedeutung dieser Konstanten erklärt und Hinweise für deren Festlegung gegeben.

Jede Veränderung der Konstanten bedingt eine neue Kompilation, sowohl der Anwenderprogramme als auch von SCHEMA.P!

4.1 SETS.CON (vgl. B 2.5)

maxpointer	=	maximale Anzahl von Zeigern für eine Entität (vgl. B 2.5)
stringlength	=	Anzahl Buchstaben im Typ string (für DEC-10 z.B. 5, 10, 15 ...)
fnlength	=	Länge eines filenamens mit Extension
recdatalength	=	Länge der Daten des längsten varianten Teils des *element*-Records (vgl. 2.3)
reclength	=	Totale Länge von *element* (exakt!) (*maxpointer* + *recdatalength* + 6; 6 für *element.recnr*, *wo* und *element.eltyp*)
matchstring matchint	=	Konstanten, die beim Direktzugriff verwendet werden können, um anzugeben, dass nicht nach den Werten in diesem Feld gesucht werden soll (vgl. SUCHE) *matchstring* muss die Länge stringlength haben.

```
    fnlength = 10;      (*  Laenge des Filenamens                              *)
  matchint = maxint;

(* vom benuetzer anzupassen :                                                  *)
     stringlength = 20;  (*  maximale Anzahl Buchstaben pro String            *)
     matchstring = '********************'; (*  stringlength Sterne !!          *)
     maxpointer = 10;    (*  maximale Zahl der Pointer pro Element           *)
     recdatalength = 14; (*  Datenfelderlaenge des laengsten Records in Worten *)
     reclength = 30;     (*  genau maxpointer + recdatalength + 6            *)
     id_pandafile = 100000001;  (* fuer jedes neue schema erhoehen *)
```

4.2 VEM.CON

Diese Werte haben keinen Einfluss auf die Art der Speicherung und dürfen deshalb von Lauf zu Lauf verändert werden:

heap_limit	=	Anzahl der Elemente, die gleichzeitig im Hauptspeicher aufbewahrt werden können. Dies beeinflusst die Hauptspeicher-Anforderung (heap_limit x reclength), aber auch die Zahl der Massenspeicher-Zugriffe. Muss grösser sein, als die Zahl der von der längsten Transaktion berührten Entitäten. Sollte so gross gewählt werden, dass möglichst alle häufig benützten Entitäten im Hauptspeicher behalten werden können.
maxhash1	=	die Grösse der Tabelle, für die Umsetzung von *felp* in *elp* (vgl. A 5.3), bevorzugt eine Primzahl etwa um 30 % grösser als *heap-limit*
maxhash	=	maxhash1-1
fehlermax	=	Höchste verwendete Fehlernummer in PANDA

```
fehlermax = 37;  (* Hoechste verwendete Fehlermeldung in NAMEE (PRI.PRO) *)
sysfelp = 2;  (* fuer system-rec in sets *)
maxhash = 4086; (* maxhash1 - 1 *)
maxhash1 = 4087; (* Primzahl etwas groesser (ca 30%) als heap_limit *)
hashstep = 3;
maxviertel = 1020; (* kontrolliert Warnung auf Ueberfuellung *)
heap_limit =3000; (* Anzahl maximal auf dem heap befindlicher Elemente
.....................muss ca. 70 % von maxhash sein *)
ok = -1;   (* zur Kontrolle von inkonsistenten updates *)
nok = 1;
```

Uebrige Konstanten (nicht verändern):

ok, nok	zwei Konstanten zum Kennzeichnen von DAB Dateien als in Ordnung, bzw. nicht in Ordnung weil Transaktion nicht korrekt zu Ende geführt.
sysfelp	ist der *felp* des *system*-Records

4.3 DZ.CON

dzmax	=	die Anzahl der Eintragungen im Hilfsfile für den Direktzugriff (möglichst eine Primzahl). Soll so gewählt werden, dass die Anzahl der Eintragungen bei voller Belegung der Datenbasis zwischen 1/3 und 2/3 von *dzmax* beträgt. Vorkommen von Entitäten, die über zwei Schlüssel erreichbar sind, müssen doppelt gezählt werden!
hash1, hash6	=	Konstanten für die Hash-Funktion (vgl. C 3.)
falt	=	Anzahl der aufeinander folgenden Buchstaben, die für die Hashwert-Berechnung verwendet werden.
faltbits	=	max. Zahl der verschiedenen Werte eines Buchstabens
dzleer	=	1 =Marke für ungültiger Eintrag im Hilfsfile. Muss ≠ 0 sein und darf nicht gültiger *felp* sein.
dzhashmax	=	Grösse der Tabelle für DZ-Aenderungen während einer Transaktion
id_dzfile	=	sollte bei jeder Aenderung in DZ.CON heraufgesetzt werden (vgl. *id_pandafile* in SETS.CON).

```
     dzmax = 9017;
(* sollte eine primzahl so gross sein, dass der
..................... ... dz-hash-file nur etwa zu 60 % belegt wird*)
     hash1 = 29797;(* primzahl *)
     hash6 =  3; (* primzahl, hash1 * hash6 ~ maxint*)
     hash4 = 27619;
     falt = 5;  (* laenge der faltung *)
     faltbits = 16;  (* vierbit *)
     dzleer = 1; (* markiert eine zelle als deleted , darf nicht
.................gueltiger wert sein *)
     dztaskmax = 100; (* anzahl moeglicher task fuer dz in einer
.................. transaktion *)
     id_dzfile = 1000000001;
```

4.4 RAUML.CON

feldmaxklasse = höchste Klasse der Einteilung in Felder

```
feldmaxklasse = 18;  (* maximale klassenzahl *)
```

5. Momentane Einschränkungen

5.1 Löschen von Daten

Werden Datensätze in der Datenbasis gelöscht, so werden sie dadurch für die weitere Verarbeitung unzugänglich. Der von den gelöschten Datensätzen beanspruchte Platz in der Datei kann noch nicht wieder verwendet werden und geht verloren.

6. Aenderungen am SCHEMA bei bestehender Datenbasis

In beschränktem Masse sind Aenderungen am SCHEMA möglich, auch wenn die Daten bereits geladen sind. Folgende Einschränkungen gelten dabei:

- *recdatalength* darf nicht verändert werden, d.h. Datenfelder dürfen nur soweit eingeführt werden, wie Platz vorgesehen ist.
- *maxpointer* darf nicht verändert werden, d.h. neue Sets können nur dort geschaffen werden, wo noch nicht alle Zeiger belegt sind.
- *reclength* darf nicht verändert werden.

Werden neue Datenfelder oder neue Sets eingeführt, so sind diese bei allen bestehenden Elementen leer bzw. undefiniert. In einem speziellen Programm können diese gefüllt werden.

- Neue Elemente in *eltypes* oder neue Sets in *settypes* können nicht eingefügt werden; allenfalls kann Platz vorher reserviert werden. Die Element- und Set-Bezeichnungen dürfen ohne weiteres verändert werden.

Datentypen in Elementen

Werden Datentypen von Element-Feldern geändert (z.B. von *real* auf *integer*), so müssen alle Programme, die diese Felder benützen, angepasst werden. Der Compiler gibt entsprechende Fehlermeldungen aus.

Set-Deklarationen in SETS.INI Sets können nicht weggelassen werden. Hingegen können owner-pointer (und später zu implementieren: B*-Bäume für sorted-sets) weggelassen werden, ohne dass die Programme angepasst werden müssten. Allenfalls ergibt sich aber eine, evtl. massive, Verlängerung der Laufzeit.

Auswirkung von Aenderungen am Schema auf bestehende Anwenderprogramme

Werden die in SETS1.TYP, SETS3.TYP festgelegten, das Schema betreffenden Namen (d.h. die Element-Namen, die Set-Namen und die Namen der Felder der Elemente) geändert, so müssen alle Programme, die diese Namen benützen, angepasst werden. Der Compiler zeigt dies bei einer Neuübersetzung an.

Werden neue Namen eingeführt, so sind die bestehenden Programme nicht zu ändern, wohl aber neu zu übersetzen.

Generell dürfte sich empfehlen, nach jeder Aenderung am Schema alle verwendeten Programme neu zu übersetzen und neu zu laden. (In Einzelfällen mag dies überflüssig sein, der Entscheid ist aber nicht ganz einfach.)

Die Konstanten in VEM.CON dürfen von Programmlauf zu Programmlauf verändert werden.

Die Konstanten in DZ.CON dürfen nicht verändert werden. Müssen sie erhöht werden, weil der dzhash-File überfüllt wird, so muss der <schema-name>.DZT file gelöscht werden und nach der Aenderung von DZ.CON mit 'do schema' neu erstellt werden (Achtung Datenfile .DAB vorher retten und nachher wieder zurückkopieren!). Danach kann mit dem Hilfsprogramm DZNEU (fehlt noch) der dzhash-File neu erstellt werden.

Schlussfolgerungen

Aenderungen am Schema bei bereits bestehenden Datenbeständen sind heikel und verlangen vertieftes Verständnis auch der internen Wirkungsweise von PANDA. Sie sind aber - im Gegensatz zu andern CODASYL-DBTG Datenbanken (z.B. DBMS-10) - grundsätzlich möglich.

P A N D A

PASCAL NETZWERK DATENBANKVERWALTUNGSSYSTEM

C. INTERNE PROGRAMM-DOKUMENTATION

1. Einleitung

Die Gliederung dieser allgemeinen Ueberlegungen folgt im wesentlichen dem schichtweisen Aufbau (vgl. Figur in A 4).

Es werden die in den Teilen A und B enthaltenen Erläuterungen vorausgesetzt. In gewissen Fällen ist zum besseren Verständnis auch der Programm-Text notwendig.

Die folgenden Ausführungen sind für den Anwendungsprogrammierer nicht nötig, können bei ihm aber das Verständnis für die Funktion von PANDA fördern und so wahrscheinlich zu einem effizienteren Einsatz führen. Dennoch ein Wort der Warnung: der Anwendungsprogrammierer lasse sich durch die folgenden Ausführungen nicht verwirren - für seine Anwendung von PANDA ist das in Teil A und B Gesagte massgebend und die hier erwähnten Methoden gelten nur innerhalb der PANDA Routinen, sind aber an der Schnittstelle zum Anwendungsprogrammierer (d.h. ausserhalb) im allgemeinen unsichtbar!

2. Abweichungen vom Standard PASCAL

Es wurden möglichst nur die in (Jensen/Wirth) beschriebenen standardisierten Teile von PASCAL verwendet. Zusätzlich wurde auf

- *Goto* und *labels*,
- geschachtelte Prozedur-Deklarationen

verzichtet, weil TSC-PASCAL aus Effizienz-Gründen diese nicht anbietet.

Benützte nicht-standard Prozeduren sind

- lesen und schreiben von random access files,
- *halt* zum Anhalten des Programmes nach schwerwiegenden Fehlern.

Diese sind in eigene Prozeduren eingeschlossen worden (random access), so dass sie leicht angepasst werden können, oder es kann leicht eine *procedure halt* deklariert werden, die die Anpassung an abweichende Implementierung übernehmen (TSC-PASCAL!).

3. Maschinenabhängige Konstanten

In MASCH.CON sind allenfalls folgende Werte zu deklarieren:

maxint = maximaler Wert, den ein Integer annehmen kann.

Hinweis:

Je nach Wert von *maxint* sind auch die konstanten *hash1* und *hash6* anzupassen, so dass bei den Berechnungen des Hash-Wertes kein Ueberlauf (integer overflow) vorkommen kann (siehe B 4.3).

4. RAF random access file

Ohne Dateien mit direktem Zugriff lässt sich wohl kaum ein Datenbank-System aufbauen!

Die notwendigen Funktionen (öffnen, schliessen, lesen, schreiben) sind in RAF.PRH bzw. für DZ in DZRAF.PRH zusammengefasst, die entsprechenden File-Deklarationen in RAF.VAR und DZRAF.VAR (für TSC-PASCAL).

In DEC-10 PASCAL sind random access files nicht vorgesehen. Die einfachste Lösung ergab sich durch die Verwendung der entsprechenden FORTRAN Befehle.

Die Numerierung der Records im File beginnt jeweils mit dem Record 1; ein Record ø , sofern vorhanden, wird nicht benützt.

Die Element-Records enthalten im Feld *element.recnr* ihre Nummer (*type=felp*); diese Nummer ist auch die Nummer ihres Platzes im <schema-Name>.DAB file (d.h. felps sind physische Adressen).

4.1 Services offered

RAF und DZRAF bietet den höheren Schichten folgende Dienstleistungen:

```
RAF:   RAFO (filename)  open file  <filename>
       RAFC             close file <filename>
       RAFR (felp)      read  element  <felp>
       RAFW (felp)      write element  <felp>

DZRAF: DZO  (filename)  open file  <filename>
       DZC              close file <filename>
       DZR  (dze)       read  eintrag  <dze>
       DZW  (dze)       write eintrag  <dze>
```

5. Virtual element manager (VEM)

5.1 Funktion

Der virtual element manager erlaubt den darüberliegenden Schichten einen einheitlichen Zugriff auf die Elemente (d.h.auf die records vom Typ *element*, wie in SETS3.Typ deklariert). VEM verwaltet den Puffer-Platz im Hauptspeicher, so dass die benützten Elemente möglichst selten vom Massenspeicher eingelesen werden müssen. Für die höheren Schichten ist beim Zugriff nicht erkennbar, ob das betreffende Element bereits im Hauptspeicher befindlich ist, oder ob es eingelesen werden muss. In diesem Sinn handelt es sich um virtual storage - es sieht so aus, als ob alles im Hauptspeicher wäre.

5.2 Funktionsweise

Auf dem *heap* wird für maximal *heap_limit* (in VEM.CON) Elemente Platz geschaffen. In diese Element-Puffer werden auf Wunsch (Funktion VR (felp) oder VW (felp)) die durch ihre Nummer (=felp) bezeichneten Elemente eingelesen und der aufrufenden PANDA-internen Routine ein *pointer* (*elp=element pointer*) zurückgegeben.

Werden mehr Elemente verlangt, als Platz vorhanden ist, so werden die am längsten nicht verwendeten Elemente ersetzt (least recently used strategy).

Am Ende jeder Transaktion (TEND) werden die geänderten Elemente auf den File geschrieben, bzw. gelöscht (TABO).

Um rasch überprüfen zu können, ob ein Element bereits im Puffer ist, wird eine Hash-Methode verwendet.

5.3 Services offered

VEM bietet den höheren Schichten folgende Funktionen an:

VINIT
VENDE
VR (felp) → elp
VW (felp) → elp
VD (felp) zum Löschen eines Elementes
VNEW → felp
TEND
TABO

VINIT muss als erstes aufgerufen werden, um die Dateien zu öffnen und die lokalen Variablen von VEM zu initialisieren. VENDE schliesst die Dateien am Schluss wieder.
TEND bzw. TABO schliesst eine Transaktion und beginnt eine neue (die erste beginnt mit VINIT!).

Alle höheren Schichten von PANDA brauchen einheitlich an ihren Schnittstellen immer *felps*, um auf Elemente zu zeigen, im innern *elps*, um auf die Daten der Elemente zugreifen zu können. Die Umwandlung von *felp* in *elp* erfolgt mittels *vr* und *vw* von VEM - und zwar muss *vr* gebraucht werden, wenn das betreffende Element nur gelesen werden soll (R=read), *vw*, wenn die Daten des betreffenden Elementes auch geändert werden sollen (W=write).

vnew dient dazu, ein neues Element zu schaffen.

5.4 Services used

VEM benützt die Funktionen von RAF.

5.5 CONT

Das erste Record des Datenfiles wird dazu benützt, gewisse Daten über den File zu speichern.

CONT^ zeigt auf dieses Element.

cont_dirty ist *true*, wenn innerhalb einer Transaktion CONT^ verändert wurde und am Schluss neu geschrieben werden muss.

In *cont^.recnr* wird die Adresse des nächsten freien Elementes im DAB-File gespeichert.

Während der Ausführung von TEND ist der DAB-File zeitweise nicht in einem konsistenten Zustand, es sind beispielsweise Elemente mit Zeigern auf neue Elemente gespeichert, bevor diese neuen Elemente gespeichert werden. Um den Benützer vor den Folgen solcher Fehler zu schützen, ist während dieser Zeit *cont^.Zeiger* |1| auf 'nok' (nicht ok) gesetzt und wird nach Abschluss aller Aenderungen wieder auf 'ok' geändert. VINIT prüft beim Oeffnen, ob der DAB-File in diesem Sinne in Ordnung ist und stoppt das Programm, sofern nicht.

Das zweite Record des Daten-Files ist das sys-Element, dessen Platz (*sysfelp=2*) ist konstant, in jeder andern Hinsicht wird es wie ein anderes Element behandelt.

5.6 Hashtable

Um rasch feststellen zu können, ob ein bestimmtes Element, gegeben durch ein *felp*, im Hauptspeicher ist und allenfalls wo, wird die *hashtable* geführt.

Aus den *felp* wird der Index in die *hashtable* berechnet (*vemhash*). Als zugehöriger Wert ist der *pointer* auf das im Hauptspeicher befindliche Element (*elp*) und der *state* dieses Elements (siehe 5.7) gespeichert.

Hashtable ist mit open adressing (Knuth 3, S.518) organisiert. Das Löschen eines Eintrages erfolgt nach den in (Knuth 3, S.527) gegebenen Algorithmen (vemempt).

Die Grösse der hashtable kann bei jedem Programmlauf angepasst werden.

5.7 LRU-Strategie

Die Puffer werden nach der least recently used (lru) strategy verwaltet, d.h. das jeweils am längsten nicht verwendete Element wird überschrieben.

Für jedes im Hauptspeicher befindliche Element wird ein Zustand (state) gespeichert. Dieser bedeutet:

empty: - dieser Eintrag in der *hashtable* ist leer.

clean: - das betreffende Element stimmt im Hauptspeicher und auf dem File überein, es kann also überschrieben werden, ohne den File nachzuführen,

- das betreffende Element wurde in der laufenden Transaktion nicht benützt.

used: - das betreffende Element stimmt im Hauptspeicher und auf dem File überein, es könnte überschrieben werden, ohne den File nachzuführen,

- das betreffende Element wurde in der laufenden Transaktion benützt.

dirty: - das betreffende Element wurde verändert und muss vor dem Ueberschreiben im File nachgeführt werden.

fresh: - das betreffende Element wurde neue gebildet und fehlt im File noch.

Am Ende der Transaktion werden die veränderten Elemente im File nachgeführt oder im Hauptspeicher gelöscht.

vorher	nach TEND	nach TABO
empty	empty	empty
clean	clean	clean
used	clean	clean
dirty	clean	empty
fresh	clean	empty

Wird Platz für ein neues Element benötigt, so wird ein Element mit Zustand *clean* überschrieben, und zwar wird zyklisch in der *hashtable* nach einem solchen gesucht; *lru-start* gibt an, wo mit Suchen begonnen werden muss.

Kann kein solches Element gefunden werden, weil eine Transaktion mehr als *heap-limit* Elemente betrifft, so wird das Programm angehalten.

5.8 Gültigkeitsdauer von *elps*

Element-Puffer können neu belegt werden, so dass der gleiche *elp* (der eigentlich auf ein Puffer-Platz zeigt) im Laufe der Zeit auf verschiedene Elemente mit unterschiedlichen *felps* zeigt. Die gewählte Strategie garantiert, dass ein *elp* zumindest während einer Transaktion immer auf das gleiche Element zeigt.

6. DZ Direktzugriff

6.1 Funktion

DZ erlaubt, Elemente zu finden, von denen die Schlüsselwerte bekannt sind. Es wird für jeden Schlüsselwert bei der Speicherung ein Hash-Wert berechnet und in einem speziellen DZ-File an jener Stelle der *felp* des zugehörigen Elementes abgelegt. Es wurde darauf verzichtet, dort auch den Schlüsselwert mitzuspeichern, weil dies Probleme mit dem Typenkonzept von PASCAL gebracht hätte; leider verursacht das möglicherweise mehr Zugriffe auf den Daten-File. Wird ein Element gesucht, so wird für den gegegebenen Schlüssel der Hash-Wert berechnet und an dieser Stelle der *felp* des Elementes und daraufhin auch das Element selber gefunden.

6.2 Services offered

DZNEU zum Einfügen eines neuen Elementes mit seinen Schlüsselwerten.
DZALT zum Testen, ob die Schlüsselwerte bei einem Element geändert haben und allenfalls den DZ-File nachführen.

DZSUCH zum Suchen eines Elementes nach gegebenen Werten.
DZTEND/DZTABO zum Abschliessen von Transaktionen.
DZINIT/DZENDE zum Initialisieren und Abschliessen.

6.3 Services used

DZ benützt für den direkten Zugriff auf das DZ-File DZRAF (DZO, DZC, DZR, DZW) und für den Zugriff auf Elemente RAF (RAFR).

(Wird via VEM zugegriffen, was möglich ist, so wird der Puffer durch irrtümlich eingelesene Elemente verstopft, die mit der Transaktion nichts zu tun haben. Hingegen muss das richtige Element so zweimal gelesen werden - was wegen der vom Betriebssystem gemachten Pufferung der Daten nicht sehr aufwendig ist.)

6.4 Hash-Funktionen

Es werden zwei verschiedene Funktionen benützt.

dzstringhash berechnet für jeden *string* einen Wert zwischen 1 und *maxint*. Es wurde eine Funktion gewählt, die für aufeinanderfolgende *strings* (z.B. P 371 und P 372) sehr unterschiedliche Werte ergibt.

Ergebnisse von *dzstringhash* und alle Integer werden mit *dzinthash* verarbeitet, so dass sie am Schluss zwischen 1 und *dzmax* liegen.

Bei der Wahl der Hash-Methode wurde folgendes berücksichtigt:

stringhash: Zeichenketten, wie sie bei uns auftauchen, unterscheiden sich häufig nur in wenigen Zeichen.
(P 001, P 002, P 003, P 004 ... sind typische Punktnummern).

Bei der Hash-Berechnung müssen diese Unterschiede ausgenützt und verstärkt werden. Es werden also zuerst die Zeichenketten gefaltet.

Zeichenkette S = 'ABCDEFGHIK'
Faltung auf *falt* (in DZ.CON) (z.B. 3) mit einfacher Addition von *ord (S [i])*

A	B	C
D	E	F
G	H	I
K	.	.

kolonnenweise Addition

falt Kolonnen

gibt *falt*(5) Werte. Von diesen werden die hintersten Bits (*faltbits*) (in denen sie sich hoffentlich unterscheiden) zu einer Zahl zusammengesetzt.

Das garantiert, dass wenn sich zwei Zeichenketten in einem Buchstaben unterscheiden, auch unterschiedliche Werte resultieren. Unterscheiden sie sich in zwei Buchstaben, die genau falt Stellen auseinanderliegen, so können sich die Unterschiede kompensieren, das scheint aber tolerierbar.

integerhash: Es wurde versucht, dafür zu sorgen, dass Werte, die nur eine Einheit differieren, dennoch um mehrere Werte auseinanderliegen, um die Bildung von Ketten im DZ-file zu vermeiden.

6.5 Transaktionskonzept

Auch in DZ wird das Transaktionskonzept verwirklicht, indem während einer Transaktion nur eine *dzliste* mit den auszuführenden Aenderungen (einfügen, löschen) aufgebaut wird, die erst bei DZTEND ausgeführt wird und im DZ-*hashfile* eingetragen werden (oder bei DZTABO gelöscht wird).

DZSUCH muss natürlich immer zuerst diese *dzliste* konsultieren, ob ein Eintrag im File allenfalls noch fehlt.

7. SETS

7.1 Funktion

Die Verbindung innerhalb der Sets wird durch einfach verkettete Listen, ev. ergänzt durch *owner_pointer*, erreicht. SETS enthält die für die Manipulation dieser Listen notwendigen Routinen.

Die Bedeutung der einzelnen Pointer in jedem Element werden in *schema* [*elementtypes, settypes, link_types*] vom Anwendungsprogrammierer festgelegt (in SETS.INI).

Es wird in FILOP geprüft (SETTEST), dass die als Owner deklarierten Elementtypen auch im Schema enthalten sind und dass jedes Set einen Owner hat.

7.2 Services offered

SETS bietet an PRI folgende Dienstleistungen:

ELGO	bestimmen des Owners
ELGN	nächstes Element
ELCARD	Anzahl member in einem Set
ELEIN	Einfügen eines Elementes in ein Set
ELN	neues Elemente schaffen
ELLOS	Lösen eines Elementes aus einem Set
ELDS	Auflösen eines Sets
ELDEZ	Löschen einer Entität. Vorher wird sie aus allen Sets, in denen sie member ist, herausgelöst und alle Sets, in denen sie owner ist, werden aufgelöst.
ELAEND	notwendige Aktionen, sofern die Datenwerte eines Elementes verändert werden.

SETSINI und SETSENDE zum Anfangen und Abschliessen der Arbeiten mit PANDA.

7.3 Services used

SETS benützt die Dienste von VEM.

8. CFANG (DEC-10)

CFANG sind vier in Assembler geschriebene kleine Routinen, die verhindern, dass der Benützer PANDA während dem Nachführen der Dateien (PTEND) unterbrechen kann und damit die Datenbank in einen inkonsistenten Zustand kommt.

8.1 Initialisieren

Beim Programm-Start (in PINIT) wird mit DISCC (dissable control-c) der sonst übliche Abbruch des Programmes unterbunden; stattdessen wird jeweils die Prozedure INTLOC ausgeführt.
Am Schluss des Programms (in PSCHLU) wird mit ENACC (enable control-c) die normale Funktion des Betriebssystems wiederhergestellt.

8.2 Wann ist Unterbruch erlaubt?

Um die Konsistenz der Datenbank zu garantieren, muss PTEND, die die Dateien nachführt, eine 'atomic transaction' sein - entweder ist sie ganz durchgeführt oder nicht. Wird das erreicht, so ist auch die ganze Benützertransaktion atomic. PTEND darf also nicht unterbrochen werden.

Ausserhalb von PTEND darf das Programm angehalten werden, wobei die Dateien geschlossen werden müssen.

Natürlich können mit CFANG nur vom Benützer ausgehende Unterbrechungen kanalisiert werden. Wird PTEND durch Hardware oder andere Defekte unterbrochen, so ist die Datenbank im allgemeinen inkonsistent (vgl. 5.5.).

8.3 PCLOSE - NOTC

NOTC signalisiert an CFANG, dass PTEND jetzt beginnt und nicht unterbrochen werden darf. (notc = not ^c terminate).

PCLOSE zeigt den Abschluss von PTEND. Ist seit NOTC ein ^C getippt worden, so werden die Dateien geschlossen und das Programm angehalten. Wird später ^C getippt, so erfolgt dasselbe sofort.

PCLOSE und NOTC werden von PTEND aufgerufen.

8.4 VARSAV

In VARSAV sind die Namen der FORTRAN Routinen zum Schliessen der Dateien eingetragen.

9. Programmers Interface

In dieser Schicht werden die aussen sichtbaren Dienstleistungen zusammengestellt.

9.1 Services offered

siehe Benützeranleitung.

9.2 Services used

PRI benützt die Dienstleistungen von SETS, DZ, VEM und CFANG.

9.3 Eingangstest

PRI muss möglichst viele Fehler des Benützers erkennnen. Dazu gehört, dass überall kontrolliert wird, ob *element.recnr* im richtigen Bereich liegt; das schützt vor nichtinitialisierten übergebenen Variablen, die als Positionsanzeiger verwendet werden sollen.

9.4 Fehlermeldungen

NAMEE druckt, sofern gewünscht, einen erklärenden Text, wenn ein Fehler angetroffen wird.

D. HINWEISE FÜR DEN WEITEREN AUSBAU

PANDA kann in verschiedener Hinsicht ausgebaut werden:

- Datenkonsistenz
- Datenschutz
- Datensicherung
- Sortierte Sets
- Kontrolle der physischen Speicherung
- Temporäre Sets
- Raumbezogene Speicher- und Zugriffsalgorithmen
- multi-user Betrieb

Keine dieser Erweiterungen sollte zu Aenderungen bestehender Programme führen - sie sollten nachher nur schneller ablaufen oder sicherer verwendbar werden. Um dies zu erreichen, wurden die am Programmierer-Interface sichtbaren Einflüsse von sortierten Sets und raumbezogener Speicherung mit einer primitiven Implementierung vorgenommen (GIB MIT, GIB IN).

Der weitere Ausbau soll so erfolgen, wie sich von der Anwendung her die Nachfrage ergibt. Wichtig bleibt beim Ausbau, dass die Struktur der Schichten nicht verwischt wird.

Anderseits sollte die Programmierung von Anwendungen schon heute möglichst so erfolgen, wie wenn alle beschriebenen Erweiterungen bereits realisiert wären, d.h. es soll bei der Programmierung nicht auf effiziente oder (noch) nicht effiziente Implementierung im heutigen PANDA abgestellt werden.

1. Datenkonsistenz

Die Kontrolle der Datenkonsistenz ist in einem Datenbanksystem von grösster Bedeutung.

1.1 Static domains

PANDA erlaubt heute, wie andere DBMS, Kontrollen über den Wertebereich von Datenfeldern (Beispiel: Richtung zwischen 0 und 400 g) - sog. static domains |Thurnherr 80|.

1.2 Dynamic domains

Die Bedingung, dass beispielsweise ein Haus nur mit einem Strassennamen gespeichert werden kann, der als Strasse schon gespeichert ist, wird als dynamic domain bezeichnet. Dies ist noch nicht implementiert. Im CODASYL-DBTG Vorschlag sind dynamic domains eigentlich, wenn auch nicht unter diesem Namen, bereits enthalten. Sog. automatic member (Beispiel: ein Haus wird beim Speichern automatisch member in einem Strasse-Haus-Set, somit hat jedes Haus eine gespeicherte Strasse als Adresse, nämlich den owner in diesem Set.)

Das in PANDA gewählte Konzept des Cursors macht in diesem Falle die Gestaltung der Schnittstelle für den Anwendungsprogrammierer etwas schwieriger:

Nach dem CODASYL-DBTG Vorschlag wird beim Speichern, je nach dem was im Schema bestimmt wurde, auf gewisse Datenfelder abgestellt, um den owner des Sets bestimmen zu können, in das das neue Record eingefügt wird. Diese Schnittstelle ist unbefriedigend, indem nicht aus dem Programm-Code allein, sondern nur in Verbindung mit dem Schema festgestellt werden kann, was geschieht.

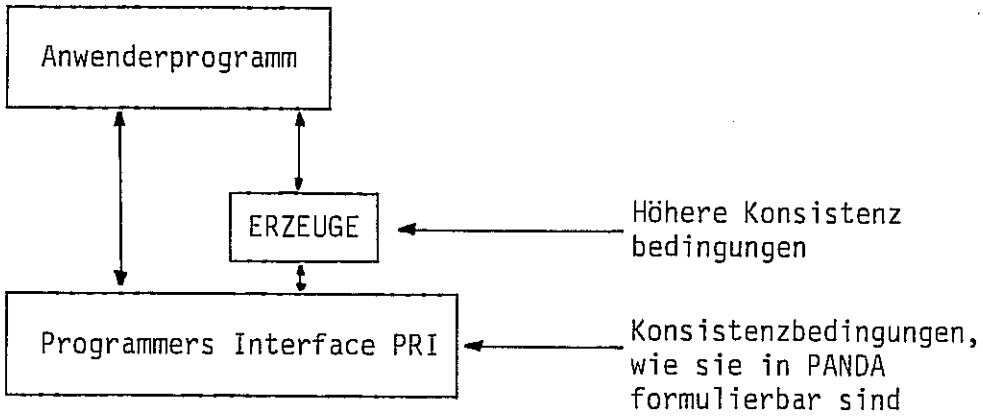
Es dürfte hier einfacher sein, die zusammengehörenden, Konsistenz erzwingenden Operationen in einer Benützerprozedur zusammenzufassen (speichere Haus), so dass Konsistenz dieser Art zusammen mit komplexeren Konsistenzbedingungen nur auf einer höheren (anwendungsnäheren) Schicht als PRI erreicht wird (siehe nächster Abschnitt).

1.3 Komplexere Konsistenzbedingungen

Der CODASYL-DBTG Vorschlag enthält auch Methoden zu Angaben beliebiger Prozeduren zur Ueberprüfung von Konsistenzbedingungen (in DBMS-10 nicht implementiert!). Ein ähnliches Vorgehen kann in PANDA gewählt werden:

Diese Möglichkeit sollte ausgenützt werden, um die den geometrisch-technischen Problemen angepassten Konsistenzbedingungen weiter zu erforschen.

Für bestimmte in einem Schema beschriebenen Entitäten, an deren Konsistenz besondere Anforderungen gestellt werden, müssen spezielle ERZEUGE Prozeduren erstellt werden. Das führt zu einer schichtweisen Vorstellung von Konsistenz



Zusätzlich muss verhindert werden, dass der Anwenderprogrammierer über die normale Schnittstelle PRI die Konstistenzbedingungen umgeht. Es scheint denkbar, wie übrigens auch im CODASYL-Vorschlag enthalten, im SCHEMA festzulegen, welche Befehle auf welche Entitäten, bzw. Sets angewendet werden dürfen.

2. Datenschutz

PANDA enthält heute keine Vorkehrungen für den Datenschutz.

Ein Schutz vor nicht erlaubtem Zugriff oder Veränderung lässt sich, analog wie bei der Datenkonsistenz, einfach erreichen, indem für den Zugriff spezielle Prozeduren bereitgestellt werden, die vorerst die Berechtigung prüfen. Auch hier muss natürlich verhindert werden, dass die normale Schnittstelle PRI benützt und der Schutz umgangen wird.

3. Datensicherung

Zusätzlich zum vorhandenen Transaktionskonzept kann erweiterte Datensicherung mit zwei Verfahren erreicht werden.

before images

Vor jeder Aenderung der Daten werden die vorher gültigen Daten in einer speziellen Datei chronologisch abgelegt. Wird die Datenbank verdorben, so kann ein früherer Zustand wiederhergestellt werden, indem die gespeicherten 'before images' in umgekehrter Reihenfolge (die letzten zuerst) wieder in die Datenbank eingefügt werden und so alle Aenderungen rückgängig gemacht werden.

after images

Nach jeder Aenderung werden die gültigen Daten chronologisch in eine Datei geschrieben. Geht der Datenbestand verloren, so lässt er sich aus einer Kopie wiederherstellen, indem alle gespeicherten 'after images' (seit der Herstellung der Kopie) in die Kopie eingefügt werden. Diese kommt damit auf den neuesten Stand.

Beide dieser Möglichkeiten lassen sich durch eine kleine Aenderung in TEND erreichen (einfacher sind 'after images'), der Aufwand zur Ausführungszeit ist aber beträchtlich. Datensicherung mit Hilfe von Journaling sollte deshalb nur angewandt werden, wenn es wirklich erforderlich ist.

Mit 'before images' liesse sich eine Verallgemeinerung des Transaktionskonzeptes verwirklichen, so dass dem Anwendungsprogrammierer stufenweise mehrere ineinander geschachtelte Transaktionen zur Verfügung stünden; PTEND wäre dann um ein Argument 'Stufen-Nummer' zu ergänzen. 'before images' scheinen beim gewählten einfachen Transaktionsmodell sonst nicht nötig.

Zusätzlich muss für die DZ-hashfiles eine Wiederherstellung vorgesehen werden (z.B. indem die dzlist am Ende jeder Transaktion in ein Journal geschrieben wird) oder aber dieser File wird aufgrund der wiederhergestellten Daten vollständig neu erstellt (Hilfsprogramm DZNEU).

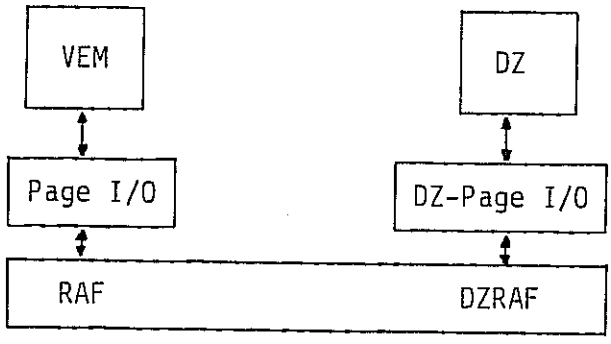
4. Sortierte Sets

Die jetzige Implementierung ist nur für Sets mit wenigen (<20) Membern geeignet. Sollen grössere Sets sortiert werden, so ist der Zugriff zu verbessern. Am einfachsten wird ein B*-Baum benützt, der in einem neuen Element-Typ untergebracht wird.
(Denert / Franck, S. 185) (Knuth 3, S. 473)

5. Kontrolle der physischen Speicherung

Die im Moment in PANDA angewandte Methode des Zugriffes auf die Daten greift auf jedes Element einzeln zu. Werden einige Elemente benötigt, die unmittelbar aufeinanderfolgend gespeichert sind, so kann dies - je nach Betriebssystem - nicht ausgenützt werden. Das könnte verbessert werden, indem mehrere Elemente zu einer 'Seite' zusammengefasst würden und nur noch ganze Seiten gelesen und geschrieben würden.

Dies bedingt eine neue Schicht, die zwischen VEM (und analog DZ) und die Lese- und Schreiboperationen des Betriebssystems tritt.



Es bietet kein Problem, eine solche Schicht einzubauen. Sie dürfte, als zusätzlicher Puffer, zu einer Beschleunigung von PANDA beitragen.

Diese Schicht sollte ebenfalls noch folgende 'Pendenzen' erledigen:

- Wiederverwendung des Speicherplatzes von gelöschten Elementen
- Speicherung nur der benützten Daten der Elemente
- Beschleunigung des raumbezogenen Zugriffes

Gleichzeitig muss das Konzept dieser Schicht genau bedacht werden, um innerhalb des PASCAL Typenkonzeptes verwirklicht werden zu können: Eine Typenkonversion zwischen der Seite und dem Element muss in irgendwelcher Form stattfinden. Dazu muss das 'loophole' im Typenkonzept von PASCAL, das Record mit variantem Teil, benützt werden.

Die Seite wird deklariert als

```
type  seite  =  array  [1 ... page size]  of words;
```

wo words auf der DEC-10 z.B. als type integer definiert werden kann.

Im element Record wird eine zusätzliche Variante

```
strukturlos: (w: array  [1 ... recdatalength]  of words);
```

eingefügt.

Schliesslich wird noch eine neue Tabelle in SETS.INI eingeführt, die für jedes Element die Datenlänge und die Anzahl der belegten Zeiger enthält:

```
var laengen : array [eltypes] of
                record of
                   pointerzahl : integer;
                   datalength  : integer;
                end;
```

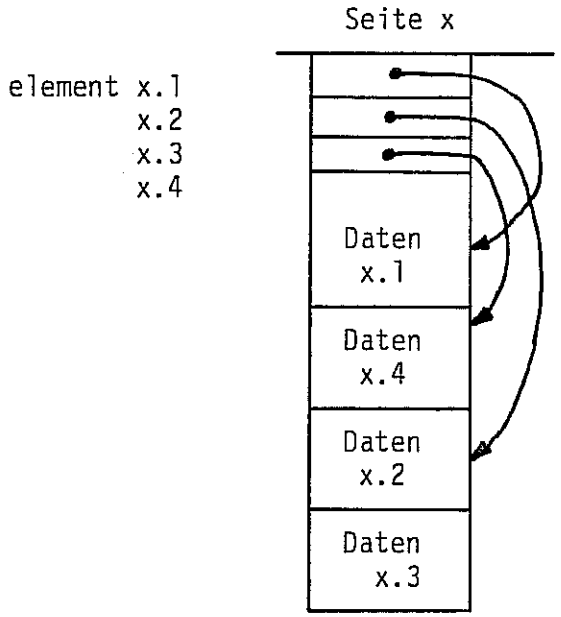
Damit lässt sich der Transfer eines Elementes *e* vom Typ *eltyp* an der Stelle *s* in der Seite *p* folgendermassen lösen:

```
with e do
  begin
    nr :=  p[s];
       (*  uebertragen eltyp *)
    s  :=  s + 1;
    eltyp :=  et [p[s]];   (* 1) *)
    s  :=  s + 1;
       (* Uebertragung der Zeiger *)
    for i :=  1 to laengen [eltyp].pointerzahl do
      begin
        zeiger[i] :=  p[s];
        s :=  s + 1
      end;
        (* Uebertragen der Daten *)
    for i :=  1 to laengen [e].datalength do
      begin
        w[i] :=  p[s];
        s :=  s + 1;
      end;
  end;
```

1) Für die Uebertragung der Elementtypen ist in SETS.INI eine spezielle Tabelle *et* anzulegen. In der andern Richtung kann die Umwandlung mit ord(eltypes) erfolgen.

Es sollte aber, ähnlich wie in DBMS-10 und in verschiedenen relationalen Datenbanksystemen,mit den felps nicht direkt die Stelle auf der Seite adressiert werden, sondern sog. TIDs verwendet werden. (TID = tuple identifier).

Ein felp setzt sich dann zusammen aus Seitennummer und Nummer des Elementes auf der Seite. Am Anfang der Seite sind einige Felder reserviert, die je für das betreffende Element die Position der Daten angibt:



Diese Lösung führt zu einer, zwar geringen, Unabhängigkeit der felps von der physischen Speicherung: die Daten von Elementen können innerhalb einer Seite verschoben werden, ohne dass die felps der Elemente ändern, d.h. die Zeiger auf diese Elemente, die in andern Elementen gespeichert sind, müssen nicht angepasst werden.

Schliesslich sollte auf jeder Seite eine Angabe über den noch freien Raum gespeichert werden, die auch zur Wiederverwendung von frei gewordenem Speicherplatz verwendet werden kann. Die Seiten mit freiem Platz sind in einer Kette untereinander zu verbinden (Wurzel = *cont*).

Eine Verminderung der physischen Zugriffe auf den Massenspeicher wird erreicht, indem ständig mehrere Seiten im Hauptspeicher behalten werden können und nach einem least recently used Verfahren ersetzt werden.

(type puffer = array[]...n] of seiten;).

6. Temporäre Sets

Die Verwendung der Datenbank innerhalb von Anwenderprogrammen lässt etwa den Wunsch aufkommen, Beziehungen zwischen Elementen aufbauen zu können, die nicht permanent gelten, sondern nur während der Laufzeit dieser Programme gelten. Damit Operationen auf solchen Sets weniger Zeit in Anspruch nehmen - was besonders für Graphik wichtig ist - aber auch damit die entsprechenden Felder nicht Platz im Massenspeicher belegen, sollten solche Beziehungen speziell bezeichnet und angepasst verwaltet werden können. Diese Aenderungen sollten im Zusammenhang mit der Kontrolle der physischen Speicherung erfolgen.

7. Raumbezogene Speicherungs- und Zugriffsalgorithmen

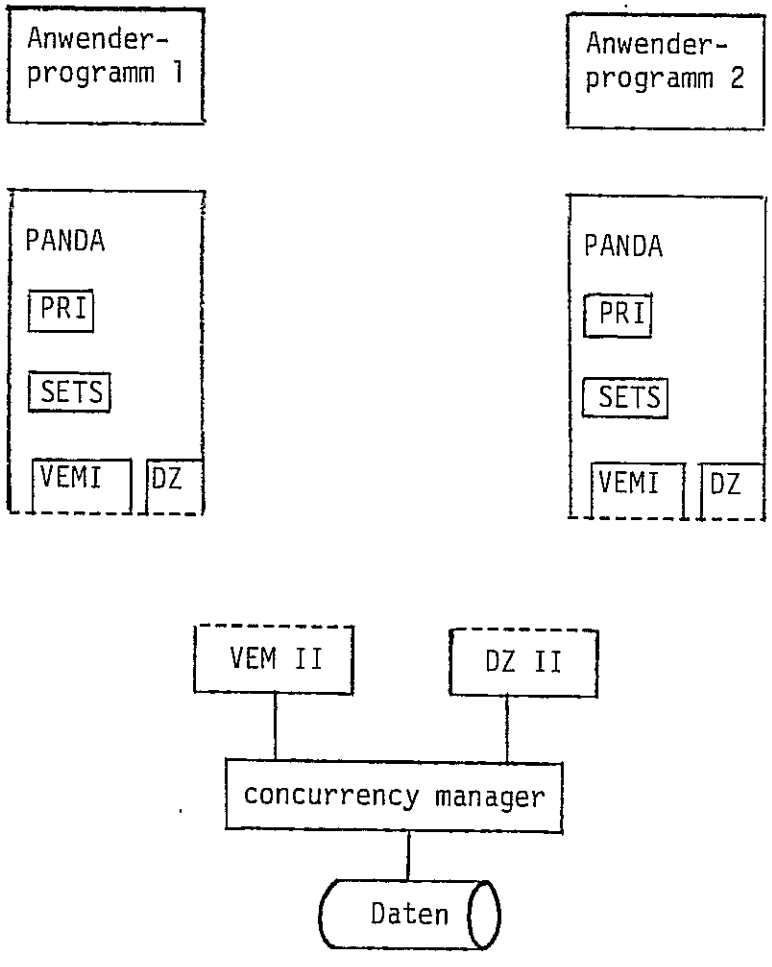
Die in |Frank 80| |Frank 81| beschriebenen Methoden lassen sich von der Logik her ohne weiteres einbauen, auf der DEC-10 können sogar die vorhandenen FORTRAN-Programme zur Berechnung der Feldnummern verwendet werden.

Die dabei aber ebenfalls angestrebte Beschleunigung des Zugriffes durch die Beeinflussung der physischen Speicherung kann nicht ohne zusätzliche Schicht zur Zusammenfassung der einzelnen Elemente zu Datenbank-Seiten verwirklicht werden (vgl. vorangehenden Abschnitt).

Bei der Umstellung sollte versucht werden, die Teilungen der Felder jeweils in nur zwei Teile, entweder in N-S oder in W-E Richtung vorzunehmen, um die mittlere Speicherbelegung höher zu halten. Einfacher ist vielleicht, mit dem Teilen zuzuwarten, bis ein Feld den doppelten minimalen Datenbestand aufweist und erst nachher zu teilen.

8. Multi-user Anwendung

PANDA ist nicht für gleichzeitigen Zugriff mehrerer Anwenderprogramme auf einen Datenbestand eingerichtet. Eine Erweiterung scheint möglich, indem pro Anwenderprozess der grösste Teil der PANDA Routinen lokal verwendet wird. Einzig die Teile von VEM und DZ, die auf die Dateien zugreifen, werden ausgegliedert und mit einem zentralen concurrency manager zusammengefasst.



Im concurrency manager müssen die für die Vermeidung von gegenseitigen Störungen notwendigen Tabellen geführt werden.

Für unsere Anwendungen, bei denen Störungen eigentlich sehr unwahrscheinlich sind, könnten 'optimistic schedules' die beste Lösung darstellen. Sie basieren auf dem Gedanken, dass man zuerst alle Benützer gewähren lässt und erst am Schluss einer Transaktion prüft, ob ein Konflikt aufgetreten ist. Wird ein solcher festgestellt, so wird die Transaktion rückgängig gemacht. Dies erfordert einen wesentlich geringeren Aufwand als die vorbeugende Verhinderung von Konflikten. Nachteilig ist höchstens, dass die Benützer-Transaktionen bei diesem Verfahren kurz gehalten werden müssen, damit nicht zuviele Eingaben wiederholt werden müssen |Kung 81|.

9. Higher level programmer interface

Es könnte sinnvoll sein, eine höhere Schnittstelle für den Anwendungsprogrammierer zu gestalten. Als Vorbild kann beispielsweise die relationale Schnittstelle in PASCAL/R oder System/R herangezogen werden und entweder eine relationale Spezifikation (Prädikatenkalkül oder -algebra) oder die in MAPQUERY vorgesehenen Ausdrücke verwendet werden. |Schmidt 77|

Eine wesentliche Weiterentwicklung ergäbe sich durch die Anwendung 'funktionaler' Ueberlegungen |Shipman 81| |Zanilo 81|, sowohl für das programmers interface als auch für die Datenbeschreibungssprache. Damit liessen sich die Ansprüche an Datenunabhängigkeit wohl optimal befriedigen.

10. Datenbeschreibungssprache

Sollte sich im Laufe der Anwendung zeigen, dass die gewählte Schnittstelle zum Anwenderprogrammierer geeignet ist, so könnte auch eine Vereinfachung der Datenbeschreibung ins Auge gefasst werden.

Statt dass der Datenbank-Administrator die verschiedenen Dateien (vgl. B 2) erstellt, die heute in PANDA für die Erstellung der SCHEMA Datei benötigt werden, könnte er in einer einzigen Datei seine Daten, ihre Verknüpfungen und Konsistenzbedingungen beschreiben.

Dazu wäre der Entwurf einer der Ideen von PANDA entsprechenden Datenbeschreibungssprache (DDL) notwendig. Als Grundlage wäre einerseits der CODASYL-DBTG Vorschlag von 1978 zu benützen und andererseits neuere Ideen, vor allem |Thurherr 80| aber auch |Shipman 81| |Hammer 81| und |Zaniolo 81|. Eng damit verbunden ist natürlich eine Ueberprüfung des programmers interface.

Das Erstellen eines Compilers für diese Sprache, der die heute vom Datenbank-Administrativ direkt erstellten PASCAL-Code Stücke produziert, bringt dann, je nach der Komplexität der DDL, unterschiedlichen Aufwand.

11. Datenunabhängigkeit

Mit Datenunabhängigkeit wird oft die Möglichkeit der Aenderung der internen Repräsentation der Daten in der Datenbank ohne Folgen für die Anwenderprogramme bezeichnet. Zusätzlich wird auch verlangt, dass die Anwenderprogramme von den im Schema deklarierten physischen Zugriffspfade unabhängig seien.

Diese Forderungen werden von PANDA nicht erfüllt.

Erstens ist beim strengen Typenkonzept von PASCAL eine Typenkonversion nicht leicht möglich. Prinzipiell wären Umwandlungen beim Zugriff (nicht in PASCAL geschrieben) und bei der Erweiterung zur Datenspeicherung auf Seiten (vgl. 'Kontrolle der physischen Speicherung') einbaubar.

Diese Lösung würde etwa dem (in DBMS-10 nicht implementierten) CODASYL-Vorschlag folgen.

Die bei relational aufgebauten Datenbanksystemen angewandte Technik, dass der Programmierer bei jedem Aufruf spezifizieren muss, in welcher Form er das Ergebnis benötigt (vgl.|Oracle 79|)scheint mir wenig programmier-freundlich und macht die DB-Aufrufe komplizierter und bestimmt auch fehleranfälliger. Eine praktische Bedeutung ist bei der Programmierung in PASCAL kaum mehr zu sehen (für COBOL vielleicht gegeben, da dort eine viel grössere Zahl von verschiedenartigen Datendeklarationen möglich sind).

Häufig wird als Argument angeführt, Datenunabhängigkeit erlaube, die Datenbank-interne Daten-Repräsentation zu ändern, ohne dass die Benützerprogramme angepasst werden müssten. Dies gilt aber nur mit Einschränkungen: wird die Länge einer Zeichenkette verkürzt, so scheint es zulässig, alte Programme weiter zu verwenden. Wird sie aber verlängert, so muss diese Verlängerung in den alten Programmen ebenfalls berücksichtigt werden, da sonst Informationen unterdrückt werden.

Zweitens wird beispielsweise im CODASYL-DBTG Vorschlag die Definition eines Sub-Schemas (auch etwa externe Sicht genannt) vorgesehen. Dies soll ermöglichen, dass der Record-Aufbau vom Programm aus gesehen nicht mit dem Datenbank-internen übereinstimmt. Insbesondere sollen einzelne, für diese Anwendung unnötige Felder weggelassen werden können. Dies könnte in PANDA beim Kopieren der Elemente von der Seite in den Element-Puffer geschehen. An dieser Stelle könnten auch 'virtuelle' Elemente, die nicht gespeichert, sondern bei jedem Zugriff berechnet werden, generiert werden.

Dies scheint beides für die geplanten Anwendungen von PANDA im Moment nicht notwendig.

Drittens wird die Unabhängigkeit der Anwenderprogramme von den deklarierten physischen Zugriffspfaden verlangt. Dies wird beispielsweise vom System/R erreicht, wo erst zur Ausführungszeit vom System entschieden wird, welche Zugriffspfade vorhanden sind und wie diese ausgenützt werden können.

PANDA ist in diesem Sinne von den deklarierten physischen Zugriffspfaden unabhängig: GIB OWNER funktioniert (von Anwendungsprogrammierer aus) gleich, ob owner-pointer vorgesehen sind oder nicht. Gleiches gilt für FIND MIT, unabhängig ob B*-Bäume für die sortierten Sets vorhanden sind oder nicht. (Für den direkten Zugriff DZ ist kein Ersatzmechanismus vorgesehen - dort besteht also eine Abhängigkeit vom deklarierten Zugriffspfad; hingegen ist es einfach eine Routine zu schreiben, die die DZ-files nach Aenderungen am Schema neu aufbaut.)

Es gilt hingegen nicht im Sinne, dass der Zugriff von den deklarierten Sets unabhängig wäre; diese bilden ja die logischen Zugriffspfade, tragen also Information und können deshalb nicht weggelassen werden, ohne dass diese Information eben wegfällt. Gleiches gilt beim relationalen Datenmodell für die Felder in einem Tuple, die für die Beziehung zwischen verschiedenen Tabellen dienen.

In diesem Sinne ist Datenunabhängigkeit ein relativer Begriff, der oft für ein System nur in einer eingeschränkten Bedeutung in Anspruch genommen werden darf. Dennoch ist er von grosser praktischer Bedeutung, indem damit der Anpassungsaufwand an bestehenden Programmen bei Aenderungen des Datenbank-Schemas beschrieben wird. Die für PANDA gültigen Regeln sind in der Bedienungsanleitung (Teil B) aufgeführt. Eine Erweiterung scheint, im Vergleich zu andern Systemen, nicht zwingend notwendig.

Hingegen scheint es einfach, Konversionsroutinen zu schreiben, die die ganze Datenbank von einer Repräsentation in eine andere umwandeln.

Hingegen sind die PANDA-Daten von der benützten Anlage weitgehend unabhängig. PANDA Daten können von einer Anlage auf eine andere übertragen werden, sofern nur der Transfer von integer, string etc. möglich ist.

ANHANG

1. INCLUD

INCLUD ist eine in PASCAL geschriebene Utility. Sie ermoeglicht es, Texte aus anderen Files vor der Compilation in ein PASCAL Programm einzufuegen, wie das beispielsweise in FORTRAN moeglich ist.
Es lassen sich damit z.B. Deklarationen von Datentypen, die in mehreren Programmen gleich verwendet werden, einkopieren.

Gleichzeitig setzt INCLUD auch den Befehl fuer den nachfolgenden Compile Class Command: damit koennen in einem PASCAL Quellenprogramm auch gleich die Befehle fuer das Laden eingefuegt werden.

Beispiel:

```
program a;
#load a.pas, x.lib    (* mit a muss auch die x.lib geladen werden *)

const
    %a.con
    ......          (* es wird der inhalt von a.con hier eingefuegt *)
type
    %a. typ
    ......


var  .....



%ut.pro             (* der text in ut.pro wird hier eingefuegt *)

begin
end.
```

Am einfachsten wird INCLUD mit den mic commands

```
do pas:comp a       (* compilieren von a *)
do pas:ls a         (* compilieren, laden und save von a *)
do pas:exe a        (* compilieren, laden und ausfuehren *)
```

ausgefuehrt, aus denen auch ersichtlich ist, wie INCLUD aufgerufen wird. INCLUD erstellt (aus Sicherheitsgruenden) im directory, in dem es laeuft, zwei scratch files (121212.ttp und 343434.ttp), die nicht automatisch geloescht werden.
Das PASCAL Quellenprorgramm muss die Extension .P haben. INCLUD erzeugt daraus ein file mit dem gleichen Namen und der Extension .PAS, das nachher kompiliert wird.

2. Die Verwendung von Funktionen mit Nebeneffekten

Die PANDA-Funktionen sind sog. Funktionen mit Seiteneffekten, bei deren Anwendung gewisse Vorsichtsmassnahmen notwendig sind. Diese werden hier läutert.

2.1 Begriff Funktion mit Nebeneffekten

Man spricht in der Programmierung von Seiteneffekten (side effects), wenn eine Prozedur oder Funktion neben den offensichtlichen Veränderungen an den über Parameter übergebenen Daten noch andere Veränderungen (eben side effects) vornimmt. Am häufigsten kommen wohl die verpönten Aenderungen von globalen Variablen vor.

Bei (Pascal-)Funktionen kann als Hauptzweck einer Funktion der zurückgegebene Wert (true oder false) angesehen werden; alle Aenderungen an übergebene Parameter können dann als Nebeneffekte angesehen werden.

2.2 Anwendung von PANDA-Funktionen

Alle PANDA-Funktionen haben nach dieser Definition Nebeneffekte. Sie geben als Wert 'true' oder 'false', je nachdem, ob sie richtig oder falsch durchlaufen worden sind, und die Ergebnisse stehen in den übergebenen Datenfeldern.

Dies bietet keine Probleme, sofern PANDA-Funktionen in Konstruktionen wie:

```
if <panda-function> (a, b, c)
        then <statement> ;
```

vorkommen.

Die PANDA-Function wird ausgeführt und je nach Ergebnis wird <statement> ausgeführt oder nicht.

Das lässt sich für gib-next auch günstig für die Steuerung von Wiederholungen verwenden:

```
e := initialwert;
while pgibn (e, s, fehler)
do    <statement>;
```

Das <statement> wird für jedes Element im Set s genau einmal ausgeführt.

2.3 Gefährliche Fälle

Wird eine PANDA-Funktion mit anderen boolschen Ausdrücken zu einer komplexen Bedingung verbunden, so ist Vorsicht geboten: Um die komplexen Bedingungen auszuwerten, müssen deren Teile ausgewertet werden, das hat bei der PANDA-Funktion eventuelle Auswirkungen (die Seiteneffekte).

Beispiel (falsch!)

```
found := false
p     := initialwert;
while       pgibn (p, s1, fehler)
            and not found
  do
    if  pgibo (p, e, s2, fehler)
      then  found := true;
```

Was bewirken soll, dass ausgehend von einem Element *initialwert* in einem Set s1 alle Elemente untersucht werden, bis zu einem ein owner gefunden wird.

Dies wird aber nicht erreicht: nach dem Finden eines owners ist *found* zwar true und der while loop bricht ab, aber 'pgibn' wird zum Test, ob der while loop abbrechen soll, nochmals ausgeführt, mit dem Effekt, dass *p* bereits auf das nächste (nicht zu e passende) Element zeigt.

Die korrekte Formulierung muss sein:

```
repeat
.   endeset := pgibn (p, s1, fehler);
.   if not endeset then
.       if pgibo (p, e, s2, fehler)
.       then
.           found := true;
until endeset or found or (fehler <>0);
```

2.4 Schlussfolgerung

Kombinierte Bedingungen von PANDA-Funktionen mit anderen boolschen Ausdrücken sind besonders zu prüfen. Die PANDA-Funktion wird beim Entscheid ausgeführt und hat ihre Auswirkungen, auch wenn wegen einem anderen Teil der Bedingung diese zu 'false' evaluiert wird.

Besonders gefährlich sind solche kombinierten Ausdrücke bei der Steuerung von Wiederholungen, wo sie im allgemeinen zu zuvielen Ausführungen der PANDA-Funktion führen. In diesen Fällen sollte immer eine zusätzlich boolsche Variable eingeführt werden (siehe Beispiel).

2.5 Hinweis auf Abbruchkriterium bei Array-Behandlung

Das vorgehend behandelte Problem ist verwandt mit dem Abbruch-kriterium-Problem bei Arrays:

```
var
    a : array [1 ... max] of integer;
    i : integer;
begin
    i := 1
    while  a [i] <> n
        and      i <= max
    do begin <statement>
                  :
                  :
                  :
              i := i + 1;
       end;
```

führt zum Ueberschreiten der Array-Grenzen mit i = max + 1.
(Wenn i = max wird der loop ausgeführt und i := max + 1.
Nun wird getestet, ob

$$i \leq max \quad \rightarrow \quad \text{false}$$
$$a[i] \neq n \quad \text{d.h.} \quad a[max + 1] \neq n$$

was aber die Array-Grenzen überschreitet und das Programm abbricht, obgleich das Ergebnis der Bedingung auf jeden Fall falsch ist.) Ein häufiger Fehler!

3. Handhabung DEC-10

Die Original Dateien befinden sich auf [251,1], und zwar müssen dort folgende Files vorhanden sein:

COMP.MIC
EXE.MIC
SCHEMA.MIC
PSTART.MIC

SCHEMA.P
FILOP.P
DRUCK.P

SETS.CON
VEM.CON
DZ.CON
RAUML.CON
METER.CON

METER.TYP
SETS1.TYP
SETS2.TYP
SETS3.TYP
RAUML.TYP
VEM.TYP
DZ.TYP

RAUML.VAR
VEM.VAR
SETS.VAR
DZ.VAR

CFANG.PRH
RAF.PRH
DZRAF.PRH
VEM.PRH
UTILI.PRH

METER.PRO
DZ.PRO
VEM.PRO
SETS.PRO
RAUML.PRO
PRI.PRO
UTILI.PRO
FDUMMY.PRO

DZ.INI
SETS.INI
RAUML.INI

CFANG.MAC

RAF.FOR
DZRAF.FOR

Für das Arbeiten mit PANDA muss [251.1] als LIB gesetzt sein. (PATH LIB: [251.1]).

Die in B 1.2 aufgezählten Files müssen vom Benützer in sein Arbeits-Directory kopiert und die notwendigen Aenderungen mit dem Editor ausgeführt werden.

Dann wird mit 'DO SCHEMA <schema-name>' ein <schema-name>.REL File erzeugt, der alle Datenbankprozeduren enthält. Do SCHEMA erzeugt auch das Programm FILOP und eröffnet die Dateien.

Jedes Anwenderprogramm muss folgende INCLUD-Befehle enthalten:

```
CONST
% sets.con
% rauml.con

TYPE
% sets1.typ
% meter.typ
% sets3.typ
% rauml.typ

VAR
(*Prozedurdeklarationen*)
% raf.prh
% fdummy.pro
% pri.prh
```

Die Files raf.prh und fdummy.pro sind nur wegen einer unerklärbaren Eigenheit beim Zusammenspiel von PASCAL und LINK notwendig (sonst ergibt sich beim Laden 'unsatisfied global symbols' !).

4. PANDA-Fehlermeldungen

Fehler Nr.	Fehlermeldung
1	PANDA : Set ist kein gueltiger Set-typ
2	PANDA : kein Platz auf dem heap - sollte nicht auftauchen
3	PANDA : element ist nicht von gueltigem Typ
4	PANDA : element kommt nicht im Set vor
5	PANDA :
6	PANDA : element ist bereits im Set
7	PANDA : position ist nicht im gleichen Set-Vorkommen wie owner
8	PANDA : owner ist nicht owner-typ im Set
9	PANDA :
10	PANDA : element kommt nicht im Set vor
11	PANDA :
12	PANDA : Verkettung nicht in Ordnung - sollte nicht auftauchen
13	PANDA : element ist owner im Set
14	PANDA :
15	PANDA : Element-typ nicht fuer Direktzugriff vorgesehen
16	PANDA : keine Entitaet mit diesen Werten
17	PANDA : pointer zeigt nicht auf Entitaet vom angegebenen Typ
18	PANDA : pointer ist nicht gueltig
19	PANDA : im owner kein set pointer
20	PANDA : kein owner fuer set
21	PANDA : pointer doppelt verwendet
22	PANDA : eltyp nicht zulaessig
23	PANDA : Wert kommt im Set bereits vor
24	PANDA :
25	PANDA : keine Entitaet mit diesen Werten
26	PANDA : set 1 ist nicht sortiert
27	PANDA : Wert kommt schon vor
28	PANDA : sys element kann nicht eroeffnet werden
29	PANDA : leerel kann nicht eroeffnet werden
30	PANDA : elklei fuer ein Set fehlt
31	PANDA : dzhash fuer ein Set fehlt
32	PANDA : dzgleich fuer ein Set fehlt
33	PANDA : maxklasse < minklasse
34	PANDA : maxklasse < 0
35	PANDA : minklasse < 0
36	PANDA : elk_el Set aber keine minklasse
37	PANDA : rsown.status nicht im erlaubten Bereich
38	PANDA : .....
39	PANDA : .....

Erläuterungen zu den einzelnen Fehlern

zu 1: Der im Aufruf übergebene Set-Typ ist in SETS1.TYP nicht deklariert. (Tippfehler?)

zu 2: Zu wenig Run-Core verlangt. Erhöhe 'run-core' entweder beim Kompilieren ([Kisicki] Ziffer 1.5.1) oder mit RUN <programname>n. Wenn der Fehler nicht verschwindet,mit PANDA-Spezialist besprechen.

zu 3: *eltyp* enthält einen Wert, der in SETS1.TYP nicht deklariert ist. Tippfehler oder *eltyp* überhaupt nicht initialisiert?

zu 4: Der Aufruf erfordert, dass ein gegebenes Element im durch den gegebenen Owner bezeichneten Set enthalten ist.

zu 6: Ein Element kann nur in ein Set des gleichen Types eingefügt werden; wenn es bereits in einem andern Set des gleichen Types member ist, muss es aus diesem zuerst entfernt werden.

zu 7: Die angegebene Position, nach der das neue Element eingefügt werden soll, ist nicht in dem Set, das durch den gegebenen Owner bezeichnet wird.

zu 8: Das als Owner gegebene Element ist nicht vom Element-Typ, der als Owner für das gegebene Set deklariert ist.

zu 10: wie Fehler 4

zu 12: Dieser Fehler bedeutet, dass die Zeigerkette nicht richtig verbunden ist. PANDA-Spezialist konsultieren!

zu 13: Das aus einem Set herauszulösende Element ist Owner in diesem; nur member Elemente können aus einem Set entfernt werden!

zu 15: Für den Element-Typ, der in *eltyp* gegeben ist, wurde in DZINIT der Direktzugriff nicht verlangt.

zu 16: Ein Element mit den gegebenen Werten wurde nicht gespeichert - nur Elemente einer mit PTEND abgeschlossenen Transaktion sind in der Datenbank enthalten.

zu 17: Der im gegebenen Element in *recnr* enthaltene *felp* zeigt auf ein Element, das einen andern *eltyp* hat, als das gegebene.
Die Benützer dürfen bei einem von PANDA gegebenen Element weder *recnr* noch *eltyp* verändern!

zu 18: Der im gegebenen Element im Feld *recnr* gespeicherte *felp* ist nicht gültig. Benützer dürfen *felp's* nicht verändern!

zu 19: Fehler in SETS.INI:
Ein Elementtyp wurde als Owner in einem Settyp bezeichnet, beim Elementtyp sind aber keine Zeiger für diesen Settyp reserviert.

zu 20: Fehler in SETS.INI:
Für einen Settyp wurde kein Owner Elementtyp bezeichnet.

zu 21: Fehler in SETS.INI:
Ein Zeiger in einem Elementtyp kann nur für einen Settyp verwendet werden. Alle Zeigernummern für einen Elementtyp müssen verschieden sein.

zu 22: wie Fehler 3

zu 23: In einem sortierten Set müssen alle Elemente verschiedene Werte für das Feld, das die Sortierreihenfolge bestimmt, aufweisen. Sie wollen ein Element mit Werten einfügen, die im Set bereits vorkommen.

zu 25: Ein Element mit den gegebenen Werten kommt in diesem sortierten Set nicht vor.

zu 26: Wenn ein Element in einem Set aufgrund gegebener Werte gesucht werden soll, so muss der Settyp als 'sortiert' in SETS.INI deklariert werden.

zu 27: Bei Elementen, für die Direktzugriff möglich ist, dürfen nicht zwei Elemente die gleichen Schlüsselwerte aufweisen.

zu 28: Das *sys* Element wird beim Initialisieren der Daten erstellt; der Benützer kann kein zweites Element des gleichen Typs erstellen.

zu 29: Das *leerel* Element wird beim Initialisieren der Daten erstellt; der Benützer kann kein zweites Element des gleichen Typs erstellen.

zu 30: Fehler in SETS.INI:
Sofern ein Settyp als 'sortiert' deklariert ist, muss eine entsprechende Vergleichsoperation in ELKLEI (in SETS.INI) eingefügt werden.

zu 31: Fehler in SETS.INI:
Sofern für einen Elementtyp Direktzugriff vorgesehen ist, müssen in DZELHASH (in DZ.INI) die für den Zugriff verwendeten Felder angegeben werden.

zu 32: Fehler in SETS.INI:
Sofern für einen Elementtyp Direktzugriff vorgesehen ist, müssen in DZGLEICH (in DZ.INI) Methoden für den Vergleich zweier Elemente dieses Typs eingefügt werden.

zu 33: Fehler in RAUML.INI:
Für Elementtypen, auf die raumbezogen zugegriffen werden muss, sind passende Feldgrössen anzugeben. Dabei muss die minimale Klasse immer kleiner (oder gleich gross) wie die maximale Klasse gewählt werden.

zu 34: Fehler in RAUML.INI:
Für Elementtypen, auf die raumbezogen zugegriffen werden muss, sind passende Feldgrössen anzugeben. Dabei muss die maximale Klasse grösser als 0 gewählt werden.

zu 35: Fehler in RAUML.INI:
Für Elementtypen, auf die raumbezogen zugegriffen werden muss, sind passende Feldgrössen anzugeben. Dabei muss die minimale Klasse grösser als 0 gewählt werden.

zu 36: Fehler in RAUML.INI:
Für Elementtypen, auf die raumbezogen zugegriffen werden muss, sind passende Feldgrössen anzugeben. Ein Elementtyp wurde in SETS.INI für raumbezogenen Zugriff vorgesehen aber in RAUML.INI ist keine minimale Klasse gegeben.

zu 37: Beim raumbezogenen Suchen muss *rsown.status* zu Beginn mit dem Wert 0 initialisiert werden und darf nachher nicht mehr verändert werden.

5. Literaturverzeichnis:

Astrahan, M.M. et. al.
System R: A relational approach to database management
ACM TODS 1, 97 (June 1976)

Bachmann, W.C.
The programmer as a navigator
ACM Communications 1973

CODASYL
Codasyl Data Base Task Group (DBTG) Report
April 1977

CODASYL DDL
Journal of Development
June 1973

CODASYL
Data Description Language Comittee
Journal of Development
January 1978

Codd, E.F.
Relational Database
A Practical Foundation fpr Productivity
ACM Communications Vol. 25 No. 2 p. 109
February 1982

Digital Equipment Corp.
Data Base Management System (DBMS-10)
Administrator's Procedures Manual
Maynard, Mass. 1977

Digital Equipment Corp.
Data Base Management System (DBMS-10)
Programmer's Procedures Manual
Maynard, Mass. 1977

Denert, Ernst, Franck, Reinhold
Datenstrukturen
Mannheim, Bibligraphisches Institut 1977
(Reihe Informatik 22)

Denert, E.
Software Modularisierung
Informatik Spektrum 4/1979 S. 204

Fagin, R.
A Normal Form for Relational Databases that is Based on Domains and Keys
ACM TODS Vol. 6 No. 3 September 1981 p. 387

Frank, A.
Datenspeicherung für schnellen Zugriff auf Daten räumlich benachbarter Objekte
in: Nachrichten aus dem Karten- und Vermessungswesen
Institut für angewandte Geodäsie
Frankfurt a.M. Reihe I Heft 85 S. 37

Frank, A.
Application of DBMS to Land Information Systems
7th Internat. Conference on Very Large Data Bases
Cannes September 1981 p. 448

Frank, A., Tamminen, M.
Management of spatially referenced data
Proceedings
Land Information at the Local Level (Leick, A. Ed.)
University of Maine Orono 1982

Frank, A.
Mengen in PANDA
Beschreibung der Routinen MENGE.PRC
zur Behandlung von Mengen
Institut für Geodäsie und Photogrammetrie
ETH Zürich 1982

Hammer, M, Mc Leod, D.
Database Description with SDM: A Semantic Database Model
ACM TODS Vol. 6 No. 3 September 1981 p. 351

Jensen, K., Wirth, N.
Pascal
User Manual and Report
Lecture Notes in Computer Science 18
Springer Verlag Berlin 1974

Kaehler, T.
Virtual Memory for an Object-Oriented Language
BYTE Vol. 6 No. 8 August 1981 p. 378

Kernighan, B.W., Plauger, P.J.
Software Tools in Pascal
Addison-Wesley (Reading, Mass.) 1981

Kimm, R., Koch, W., Sinousmeier W., Tontsch, F.
Einführung in Software Engineering
W. de Gruyler Berlin 1979

Kisicki, E. Nagel, H.-H.
PASCAL for the DECSystem-10
Institut für Informatik
Universität Hamburg
Mitteilung Nr. 37, 1976

Knuth, D.
The art of computer programming
Vol. 3 Sorting and Searching
Addison Wesley 1973

Kung, H.T., Robinson, J.T.
On optimistic Methods for Concurrency Control
ACM Transaction on Database Systems
Vol. 6 No. 2 Junie 1981 p. 213

Lampson, B.W., Paul, M., Siegert, H.J. (Eds.)
Distributed Systems - Architecture and Implementation
Springer Verlag 1981

Lewis
Proceedings of the SIGSMALL-SIGMOD Conference 1981

Nievergelt, J., Hinterberg, H., Sevcik, K.C.
The GRID FILE
an adaptable, symmetric multi-key file structure
Institut für Informatik
ETH Zürich 1981 Bericht 46

Oracle
Oracle - Introduction
Relational Software Inc.
Menlo Park, Cal 1979

Rebsamen, J., Reimer, M., Ursprung, P., Zehnder, C.A.
LIDAS
A Database System for the Personal Computer Lilite
The Database Management
Eidg. Technische Hochschule Zürich
Institut für Informatik
Bericht 50 June 1982

Reuter, A.
Fehlerbehandlung in Datenbanksystemen (Datenbank-Recovery)
Hauser Verlag, München 1981

Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation
ACM Trans. on Database
Vol. 2 No. 3 September 1977 p. 247

Shipman, D.W.
The Functional Data Model and the Data Language DAPLEX
ACM TODS March 1981 Vol. 6 No. 1 p. 140

Tamminen, M.
Management of spatially referenced data
Conceptual study of system requirements and structure
Helsinki University of Technology 1981

Thurnherr, B.
Konzept und Sprache für den Entwurf konsistenter Datenbanken
Diss. ETH Nr. 6526
Zürich 1980

TSC-Pascal
Uniflex Pascal User's Manual
Technical Systems Consultants Inc. 1981

Zaniolo, C.,
Melkanoff, M.A.

On the Design of Relational Database Schemata

ACM Transactions on Database Systems
Vol. 6 No. 1 March 1981 p. 1

Zehnder, C.A.

Informationssysteme und Datenbanken

Verlag der Fachvereine
Zürich 1981

ACM	=	Association of Computing Machinery
CODASYL	=	Committee on Data System Languages
TODS	=	Transactions on Database Systems