Kapitel 1: Einführung

1.1 Grundprinzipien und Stand der Technik

Datenbanksysteme bilden das Softwarerückgrat der meisten computergestützten Informationssysteme. Sie stellen dem Anwendungsentwickler generische Operationen zum Suchen in sehr großen Datenbeständen sowie zum Einfügen, Ändern und Löschen von Daten zur Verfügung. Operationen werden in einer deskriptiven Datenmanipulationssprache spezifiziert, meist in der standardisierten Sprache SQL (Structured Query Language), und die Operationsaufrufe werden in Anwendungsprogramme eingebettet, die in einer konventionellen Programmiersprache oder einer sogenannten "'4th Generation Language"' (4GL) geschrieben sind. Die folgenden Beispiele nennen einige typische Informationssysteme, die auf Datenbanksystemen beruhen.

- *Flugreservationen und −buchungen einer Fluggesellschaft*. Derartige Systeme verwalten typischerweise einige Millionen Reservationen und Buchungen (bis zu einem Jahr im voraus) und müssen in Spitzenzeiten bis zu 100 Such− und Änderungsoperationen pro Sekunde ausführen können. Darüber hinaus müssen solche Systeme praktisch rund um die Uhr an 365 Tagen im Jahr betriebsbereit sein. Eine Ausfallzeit von mehr als wenigen Minuten im Jahr gilt bereits als unakzeptabel.

- *Verwaltung eines Teilelagers eines Automobilherstellers*. Dies umfaßt u.U. mehrere Millionen Teile für alle Modelle, für die der Hersteller die Lieferung von Ersatzteilen garantiert. Das System dient u.a. zur Buchführung über die Lagerbestände und zum Auffinden der Teile in roboterbedienten Lagerhallen. Zusätzlich gibt das System Auskunft darüber, welche Teile für welche Modelle verwendet werden können, welche Teile untereinander kompatibel sind, usw.

- *Entscheidungsunterstützung für den Wertschriftenhandel einer Bank*. Börsenkurse und damit zusammenhängende Informationen (z.B. Wirtschaftsnachrichten) werden über entsprechende Datenleitungen von verschiedenen Börsen kontinuierlich erfaßt, und es werden komplexe Suchoperationen auf den Daten spezifiziert, die die Händler auf interessante Konstellationen aufmerksam machen sollen. Dies ist ein Beispiel eines Echtzeit−Informationssystems mit harten Anforderungen an die Antwortzeit der Suchoperationen und extrem hohen Änderungsraten der Daten (z.T. mehr als 100 Kurse pro Sekunde).

- *Verwaltung von Patientendaten in einem Krankenhaus*. Dies beinhaltet Informationen über die Krankengeschichte eines Patienten, Untersuchungsergebnisse, Einsatz von Medikamenten, Behandlungen, bis hin zur Kostenabrechnung. über die für administrative Zwecke notwendigen Such− und Änderungsoperationen hinaus ist u.a. die Unterstützung der Suche nach ähnlichen Krankheitsbildern wünschenswert. Zusätzlich wird die Anbindung von Archivsystemen für Ultraschall− und Röntgenbilder, Computertomographien u.ä. angestrebt.

Aus den aufgeführten Anwendungen geht klar hervor, daß Datenbanksysteme u.U. riesige Datenmengen zu verwalten haben. Typische Datenbankgrößen in kommerziellen Anwendungen liegen im Bereich zwischen 10 und einigen 100 Gigabytes, sehr große Datenbanken umfassen mehr als 1 Terabyte. Daraus folgt zwangsläufig, daß Datenbanken in erster Linie auf Sekundärspeichermedien liegen, in aller Regel auf Magnetplatten. Zwischen Hauptspeicher und Magnetplatten aber klafft ein riesiger Geschwindigkeitsunterschied von 5 bis 6 Zehnerpotenzen. Wenn wir uns beispielsweise die Geschwindigkeit verschiedener Speicherhierarchiestufen als eine räumliche Entfernung der Daten vom Prozessor vorstellen und annehmen, daß der CPU−Cache im selben Raum wie der Prozessor, also vielleicht 1 Meter entfernt, ist, dann befindet sich der Hauptspeicher im Nebenraum (ca. 10 Meter entfernt), die Magnetplatten aber stehen 1000 bis 10000 Kilometer entfernt in Moskau oder gar in Tokio. Aus diesem gewaltigen Geschwindigkeitsunterschied sowie aus der Nebenbedingung, daß Daten auf Magnetplatten nur blockweise adressiert werden können, ergibt sich die Notwendigkeit, daß Datenbanken anders als mit klassischen, hauptspeicherorientierten Datenstrukturen organisiert werden. Diese Überlegung führt auf das folgende Grundprinzip.

Grundprinzip 1:
Um große Datenmengen effizient verwalten zu können, sind Datenbanksysteme im Hinblick auf Sekundärspeicherzugriffe optimiert.

Insbesondere sind auch die klassischen, RAM−orientierten Modelle der Komplexitätstheorie für Datenbanksysteme nur von untergeordneter Bedeutung; die entscheidende Effizienzmetrik ist vielmehr die Anzahl der Sekundärspeicherzugriffe.

Da Datenbanksysteme generische Systeme sind, also gleichermaßen in Bankkonten wie auch in Patientendaten suchen und ändern können, sind universelle Optimierungen der Systemeffizienz nur eingeschränkt möglich. Vielmehr hängt der Nutzen einer Optimierungsmaßnahme häufig von den spezifischen Lastmerkmalen der jeweiligen Anwendung ab. In einer Anwendung mit geringer Änderungsrate kann man Daten − ggf. unter Einführung von Redundanz − so speichern, daß verschiedenartige Suchoperationen extrem effizient ausgeführt werden können; in einer Anwendung mit hoher Änderungsrate hingegen kann dies unakzeptabel sein, wenn die Aktualisierung der entsprechenden Datenstrukturen zu aufwendig ist. Aus diesem Grund unterstützt ein Datenbanksystem in der Regel ein breites Spektrum verschiedener Speicherungs− und Zugriffsstrukturen und überläßt es einem Tuning−Spezialisten, diese Strukturen für eine Anwendung festzulegen. Da sich aber die Lastmerkmale einer Anwendung im Laufe der Zeit ändern (z.B. durch Erweiterung der Anwendungsfunktionalität), müssen diese Tuning−Entscheidungen hin und wieder revidiert werden.

Solche Änderungen der Speicherungs− und Zugriffsstrukturen sind nicht gerade eine billige Maßnahme, da dazu häufig große Datenmengen vom Datenbanksystem umgespeichert werden müssen. Von fundamentaler Bedeutung ist jedoch, daß bei einer solchen Revision der Datenspeicherung die Anwendungsprogramme auf keinen Fall geändert werden müssen, denn der entsprechende Änderungsaufwand wäre kostenmäßig in den meisten Fällen absolut unakzeptabel. Datenbanksysteme stellen diese sogenannte *Programm−Daten−Unabhängigkeit* (häufig auch nur: Datenunabhängigkeit) sicher, indem sie die eigentlichen Speicherungs− und Zugriffsstrukturen vor dem Anwendungsprogramm verbergen. Mit anderen Worten: Datenbanksysteme wenden das Prinzip der Abstrakten Datentypen auf generische Weise an, indem sie die verwalteten Daten kapseln und nur deskriptive Operationen anbieten, die unabhängig von den zugrundeliegenden konkreten Datenstrukturen sind. In eingeschränkter Form gilt dieses Prinzip sogar auch bei einer anwendungsbeding-

ten Änderung des Datenformats (z.B. der Umstellung der Postleitzahlen in Deutschland) sowie bei der nachträglichen Erweiterung existierender Daten um zusätzliche Informationen (z.B. der Hinzunahme einer Kreditkartennummer für alle Privatpatienten einer Klinik).

Grundprinzip 2:
Die Speicherung von Daten kann geändert werden, ohne daß Anwendungsprogramme geändert werden müssen (Programm–Daten–Unabhängigkeit).

Dieses fundamentale Prinzip hat ganz entscheidend zum kommerziellen Erfolg von Datenbanksystemen beigetragen. Etliche Informationssysteme, die ohne echtes Datenbanksystem entwickelt wurden (also beispielsweise auf der Basis eines File–Systems), sind mittelfristig gescheitert, weil sie aufgrund fehlender Programm–Daten–Unabhängigkeit unakzeptable Programmwartungskosten verursachten.

Das Grundprinzip der Programm–Daten–Unabhängigkeit und die damit einhergehende Konzeption deskriptiver Datenmanipulationssprachen wurde vor allem durch die Entwicklung *relationaler Datenbanksysteme* in den Siebziger und Achtziger Jahren entscheidend vorangebracht. Bei diesen Systemen, die heute den Markt dominieren, werden alle Daten konzeptionell in Form von Tabellen (mathematisch: Relationen), repräsentiert. Einen Schritt weiter geht man seit einigen Jahren mit *objektorientierten Datenbanksystemen (*bzw. entsprechenden "'postrelationalen"' Weiterentwicklungen relationaler Systeme). Diese Systemklasse erlaubt es insbesondere, Aggregationsbeziehungen zwischen Datenelementen in Form "'komplexer Objekte"' direkt zu repräsentieren; beispielsweise können alle Teile eines Motors zu einem Objekt "'Motor"' zusammengefaßt werden – – eine Form der Aggregation, die mit relationalen Systemen nur auf umständliche Art möglich ist. Generische Such– und Änderungsoperationen werden sowohl auf der Ebene der Gesamtobjekte als auch auf der Ebene der Teilobjekte angeboten. Darüber hinaus ist es in objektorientierten Datenbanksystemen auch möglich, anwendungsspezifische Operationen für eine Menge von Objekten zu definieren, und diese können in Ausdrücken der Datenmanipulationssprache verwendet werden. Das Datenbanksystem, welches an sich als generisches Softwarepaket konzipiert ist, wird damit anwendungsspezifisch erweiterbar. Beispielsweise kann man eine Funktion "'Volumen"' für geometrische Objekte definieren, die auf Polygonen für die Begrenzungsflächen eines dreidimensionalen Körpers operiert. Wie in objektorientierten Programmiersprachen können diese Funktionen mittels "'Vererbung"' auf flexible Art und Weise für speziellere Objekte wiederverwendet werden (z.B. für die speziellen Körper, die beim mechanischen CAD auftreten); dabei können bei Bedarf spezifische Details "'überschrieben"', d.h. neu festgelegt werden (z.B. die Interpolationsvorschrift zwischen Stützpunkten eines Polygons). Das Prinzip der Programm–Daten–Unabhängigkeit sollte auch für objektorientierte Datenbanksysteme unter Einschluß der anwendungsspezifischen Operationen gelten. Allerdings gibt es durchaus noch offene Fragen in diesem Zusammenhang, die Gegenstand aktueller Forschung sind, beispielsweise die Konzeption und effiziente Implementierung objektorientierter Datenmanipulationssprachen und deren Einbettung in objektorientierte Programmiersprachen.

Die Effektivität eines Informationssystems steht und fällt mit der Korrektheit der zugrundeliegenden Daten. Eine Form der Datenkorrektheit, die von Datenbanksystemen sehr weitreichend unterstützt wird, ist die *Konsistenz* der Daten (oder auch: *Integrität* der Daten). Diese fordert, daß bestimmte Zusammenhänge zwischen verschiedenen Datenelementen, die in der realen Welt gelten sollen, auch in der Datenbank gewährleistet werden. Beispielsweise soll das Konto eines Kunden bei einem Versandhaus nur dann belastet werden, wenn eine entsprechende Bestellung ausgeliefert wird. Datenbanksysteme bieten die

Möglichkeit an, daß solche anwendungsspezifischen Konsistenzbedingungen spezifiziert werden, und die Systeme überwachen die Einhaltung dieser Bedingungen. Änderungen, die die Konsistenz verletzen würden, werden zurückgewiesen. Damit werden Systeme gewissermaßen "'immun"' gegen fehlerhafte bzw. inkonsistente Dateneingaben. Diese Vorgehensweise ist zwar grundsätzlich auf Plausibilitätsprüfungen beschränkt, da auch fehlerhafte Eingaben formal konsistent sein können; in vielen Anwendungen aber können dadurch fast alle Eingabefehler erkannt und eliminiert werden.

Einige Klassen von Konsistenzbedingungen können leichter spezifiziert und vom System überwacht werden, wenn die Datenbank selbst daraufhin entworfen wurde. In diese Klasse fallen beispielsweise Bedingungen vom Typ "'keine zwei Konten dürfen dieselbe Kontonummer haben"' oder "'die bei einer Buchung angegebene Kontonummer muß in der Datenbank existieren"'. Die Datenbankforschung hat verschiedene Methoden des sogenannten konzeptionellen (oder auch: logischen) Datenbankentwurfs hervorgebracht, die in diesem Sinne auf eine hohe Qualitätssicherung für die Daten abzielen. Diese Methoden, z.B. die relationale Entwurfstheorie und vor allem der Entwurf mit "'Entity–Relationship–Modellen"', haben in der Praxis weite Verbreitung gefunden. Moderne Datenbanksysteme gehen noch einen Schritt weiter und erlauben es, anwendungsspezifische Konsistenzbedingungen mit den Sprachmitteln der Datenmanipulationssprache deskriptiv zu spezifizieren. Mit Hilfe sogenannter *aktiver Datenbankmechanismen (*z.B. "'Trigger"' und weitergehende Konzepte) ist es ferner möglich, spezifische Reaktionen beim Erkennen einer potentiellen Konsistenzverletzung festzulegen; beispielsweise können anstatt einer Zurückweisung der Änderung bestimmte Folgeänderungen automatisch ausgelöst werden. Diese Delegation der Verantwortung für die Datenkonsistenz an das Datenbanksystem stellt eine fundamentale Vereinfachung für die Anwendungsentwicklung dar.

Grundprinzip 3:
Die Konsistenz der Daten wird vom Datenbanksystem gewährleistet.

Die Konsistenz der Daten ist nicht nur durch Eingabefehler gefährdet, sondern auch durch "'Interferenzeffekte"' paralleler Änderungsoperationen im Mehrbenutzerbetrieb sowie durch Fehler in Anwendungsprogrammen und Ausfälle des Datenbanksystems selbst. Beispielsweise ist es möglich, daß ein Bankkonto "'Geld verliert"', wenn etwa zwei Operationen zur Einzahlung von jeweils 100 DM zunächst beide den Kontostand lesen, jeweils 100 DM dazu addieren und dann beide denselben Wert in die Datenbank zurückschreiben; die zuerst schreibende Einzahlung wird von der späteren fälschlicherweise überschrieben und geht praktisch verloren. Vermeiden ließe sich dieser Fehler natürlich durch entsprechende Synchronisationsmaßnahmen (z.B. unter Verwendung von Semaphoren) von Seiten der Anwendungsprogramme; dies aber wäre eine erhebliche Komplikation der Anwendungsentwicklung für Mehrbenutzersysteme mit einer entsprechenden Einbuße bei der Entwicklungsproduktivität. Ähnliche Fehlereffekte wie der eben beschriebene ergeben sich potentiell auch beim unkontrollierten Abbruch eines Anwendungsprogramms oder einem Ausfall des Datenbanksystems. Beispielsweise würde eine überweisung, die nach dem Belasten des ersten Kontos, aber vor der Gutschrift auf das zweite Konto durch einen Fehler abgebrochen wird, die Datenbank inkonsistent hinterlassen und wie zuvor "'Geld verlieren"'.

Datenbanksysteme stellen alle Mechanismen bereit, um Fehlereffekte dieser Art garantiert zu verhindern. Dazu wird vom Anwendungsentwickler verlangt, daß Folgen von Datenbankoperationen innerhalb eines Anwendungsprogramms zu konsistenzerhaltenden Einheiten, sogenannten *Transaktionen*, zusammengefaßt und explizit spezifiziert werden. Beispielsweise bilden bei einer überweisung die Änderungsoperationen beider Konten zu-

sammen eine Transaktion. Das Datenbanksystem gewährleistet dann die "'(virtuelle) Isolation"' von Transaktionen im Mehrbenutzerbetrieb, indem es (z.B. für das erste Beispiel) entsprechende Synchronisationsmaßnahmen für parallele Transaktionen automatisch generiert, und die "'Atomarität"' von Transaktionen, indem die Änderungen unvollständig ausgeführter Transaktionen (wie im zweiten Beispiel) mit Hilfe eines entsprechenden Logbuchs automatisch rückgängig gemacht werden. Mit Hilfe des Logbuchs wird darüber hinaus sichergestellt, daß Änderungen abgeschlossener Transaktionen wirklich "'dauerhaft"' sind, also weder durch Ausfälle des Datenbanksystems noch durch sonstige Fehler (z.B. Plattenfehler) verloren gehen. Diese systemgarantierten Eigenschaften von Transaktionen werden auch als *ACID−Prinzip* bezeichnet.

Grundprinzip 4:
Datenbankänderungen werden innerhalb von Transaktionen durchgeführt, die atomar (*atomic*), konsistenzerhaltend (*consistency−preserving*), isoliert (*isolated*) und dauerhaft (*durable*) sind.

Die Unterstützung von Transaktionen ist ein Grundpfeiler aller modernen Datenbanksysteme und ein Schlüsselkonzept zur Sicherstellung der Datenkonsistenz. Die dazu notwendigen Synchronisations− und Fehlertoleranzmaßnahmen werden vom Datenbanksystem sowohl für zentralisierte als auch für verteilte Datenbanken realisiert. Wegen seiner Allgemeinheit und Leistungsfähigkeit finden das Transaktionskonzept und die entsprechenden Implementierungstechniken der Transaktionsverwaltung auch in Betriebssystemen sowie in persistenten Programmiersprachen zunehmende Beachtung. Die X/Open−Gruppe beispielsweise sieht in ihren Standards Transaktionen als universelles Konzept vor, um atomare Prozesse mit Aufrufen beliebiger verteilter Ressourcenmanager zu realisieren.

Der Stand der Technik auf dem Datenbankgebiet läßt sich folgendermaßen zusammenfassen. Datenbanksysteme sind ausgereifte Systeme, ohne die zahlreiche computergestützte Informationssysteme undenkbar wären. Der Markt wird heute von relationalen Systemen dominiert; der z.Zt. noch kleine Anteil objektorientierter Systeme wächst langsam aber ständig, und viele Anbieter relationaler Systeme sind im Begriff, objektorientierte Konzepte in ihre Systeme zu integrieren.

Zukunftsperspektiven

Die Forschungsaktivitäten in der Datenbanktechnik verschieben sich von einer eher technologiegetriebenen zu einer eher anwendungsgetriebenen Ausrichtung. Ein zentrales Thema ist dabei die Interoperabilität zwischen verschiedenen, weitgehend unabhängigen Datenbanken untereinander sowie mit den verschiedenen Anwendungssystemen vor dem Hintergrund einer verteilten, hochgradig heterogenen Informationslandschaft. Ziel ist es, den Anwendungen einen systemübergreifenden Datenzugriff auf möglichst bequeme Art zu ermöglichen. Im folgenden sind drei Beispielszenarien aufgeführt, für die diese Forderung essentiell ist.

- Integrierte Auswertung verschiedenartiger Informationsquellen zur intelligenten Steuerung des Straßenverkehrs. Die Grundlage solcher Leitsysteme bilden einerseits geographische Daten und Daten über das Straßennetz sowie andererseits ständig aktualisierte Daten über die Verkehrsdichte, die Straßenbeschaffenheit, die Wetterlage und −vorhersage, Auslastung von Parkhäusern usw.
- Integrierte und weitestgehend automatisierte Abwicklung von Vorgängen, sogenannten "Workflows", die eine Vielzahl von Informationssystemen verschiedener Institutionen betreffen. Beispielsweise beinhaltet der Erwerb eines Einfamilienhauses Prüfungen und Eintragungen beim Grundbuchamt, der finanzierenden Bank, dem Notar usw. Gleichzeitig sollten auch alle mit dem Umzug verbundenen Formalitäten wie beispielsweise die Ab− und Anmeldung von Gas/Wasser/Strom, Telefon, Internetanschluß usw. elektronisch abgewickelt werden.
- Prüfung von Firmenkrediten bei einer Bank mit entsprechenden Bonitätsprüfungen und Risikoabschätzungen bezüglich des Kreditnehmers sowie seiner finanziellen Verflechtungen mit anderen nationalen und internationalen Unternehmen. Dies beinhaltet die Unterstützung verteilter Arbeitsabläufe, die sich über verschiedene Geschäftssparten und verschiedene internationale Niederlassungen einer Bank erstrecken und Zugriff auf ein breites Spektrum von Informationssystemen erfordern − von Wirtschaftsnachrichten und Börsenkursen bis zu Informationen über die Aufsichtsratsmitglieder des potentiellen Kreditnehmers.

Den Hintergrund derartiger Anwendungsszenarien bildet eine zukünftige Informationslandschaft von großflächigen Netzen mit einer Vielzahl von Rechnern unterschiedlichsten Leistungsvermögens. Diese Netze werden als "offene Systeme" konzipiert sein und damit eine technische Infrastruktur bereitstellen, die das Einbetten und Zusammenwirken von Geräten, System− und Anwendungssoftware unterschiedlichster Herkunft, Funktionalität und Leistung gestattet. Damit ermöglichen diese Netze einen kooperativen Arbeitsstil völlig neuer Qualität, der von dem Stichwort "verteilte Anwendungen" geprägt ist. In zunehmendem Maße werden Anwendungen entstehen, die verschiedenartige Informationssysteme und sonstige Dienste übergreifend nutzen. Hierfür müssen Prinzipien und Methoden erstellt werden, um ein geordnetes Zusammenwirken dieser Vielfalt und Heterogenität zu erreichen.

1.2 Komponenten eines Informationssystems (IS)

IS = Daten + Programme

häufig:

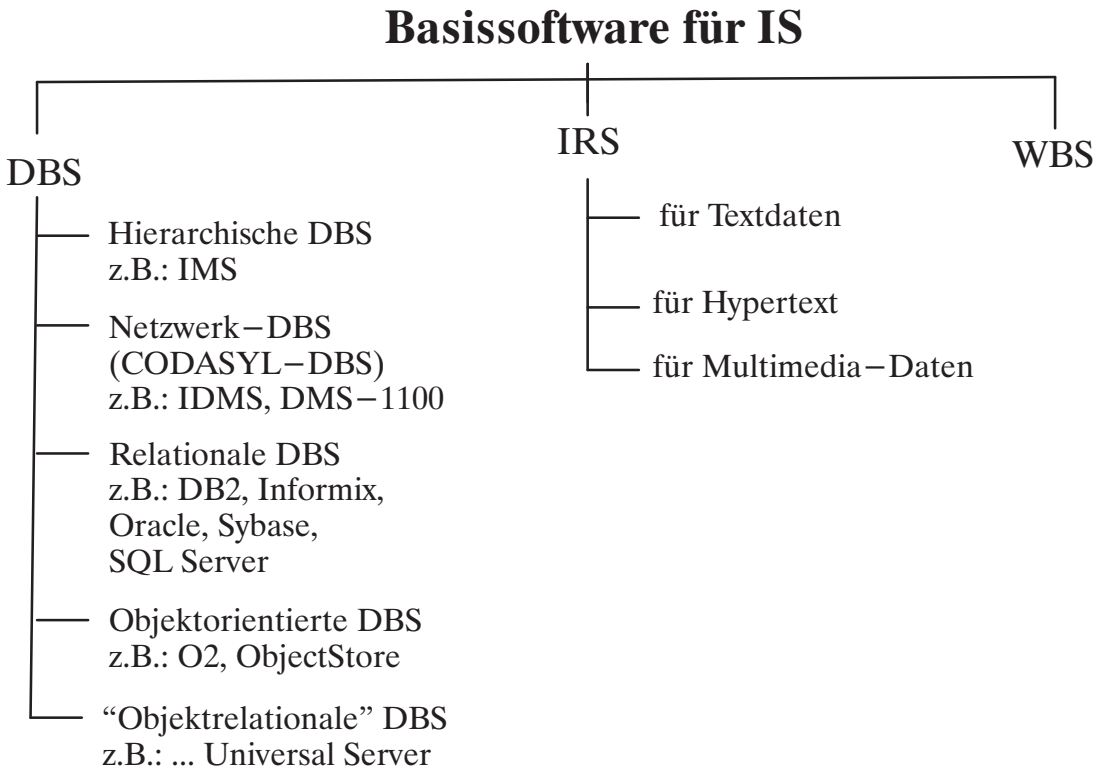
IS = (strukturierte) Daten + } *Teil III der Vorl.*
Datenbanksystem (DBS) + } *Teil I der Vorl.*
Anwendungsprogramme (APs) } *Teil II der Vorl.*
[+ "Middleware" für Client/Server−Architektur
(TP−Monitor, Object−Request−Broker, etc.)]
[+ 4GL, GUIs, ...]

oder:

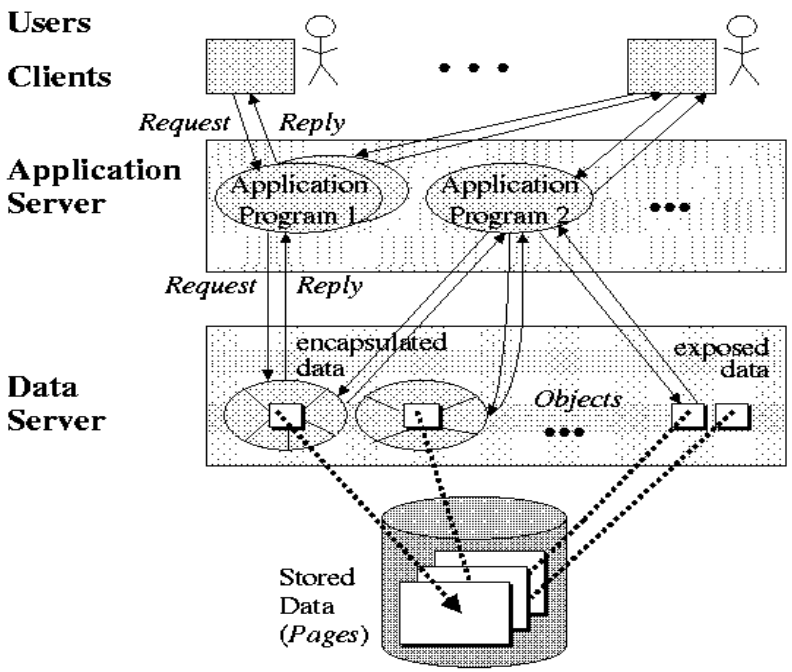
IS = (unstrukturierte) Daten +
Information−Retrieval−System (IRS)
[+ GUIs, APs, ...]

oder:

IS = Daten + Regeln +
Wissensbasiertes System (WBS) +
[+ GUIs, APs, ...]

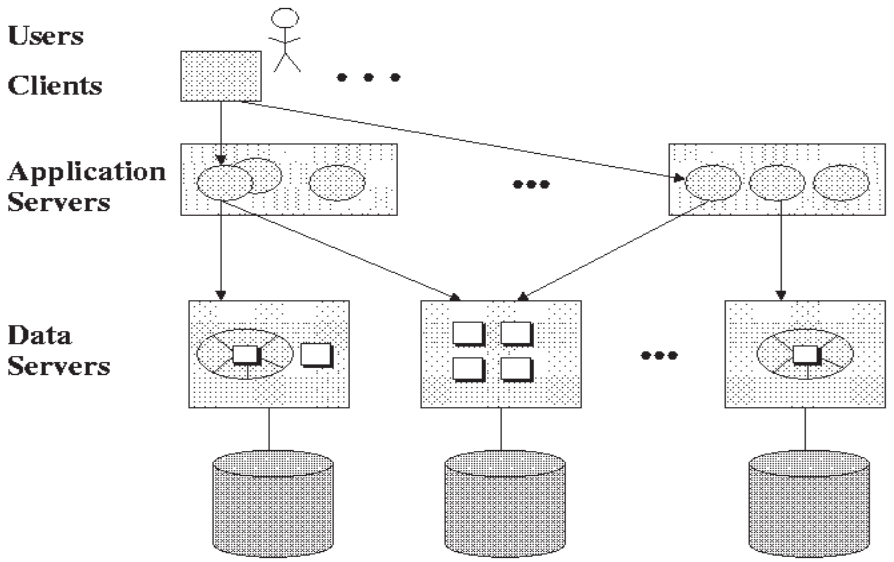


Typische Architektur von IS



Die in der Praxis wichtigsten Varianten sind:
- 2−Ebenen−Architektur: Clients + Daten−Server
- 3−Ebenen−Architektur: Clients + Applikations−Server + Daten−Server

Verteilte (”Föderierte”) Variante



1.3 Vergleich einer IS−Realisierung mit File−System gegenüber einer IS−Realisierung mit einem DBS

mit File−System **mit DBS**

Daten: Kunden, Bestellungen, Produkte, Lager, ...
Anwendungen: Erfassung von Bestellungen, Lieferungen und Zahlungen, Mahnungen, Marketing, ...

Datendefinition:

```
TYPE
Kunde =   RECORD
          KNr: INTEGER;
          Name: ARRAY[1..30] OF CHAR
          Stadt: ARRAY[1..30] OF CHAR;
          Saldo: REAL;
          Rabatt: REAL;
          END;
Produkt=  RECORD
          PNr: INTEGER;
          Bez: ARRAY[1..30] OF CHAR;
          Gewicht: REAL;
          Preis: REAL;
          Lagerort: ARRAY[1..30] OF CHAR;
          Vorrat: INTEGER;
          END;
Bestellung= RECORD
          BestNr: INTEGER;
          Monat: [1..12];

          Tag: [1..31];

          KNr: INTEGER;
          PNr: INTEGER;
          Menge: INTEGER;
          Summe: REAL;
          Status: (neu,geliefert,bezahlt);

          END;
VAR       KFile:   FILE OF Kunde;
          PFile:   FILE OF Produkt;
          BFile:   FILE OF Bestellung;
```

```
CREATE TABLE Kunden (
   KNr INTEGER CHECK (KNr>0),
   Name VARCHAR(30),
   Stadt VARCHAR(30),
   Saldo FLOAT,
   Rabatt FLOAT )

CREATE TABLE Produkte (
   PNr INTEGER CHECK (PNr>0),
   Bez VARCHAR(30),
   Gewicht FLOAT,
   Preis FLOAT,
   Lagerort VARCHAR(30),
   Vorrat INTEGER CHECK (Vorrat>=0))

CREATE TABLE Bestellungen (
   BestNr INTEGER CHECK (BestNr>0),
   Monat INTEGER CHECK
      (Monat BETWEEN 1 AND 12),
   Tag INTEGER CHECK
      (Tag BETWEEN 1 AND 31),
   KNr INTEGER CHECK (KNr>0),
   PNr INTEGER CHECK (PNr>0),
   Menge INTEGER CHECK (Menge>0),
   Summe FLOAT,
   Status VARCHAR(9) CHECK
      (Status IN ('neu','geliefert','bezahlt')))
```

Datendefinition in jedem Programm wiederholt deklariert
−> Gefahr inkonsistenter Definitionen
(IMPORT nicht erzwingbar)

Datendefinition einmal im Datenbankkatalog abgelegt
−> verbindlich für alle APs
(IMPORT erzwungen)

Persistente Speicherung:

Realistisches Datenvolumen (Belieferung von Einzelhandelsfilialen):

Kunden:	10000 Datensätze * 200 Bytes	= 2 MBytes
Produkte:	100000 Datensätze * 200 Bytes	= 20 MBytes
Bestellungen:	50 * 10000 * 10000 Datensätze * 50 Bytes	= 250 GBytes

(ein Kunde bestellt jede Woche ca. 10% der Produktpalette nach)

Eine Datei muß typischerweise auf eine Magnetplatte passen.

Datenbanken können über viele Magnetplatten verteilt sein.

Datenzugriff:

Welche seit dem 21.10. bestellten Produkte am Lagerort Homburg gehen zur Neige (Vorrat < 100)?

```
VAR B: Bestellung;
      P: Produkt;
open (BFile);
WHILE NOT eof(BFile) DO
      B := getnext (BFile);
      IF B.Monat = 10 AND
         B.Tag >= 21
      THEN
            open (PFile);
            WHILE NOT eof(PFile) DO
               P := getnext (PFile);
               IF P.PNr = B.PNr AND
                  P.Lagerort = 'Homburg' AND
                  P.Vorrat < 100
               THEN WriteInt (PNr, Menge);
               END;
            END;
            close (BFile);
      END;
END;
```

```
SELECT PNr, Menge
FROM Bestellungen, Produkte
WHERE Monat = 10 AND Tag>=21
AND Lagerort='Homburg'
AND Vorrat < 100
AND Bestellungen.PNr=Produkte.PNr
```

oder: beide Dateien nach PNr sortieren und danach mischen (vorteilhaft bei wenig Hauptspeicher)

(deklarative) Anfrage bleibt unverändert gültig −> automatische Optimierung

Effizienter Datenzugriff:

PFile ist intern als Suchbaum organisiert
mit PNr als Suchschlüssel
(−>Neues Programm für den Aufbau
von PFile schreiben)

```
CREATE INDEX P−PNr−Index
ON Produkte ( PNr)
```

−> Programm ändern:
innere Schleife ersetzen durch:

```
P := search (B.PNr, PFile);
IF found
THEN
   IF P.Lagerort = 'Homburg' AND
      P.Vorrat < 100
   THEN WriteInt (PNr, Menge);
   END;
END;
```

(deklarative) Anfrage
bleibt unverändert gültig
−> automatische Optimierung mit
Ausnutzung des Index

Noch effizienterer Datenzugriff:

Zusätzlich gibt es einen Suchbaum für
BFile mit Monat*100+Tag als Suchschlüssel

```
CREATE INDEX B−Datum−Index
ON Bestellungen ( Monat Tag )
```

−> Programm ändern:
äußere Schleife modifizieren:

```
FOR schl:= 1021 TO 1024 DO
   B := search (schl, BFile);
   WHILE found DO
      P := search (B.PNr, PFile);
      ...
   END;
END;
```

(deklarative) Anfrage
bleibt unverändert gültig
−> automatische Optimierung
mit Ausnutzung beider Indizes

Berücksichtigung von Zielkonflikten (globale Optimierung von Kosten/Leistung):

Der Suchbaum für BFile ist zu "teuer"
in Bezug auf Speicherplatz oder die
Laufzeit von Einfügeoperationen

```
DROP INDEX B−Datum−Index
```

äußere Schleife wieder
wie zuvor ändern

(deklarative) Anfrage
bleibt unverändert gültig

Datenunabhängigkeit:

”Kapselung” der Datenorganisation:
Programme bleiben bei Änderungen
der physischen Datenorganisation oder
Erweiterungen des ”Datenschemas”
weiterhin unverändert gültig.

Erweiterung der gespeicherten Daten:

1) Ab sofort wird für jede Bestellung zusätzlich eine Lieferfrist (in Tagen) gespeichert.

```
TYPE Bestellung= RECORD
                   ...
                   Frist: CARDINAL;
                 END;
```

```
ALTER TABLE Bestellungen
ADD (Frist INTEGER
CHECK (Frist>0) )
```

Bestellungen in Best umbenennen:

```
UPDATE ALL_CATALOG
SET TABLE_NAME='Best'
WHERE TABLE_NAME='Bestellungen'
```

```
CREATE VIEW Bestellungen AS
SELECT BestNr, Monat, Tag, KNr,
PNr, Menge, Summe, Status
FROM Best
```

−> Datei neu aufbauen und
−> u.U. Programme ändern

−> nur das Programm zur Erfassung von Bestellungen wird geändert

2) Ab sofort kann ein Produkt in mehreren Lagern bereitgehalten werden.

−> wie bei 1)

```
CREATE TABLE Lager
(Ort VARCHAR(30),
PNr INTEGER,
Vorrat INTEGER)
```

```
ALTER TABLE Produkte
DROP Vorrat, Lagerort
```

Produkte in Prod umbenennen

```
CREATE VIEW Produkte
(PNr, Bez, Gewicht, Preis, Vorrat)
AS SELECT
   PNr, Bez, Gewicht, Preis, SUM(Vorrat)
   FROM Prod, Lager
   WHERE Prod.PNr=Lager.PNr
   GROUP BY PNr
```

−> alle Programme, die nur am Gesamtlagervorrat interessiert sind, laufen unverändert

Datenintegrität:

1) Produktnummern und Produktbezeichnungen sollen eindeutig sein.

Integritätsverletzung möglich aufgrund fehlerhafter Eingaben oder fehlender Abstimmung zwischen verschiedenen Programmen (z.B. ein Programm für die Erfassung von Produkten, ein anderes Programm für die Änderung von Produktbezeichnungen)

```
CREATE TABLE Produkte (
PNr INTEGER CHECK (PNR>0)
PRIMARY KEY,
Bez VARCHAR(30) UNIQUE,
... )
−> Integritätsverletzung unmöglich
```

2) Zu jeder Bestellung muß ein Produkt mit der entsprechende PNr gespeichert sein.

Integritätsverletzung möglich, wenn ein Datensatz aus PFile gelöscht wird, ohne zu prüfen, ob es eine Bestellung für das entsprechende Produkt gibt

```
CREATE TABLE Bestellungen (
...,
FOREIGN KEY (PNr)
REFERENCES Produkte
ON DELETE NO ACTION )
−> Integritätsverletzung unmöglich
```

3) Spezifische Integritätsbedingungen:

Kontrolle nur durch Programme

in der Datenbank definiert, z.B.:

```
CREATE TABLE Kunden (
...,
Rabatt FLOAT CHECK (
Rabatt BETWEEN 0.0 AND 0.5 ),
... ,
CONSTRAINT Saldo−Bedingung CHECK (
Saldo = − SELECT SUM (Summe)
          FROM Bestellungen
          WHERE Bestellungen.KNr=KNr
          AND Status <> 'bezahlt' ) )
```

Zugriffskontrolle:

grobe Granulate bzgl. zu schützender Objekte (nur ganze Dateien) und zugreifender Subjekte (Owner/Group/World)

prädikatorientierter Schutz gegenüber individuellen Benutzern und Benutzergruppen, z.B.:

```
CREATE VIEW Meier−Best AS
SELECT * FROM Bestellungen, Kunden
WHERE Name='Meier'
AND Bestellungen.KNr=Kunden.KNr

GRANT SELECT ON Meier−Best TO Meier
```

Kontrollierter Mehrbenutzerbetrieb:

Problem: Potentielle ”Interferenz” zwischen parallel laufenden Programmen,
z.B. zwischen der Erfassung einer Zahlung und einer Suche nach säumigen Zahlern:

Zahlung:

Schritt Z1:
Bestellungsstatus in
’bezahlt’ ändern

Schritt Z2:
Kundensaldo erhöhen

Suche/Mahnungen:

Schritt M1:
Kunden mit negativem Saldo finden
Schritt M2:
Bestellungen der gefundenen
Kunden inspizieren

File−System:

−> Ausführung verhindern durch
P− und V−Operationen
(Lock und Unlock) auf
Semaphoren (Sperren)

DBS:

Zu ”isolierende” Transaktionen gegenüber
dem DBS deklarieren:
CONNECT ...
UPDATE Bestellungen SET Status = ...
UPDATE Kunden SET Saldo = ...
COMMIT
−> automatische Synchronisation

Fehlertoleranz:

Lieferung einer Bestellung:
Schritt L1: Dekrementierung des Vorrats des gelieferten Produkts
Schritt L2: Änderung des Bestellungsstatus von ’neu’ in ’geliefert’

Problem:
Systemausfall zwischen 1 und 2
−> Datenbank wird inkonsistent

−> spezielles ”Recovery”−Programm
schreiben und jeweils nach
Systemausfall starten
(für jeden Fehlerfall ein Programm)

Atomare Transaktionen
(”Recovery−Blöcke”) gegenüber dem DBS
deklarieren −> automatische Recovery,

z.B.:
CONNECT ...
UPDATE Produkte SET Vorrat = ...
UPDATE Bestellungen SET Status = ...
COMMIT