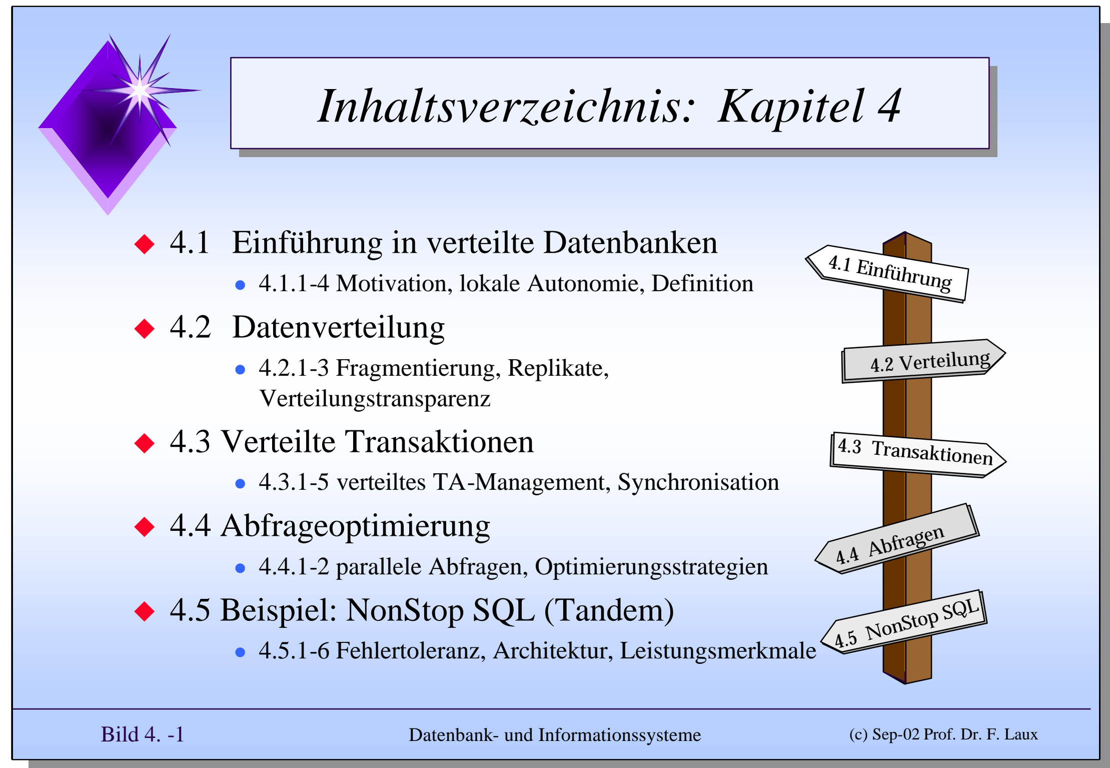
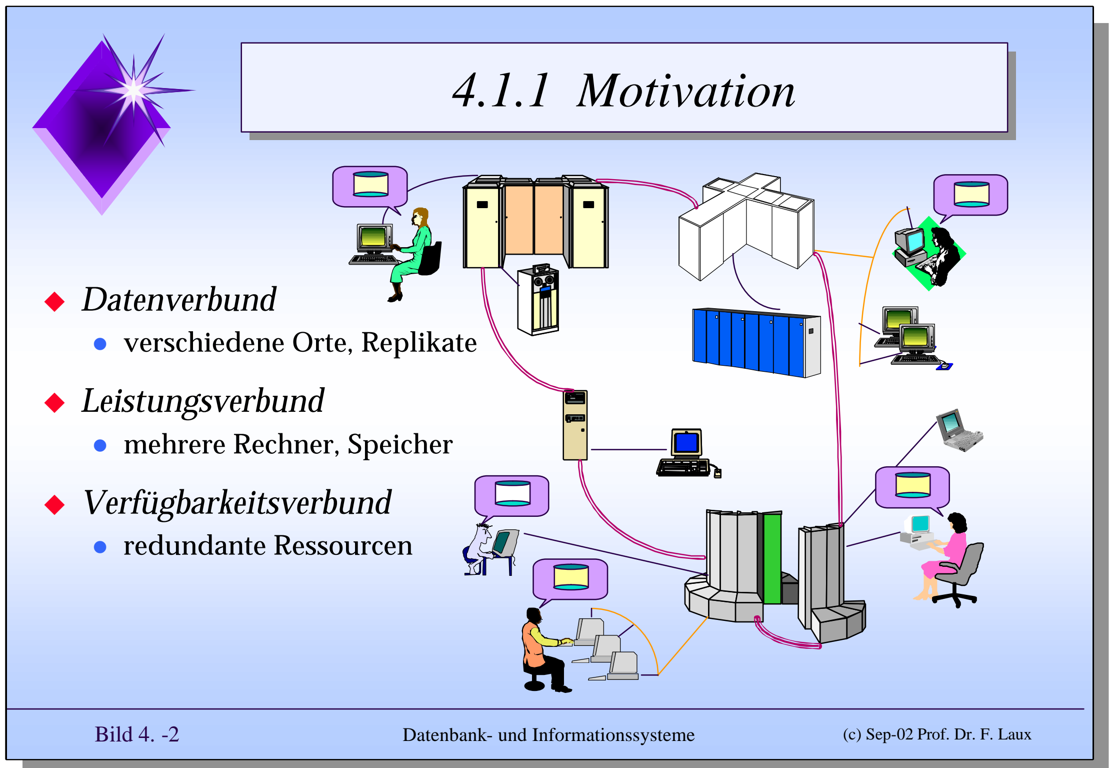


In Kapitel vier wird zunächst eine Einführung und Motivation gegeben, die uns zu einer Definition für verteilte Datenbanken führt. Wir zeigen, wie Daten sinnvoll zu verteilen sind. Kapitel 4.3 behandelt die Besonderheiten des Transaktionsmanagement verteilter Datenbanken. Welches Optimierungspotential in verteilten Systemen steckt, wird am Beispiel der Abfrageoptimierung aufgezeigt.

Zum Abschluß wird ein kommerzielles, fehlertolerantes Rechnersystem mit verteilter relationaler Datenbank vorgestellt.



Die immer komplexer werdenden Strukturen in Handel, Industrie und Dienstleistungs-unternehmen stellen neue Anforderungen an die Informationssysteme. Durch die Globalisierung der Märkte sind Unternehmen über mehrere Länder verteilt, doch ihre betrieblichen Daten müssen von jedem Ort aus erreichbar sein (*Datenverbund*). Simulationsrechnungen und entscheidungsunterstützende Systeme (Decision Support Systems, siehe Kap. 5) erfordern eine hohe Rechnerleistung (*Leistungsverbund*). Ein Flugreservierungssystem oder eine internationale Bank muß einen ständigen Betrieb garantieren, dies kann durch redundante Ressourcen erreicht werden (*Verfügbarkeitsverbund*).
Viele Unternehmen sind von ihrer Organisation her gesehen verteilten Datenbanken viel ähnlicher als zentralen (z.B. eine Bank mit ihren Filialen). Dadurch ist das Modell der verteilten Datenbank adäquater und wirtschaftlicher als das zentrale Datenbankmodell.

Eine verteilte Datenbank ist in einem Unternehmen auch dann sinnvoll einzurichten, wenn schon verschiedene Datenbanken existieren, die bisher noch unabhängig voneinander sind. Durch die Verbindung der Datenbanken werden diese global verfügbar und können immer aktuell und konsistent gehalten werden.

Für wachsende Unternehmen, in denen sich neue Zweigstellen bilden, ist eine verteilte Datenbank ebenfalls sinnvoll. Sobald eine neue Zweigstelle gebildet worden ist und diese eine neue Datenbank benötigt, kann ein neuer Rechner mit lokalem DBS dem Computernetz hinzugefügt werden. Kleinere Einheiten, wie z.B. lokale Datenbanken sind besser überschaubar und daher auch besser planbar, d.h. es stellen sich seltener Diskrepanzen zwischen Datenbank-Kapazität und Anforderungen ein, als dies bei großen Einheiten der Fall ist. Durch die Skalierbarkeit lassen sich Leistungsengpässe leichter vermeiden.

Fällt ein Rechner aus, kann bei redundantem Datenbestand (automatisch) auf eine andere Datenbasis umgeschaltet werden.

4.1.1 Anforderungen

- Eine *verteilte Datenbank (DDBS)* sollte einem Benutzer wie ein zentrales System erscheinen
  - Lokale Autonomie
  - Dauerbetrieb
  - Verteilungstransparenz
  - Parallele Queries
  - Transparenz von Hard- und Software
- Vor- und Nachteile verteilter Datenbanken

+ Datenverbund	- Kosten für Entwicklung
+ lokale Autonomie	- Verwaltungsaufwand (Overhead)
+ Verfügbarkeit	- Komplexität/Fehlerrisiko
+ Performance	
+ Datenintegrität	
+ transparenter Zugriff	

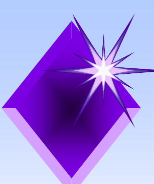
Bild 4. -3 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Der Datenbankexperte *Chris Date* hat seinem Anforderungskatalog folgende Idee vorangestellt:

Eine verteilte Datenbank (Distributed Database System, DDBS) sollte einem Benutzer oder Programmierer wie ein zentrales System erscheinen. Hardware (Computer, Netzwerk) und Software (Betriebssysteme, Datenbankmanagement) sollten transparent sein.

Neben dieser zentralen Forderung ergeben sich durch den Einsatz autonomer Computersysteme (Knoten) in einem Netzwerk weitere Vorteile wie lokale Autonomie, Dauerbetrieb und höhere Leistung (z.B. durch parallele Queries). Aber auch für die Aktualität und Integrität der Daten ergeben sich neue Möglichkeiten: automatischer Datenabgleich und online Wiederherstellung.

Diese Vorteile werden jedoch durch eine höhere Komplexität und damit auch durch höhere Entwicklungskosten und größeren Verwaltungsaufwand erkauft. Trotzdem dürfte sich in vielen Fällen insgesamt ein wirtschaftlicher Vorteil durch die bereits genannten Vorteile ergeben.



4.1.2 Lokale Autonomie

- ***Definition:*** lokal gespeicherte Daten eines DDBS können wie mit einer lokalen Datenbank verwaltet werden.
  - DDBS/Speicherort = lokales DBS; (DDBS ≠ Σ lokale DBS !)
- Eigenschaften/Anforderungen
  - lokal erzeugte bzw. verantwortete Daten werden lokal gespeichert
  - es gibt lokal durchführbare Transaktionen
  - lokales DBS nimmt an globalen Transaktionen teil
  - lokales DBS unterstützt Recovery der DDBS

Bild 4. -4 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Lokale Autonomie bedeutet, daß die lokalen Datenbanken der einzelnen Rechner auch lokal verwaltet werden. Das heißt, daß auf jedem Rechner (Knoten) ein lokales Datenbank-Managementsystem existieren muß, das für die Verwaltung, für Recovery-Mechanismen usw. zuständig ist. Ein Datenbank-Knoten ist dann autonom, wenn diese Voraussetzungen erfüllt sind und somit lokale Transaktionen möglich sind.

Allerdings bedeutet dies nicht, daß die Summe aller lokalen Datenbanken gleich dem verteilten System ist, denn es gibt auch Transaktionen, die nicht lokal ausgeführt werden können. Diese globalen Transaktionen erfordern das Zusammenspiel von mindestens zwei Knoten.

4.1.3 Definition DDBS

- *Verteiltes Datenbanksystem* :=

Ein Datenbanksystem (distributed database system, DDBS), das auf mehrere Computer verteilt ist.

- Die einzelnen Computer besitzen eine *lokale Datenbank* mit gewisser *Autonomie.*
- Jeder Computer nimmt an mindestens einer *globalen* bzw. *verteilten Transaktion* (siehe Kap. 4.3.1) teil.

Bild 4. -5 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Ein *verteiltes Datenbanksystem* (DDBS) ist ein System von Datenbanken, deren Daten miteinander in Beziehung stehen und auf verschiedenen Rechnern an verschiedenen geographischen Orten gespeichert sind. Es existiert keine zentrale Datenbank mehr, sondern jeder einzelne, am Netz angeschlossenen Computer besitzt jeweils eine *lokale Datenbank*. Sämtliche Daten im Netz können von jedem Computer aus abgefragt und bearbeitet werden.

Ein System ist nur dann eine verteilte Datenbank, wenn *lokale Transaktionen* und mindestens eine *globale Transaktionen* möglich sind. Von einer lokalen Transaktion wird gesprochen, wenn sie nur die Datenbank des Computers, von dem aus sie getätigt wird, benötigt (d.h., vom anfragenden Computer). Das bedeutet, daß Datenbanken der anderen Computer im Netz nicht angesprochen werden. Für eine globale Transaktion muß außer der zum anfragenden Rechner zugehörigen Datenbank mindestens noch eine andere angesprochen werden.

Beispiel für eine **lokale Transaktion**: Eine Bank hat verschiedene Filialen. Jede Filiale unterhält eine eigene, lokale Datenbank mit den Daten (Konten) der zu Ihrem Gebiet gehörenden Kunden. Wird nun eine Auszahlung für einen dieser Kunden vorgenommen, so ist nur die zu dieser Filiale gehörende Datenbank betroffen.

Beispiel für eine **globale Transaktion:** Wird eine Überweisung vorgenommen, bei der ein Kunde einer Filiale auf das Konto eines Kunden einer anderen Filiale überweist, so sind davon zwei verschiedene Datenbanken auf zwei verschiedenen Rechnern betroffen. Dabei muß dann natürlich sichergestellt werden, daß entweder in beiden Datenbanken die Änderungen vorgenommen wird oder in keiner (Transaktionsmechanismus).

Die einzelnen Rechner und ihre lokalen Datenbanken sind **autonom**. Das heißt, wenn ein Rechner ausfällt, stehen noch alle anderen Rechner und deren Datenbanken zur Verfügung. Damit wird die Datenverfügbarkeit im Vergleich zu zentralen Datenbanken erheblich verbessert. Selbst wenn der Fall eintreten sollte, daß das Netzwerk ausfällt, bleiben die einzelnen lokalen Rechner verfügbar.

4.1.4 Vergleich

	verteiltes Datenbanksystem	zentrales Datenbanksystem
Kontrolle	lokal und verteilt	zentral
Datenunabh.	physisch, logisch, geographisch	physisch, logisch
Redundanz	größere Redundanz => größere Autonomie	möglichst keine
Effizienz	Zugriffshilfen übers Netz sind problematisch -> RDBS	Zugriffshilfen (Pointer)
Integrität	Konsistenz -> TA, 2PC Recovery -> lokal + global Synchr. -> 2PL, Timestamp	Konsistenz -> TA Recovery -> lokal (Log) Synchron. -> 2PL

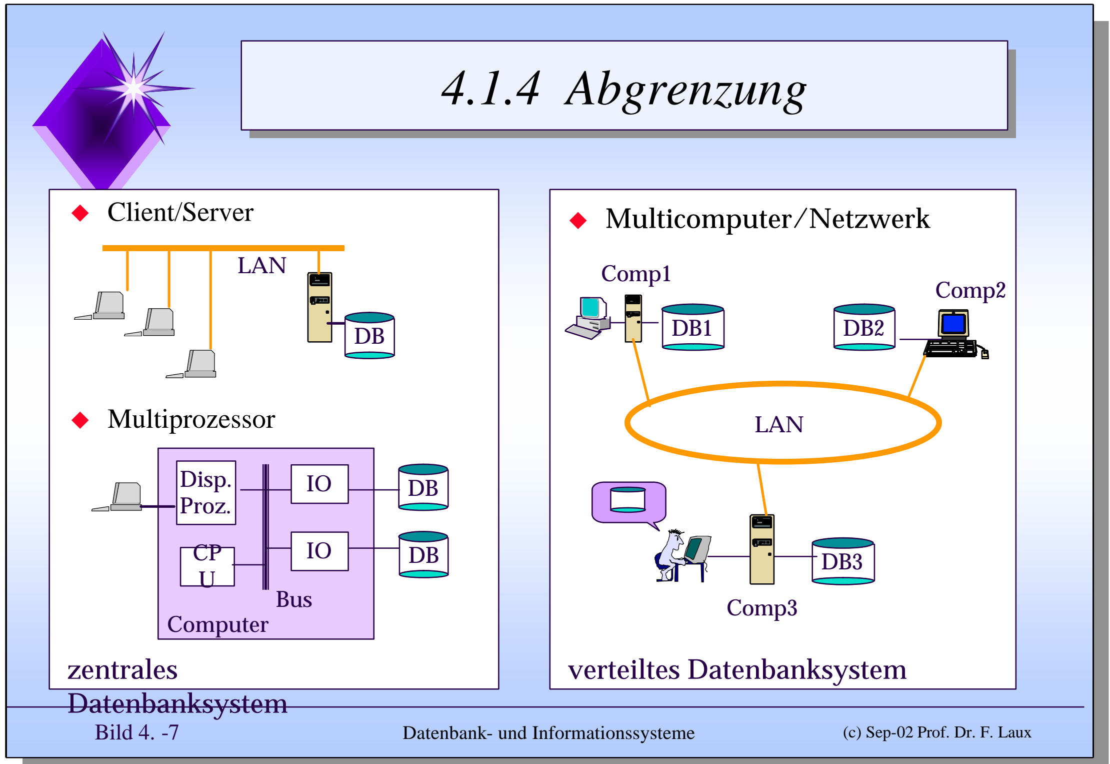
Bild 4. -6 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Wesentlicher Unterschied zwischen zentraler und verteilter Datenbank ist die Datenkontrolle. Bei einer *zentralen Datenbank* existiert ein Datenbankmanager, der alle Daten *zentral verwaltet* und für die Datensicherheit zuständig ist.
Bei einer *verteilten Datenbank* existieren normalerweise zwei verschieden Arten von Datenbankmanagern: Ein *globaler Datenbankmanager*, der für die Koordination sämtlicher lokaler Datenbanken und die Sicherheit der verteilten Daten zuständig ist, sowie auf jedem Rechner ein *lokaler Datenbankmanager*, der nur für die Verwaltung der Daten der jeweiligen lokalen Datenbank zuständig ist. Damit ist die Kontrolle über die Daten - soweit dies möglich ist - *lokal*, sonst ist sie *verteilt*. Der globale Datenbank-Manager sollte aus Sicherheitsgründen nicht ortsgebunden sein, sondern auf einem beliebigen Rechner lauffähig sein.

Bei einer zentralen Datenbank gibt es logische und physische Datenunabhängigkeit (siehe Kap1). Bei verteilten Datenbanken kommt noch eine dritte Art dazu, die geographische Unabhängigkeit. Das bedeutet, daß für Programme und Benutzer die verteilte Datenbank wie eine zentrale erscheinen muß. Das heißt, Daten können von einem Ort an einen anderen verlegt werden, ohne daß die Programme verändert werden müssen.

Bei zentralen Datenbanken besteht die Zielsetzung, jegliche Datenredundanz zu vermieden, da so weniger Speicherplatz benötigt wird und Datenkonsistenz leichter sicherzustellen ist. Bei verteilten Datenbanken muß ein geeignetes Maß an Datenredundanz gefunden werden, um ausreichende Verfügbarkeit und lokale Autonomie zu erreichen bei gleichzeitig akzeptablem Aufwand für den Datenabgleich. Beispiel: Fällt ein Rechner aus und damit seine lokale Datenbank, so ist es sinnvoll, wenn auf einen anderen Rechner ein Replikat (Kopie) davon existiert, damit die Daten weiterhin verfügbar sind. Allerdings müssen im Normalbetrieb die beiden Datenbestände ständig abgeglichen werden.

Zentrale Datenbanken könne durch eine Vielzahl von Zugriffshilfen (Indizes, Pointer) effizient konstruiert werden. Physische Zugriffshilfen (z.B. Adresspointer) sind in Netzwerken problematisch, da damit die geographische Unabhängigkeit der Daten erschwert wird.



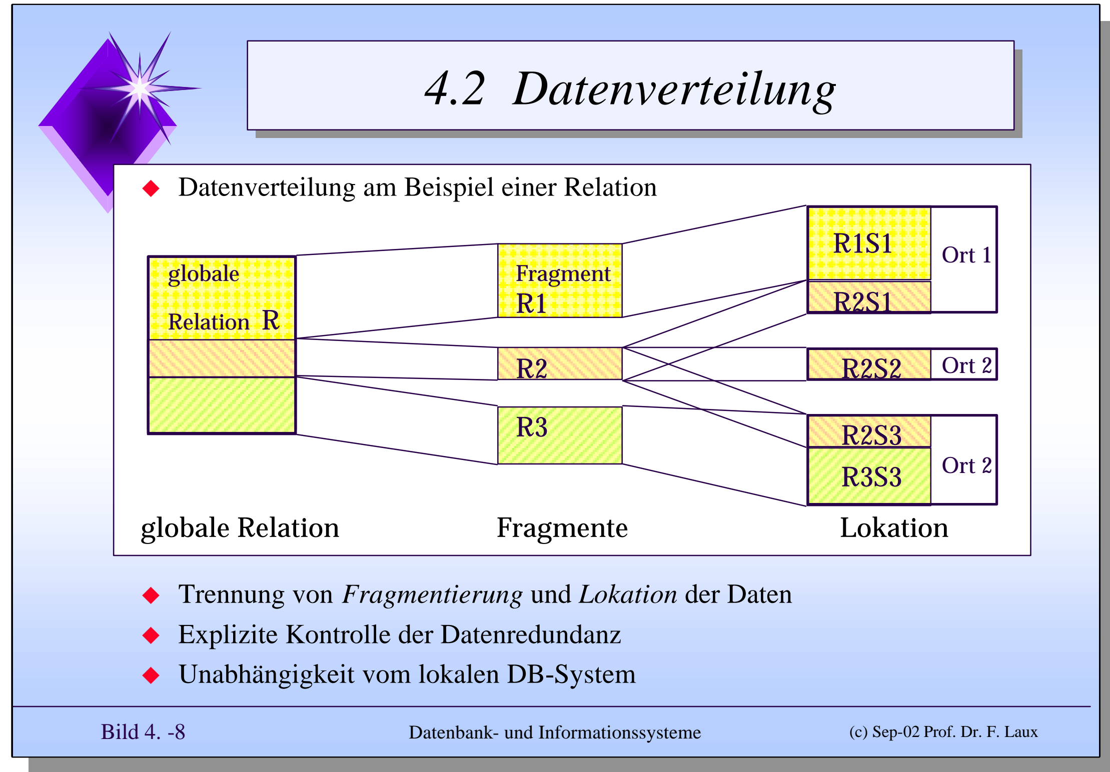
Ein verteiltes Datenbanksystem (DDBS) ist ein System von Datenbanken, welche auf verschiedenen Rechnern verteilt sind, die aber als ein Datenbanksystem benutzt werden. Die Teilsysteme müssen über lokale Autonomie verfügen. (vgl. Def. 4.1.3)

Ein verteiltes Datenbanksystem ist somit ein Multicomputersystem. Das heißt, es existiert keine Einteilung mehr in Client (Rechner, der Anfrage stellt) und Server (Rechner, der Daten zur Verfügung stellt). Bei einem verteilten Datenbanksystem kann jeder Rechner beide Rollen übernehmen, da ja auch jeder Rechner über eine Datenbank verfügt und somit Server werden kann. Es können aber auch über jeden Rechner Daten abgefragt und verändert werden, also ist jeder Rechner auch Client.

Deswegen spricht man nicht mehr wie beim zentralen Datenbanksystem von der Client-Server-Beziehung sondern von einem Multicomputer-System.

Ein klassisches Client/Server System mit zentraler Datenbank hingegen ist kein DDBS, da es nur eine zentrale Datenbank (keine Datenverteilung) aufweist. Es handelt sich jedoch um eine verteilte Anwendung.

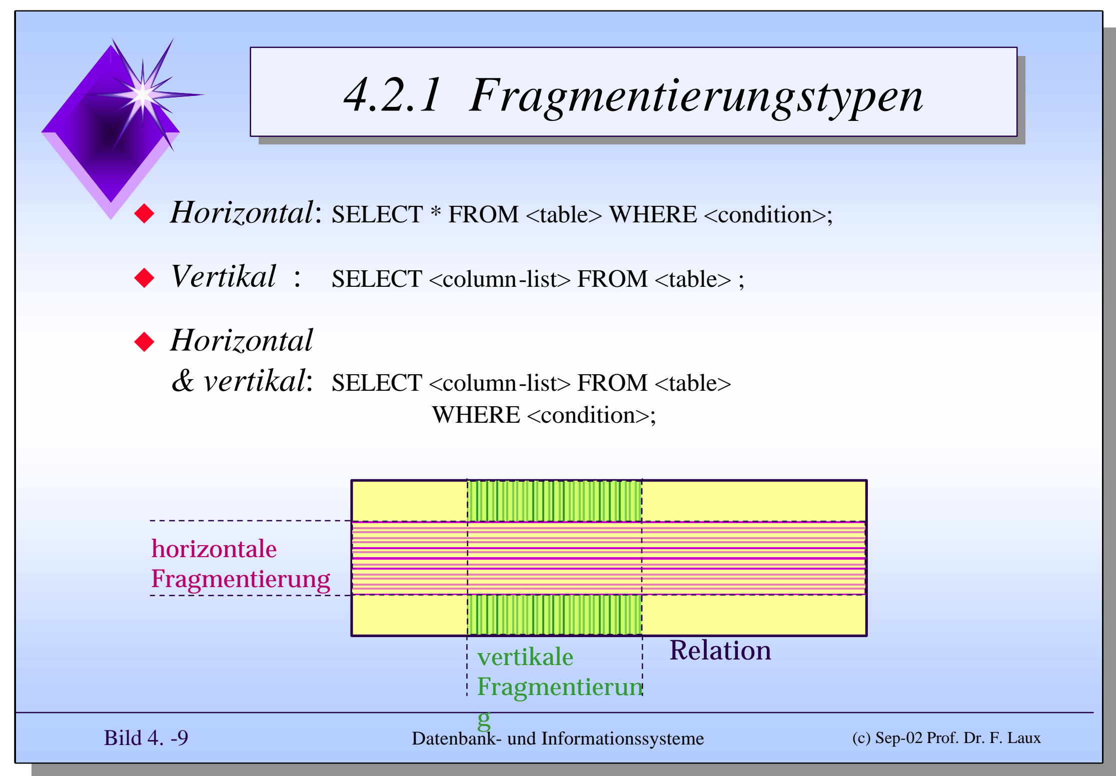
Ein Multiprozessorsystem mit mehreren Speichermedien ist ebenfalls kein DDBS, wenn der Ausfall einer Komponente (z.B. CPU, Display Prozessor, DBMS) das ganze System lahmlegt (keine lokale Autonomie).



Am Beispiel des relationalen Datenmodells wollen wir eine mögliche Datenaufteilung vorstellen.

Unter einer globalen Relation wird hier der komplette Datenbestand einer Tabelle verstanden, bevor sie auf die einzelnen Rechner aufgeteilt wird. Diese globale Relation R wird in Bruchstücke, sog. Fragmente R1 - Rn zerlegt, die jeweils eine eigene Relation bilden. Diese Fragmente wiederum werden auf die einzelnen physikalischen Orte, also die einzelnen Rechner im Netz aufgeteilt. Diese Aufteilung erfolgt unter dem Gesichtspunkt der Verfügbarkeit und lokalen Autonomie. Deshalb wird ein Fragment oft auf verschiedenen Rechner gespeichert (Replikate).

Durch diese zweistufige Aufteilung ist eine fachliche (konzeptionelle) und zusätzlich eine physische (Speicherorte) Trennung der Daten möglich. Die fachliche Trennung wird durch die Fragmente ermöglicht; die physische Trennung wird im Hinblick auf Performance und geographische Unabhängigkeit vorgenommen. Dies erlaubt die gleichzeitige Optimierung hinsichtlich Leistung und konzeptioneller Strukturierung.



Beispiel für eine Fragmentierung:

Globale Relation:

Firma

Abt-Nr	Abteilung	Anzahl Mitarb.
00922	Einkauf	25
10025	Verkauf	21
10083	Produktion	83

Horizontale Fragmentierung (aus der Relation 'Firma' werden die beiden Fragmente 'Firma1' und 'Firma2' gebildet:

Firma1

Abt-Nr	Abteilung	Anzahl Mitarb.
00922	Einkauf	25

Firma2

Abt-Nr	Abteilung	Anzahl Mitarb.
10025	Verkauf	21
10083	Produktion	83

Vertikale Fragmentierung:

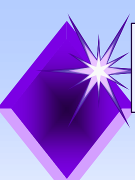
In beiden Fragmenten ist die Abt-Nr als ID-Schlüssel wiederholt. Diese Redundanz ist erforderlich, um die Originalrelation rekonstruieren zu können.

Firma3

Abt-Nr	Abteilung
00922	Einkauf
10025	Verkauf
10083	Produktion

Firma4

Abt-Nr	Anzahl Mitarb.
00922	25
10025	21
10083	83



4.2.1 Kriterien einer Fragmentierung

- *Vollständigkeit* $(R = \cup F_i)$
  - Die gesamte globale Relation muß vollständig in Fragmente aufgeteilt werden
- *Rekonstruierbarkeit* $(r \in R \Rightarrow \exists f \text{ mit: } r = f(F_i))$
  - Die globale Relation muß aus ihren Fragmenten rekonstuierbar sein
- *Disjunktheit* $(F_i \cap F_k = \emptyset)$
  - Ein Element darf nur zu genau einem (horizontalen) Fragment gehören. Überlappungen sind nur bei Schlüsselfeldern erlaubt.

Bild 4. -10 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Vollständigkeit:

Die globale Relation muß vollständig in Fragmente eingeteilt werden, sonst gehen eventuell Daten verloren.

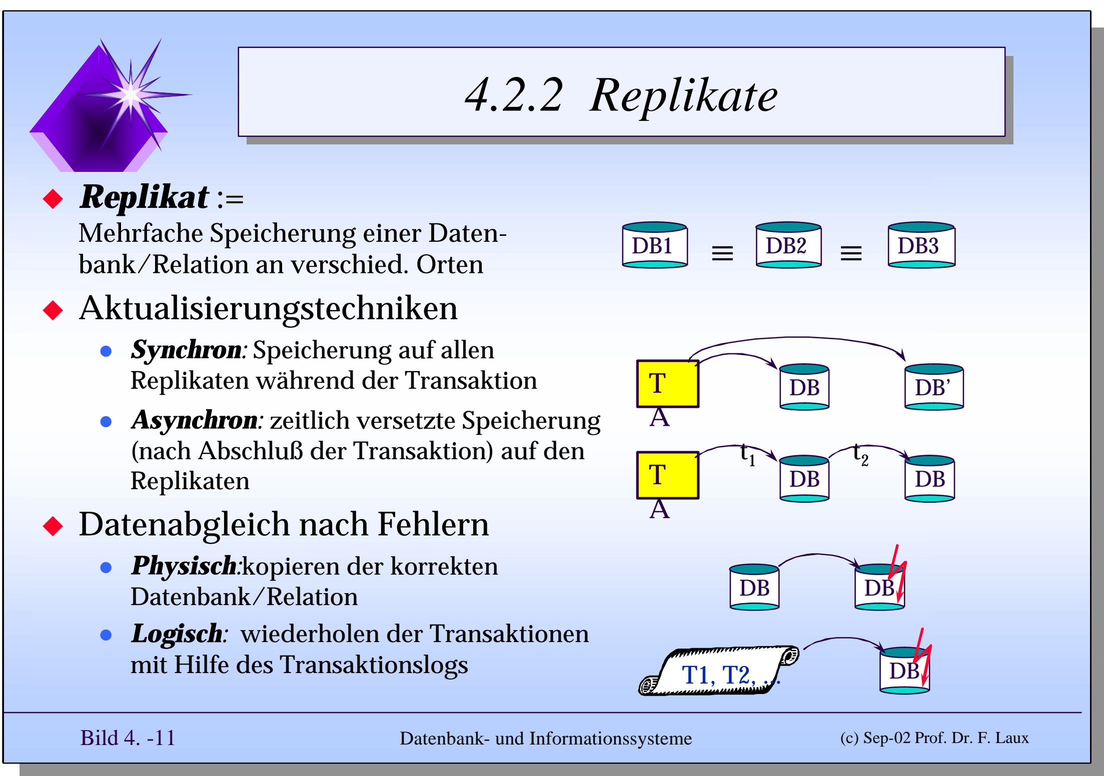
Rekonstruierbarkeit:

Die einzelnen Fragmente müssen so gewählt werden, daß es möglich ist, die ursprüngliche globale Relation aus den einzelnen Fragmenten wieder herzustellen.

Disjunktheit:

Jedes Element der globalen Relation darf nur genau zu einem Fragment gehören. D.h., die Daten in den Fragmenten dürfen sich nicht wiederholen. Eine Ausnahme bilden die Schlüssel bei der vertikalen Fragmentierung, diese müssen bei vertikaler Fragmentierung wiederholt werden, um die Rekonstuierbarkeit sicherzustellen.

ÜA: Erstellen Sie die SQL-Selektionen für das Fragmentierungsbeispiel der globalen Relation *Firma* von Seite 4-9.



Unter einem Replikat versteht man die Speicherung der gleichen Daten an verschiedenen Standorten. Z.B. kann die Kundendatei einer Bankfiliale, die bei dieser Filiale lokal gespeichert ist, zur Sicherheit in der Bankzentrale nochmals als Kopie (Replikat) gespeichert werden. Dadurch entstehen natürlich Redundanzen und damit die Gefahr von Inkonsistenzen. Deshalb müssen die Replikate auf den gleichen Stand gehalten werden. Dies kann entweder synchron oder asynchron geschehen. Im letzteren Fall muß sichergestellt werden, daß nur die aktuellen Daten gelesen werden oder dass bewusst evtl. nicht ganz aktuelle Informationen benutzt werden. Zyklische Aktualisierung, z.B. wöchentlich oder täglich ist bei nahezu statischen Daten akzeptabel.

Dies ist anders bei Tabellen und ihren Replikaten, die permanent geändert werden. Sie müssen ständig synchron gehalten werden. Hier besteht ebenfalls die Möglichkeit, die Replikate (mit Ausnahme eines, nämich dem Original) nur für Lese-Zugriffe freizugeben. Damit ist die Synchronisation dann nur unidirektional durchzuführen. Die Daten des Replikats stehen dann wie oben nur zum Lesen zur Verfügung und alle Änderungen erfolgen auf dem Original.

Dies ist aber nur dann sinnvoll, wenn die Änderungen hauptsächlich an einem Ort stattfinden und die Lesezugriffe über das Netzwerk zu zeitraubend sind.

Replikate sind dann sinnvoll, wenn die Daten oft an verschiedenen Orten benötigt werden und die Zugriffszeiten sonst zu lang würden. In solchen Fällen sind sie essentiell für die Effizienz des verteilten Systems. Hat z.B. eine Firma mehrere Filialen, von denen aus oft Zugriffe auf die Stammdaten der fabrizierten Artikel gemacht werden, so ist es sinnvoller, auf allen Filialen ein Replikat der Daten zu erstellen mit permanentem Abgleich und diese als Nur-Lese-Tabelle zu definieren.

Generell läßt sich sagen, daß Replikate dann sinnvoll sind wenn

- häufige Lesezugriffe erfolgen oder
- hohe Autonomie gefordert ist.

4.2.3 Ebenen der Transparenz

- *Verteilungstransparenz*
  - Vollständige Transparenz der Fragmentierung und Lokation einer globalen Relation
  - Bsp.: *select * from Material;*
- *Fragmentierungstransparenz*
  - Transparenz der Fragmentierung einer Relation
  - Bsp.: *select * from Material at site Lager;*
- *Ortstransparenz*
  - Transparenz der physischen Lokation eines Fragments einer Relation
  - Bsp.: *select * from Material.Fragm1 union select * from Material.Fragm2;*
- *Lokale Transparenz*
  - Transparenz der lokalen Daten (wie bei einem zentralen DBS)
  - Bsp.: *select * from Material.Fragm1 at site Lager;*

Bild 4. -12 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

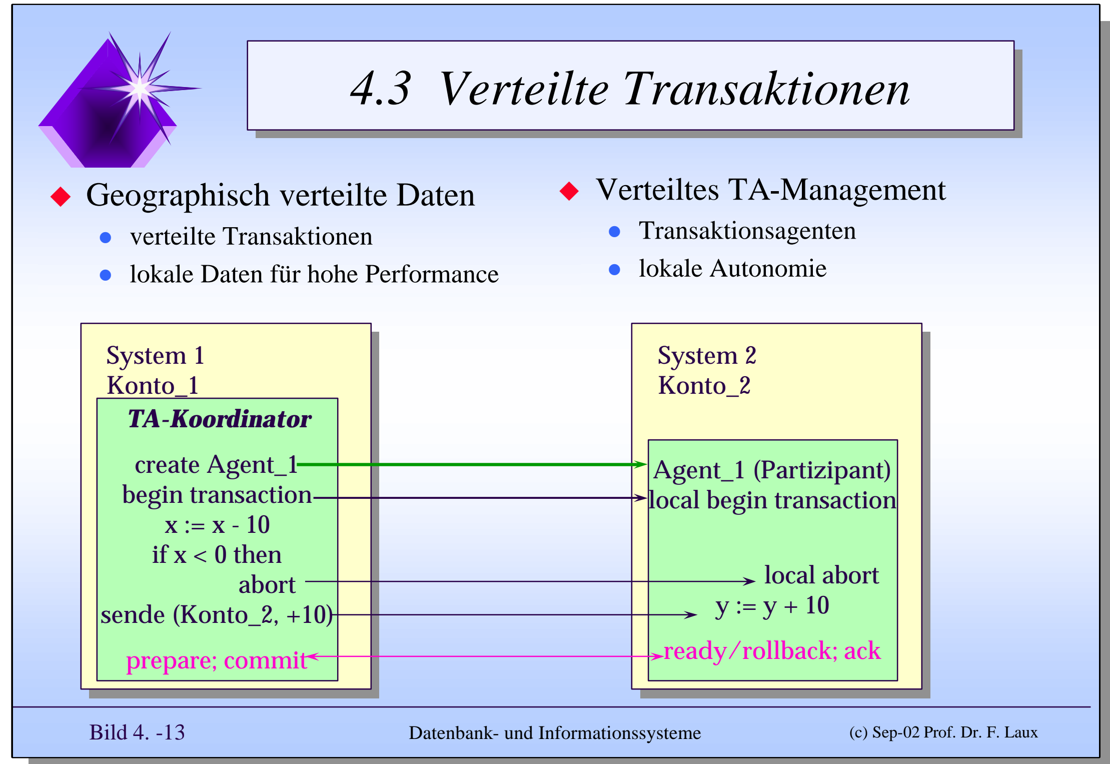
Wir verwenden den Begriff Transparenz im Sinne von unsichtbar (durchsichtig) für einen Benutzer oder ein Programm.

Bei Verteilungstransparenz ist weder die Fragmentierung noch die Speicherlokation sichtbar.

Wir haben es mit Fragmentierungstransparenz zu tun, wenn die Fragmentierung unsichtbar ist aber die Speicherorte sichtbar sind. Umgekehrt ist es bei der Ortstransparenz.

Lokale Transparenz bezieht sich auf die Unsichtbarkeit der lokalen Speichermedien. Diese Eigenschaft wird üblicherweise von einem zentralen oder verteilten DBS erfüllt.

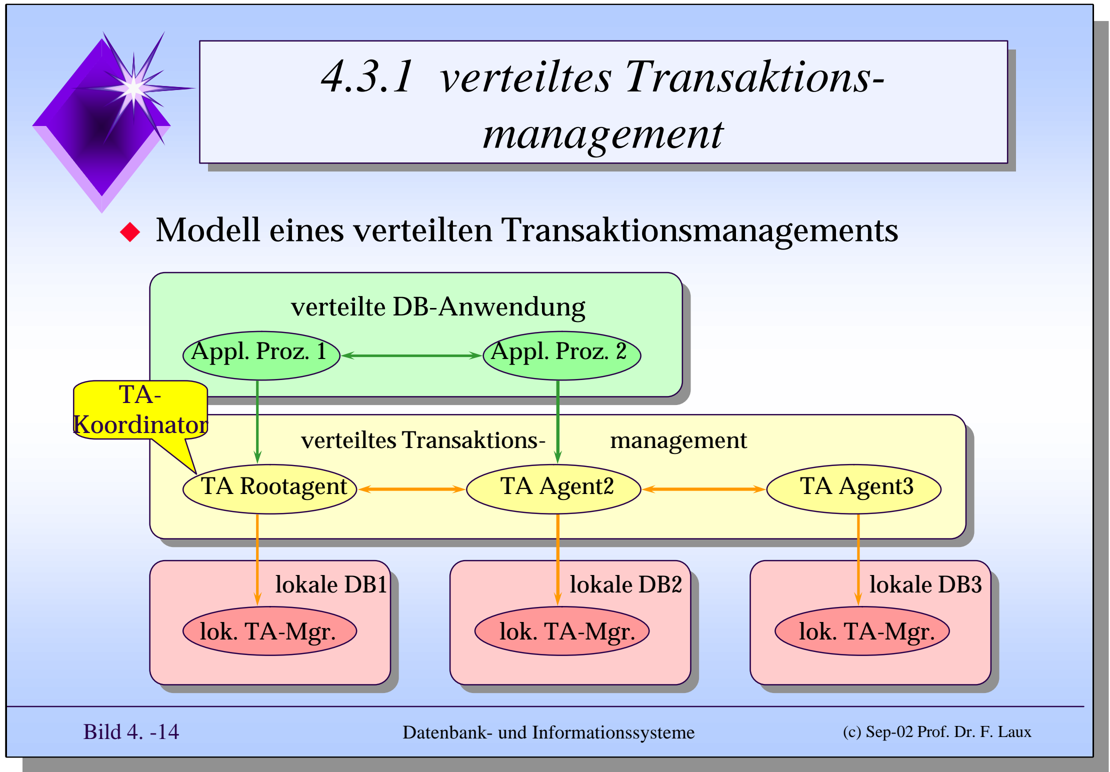
Bei einem verteilten Datenbanksystem werden im **Allokationsschema** als Teil des internen Schemas die Tabellenfragmente und ihre Speicherorte definiert.



Das Management von *verteilten Transaktionen* ist weitaus komplizierter als bei nicht verteilten. Da eine verteilte Transaktion nicht nur an einem Ort ausgeführt wird, sondern mehrere Rechner beteiligt sind, die sich an unterschiedlichen Orten befinden, muß sichergestellt werden, daß die Durchführung von allen beteiligten Rechnern vollständig und korrekt erfolgt. Nur so kann gewährleistet werden, daß danach die Datenbestände wieder konsistent sind. Für diese Aufgabe gibt es einen Koordinator, den sogenannten Transaktions-Manager. Dieser läuft meist auf demjenigen Rechner, von dem aus die Transaktion gestartet wird. Er kann aber auch auf jedem anderen Rechner sich befinden. Dieser Transaktions-Manager leitet die einzelnen Teilanfragen an Hilfsprozesse (*TA-Agenten*) auf den beteiligten Rechnern weiter und muß auch für deren Koordination sorgen. Die Agenten sind für die lokale Abwicklung der Transaktion zuständig.

Beispiel: Bei einer Bank (System1) soll eine Abbuchung von *?* 10,- von Konto_1 bei dieser Filiale und eine gleichzeitige Zubuchung des Betrages auf Konto_2 bei einer anderen Filiale (System2) getätigt werden. In diesem Fall muß gewährleistet sein, daß auch beide Buchungen vorgenommen werden. Wird nur die Abbuchung vorgenommen, aber die Zubuchung kann von dem anderen Computer nicht durchgeführt werden (da das Konto z.B. gerade für eine andere Transaktion gesperrt ist), so würden die Kontostände verfälscht werden. Es muß also einen Koordinator für die Transaktion geben, der feststellt, ob sie komplett durchgeführt werden kann oder nicht.

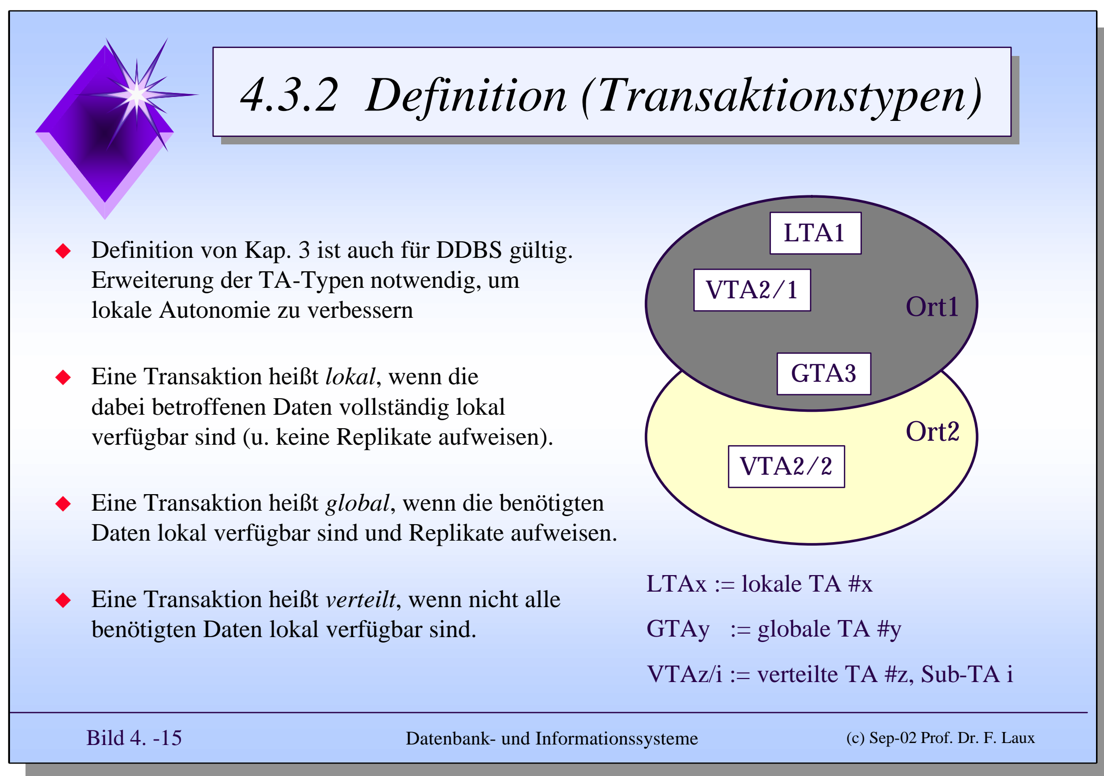
Mit dem verteilten Transaktionsmanagement werden die gleichen Ziele verfolgt, wie bei einer zentralen Datenbank. Der Unterschied liegt in der Komplexität wie uns die folgende Folie zeigt.



Eine weitere wesentlich Eigenschaft verteilter Datenbanken ist ihre verteilte Funktionalität. Eine Datenbankanwendung kann auf mehrere Rechner bzw. Prozessoren verteilt sein. Die für das Gesamtsystem zuständige Transaktionsverwaltung sollte auf wenigstens 2 Orte verteilt sein, um gegen Einzelfehler gesichert zu sein. Eine Optimale Verfügbarkeit wird durch die Verteilung auf möglichst viele Speicherorte erzielt. Da es sich dabei um eigenständige Prozesse handelt, ist ein Kommunikationsprotokoll, das ein einheitliches Vorgehen sicherstellt, unabdingbar. Deshalb verwenden DDBS das Zwei-Phasen-Commitment Protokoll (Kap. 3.2.4).

Die Daten eines Speicherortes werden von einem lokalen TA-Manager verwaltet, der auch mit der verteilten Transaktionsverwaltung (TA-Agent) zusammenarbeitet. Dies bedeutet, lokale Aufgaben werden lokal wahrgenommen, verteilte Transaktionen werden unter Mitwirkung lokaler und verteilter Transaktionsverwaltung durchgeführt.

Der TA-Manager muß verschiedene Bedingen und Transaktionstypen unterscheiden, um maximale Autonomie und Verfügbarkeit zu ermöglichen, wie wir auf den folgenden Seiten sehen werden.



Die Transaktionsdefinition von Kapitel 3 gilt auch für verteilte Datenbanken. Zum besseren Umgang mit Redundanzen und Fehlersituationen unterscheiden wir die 3 oben definierten Transaktionstypen.

Besondere Beachtung verdienen *globale Transaktionen*. Diese haben lokale Datenreplikate. Um weitgehende *lokale Autonomie* zu erzielen, kann in bestimmten Fällen (wird durch Konsistenzbedingungen festgelegt, siehe Folie 5-15) auf die Verfügbarkeit der entfernten Replikate verzichtet werden, so daß die Transaktion lokal durchgeführt werden kann. In diesem Fall muß zu einem späteren Zeitpunkt ein Datenabgleich erfolgen.

Lokale Transaktionen entsprechen den Transaktionen eines zentralen Datenbanksystems. Echt *verteilte Transaktionen* können offenbar nur durchgeführt werden, wenn alle benötigten Daten auch verfügbar sind.

Im Bereich mobiler Rechner (mobile computing) werden Transaktionen in eine Warteschlange gestellt, wenn keine online-Verbindung zur Datenbank besteht. Dabei wird davon ausgegangen, daß die Transaktion zu einem späteren Zeitpunkt (automatisch) durchgeführt werden kann. Ob diese Technik Sinn macht, hängt von fachlichen Bedingungen ab. Ein Lagersystem ist auf diese weise nicht zu führen, wohl aber eine Spesenabrechnung oder Auftragserfassung.

ÜA: Erklären Sie, warum eine Lagerhaltung nur mit online-Verbindung zu verwirklichen ist. Zeigen Sie an einem Beispiel, was beim offline-Betrieb passieren könnte.

ÜA: Nennen Sie Beispiele für Transaktionen, die verzögert (off-line) durchgeführt werden können, ohne dass diese zu schweren betrieblichen Beeinträchtigungen führen.

4.3.2 Konsistenzbedingungen

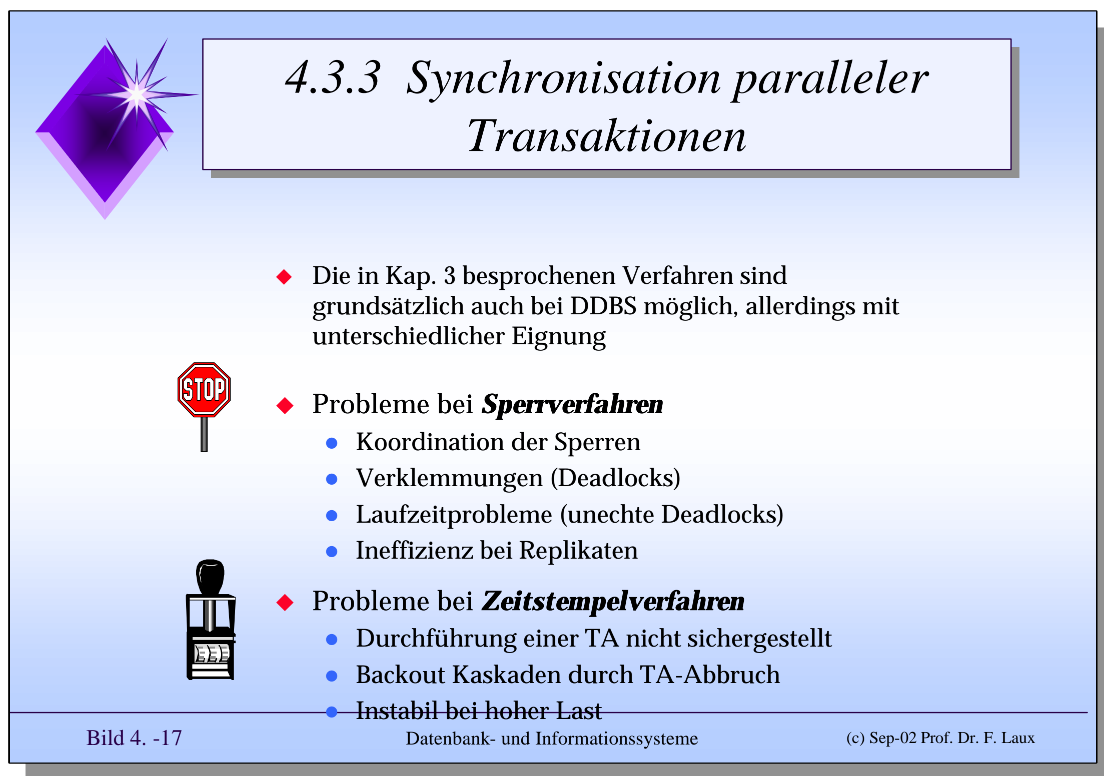
- *Starke* (strikte) *Konsistenzbedingung* (KB) :=
  Eine KB, die stets einzuhalten ist
- *Schwache* (weiche) *KB* :=
  Eine KB, die bei einer Teilverfügbarkeit des
  Systems nicht eingehalten werden muß
- Beispiel:
  Ein vert. DBS besitze eine Relation R, die an 2 Orten S1 und S2 gespeichert sei.
  Die starke KB könnte lauten: R muß an Ort S1 verfügbar sein (Original, Master Copy).
  Die schwache KB könnte dann lauten: R muß an Ort S2 verfügbar sein (Kopie, Replikat).
  => Das Original muß zur Durchführung der TA immer verfügbar sein; auf die Kopie
  kann bei Teilverfügbarkeit verzichtet werden. (Recovery: s. 4.2.2 u. 4.3.6)

Bild 4. -16 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Für verteilte Datenbanken müssen die Konsistenzbedingungen in zwei Gruppen eingeteilt werden, die orthogonal (unabhängig) zu den Definitionen in Kap. 3.2 sind:

Konsistenzanforderungen, die unbedingt einzuhalten sind, nennen wir starke bzw. strikte Bedingungen.

Schwache Konsistenzbedingungen werden bei Teilverfügbarkeit des DDBS außer Kraft gesetzt. Dadurch können globale Transaktionen lokal durchgeführt werden. Während der Teilverfügbarkeit muß sichergestellt sein, daß nur mit aktuelle Daten gearbeitet wird, d.h. die veralteten Replikate dürfen nicht verwendet werden. Wenn das System wieder voll funktionsfähig ist, müssen die Replikate aktualisiert werden.



Durch autonome, parallele Prozesse, unterschiedlichen Laufzeiten von Nachrichten und durch Datenreplikate erweist sich die Synchronisation von parallelen Transaktionen als schwieriger als bei zentralen Datenbanken. Es kann zu Verklemmungen im Gesamtsystem kommen, die lokal nicht erkannt werden können.

Beispiel: Externe Transaktionen halten lokale Ressourcen belegt. Ohne Kenntnis der externen Situation ist es nicht möglich zu entscheiden, ob eine Verklemmung vorliegt.

Durch Laufzeiten bedingt, können auf globaler Ebene scheinbare (unechte) Sperren oder Verklemmungen auftreten.

Beispiel: Wird die Freigabe einer Sperre asynchron weitergemeldet, so kann eine Neuanforderung abgewiesen werden obwohl die Sperre bereits aufgehoben wurde oder es kann ein Deadlock angezeigt werden, obwohl die Verklemmung bereits aufgelöst ist.

4.3.4 Sperrverfahren

- Um die Effizienz von Sperrverfahren bei Replikaten zu verbessern, sind folgende Varianten möglich:

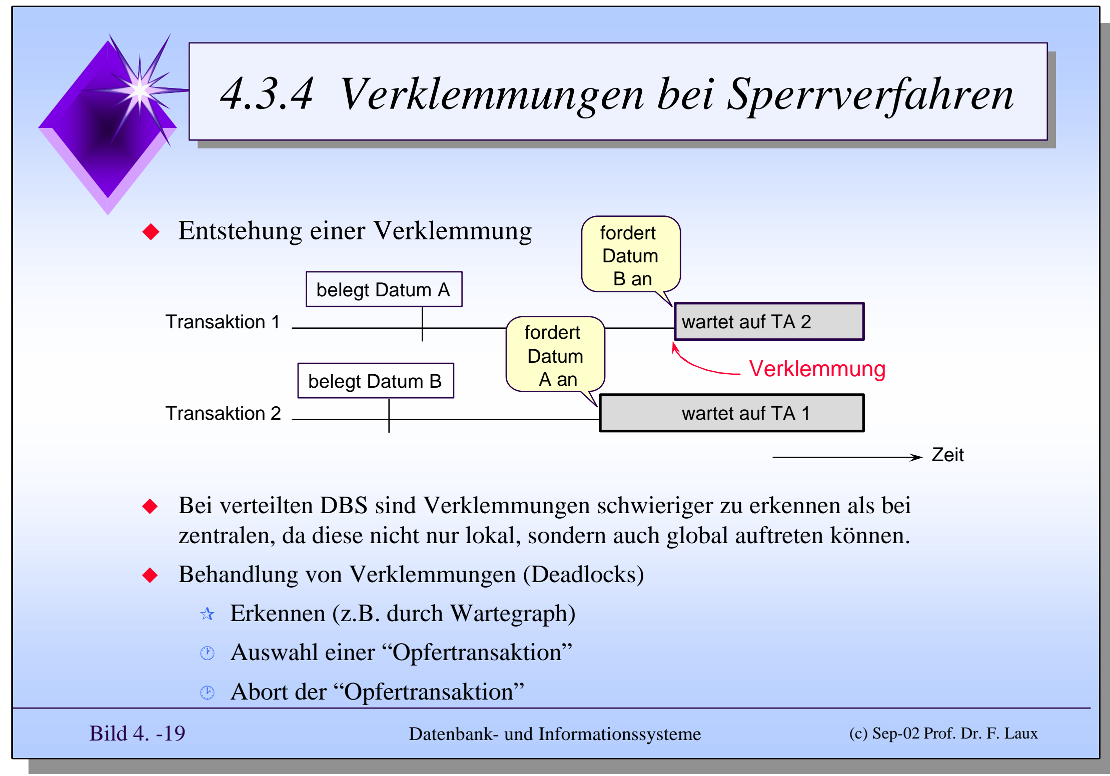
- Majoritätsprotokoll
  - die Mehrheit der Replikate wird gesperrt
- Ungleichbehandlung
  - bei WRITE-Lock werden alle Replikate gesperrt
  - bei READ-Lock wird nur 1 Replikat gesperrt
- Primärkopie
  - nur das Original wird gesperrt

Bild 4. -18 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Das ***Majoritätsprotokoll*** ist hinreichend für ein Sperrverfahren, da nur eine Transaktion die Mehrheit der Replikate sperren kann.

Bei der ***Primärkopie*** wird ein Replikat als Original ausgezeichnet; alle Sperren werden auf dieses angewandt. Dadurch erhält man die gleiche Sperrsituation, wie wenn keine Replikate vorhanden wären. Nachteilig bei diesem Verfahren ist, daß bei einem fehlerhaften Original die intakten Replikate auch nicht mehr verwendet werden dürfen.

Das ***Ungleichbehandlungsprotokoll*** erkennt ebenfalls alle Sperrkonflikte sicher, da eine Schreibsperre auf alle Replikate gesetzt wird. Es ist besonders bei vielen Lesezugriffen effizient.



Entstehung von Verklemmungen (Deadlocks):

Verklemmungen entstehen dadurch, daß Transaktionen Ressourcen *exklusiv* belegen und sich dabei wechselseitig behindern (blockieren). Da jede Transaktion auf eine Ressource wartet, die bereits belegt ist, entsteht eine Verklemmung.

Vermeidung eines Deadlocks:

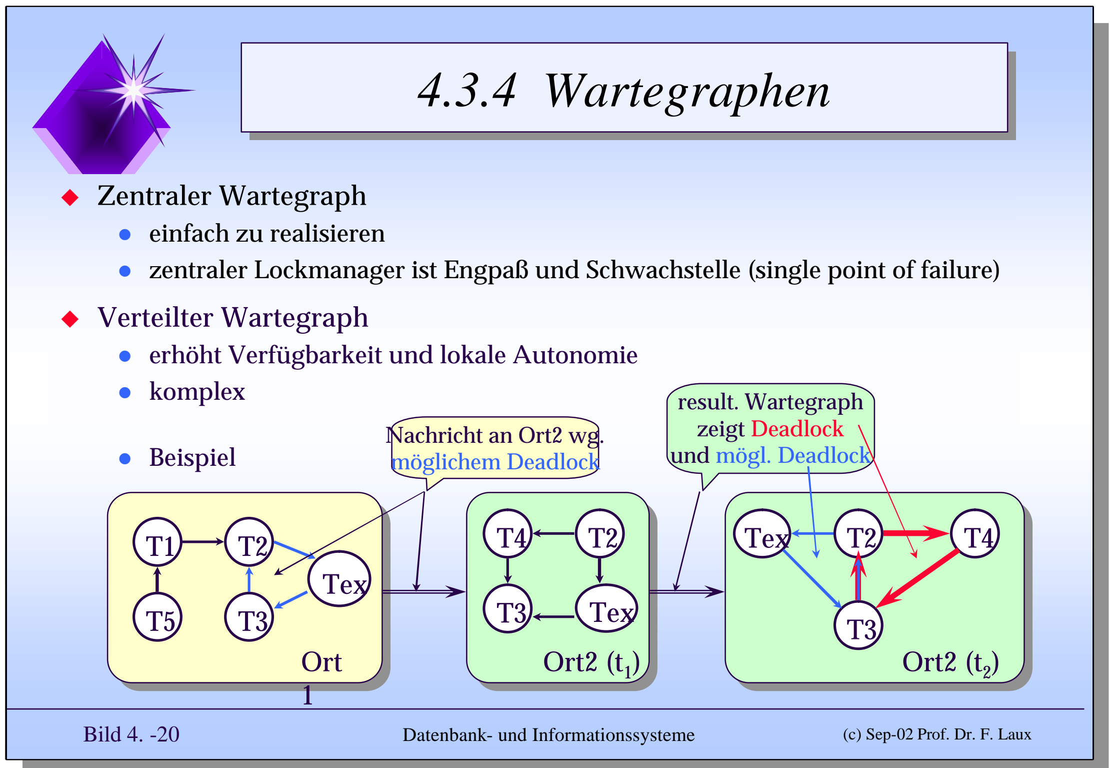
Zur Vermeidung eines Deadlocks überprüft der Transaktionsmanager vor Durchführung der Transaktion, ob diese vollständig durchführbar ist oder ob sie eine Verklemmung erzeugen würde. Der Benutzer hat dann die Möglichkeit, auf die Freigabe der Ressourcen zu warten oder die Transaktion nicht durchzuführen. Der Vorteil dieser Methode ist, daß weder Rollback noch Neustart der Transaktion notwendig ist. Für diese Überprüfung muß natürlich schon vor Ausführung der Transaktion klar sein, welche Sperren benötigt werden. Dies ist auch der Nachteil dieser Methode. Oft wird erst während der Durchführung einer Transaktion klar, welche Daten (und damit auch Sperren) benötigt werden..

Die zweite Art, Verklemmungen zu vermeiden, ist, das **Timestamp-Verfahren**, das später (Kap. 4.3.5) erläutert wird.

Behandlung von Deadlocks:

Das Transaktionsmanagement sollte eine Verklemmung erkennen und diese auflösen. Dazu muß eine der beteiligten Transaktionen abgebrochen werden, um den Verklemmungskreis aufzubrechen. Die Kriterien nach denen eine "Opfertransaktion" ausgewählt wird, können willkürlich sein. Günstig ist eine Auswahl nach dem geringsten Aufwand für das Rollback oder nach der kürzesten Transaktionsdauer.

Wie Verklemmungen erkannt werden können, wird auf der nächsten Folie gezeigt.

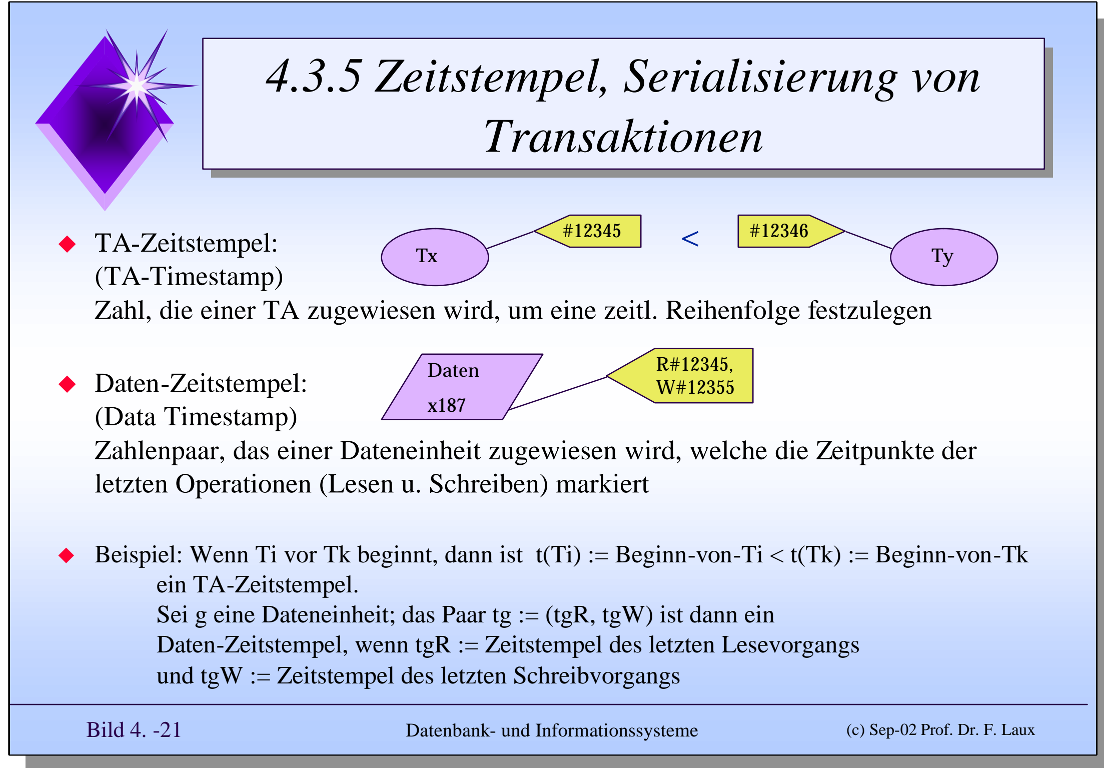


Wartegraph

Ein gerichteter Graph ist ein mathematisches Netzwerk, das aus Knoten und gerichteten Kanten besteht. Wenn eine TA als Knoten und die Reihenfolge der Sperrvergabe als Kanten (Pfeilrichtung bedeutet "wartet auf") interpretiert werden, erhalten wir einen Wartegraph (auch Sperrgraph genannt). Wenn ein Graph eine Schleife (Zyklus) enthält, befinden sich die an der Schleife beteiligten Transaktionen in einer Verklemmung.

Bei einer verteilten Datenbank ist dazu ein zentraler oder verteilter Graph erforderlich. Ein ***zentraler Wartegraph*** enthält alle Transaktionen des Gesamtsystems. Nachteilig dabei ist, daß ein hoher Kommunikationsaufwand über alle Knoten erforderlich ist und der Wartegraph sowohl einen Engpaß als auch eine Schwachstelle (single point of failure) darstellt.
Besser geeignet ist ein ***verteilter Wartegraph***. Jeder Knoten unterhält einen lokalen Graphen (*lokale Autonomie*), der auch externe Anforderungen lokaler Ressourcen enthält. Damit läßt sich nur ein lokaler Deadlock sicher feststellen. Verklemmungen, die mit externen Transaktionen zusammenhängen, können nur vermutet werden. Um solche Situationen zu klären, wird die lokale Information an andere Knoten weitergemeldet, bis mit den dort vorhandenen Graphen eine Klärung herbeigeführt werden kann. Dazu wird aus der neu eingetroffenen Information und der lokal verfügbaren ein resultierender Sperrgraph gebildet. Spätestens, wenn alle Knoten informiert wurden, ist die Gesamtinformation vorhanden, um die Situation zu klären.

Die Vorteile dieses Verfahrens sind eine erhöhte Verfügbarkeit, denn ein nicht verfügbarer Knoten kann ausgelassen werden, da er auch keine globalen Transaktionen durchführt. Gegenüber einem zentralen Wartegraphen ist er natürlich komplexer in seinen Funktionen.



Zeitstempel dienen dazu, Ereignissen oder Vorgängen (Transaktionen) Ordinalzahlen zuzuweisen, damit sie in eine zeitliche Reihenfolge gebracht werden können. Dies kann z.B. durch eine Uhrzeit oder einen Zähler geschehen. Bei zeitgleichen Ereignissen ist eine Rangfolge zu definieren, die sich in den Timestamps widerspiegelt. Seien z.B. die Ereignisse E1 und E2 zeitgleich eingetreten und E1 hat gegenüber E2 Vorrang, dann erhält E1 die kleinere Nummer.

Ein Datenbankobjekt ist durch zwei Timestamps gekennzeichnet, den Lesestempel (letzter Lesevorgang) und den Schreibstempel (letzter Schreibvorgang). Als Zeitstempel einer Transaktion wird ihr Beginn festgelegt.

Damit ist es möglich festzustellen, ob ein Objekt nach Transaktionsbeginn von einer anderen Transaktion gelesen oder geschrieben wurde.

Auf der anschließenden Folie wird ein Verfahren zur Synchronisation von Transaktionen vorgestellt, das Zeitstempel verwendet.

4.3.5 Zeitstempel-Verfahren

- Das folgende Zeitstempelverfahren stellt ein deadlock-freies Verfahren zur Serialisierung von TAs dar.
- Das Verfahren basiert auf einer zeitlichen Ordnung von TAs.
- Definitionen:
  - ts(Tx) := Zeitstempel der TA x
  - Wts(E) := letzter Schreibzugriff auf Datenelement E
  - Rts(E) := letzter Lesezugriff auf Datenelement E
- Verfahren (Algorithmus)
  - Ti will E lesen: a) ts(Ti) < Wts(E) => E geändert, Abort Ti
    b) ts(Ti) >= Wts(E) => Ti liest E,
    Rts(E) := max(Rts(E), ts(Ti))
  - Ti will E schreiben: a) ts(Ti) < Rts(E) => E gelesen, Abort Ti
    b) ts(Ti) < Wts(E) => E geändert, ignoriere 'write'
    c) sonst => Ti schreibt E,
    Wts(E) := max(Wts(E), ts(Ti))

Bild 4. -22 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Das Zeitstempel-Verfahren ist ein Synchronisationsverfahren für Datenbanken. Es basiert nicht auf Sperrmechanismen sondern auf einer zeitlichen Ordnung der Transaktionen und ist somit verklemmungsfrei.

Definitionen:

Jede Transaktion Tx, erhält einen Zeitstempel ts(Tx). Diese Zeitstempel werden aufsteigend vergeben (z.B. die Uhrzeit oder eine streng monoton wachsende Numerierung). Jedes Objekt (Datenelement) E (z.B. ein Datensatz) besitzt zwei verschiedene Zeitstempel, den Lese-Stempel Rts(E) und den Schreib-Stempel Wts(E).

Vorgehen:

Trifft eine Transaktion T ein, so gelten folgende Regeln:

Für eine Lese-Operation:

Wenn der Zeitstempel ts(T) kleiner ist als der Zeitstempel Wts(E), dann erfolgt der Abbruch von T, denn E wurde zwischenzeitlich von einer anderen TA verändert. Ansonsten wird die Lese-Operation ausgeführt und der Zeitstempel Rts(E) wird auf einen neuen Wert gesetzt. Dieser Wert ist immer der größere der beiden Zeitstempel ts(T) oder Rts(E),
d.h. Rts(E) := Max(Rts(E), ts(T)).

Für eine Schreib-Operation:

Wenn der Zeitstempel ts(T) kleiner ist, als der Zeitstempel Rts(E), dann hat eine andere TA das Objekt inzwischen gelesen. Die Transaktion T ist abzubrechen. Ist Wts(E) größer oder gleich ts(T), dann wurde E von einer andern TA bereits geändert. Unsere Transaktion T braucht nicht mehr zu schreiben, die Transaktion kann fortgesetzt werden. In allen andern Fällen kann T schreiben und den Schreibstempel aktualisieren.

4.3.5 Beispiel: Synchronisation mit Zeitstempel

t	Rts/Wts(E)	E	T1	T2
1	1 / 1	10		
2			ts(T1) := 2	
3				ts(T2) := 3
4	3 / 1			T2 liest E
5	3 / 1		T1 liest E	
6			T1 will E:=E-1 schreiben *Rollback T1**	
7	3 / 3	8		T2 schreibt E:=E-2

* T1 muß zurückgesetzt werden, da zum Zeitpunkt t = 6 gilt: ts(T1) = 2 < Rts(E) = 3.

Bild 4. -23 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Fortsetzung (Schreiboperation)

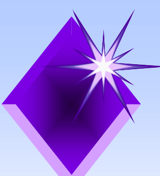
Aufgrund dieser Regeln ist gewährleistet, daß Transaktionen nach der Reihenfolge ihres Eintreffens bearbeitet werden. Falls dies nicht möglich ist, wird die Transaktion abgebrochen und mit einem neuen Zeitstempel neu gestartet. Eine Lese-Operation kann also nur ausgeführt werden, wenn ihr Zeitstempel größer (also jünger) ist als die letzte Schreib-Operation und eine Schreib-Operation kann nur ausgeführt werden, wenn nicht schon eine jüngere Operation (also mit höherem Zeitstempel) auf das entsprechende Element zugegriffen hat.

Beispiel:

Gegeben seien zwei Transaktionen T1 und T2. T1 bucht 1 Platz, T2 bucht 2 Plätze des gleichen Fluges. Dazu muß die Anzahl verfügbarer Sitze gelesen und anschließend die verbleibende Anzahl zurückgeschrieben werden. Sei E = 10 die Anzahl freier Plätze zu Zeitpunkt t = 1.

T1 kann sich nicht durchsetzen, da T2 das Datenelement E bereits gelesen hat. T1 muß zurückgesetzt werden. Hätte T1 bereits zum Zeitpunkt t = 3 gelesen, so wäre das Ergebnis allerdings das gleiche gewesen. Dieser Algorithmus bevorzugt offenbar später begonnene Transaktionen.

ÜA: Wie müßte der Algorithmus geändert werden, so daß die ältere Transaktion (also T1) sich durchsetzt?



4.3.6 Fehlerbehandlung

- *lokale Fehler*
  - Fehler, die nur einen Ort betreffen, können lokal behandelt werden (-> lokale Autonomie)
- *globale Fehler*
  - Fehler, die nur im Zusammenspiel mehrerer Orte behandelt werden können
  - dazu gehören
    - Verbindungsfehler
    - Netzwerkpartition
    - Fehler bei globalen und verteilten Transaktionen
- *Maßnahmen*
  - Verbindungsfehler: Rerouting bzw. Wiederholung der Nachricht nach Timeout (REDO)
  - Netzwerkpartitionen: lokales Weiterarbeiten (nur lokale bzw. globale TAs)
  - sonstige TA-Fehler: Abbrechen der Transaktion (wie bei einer zentralen Datenbank)

Bild 4. -24 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

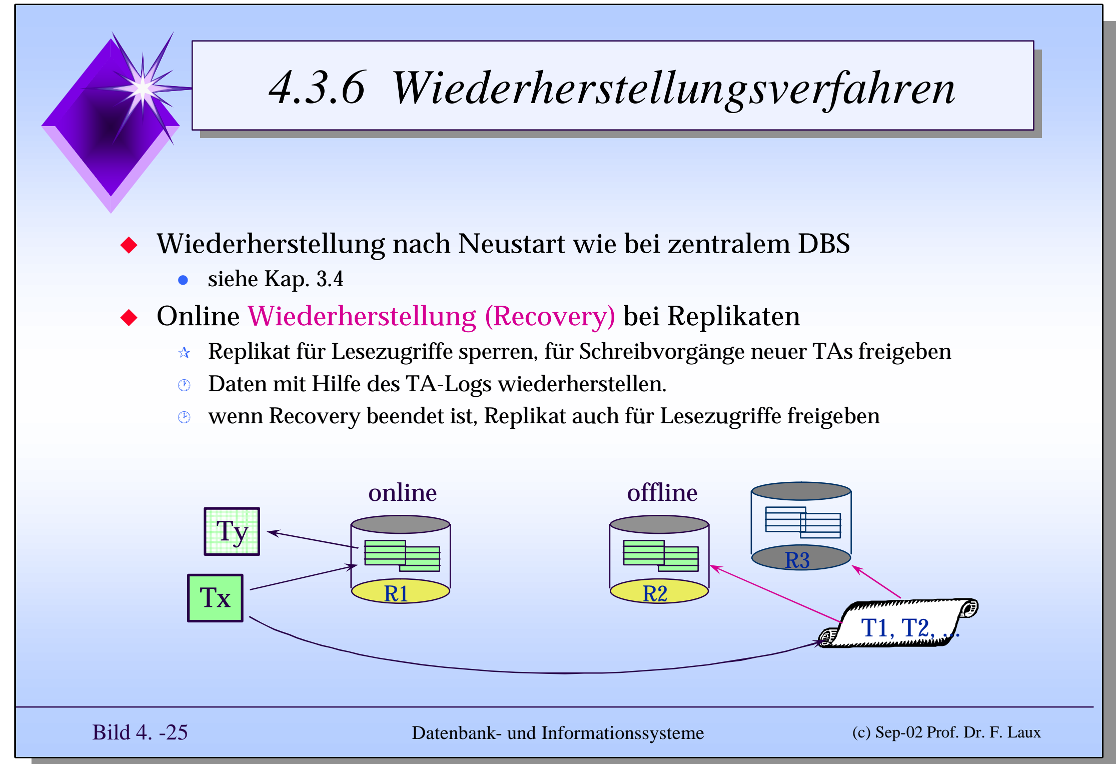
Entsprechend ihrer Auswirkung unterscheiden wir lokale und globale Fehler.

Lokale Fehler können wie bei einer zentralen DB behandelt werden. ***Globale Fehler*** hingegen erfordern die Mitwirkung weiterer Knoten und die Koordination durch das verteilte Transaktionsmanagement (siehe Kap. 4.3.1) und den Recovery Manager bei System- oder Speicherfehler.

Bei verteilten Datenbanken können gegenüber einer zentralen Datenbank weitere Fehler auftreten:

Verbindungsfehler: temporäre Fehler können durch eine Wiederholung der Nachricht behoben werden, bei permanenten Fehler ist ein alternativer Kommunikationsweg zu wählen.

Durch eine ***Netzwerkpartition*** ergeben sich zwei getrennte verteilte Datenbanken, die teilweise autonom weiterarbeiten können. Nach der Wiederherstellung des Netzwerks muß eine Recovery zum Datenabgleich gefahren werden.



Wir wissen bereits, dass bei verteilten Systemen u.U. trotz aufgetretenem Fehler mit Einschränkungen weitergearbeitet werden kann. Deshalb möchte man auch die Recovery während des laufenden Betriebs (online) durchführen können. Wenn wenigstens ein Replikat (R1) einen aktuellen Zustand besitzt, können die veraltete Replikate (R2, R3, ...) im laufenden Betrieb aktualisiert werden. Nachdem das Gesamtsystem wieder funktionsfähig ist, sind folgende Wiederherstellungsschritte erforderlich:

• Veraltete Replikate für Lesezugriffe sperren (falls nicht bereits zum Fehlerzeitpunkt geschehen) und evtl. in konsistenten Zustand bringen (notfalls Backup einspielen)

• R2, R3, ... für Schreibvorgänge laufender Transaktionen freigeben

• Transaktionen seit dem Fehlerzeitpunkt auf R2, R3, ... wiederherstellen (Roll Forward).

• Nach der Recovery alle Replikate wieder freigeben

Während der ganzen Zeit kann auf Replikat R1 normal gearbeitet werden.

4.4 verteilte Abfragen

- In zentralen Systemen ist der entscheidende Faktor für die Performance (Antwortzeit) einer Abfrage (Query) die Anzahl der Plattenzugriffe. Bei DDBS kommt der Zeitbedarf für die Kommunikation hinzu. Andere Zeitaspekte (z.B. Rechenzeit) sind meist vernachlässigbar.
- Effizienz einer Query
  - 2 Punkte sind für die Effizienz einer Query zu beachten:
    - der Leistungs- und Ressourcenbedarf
    - die Antwortzeit
- Für verteilte Systeme sind daher Systembelastung und Antwortzeit zu minimieren:
  - SysLoad (d, B, t) = MIN (d := #Disc-I/O, B := Datenmenge, t = Rechenzeit)
  - RespTime (ds, B, ts) = MIN (ds := sequent. Disc-I/O, ts := sequent. Rechenzeit)
- ds, ts ~ 1/Parallelität

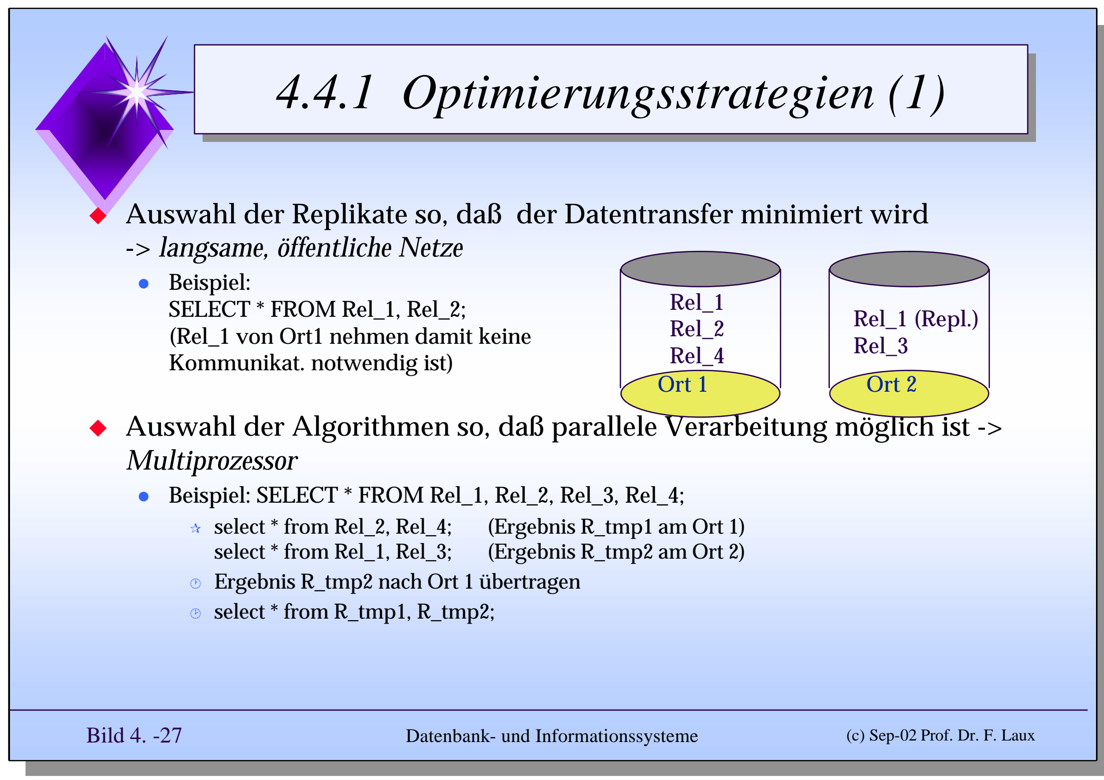
Bild 4. -26 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Wir betrachten nur Queries in verteilten relationalen Datenbanken. Weil die einzelnen Daten, die zur Abarbeitung der Query benötigt werden, an unterschiedlichen Speicherorten liegen können, muß die Abfrage in verschiedene Teile zerlegt, abgearbeitet und schließlich die Ergebnisse zusammengefügt werden. Deshalb kommen zu den üblichen Zeitfaktoren (I/O-Zeiten, Rechenzeiten) einer Query die Übertragungszeiten für die Daten hinzu.

Neben der Antwortzeit (Zeitspanne zwischen Anfrage und Bereitstellung der Ergebnisse) ist für die Effizienz einer Query auch der Ressourcenbedarf von Bedeutung. Beide Zielgrößen stehen in Konkurrenz zueinander. Eine Query kann schneller durchgeführt werden, wenn mehr Ressourcen eingesetzt werden.

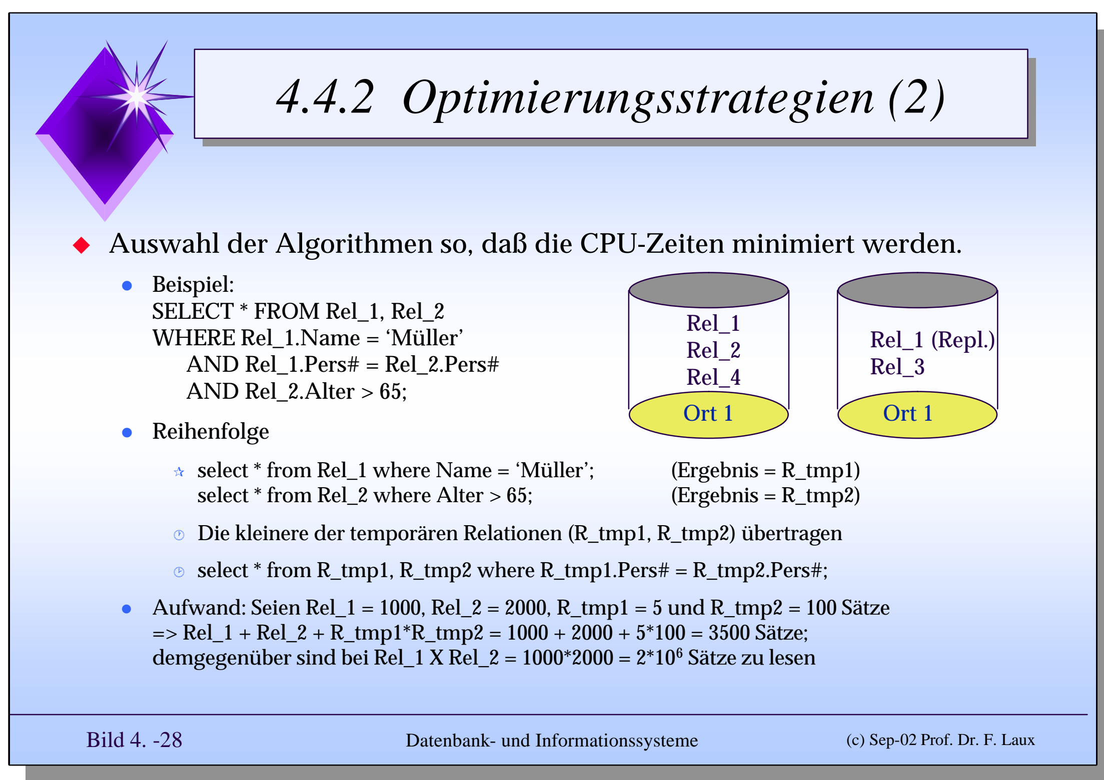
Bei verteilten Abfragen müssen also Antwortzeit und Ressoucenbedarf gemeinsam optimiert werden. Die Antwortzeit kann durch eine Aufteilung der Aufgaben (Parallelisierung) verkürzt werden. Dabei gilt: Die Antwortzeit ist ungefähr umgekehrt proportional zum Grad der Parallelisierung.

Um möglichst kurze Antwortzeiten zu erhalten, gibt es eine Reihe von Optimierungsstrategien, die auf den folgenden Seiten vorgestellt werden. Da die Maßnahmen nicht unabhängig voneinander sind, wird zu Erklärung jeweils nur ein Parameter optimiert. In Wirklichkeit sind natürlich alle Parameter gleichzeitig zu optimieren. Diese Aufgabe übernimmt der Query-Optimierer. Er überprüft, welche Daten zur Bearbeitung der globalen Query benötigt werden und wo diese liegen. Dann zerlegt er die Query in Fragmente, so daß die einzelnen Teile von den jeweiligen Rechnern bearbeitet werden können. Bei der Bearbeitung muß der Query-Prozessor aber darauf achten, daß die einzelnen Subqueries so verteilt werden, daß nach Abarbeitung aller Teile das Ergebnis vollständig und richtig ist und durch die Fragmentierung nicht verfälscht wurde. Beispielsweise kann eine falsche Reihenfolge der Bearbeitung einzelner Teilaufgaben zu ungültigen Ergebnissen führen.



Die erste der hier gezeigten Strategien ist für langsame Netzwerke oder teuren Datentransport geeignet. Die Aufteilung der Query erfolgt so, daß der Kommunikationsbedarf minimiert wird.

Bei der zweiten Strategie wird auf maximale Parallelität geachtet. Deshalb ist sie bei Multiprozessorsystemen vorteilhaft.



Die letzte der vorgestellten Optimierung versucht die CPU-Zeiten und damit die Rechner-belastung zu minimieren.

Es werden zuerst die Tabellen einzeln durchsucht und dann die Ergebnisse zusammen-getragen. Weil nur eine Teilmenge einer Tabelle übertragen werden muß, wird auch die übertragene Datenmenge geringer. Generell gilt für die Durchführung einer Abfrage, daß zuerst das selektivste Kriterium abgearbeitet wird, da dieses die geringste Anzahl von Sätzen liefert.

Die bisherigen Beispiele haben gezeigt, daß nach verschieden Kriterien optimiert werden kann und sich die Optimierungskriterien gegenseitig beeinflussen. Im allgemeinen ist es zu zeit- und ressourcenaufwendig, wenn vor jeder Query jedesmal alle Möglichkeiten überprüft werden müßten, bevor die Bearbeitung der Query gestartet wird. Denn je komplexer eine Query ist und je mehr Bedingungen beachtet werden müssen, desto schwieriger und vielfältiger sind die Lösungswege.

Deshalb werden häufig zwei Strategien angewandt:

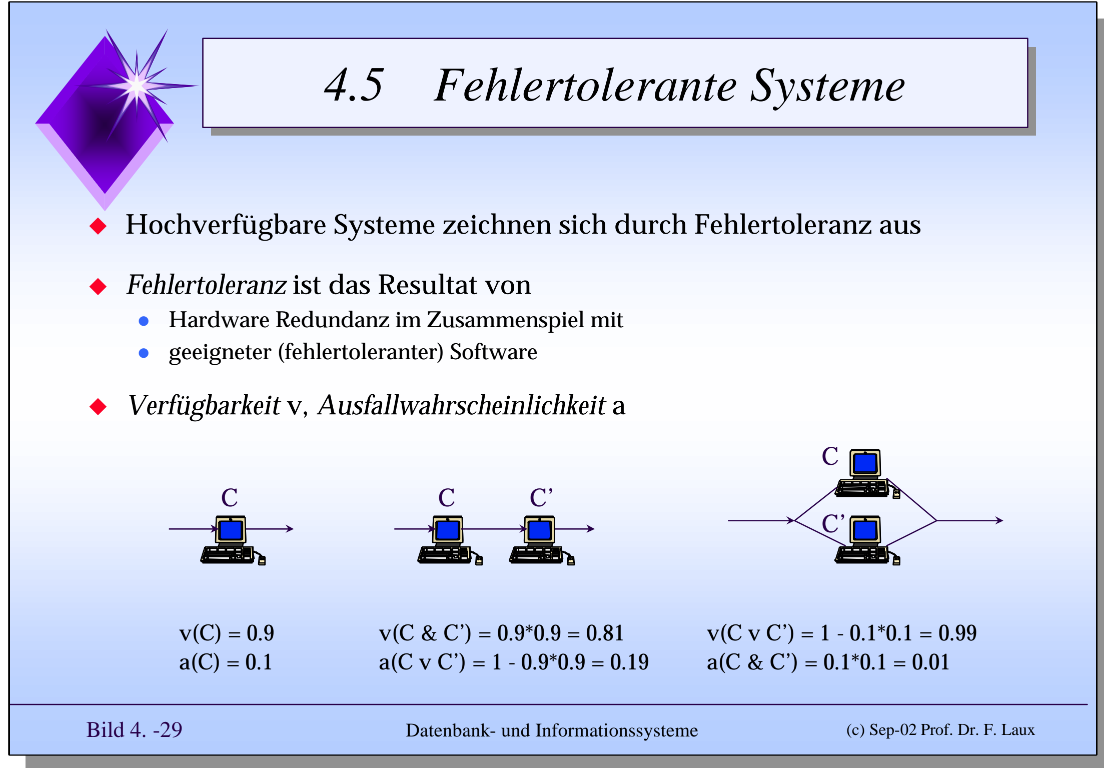
Regelbasierter Optimierer:

Bei diesem existieren heuristische Regeln (Erfahrungen), wie Queries abgearbeitet werden sollen. So werden z.B. Regeln festgelegt, welche Bedingungen zuerst beachtet werden sollen.

Statistikbasierter Optimierer:

Dieser Optimierer speichert die Zeiten von einzelnen Zugriffen und kann dann bei gleichen oder ähnlichen Queries aus der Statistik entnehmen, welche Zugriffswege und welche Dekomposition der Query vermutlich am günstigsten sind.

ÜA: Überlegen Sie sich, wie ein Optimierer das selektivste Kriterium einer Abfrage herausfinden kann.



Informationssysteme erfordern bei unternehmenskritischen Anwendungen eine ständige Verfügbarkeit. Wartung und Reparatur müssen "online" erfolgen. Beispiele hierfür sind internationale Reservierungssysteme, Blutbanken und Rettungsinformationssysteme.

Hochverfügbare Systeme müssen fehlertolerant sein. Dies wird durch redundante Hardware erreicht. Wenn wir zwei gleiche Komponenten parallel schalten, so ist das System auch mit nur einer Komponente noch funktionsfähig. Da nur eine von beiden Komponenten verfügbar sein muss, ist die Ausfallwahrscheinlichkeit eine Größenordnung kleiner als bei einer Komponente.

Durch die zunehmende Komponentenzahl bei redundanten Systemen ist allerdings die Wahrscheinlichkeit eines Einzelfehlers höher als bei einem einfachen System. Aus diesem Grund ist es wichtig, dass solche Systeme auch im laufenden Betrieb (online) gewartet und repariert werden können. Durch geeignete Softwareunterstützung ist es möglich, einzelne Komponenten zu deaktivieren, zu testen und ggf. zu ersetzen. Die Wiederinbetriebnahme redundanter Komponenten (z.B. gespiegelte Platten) muss ebenfalls durch entsprechende Software unterstützt werden (z.B. Datenreplikat erstellen).

4.5.1 Fehlertoleranz (Begriffe)

- *Fehlertoleranz* :=
  Ein System, das trotz Fehler weiterarbeiten kann
- *Hot Stand-By* :=
  2 aktive Systeme arbeiten an der gleichen Aufgabe
- *Cold Stand-By* :=
  Ein System ist aktiv, das andere überwacht und übernimmt im Fehlerfall
- *Fehlererkennung* mittels Mehrheitsprinzip :=
  n >= 3 Systeme bearbeiten die gl. Aufgabe; das *Mehrheitsergebnis* wird als richtig erachtet.
- *Fehlererkennung* durch Funktionsprüfung
  ein aktives System wird ständig durch ein *Testsystem* überprüft; wenn ein Fehler erkannt wird, wird die Aufgabe an ein anderes System übergeben.

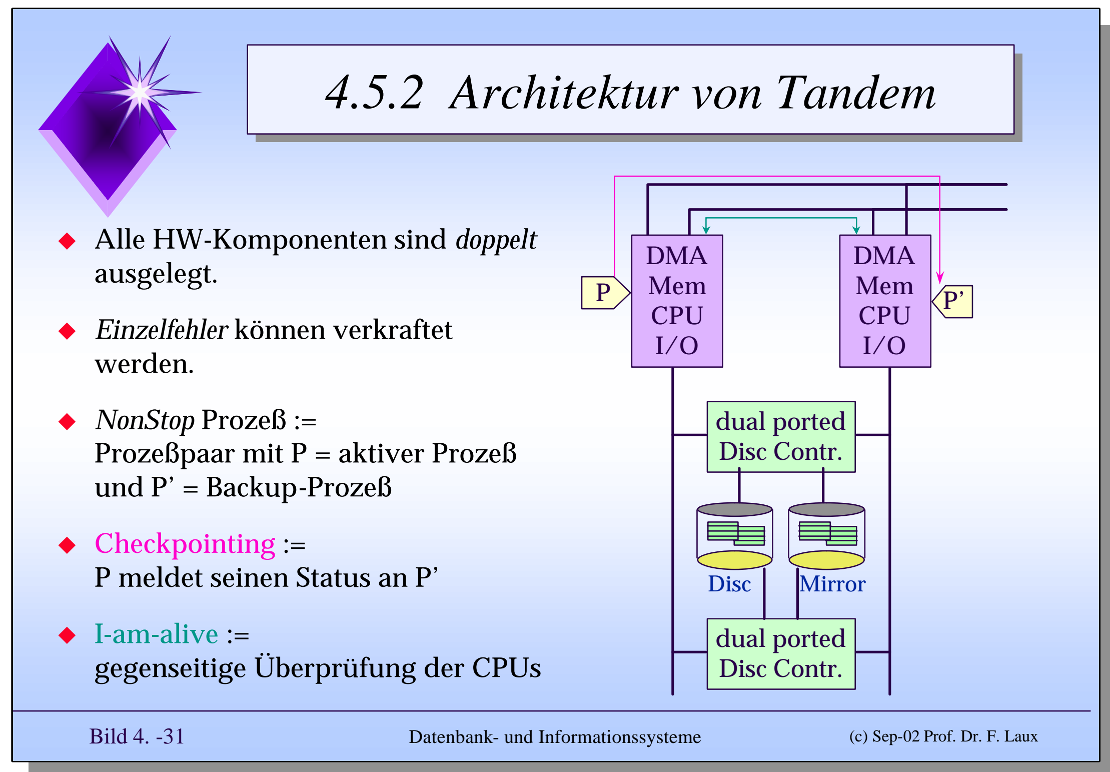
Bild 4. -30 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Softwareseitig werden fehlertolerante Systeme durch verschiedene Konzepte unterstützt:

Entweder arbeiten mindestens 2 Systeme an der gleichen Aufgabe parallel (*hot stand-by*) oder ein System wartet auf seinen Einsatz im Fehlerfall (*cold stand-by*).

Bei einem hot stand-by kann verzögerungsfrei nach einem Fehler weitergearbeitet werden, da das Reservesystem bereits aktiv ist und den aktuellen Status besitzt. Die erforderliche Gesamtkapazität ist doppelt so hoch, wie ohne Fehlertoleranz.

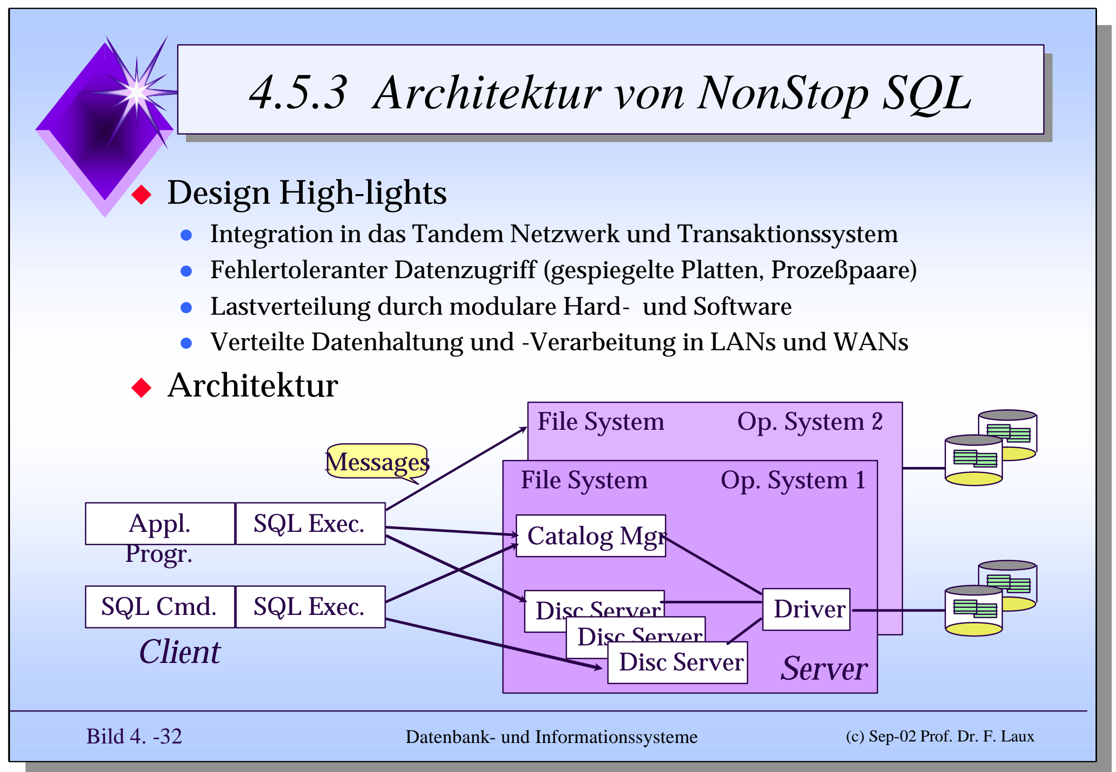
Im Falle eines cold stand-by ist das Reservesystem nicht aktiv, solange kein Fehler aufgetreten ist. Es kann in dieser Zeit für andere Aufgaben verwendet werden. Allerdings muß es von Zeit zu Zeit über den aktuellen Zustand des aktiven Systems und seiner Programme informiert werden. Dies kann durch eine Protokolldatei, durch Checkpointing (Statusmeldungen an das Reservesystem) oder manuell geschehen.



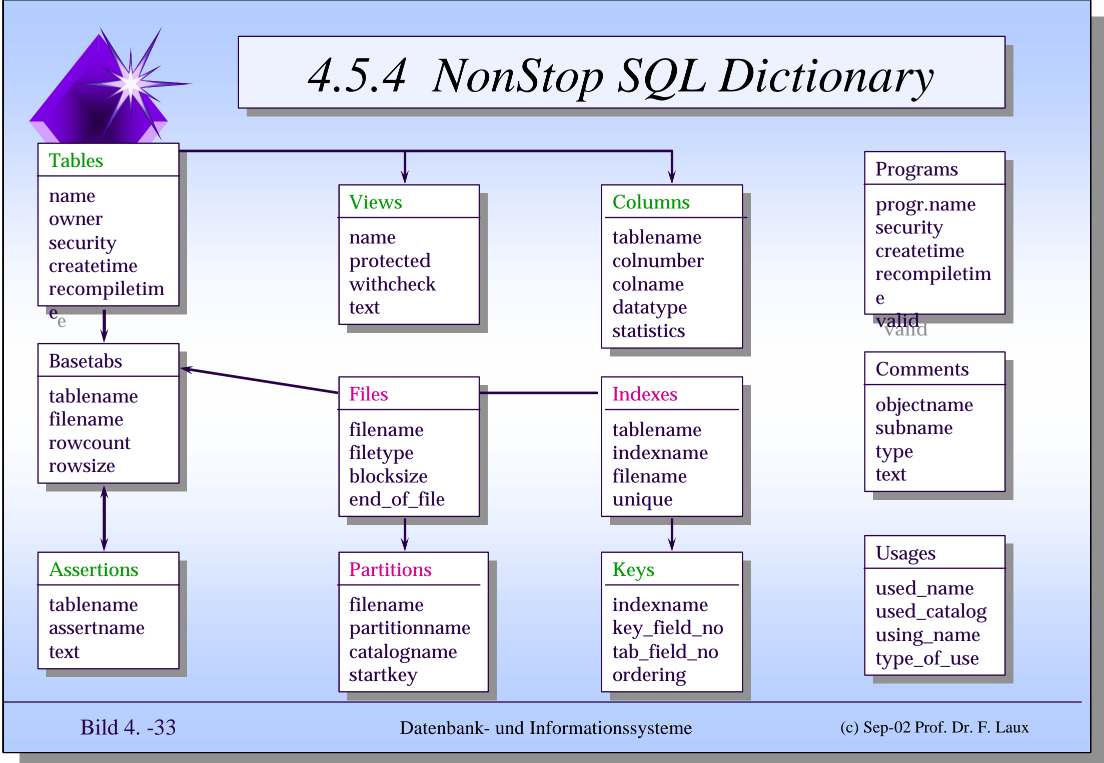
Computersysteme der Fa. Tandem (jetzt Fa. Compaq) haben eine redundante Hardware-Architektur, so daß beliebige Einzelfehler toleriert werden können. Softwareseitig wird mit Prozeßpaaren und mit Checkpointing gearbeitet, das eine Mischung von beiden Stand-by Konzepten darstellt.

Jede Komponente ist doppelt vorhanden. Dadurch können prinzipiell Einzelfehler verkraftet werden. Allerdings ist eine Erkennung der Fehlersituation erforderlich. Dies wird durch zyklische Selbsttest der Hardware und Kontrollnachrichten ("I-am-alive" Messages) erreicht. Fehlertolerante Aufgaben werden immer von einem Prozeßpaar (ein Nonstop Prozeß besteht aus Primär- und Backup-Prozeß) durchgeführt, wobei der Primärprozess aktiv ist und seinen Status an den Reserveprozeß (Backup) meldet, der ansonsten inaktiv ist. Kritische Operationen werden ähnlich einer Transaktion abgesichert.

In einem Fehlerfall übernimmt der Backup Prozeß in einem anderen Prozessor die Aufgaben. Im Normalbetrieb benachrichtigt der Primärprozeß seinen Backup mit Statusinformationen z.B. über nicht wiederholbaren Operationen (z.B. Benutzereingabe, Schreiboperation auf einer Datei). Dadurch ist sichergestellt, daß im Fehlerfall der Backup- Prozeß ohne Datenverlust oder -verfälschung weiterarbeiten kann.



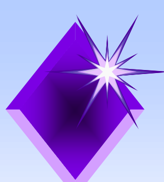
Das Datenbanksystem ist nach dem zuvor beschriebenen Konzept als Client-Server Architektur realisiert. Alle funktionswichtigen Datenbankprozesse (Disc Manager, Catalog Manager, Disc Driver) sind als Nonstop Prozesse implementiert. Das Betriebssystem ist nachrichtenbasierend (message passing), so daß Prozesse über Nachrichten kommunizieren. Dabei ist es unwesentlich ob sich die Prozesse in der gleichen CPU oder in verschiedenen CPUs befinden. Zur Lastverteilung ist es möglich, weitere Prozesse eines Programms in anderen CPUs zu starten und damit Aufgaben zu parallelisieren. Das System arbeitet transparent in öffentlichen und lokalen Netzwerken (WAN (Wide Area Network), LAN (Local Area Network)).



Das Data Dictionary ist verhältnismäßig einfach aufgebaut. Allerdings sind nicht alle Attribute in der Folie wiedergegeben.

Zu jeder globalen Tabelle (*Tables*) kann es viele *Views* und Spalten (*Columns*) geben. Eine Tabelle besteht aus mehreren Basistabellen (Fragmente), die auf genau eine Datei abgebildet werden. Eine Datei hingegen kann mehrere Fragmente aufnehmen. Die geographische Verteilung der Daten wird durch erreicht, dass eine Datei über mehrere Partitionen (logische Platten) verteilt werden kann. Für jede Datei können mehrere Indizes (*Indexes*) angelegt werden, deren Zugriffsschlüssel in der Tabelle *Keys* definiert werden.

Im Dictionary sind auch die Datenbankprogramme (Programs), Art der Verwendung (*Usages*) und Kommentare (*Comments*) abgelegt. Damit ist es möglich, festzustellen, welche Programme von einer Datenbankänderung betroffen sind. Aus diesen Einträgen kann auch der Laufzeitcompiler (dynamische Recompilation) feststellen, ob ein Programm recompiliert werden muß (vgl. Bild 4-35)



4.5.5 Leistungsmerkmale

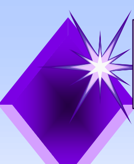
- Transaktionsmanagement
  - Zwei-Phasen-Commitment (Begin, Commit, Rollback, Free-Resource)
  - Synchronisation paralleler TA durch Locking auf Datei- und Satzebene im READ- und WRITE-Modus
  - Wartezeit für Locks wird durch User definiert (0 ... ∞)
  - Auflösung von Verklemmungen durch Timeouts
  - Transaktionslog (Transaktion Monitoring Facility) zur Wiederherstellung
  - Online Recovery
- Lokale Autonomie
  - freie lokale Namenswahl (Konvention: \site.$disc.dir.table)
  - Ortstransparenz durch Alias-Namen
  - dynamische Zugriffstrategie, falls Speicherort ausfällt
  - bei fragment. Tabellen kann nach ‘disconnect’ lokal weitergearbeitet werden

Bild 4. -34 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Das Transaktionsmanagement arbeitet mit einem Zwei-Phasen-Commitment Protokoll. Die Freigabe von Ressourcen kann manuell erfolgen, um den Speicher effizienter zu nutzen. Die Wartezeit auf Sperranforderungen kann durch ein Timeout begrenzt werden. Damit können auch Deadlocks aufgelöst werden. Ein echte Deadlockerkennung ist nicht vorhanden.

Eine Wiederherstellung der Datenbank mit Hilfe des Transaktionslogs und durch Betriebssystem Utilities (z.B. Revive (Aktualisierung und Freischalten von Replikaten)) ist im laufenden Betrieb möglich.

Tabellennamen können lokal vergeben werden, ohne daß dadurch globale Konflikte entstehen. Die globalen Namen spiegeln die physische Lokation wieder indem der Speicherort vor den lokalen Namen gesetzt wird. Dadurch sind die globalen Namen ortsabhängig. Durch die Zuweisung von Alias-Namen erreicht man eine quasi Ortstransparenz. Sind fragmentierte Tabellen von Speicherfehlern betroffen, müssen die betroffenen Fragmente durch einen "disconnect"-Befehl abgehängt werden, damit weitergearbeitet werden kann.

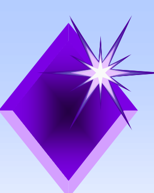


4.5.5 statisches u. dynamisches Binden

- Statisches Binden von Anwendung mit der SQL-Datenbank
  - höchste Ausführungsgeschwindigkeit
- Dynamische Recompilation und erneutes Binden bei DB-Änderung und in Fehlersituationen
  - SQL-Source wird im Programmfile mitgeführt -> zur Erkennung von Änderungen in der Datenbank
  - automatische Recompilation beim Programmstart, falls DB geändert wurde
  - erneutes Binden beim DB-Zugriff, falls vorgesehener Speicherort nicht verfügbar ist (Änderung der Zugriffstrategie)

Bild 4. -35 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Tandem kombiniert statisches Binden (zur Übersetzungszeit) mit quasi-dynamischem Binden (Recompilation zur Startzeit und erneutes Binden) falls sich die Datenbank bzw. ihre Zugriffspfade seit der letzten Übersetzung geändert haben. Dadurch erzielen die Programme die hohe Ausführungsgeschwindigkeit von statisch gebundenen Programmen und bieten gleichzeitig die Flexibilität von interpretierten Programmen. Ist eine Recompilation erforderlich, erhöht sich natürlich die Zeit für den Programmstart um die Compilations- und Bindezeit.



Aufgaben

- Ein Textilunternehmen mit Produktion und Lager in Albstadt unterhält Verkaufsstellen (sogenannte Showrooms) in München, Düsseldorf, Paris und Mailand. Den Verkaufsstellen sind disjunkte Verkaufsgebiete zugeordnet (z.B. Mailand - Italien, München = Süddeutschland, etc). Der Verkauf erfolgt ausschließlich durch die Showrooms. Wobei diese die Artikel aus dem Zentrallager in Albstadt anfordern. Die Auslieferung an den Kunden erfolgt durch die Verkaufsstellen
- Entwerfen sie eine geeignete verteilte Datenbank

Bild 4. -36 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux

Für die Produktion werden folgende Tabellen benötigt:

Artikelstamm (ArtNr (ID), Bezeichnung, ArtikelTyp, Preis, Lieferant)
ArtikelTyp = {Produkt, Rohmaterial}

Stückliste (ProdNr (ID) (FS), TeileNr (ID) (FS), Menge),

ProdNr, TeileNr = Fremdschlüssel zu Artikelstamm

Für das Lager werden folgende Tabellen benötigt:

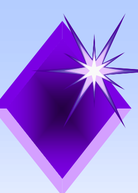
Lager (Lagerort (ID), ArtNr, Menge)

Für die Verkaufsstellen werden folgende Tabellen benötigt:

Kunde (KdNr (ID), Name, Adresse)

AuftrPos (AuftrNr (ID)(FS), Pos (ID), ArtNr (FS), Menge)

Fragmentieren Sie die globalen Tabellen und verteilen Sie diese auf die Standorte des Unternehmens.



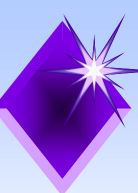
Aufgaben (Fortsetzung)

- Für die Produktion werden folgende Tabellen benötigt:
- Artikelstamm (ArtNr (ID), Bezeichnung, ArtikelTyp, Preis, Lieferant) ArtikelTyp = {Produkt, Rohmaterial}
- Stückliste (ProdNr (ID) (FS), TeileNr (ID) (FS), Menge),
- ProdNr, TeileNr = Fremdschlüssel zu Artikelstamm

Aufgaben (Fortsetzung)

- Für das Lager werden folgende Tabellen benötigt:
- Lager (Lagerort (ID), ArtNr, Menge)
- Für die Verkaufsstellen werden folgende Tabellen benötigt:
- Kunde (KdNr (ID), Name, Adresse)

Bild 4. -38 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux



Aufgaben (Fortsetzung)

- Auftrag (AuftrNr (ID), KdNr (FS), Datum, GesPreis, Status)
- AuftrPos (AuftrNR (ID)(FS), Pos (ID), ArtNr (FS), Menge)

- Fragmentieren Sie die globalen Tabellen und verteilen Sie diese auf die Standorte des Unternehmens.

Bild 4. -39 Datenbank- und Informationssysteme (c) Sep-02 Prof. Dr. F. Laux