

Verteilte Datenbanken (VDB)

1. Einleitung (® [DBIS])

- Datenbanken und Informationssysteme → [DBIS]
- Rechnernetze
 - Rechnernetze := in Netzwerk miteinander verbundene autonome Rechner, mit Möglichkeit des Mitteilungsversenden an andere Knoten
 - Arten: LAN (Local Area Network), MAN (Metropolitan), WAN (Wide), PIN (Infra-Red)
 - Netztopologie: Aufbau, Routing, Protokoll
 - Typen:
 - Hostbasiert („Dummy Terminals“)
 - Client/Server (Shared Device, File Server vs. Database Server)
- verteilte Datenbanken
 - logisch miteinander verbundene Datenbanken mit Daten auf verschiedenen Knoten (DBS, DBMS, IS, ... analog zu [DBIS])
 - Ursachen:
 - historisch/strukturell
 - Zusammenschlüsse von Firmen
Zusammenfassung der Daten meist nicht möglich, wegen: unterschiedlichen Systemen, Umfang der Daten, gleiche Dinge haben unterschiedliche Namen, unterschiedliche Dinge haben gleiche Namen, Konsistenz der Daten nicht gegeben, ...)
 - Filialnetz
 - Speicher-/Rechenkapazität
 - Varianten
 - verteiltes Multi-DBS (keine Kooperation)
 - verteiltes föderiertes DBS (kooperierende, teilweise autonome Systeme)
 - verteiltes DBS im engeren Sinne (enge Kopplung, meist: ein Anbieter)
 - Vor- und Nachteile der Verteilung
 - Lokale Autonomie (Zugriff auf lokale Daten, auch bei Netzausfall)
 - Performance (parallele Bearbeitung an verschiedenen Knoten, Replikate ⇒ Leseanfragen gleichzeitig, allerdings keine Updates)
 - Verlässlichkeit, Verfügbarkeit (durch Replikation)
 - Wirtschaftlichkeit (Daten dort, wo am häufigsten benötigt)
 - Organisationsstruktur der Firma kann gut abgebildet werden
 - fehlende Erfahrung (Forschung)
 - Umstellungsschwierigkeiten (keine Werkzeuge oder Methoden)
 - Kosten (Personal-, Hard-, Software- und Kommunikationskosten i.A. unbekannt)

2. relationales Datenbanksystem

siehe → [DBIS]

3. Entwurf verteilter Datenbanken

- Top-Down (globales konzeptuelles Schema, anschl. Lokale Schemata, sinnvoll bei Neuentwurf)
- Bottom-Up (lokale Schemata vorhanden \Rightarrow globales Schema, sinnvoll wenn bereits lokale DBS bestehen)
- Datenfragmentierung
 - Regeln:
 - Vollständigkeit
 - Rekonstruierbarkeit
 - Disjunktheit (keine überlappenden Datenanteile, nur PS überlappend bei vertikaler Fragmentierung)
 - Probleme:
 - Aufwand (teilweise mehr JOINS und Vereinigungen für Operationen)
 - Integritätsüberwachung erschwert
 - SQL Modellierung Fragmente: `DEFINE FRAGMENT FragmentName AS (SELECT-Statement)`
 - horizontale Fragmentierung
Aufteilung zeilenweise durch Select-Operation (z.B. KundenKarlsruhe)
es gilt: $r = r1 \cup r2 \cup \dots \cup rN$
 - abgeleitete horizontale Fragmentierung
Aufteilung zeilenweise durch Selektion auf Attribute einer anderen Relation (z.B. TeileKarlsruhe als Selektion von Teile mit Ort des Lieferanten = Karlsruhe)
 - vertikale Fragmentierung
Projektion von einzelnen Attributen, Schlüssel jeweils aufnehmen
es gilt: $r = r1 * r2 * \dots * rN$
 - hybride / gemischte Fragmentierung
Kombination horizontaler und vertikaler Fragmentierungen
- Datenallokation
 - Verteilung der Daten auf die Fragmente
 - replizierte \leftrightarrow nicht replizierte \leftrightarrow teilweise replizierte Speicherung
Vor- und Nachteile der Replikation:
Erhöhung der Systemverfügbarkeit
paralleles Lesen
Probleme bzw. Aufwand beim Update
 - Kriterien: Speicher-, Kommunikationskosten und Verlässlichkeit
- Transparenz
 - Fragmentierungstransparenz (Anfrage an *Kunde* nicht an *KundeKarlsruhe* oder *Kunde1*)
 - Ortstransparenz (Anfrage ohne Kenntnis des physischen Ortes der Daten)
z.B.: `ORACLE SQL*NET: CREATE SYNONYM relation1 FOR relation@knoten; (Admin)`
 - Replikationstransparenz (Anfrage ohne Kenntnis des physischen Ortes der Daten)
 - Anwendungsprogramme müssen bei Änderung der Datenbank (Fragmentierung, Replikationen, ...) bei vorliegender Transparenz nicht geändert werden!
- globales Data Dictionary
 - wie bei herkömmlichen Datenbanksystemen (\rightarrow [DBIS]) jedoch zusätzlich:
 - Fragmentierung
 - Datenallokation
 - Verteilung des Data Dictionary:
 - auf einem Knoten (Ausfall Knoten?)
 - auf allen Knoten (dort nur Informationen über Daten an diesem Knoten, Probleme bei Anfrage an alle Knoten)
 - vollständige Kopien an allen Knoten (Speicherplatz?, Änderungen?)
 - Kombination ist Regelfall: zentrale Lösung mit dezentralen Backups für Ausfall

- Berechnungsmodell
 Ann.: Fragmentierung vorgegeben, keine redundanten Daten
 LP: $\min Z = QST + QTP + UST + UTP + SK$ (Gesamtkosten)
 unter: $\sum d_{pk} = 1, \sum t_{mk} = 1$ (nicht redundante Speicherung)
 $\sum d_{pk} G_p \leq K_k$ (Speicherkapazität)
 mit: $i = 1, \dots, N$ (Knoten)
 $p = 1, \dots, P$ (Daten-Fragmente)
 $m = 1, \dots, M$ (Transaktions-Monitore TM)
 $QST = \sum q_{imp} t_{mk} c_{ik}^q$ (Anfragekosten Start \rightarrow TMs)
 $QTP = \sum q_{imp} t_{mk} d_{pj} c_{ik}^q$ (Anfragekosten TMs \rightarrow Fragmente)
 $UST = \sum u_{imp} t_{mk} c_{ik}^u$ (Updatekosten Start \rightarrow TMs)
 $UTP = \sum u_{imp} t_{mk} d_{pj} c_{ik}^u$ (Updatekosten TMs \rightarrow Fragmente)
 $SK = \sum d_{pk} G_p S_k$ (Speicherkosten für Fragmente)
 q_{imp} (geschätzte # Anfragen in Knoten i initiiert über TM m an Fragment p)
 u_{imp} (geschätzte # Updates in Knoten i initiiert über TM m an Fragment p)
 c_{ij}^q (durchschnittliche Kommunikationskosten pro Anfrageeinheit von Knoten i nach j)
 c_{ij}^u (durchschnittliche Kommunikationskosten pro Updateeinheit von Knoten i nach j)
 G_p (Größe Fragment p)
 K_j (Speicherkapazität an Knoten j)
 S_j (Speicherkosten pro EH am Knoten j)
 d_{pk} (=1 wenn Fragment p an Knoten k gespeichert, =0 sonst)
 t_{mk} (=1 wenn TM m ist am Knoten k implementiert, =0 sonst)

4. Transaktionskonzept

- Probleme
 - verlorengegangene Änderung (lost Update)
 - inkonsistente Sicht
 - Phantom Problem
 - inkonsistente Datenbank
- Menge von Aktionen, die zusammengehören sind nur in ihrer Gesamtheit sinnvoll \Rightarrow Transaktion
- Transaktion muß ablaufen, als wäre sie die einzige im System
- ACID-Prinzip
 - Atomicity (Unteilbarkeit): Transaktion ganz oder gar nicht ausführen
 \Rightarrow BEGIN OF TRANSACTION (BOT) und END OF TRANSACTION (EOT)
 \Rightarrow Recovery sorgt für zurücksetzen auf Zustand vor BOT, falls Fehler zwischen BOT und EOT
 \Rightarrow kein Zugriff auf Daten, bis EOT erreicht
 - Consistency (Konsistenz): nach Transaktion herrscht korrekter Zustand (nur zwischendurch sind inkonsistente Zustände möglich)
 - Isolation: Transaktion läuft so ab, als wäre sie die einzige
 - Durability (Dauerhaftigkeit): Transaktion erfolgreich \Rightarrow Transaktion überdauert nachfolgende Fehlerfälle
- Serialisierbarkeit
 - Def.: System von parallel ablaufenden Transaktionen ist korrekt synchronisiert, wenn es serialisierbar ist, d.h. wenn es mindestens eine (gedachte) serielle Ausführung derselben Transaktion gibt, die denselben Datenbankzustand und dieselben Ausgabendaten der Transaktionen liefert.
 - Objekt x: Datenbankeinheit; Aktion $a \in$ Transaktion T_i : jeder Zugriff auf ein Objekt, Aktionen sind atomar: Lesen $r_i(x)$, Schreiben $w_i(x)$
 - $<$ -Relation: $a_{im} < a_{in} \Leftrightarrow a_{im}$ steht vor a_{in}
 - Schedule: $T^* = T_1, \dots, T_n$
 - Konflikt: zwei Transaktionen stehen im Konflikt zueinander, wenn sie auf dasselbe Objekt zugreifen und mindestens eine der Operationen ein write ist.
 - abhängig: Transaktionen sind abhängig $\Leftrightarrow a_{im} \in T_i$ und $a_{jn} \in T_j$ mit $a_{im} < a_{jn}$ und a_{im}, a_{jn} stehen im Konflikt zueinander
 - S nicht serialisierbar $\Leftrightarrow T_i \in S$ abhängig von $T_j \in S$ und T_j abhängig von T_i
 - G: Abhängigkeitsgraph eines Schedules S: mit G zyklenfrei $\Rightarrow S$ serialisierbar
- **Zweiphasen-Commit-Protokoll**
 - Phase 1:

- Koordinator schickt prepare-to-COMMIT (PtC) an Agenten
 - Koordinator wartet auf ready-to-COMMIT (RtC) oder ABORT der Agenten (Möglichkeit der Durchführung von Teiltransaktionen)
 - Phase 2:
 - Falls *alle* Agenten RtC gesendet haben schickt Koordinator COMMIT sonst ABORT
 - Agenten bestätigen Erhalt der Mitteilung und führen Operation durch
 - Koordinator selbst führt COMMIT oder ABORT durch
 - Fehler Situationen bei Subtransaktion:
 - Ausfall • : Sub-Transaktion kann nicht initiiert werden
 - Ausfall , : Primär-Transaktion erhält keine RtC bzw. ABORT Meldung
 - Ausfall \mathcal{F} : Primär-Transaktion hat RtC oder ABORT empfangen kann aber globales COMMIT/ABORT nicht zustellen
 - Fehler Situationen bei Primärtransaktion:
 - Ausfall • : Primär-Transaktion kann globales COMMIT/ABORT nicht entscheiden
 - Ausfall , : Ausfall Primär-Transaktion nach globalem COMMIT/ABORT
 - Variante: Verteiltes Zwei-Phasen-Commit-Protokoll
Änderung: Jeder Agent schickt RtC/ABORT an alle anderen Agenten und Koordinator
 - Variante: Zwei-Phasen-Commit-Protokoll mit Timeouts
Mitteilung länger als vorgegebene Zeit verzögert \Rightarrow Timeout
 - Timeout • : Agent wartet auf COMMIT/ABORT \Rightarrow sende help-me
 - Timeout , : Koordinator wartet auf COMMIT/ABORT \Rightarrow entscheide ABORT
 - Timeout \mathcal{F} : Agent erhält kein prepare-to-COMMIT \Rightarrow entscheide ABORT
 - Variante: Lineares Zwei-Phasen-Commit-Protokoll
...
- \Rightarrow Abhilfe, wenn Koordinator blockiert: Agenten wählen neuen Koordinator
 \Rightarrow mehr Sicherheit: Drei-Phasen-Commit-Protokoll
- **mögliche Fehler in verteilten Datenbanken**
 - Transaktionsfehler \Rightarrow Abbruch einer Transaktion
 - Knotenfehler (Systemfehler) \Rightarrow Arbeitsspeicherinhalt verloren
 - Medienfehler \Rightarrow Verlust der Datenbank
 - Kommunikationsfehler (insbes. VDB)

5. Anfragebearbeitung

- totaler Zeitaufwand: \sum aller Kommunikations- + I/O + CPU-Zeiten
- Antwortzeit (\leq totaler Zeitaufwand bei paralleler Bearbeitung)
- wichtig: Die Optimierung sollte nicht länger dauern, als die erzielte Zeitersparnis
- Transformation globaler Anfragen in lokale
 1. Relation temporär physisch erzeugen \Rightarrow einfach, aber aufwendig in Ausführung
 2. Transformation der Anfrage, damit unmittelbar auf Teilrelationen anwendbar \Rightarrow effizienter Darstellung durch Operatorbäume
 1. Erkennung gemeinsamer Teilausdrücke (z.B. Differenz)
 2. Eliminierung von überflüssigen Teilausdrücken
$$r * r \Leftrightarrow r \cup r \Leftrightarrow r \cup \sigma[F] r \Leftrightarrow r$$
$$r \setminus r \Leftrightarrow \emptyset$$
$$r * \sigma[F] r \Leftrightarrow \sigma[F] r$$
$$r \setminus \sigma[F] r \Leftrightarrow \sigma[\neg F] r$$
$$(\sigma[F_1] r) * (\sigma[F_2] r) \Leftrightarrow \sigma[F_1 \wedge F_2] r \text{ (ditto: } \cup \hat{=} F_1 \vee F_2 \text{ bzw. } \setminus \hat{=} F_1 \wedge \neg F_2$$

Streiche Teilrelationen, die aufgrund ihrer Fragmentierung nichts zum Ergebnis beitragen
 3. Parallel- Ausführung von Teilanfragen
Selektionen und Projektionen so weit wie möglich reinziehen
 $\hat{=}$ Ausführungsplan (Operatorbaum + Kommunikations- und Ortsaspekte)
 4. Auswahl einer geschickten Reihenfolge
- Behandlung von Kopien
 - Anfrage: benutze Kopie mit geringsten Übertragungskosten
 - Update: alle Kopien ändern

- Übertragungskosten: $TC(x) = C_0 + x C_1$
- Übertragungsdauer: $TD(x) = D_0 + x D_1$
- **Berechnung von Joins**
 - Nested-Loop-Join
kein Index vorhanden
A sendet je ein Joinattribut an B, bei Treffer übermitteln beide das gesamte Tupel an C
à Vergleiche über Knotengrenzen nicht sinnvoll \Rightarrow sende kleinere Relation an anderen Knoten, um dort Join auszuführen
 - Sort-Merge-Join
nach Joinattribut sortierte Tabellen werden parallel durchsucht
A sendet Joinattribut an B, bei Treffer sendet B "Treffer, Next A=..." und Tupel an C, dann sendet A Tupel an C und "Treffer, NEXT B=..." oder "kein Treffer, NEXT B=...", ...
 - Semi-Join
Schicke nur Joinattribut (π) von Knoten A nach B, dort Join, sende Ergebnistupel an C und Joinattribut an A, A sendet Ergebnistupel ebenfalls an C
- Bestimmung einer optimalen Ausführungsstrategie
 - Relations-Profil: card(r) (Anzahl Tupel), size(A), size(r), val(A) (Anzahl versch. Werte), alloc(r) (Speicherort)

6. Mehrbenutzerkontrolle

- verantwortlich: Scheduler
- **Updates mit Replikaten**
 - Synchrone Änderung
 - schärfste Version: Änderungen sofort auf allen Kopien durchführen
 - Nachteil: Kopie nicht verfügbar \Rightarrow Änderung kann nicht ausgeführt werden
 - Vorteil: stets aktueller Stand
 - Teilsynchrone Änderung
 - nur verfügbare Kopien werden simultan geändert
 - Primärknoten merkt sich: pending updates à werden bei Wiederanlauf der Knoten durchgeführt
 - Vorteil: bessere Update Verfügbarkeit
 - Nachteil: komplexe Wiederanlauf- Prozeduren
 - Asynchrone Änderung
 - Änderungs- Transaktion führt nur Änderung auf einer Kopie durch
 - DBMS übernimmt weitere Änderungen, TA muß nicht warten
 - synchrones Sperren
alle Kopien werden simultan gesperrt
 - asynchrones Sperren
nur Primärkopie wird gesperrt, weitere Änderungs- TA's müssen diese sperren und dürfen nur diese ändern
à Abhängigkeit von der Verfügbarkeit einer bestimmten Kopie
à Kopien haben evtl. unterschiedliche Aktualitätsstände
- **Serialisierbarkeit** (im verteilten Fall)
 \Leftrightarrow Serialisierbarkeit aller lokalen Schedules \wedge globale Ordnung $T_1 < T_2 < \dots < T_n$ und gleiche lokale Ordnung an allen Knoten $T_1^K > T_2^K > \dots > T_n^K$.

- **Sperrverfahren**
 - Lesesperre: Shared Lock (s), dabei können andere TA's ebenfalls Lesesperren jedoch keine Schreibsperren setzen
 - Schreibsperre: Exclusive Lock (x), andere TA's dürfen keine Sperren setzen \Rightarrow jeglicher Zugriff verwehrt
 - **2-Phasen-Sperrprotokoll**
 - wohlgeformte TA's : d.h. TA sperrt Objekt vor Zugriff und gibt es spätestens am Ende der Transaktion wieder frei
 - 2-phasige-TA: nachdem 1 Sperre freigegeben wurde, kann keine neue Sperre mehr angefordert werden
 - alle Schreibsperren werden gleichzeitig freigegeben (gilt nicht immer, aber mit 2-Ph-Commit!)
 - Satz: jedes System 2-phasiger TA's ist (global und verteilt) serialisierbar (Umkehrung gilt nicht immer, da 2-Ph-Sperrprotokoll unnötig restriktiv)
 - Sperrverwaltung
 - zentral \Rightarrow Abhängigkeit von einem Knoten
 - verteilt (jeder Knoten hat eigenen Lock-Manager)
 - mit Primärkopie (Sperren der Primärkopie \Rightarrow Lesen einer beliebigen, bzw. Update aller Kopien)
- **Deadlock**
 - Deadlock := TA1 wartet auf Freigabe einer Sperre, die von TA2 gehalten wird und umgekehrt
 - es können auch globale, knotenübergreifende Deadlocks auftreten
 - Auflösung durch Timeouts
 - Nachteil: Wartezeit wegen (Netz-)Überlastung \Rightarrow Abbruch unnötig
 - Nachteil: Timeout zu groß \Rightarrow im Deadlockfall wartet TA zu lange
 - Deadlockerkennung durch Zyklenuntersuchung des Wartet-auf-Graphen
 - zentralisierte Deadlockerkennung
 - ein Knoten faßt alle lokalen Wartet-auf-Graphen zusammen
 - Nachteil: großer Kommunikationsaufwand
 - hierarchische Deadlockerkennung
 - senden des lokalen Wartet-auf-Graphen an übergeordneten Knoten
 - Vorteil: Abhängigkeit vom Zentralknoten geringer, als im zentralen Fall
 - Nachteil: aufwendige Realisierung
 - verteilte Deadlockerkennung
 - jeder Knoten erstellt lokalen Wartet-auf-Graphen mit [EXT] für alle externen Beziehungen
 - Zyklus mit [EXT] \Rightarrow potentieller globaler Deadlock
 - \Rightarrow Mitteilung an die Knoten, die evtl. zum Deadlock beitragen
 - Vorteil: Verringerung des Kommunikationsaufwandes
 - Nachteil: Deadlocks falsch erkannt \Rightarrow unnötige Rücksetzung der TA

7. Fehlerbehandlung

- **Recovery** := Wiederherstellen eines konsistenten Datenbankzustandes nach aufgetretenen Fehlern
- **Logfile:**
 - Before Image: Objektwerte vor Änderungen \Rightarrow UNDO möglich
 - After Image: Objektwerte nach Änderung \Rightarrow REDO möglich
 - Dump (Kopie der Datenbank, alter aber konsistenter Zustand)
- Kurzzeit- Recovery
 - Zurücksetzen einer abgebrochenen TA
 - Wiederanlauf des lokalen DBMS nach Systemabsturz (Einspielen von Änderungen, Zurücksetzen nicht abgeschlossener Änderungen, evtl. Neustart abgebrochener TA's)
 - Sichern von Before- /After- Images
 - Behandlung einfacher Lese-/Schreibfehler
 - verteilt: Wiedereingliederung
 - mit Primärkopie:
nur Anforderung der Änderungen von Primärkopie
Ausfall der Primärkopie \Rightarrow Rundumanfrage, ob neue Primärkopie
 - bei teilsynchroner Änderung:
Erfragen des neuesten Standes, exklusives Sperren der Kopien
- Langzeit- Recovery
 - bei Hardware- oder Plattenfehlern
 - Herstellen eines Sicherungspunktes:
 - strikt synchronisiert
Anhalten der DB, an allen Knoten gleichzeitig Sicherungspunkt herstellen
 \Rightarrow Sicherungstransaktion, die exklusiv alle Daten an allen Knoten sperrt
 - lose synchronisiert
Verschiebung des lokalen Sicherungspunktes bis Sub-Transaktion initiiert wird, die sich auf späteren Sicherungspunkt bezieht
 - nicht synchronisiert
Knoten entscheidet selbst über lokale Sicherungspunkte
Abhängigkeiten aus Logfile ableitbar