Datenföderalismus

Methoden und Probleme Verteilter Datenbanksysteme

Dr. Erhard Rahm

Alle Informationen in einer Hand, auf einem Rechner, dieses Konzept ist den heutigen Anforderungen nicht mehr gewachsen. Zu lange dauert es, wenn beispielsweise die Frankfurter Filiale eines Großunternehmens die Zentrale in Tokio um Informationen bittet, die vor Ort dringend benötigt werden. Das geht auch anders. Softwareingenieure und Programmierer sind aufgefordert, sich mit einer neuen, flexibleren Form der Datenhaltung bekannt zu machen: den Verteilten Datenbanksystemen.

Herkömmlicherweise werden die harten Fakten eines Unternehmens in einer zentralen Datenbank (DB) gespeichert, auf die der Benutzer über ein Datenbank-Managementsystem (DBMS) zugreift. Das DBMS verbirgt sämtliche Interna der Datenorganisation und -speicherung dem Anwender gegenüber; der Zugriff erfolgt in der Regel über eine deskriptive und ausdrucksstarke Anfragesprache wie zum Beispiel SQL. Datenbanksystem und Anwendungsprogramme laufen meist auf ein und demselben Rechner, die eigentlichen Benutzer (zum Beispiel Sachbearbeiter einzelner Abteilungen) sind mit der Zentralen nur über Terminals oder andere Endgeräte verbunden.

Diese, an einem Rechenzentrum orientierte Systemstruktur wird jedoch in vielen Fällen der Organisationsform eines Unternehmens nicht mehr gerecht. So schreibt das zentralistische Modell beispielsweise einer Großbank mit mehreren Filialen vor, sämtliche Konten und Personaldaten aller Zweigstellen zentral zu speichern. Dies führt zum einen zu einem hohen Kommunikationsaufwand zwischen Filialen und Zentrale, da in den Zweigstellen kein lokaler Datenzugriff möglich ist. Zudem können die starken Abhängigkeiten von der Zentrale zu Akzeptanzproblemen führen und gegebenenfalls eine unkontrollierte Verwaltung lokaler Daten hervorrufen. Führen zum Beispiel die Zweigstellen Kopien der sie betreffenden Personaldaten, dann sind dort eigene Anwendungen zu entwickeln, die die Kopien der einzelnen Datenbestände auf den gleichen Aktualitätsstand bringen (Konsistenz). Denkt man an die Möglichkeit, private Datenbanken auf PCs und Workstations zu halten, vervielfachen sich derartige Datenverwaltungs- und Konsistenzprobleme noch.

Signifikante Vorteile

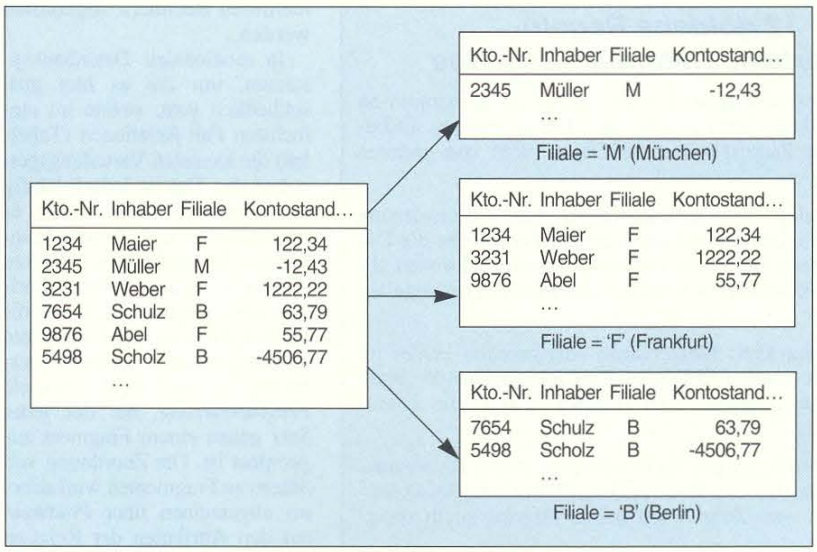
Verteilte Datenbanksysteme gestatten es dagegen, die Systemstruktur an die jeweilige betriebliche Organisationsstruktur anzupassen, ohne die Konsistenz des Systems zu gefährden. Eine logische Datenbank wird dabei unter mehreren Datenbanksystemen physisch aufgeteilt, die einzelnen DBMS laufen in der Regel auf verschiedenen, geographisch verteilten Rechnern. Um Datenbankoperationen über dem verteilten Gesamtbestand abzuarbeiten, kooperieren die einzelnen Rechner eng miteinander. Die Kooperation findet auf der Ebene der Datenbanksysteme statt. Dadurch bleiben dem Benutzer mit seinen Anwendungsprogrammen sämtliche Aspekte der Verteilung verborgen. Für ihn sieht die Zugriffsschnittstelle genauso aus wie beim zentralen Fall, beispielsweise bei Mainframes. Eine solche *Verteilungstransparenz* zu gewährleisten bleibt die Hauptforderung an Verteilte Datenbanksysteme. Sie ist wesentlich für eine einfache Bedienung, da trotz der Verteilung sämtliche Verwaltungsfunktionen Aufgabe der Datenbank-Software bleiben.

Insbesondere haben Änderungen in der Datenverteilung keine Auswirkungen auf bestehende Anwendungsprogramme. In der erwähnten Bankanwendung ließe sich so auf einfache Weise eine Anpassung an die betriebliche Organisationsstruktur erreichen. Jeder Zweigstelle wird ein lokales DBMS zugeordnet, das die sie betreffenden Konto- und Personaldaten selbst verwaltet. Durch Kooperation mit den DBMS anderer Zweigstellen ist weiterhin der Zugriff auf den gesamten Datenbestand gewährleistet.

Neben der Unterstützung dezentraler Organisationsformen bieten Verteilte Datenbanksysteme eine Reihe weiterer Vorteile, was die Leistungsfähigkeit, Verfügbarkeit und Kosteneffektivität betrifft. Ein einzelner Rechner kann leicht zum Systemengpaß mutieren und dadurch Durchsatz und Antwortzeiten beim Datenbankzugriff beeinträchtigen. Bei Verteilten

Datenbanksystemen dagegen wirkt die Verarbeitungskapazität mehrerer Rechner derartigem Fehlverhalten entgegen; zudem läßt sich die Leistungsfähigkeit durch Erhöhung der Rechneranzahl inkrementell erweitern. Da die Mehrzahl der Zugriffe nur lokal gespeicherte Daten betrifft, entfallen Kommunikationsverzögerungen mit entfernten Rechnern weitgehend. Das unterstützt zusätzlich kurze Antwortzeiten.

Ein weiterer Nachteil der zentralisierten Systemarchitektur besteht in ihrer geringen Verfügbarkeit. Wenn nicht ein Reserverechner die Verarbeitung fortsetzen kann, legt der Ausfall der zentralen DB-Maschine den gesamten Betrieb lahm. Im verteilten Fall dagegen blockiert ein Rechnerausfall lediglich die von ihm verwaltete Datenmenge; die DB-Verarbeitung der übrigen Rechner (in anderen Zweigstellen) bleibt davon unberührt. Eine weitere Steigerung der Verfügbarkeit wird erreicht, wenn Teile der Datenbank unter der Kontrolle der DBMS repliziert, das heißt mehrfach gespeichert werden. Liegen die vom Ausfall betroffenen Daten auf einem weiteren Rechner vor, kann der Benutzer nach einem solchen Ausfall weiterhin auf die gesamte Datenbank zugreifen. Schließlich arbeiten Verteilte Systeme kostengünstiger als zentrale Datenspeicher. In letzterem Fall



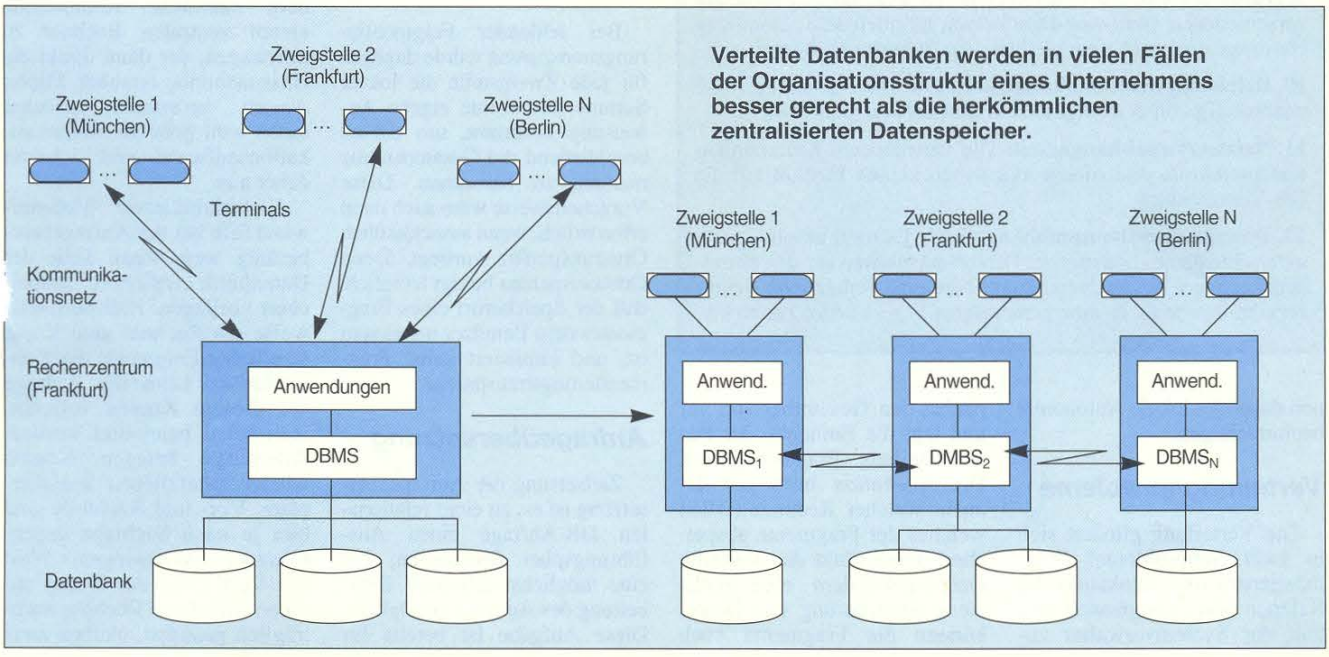
Horizontale Fragmentierung der Kontorelation nach Filialen: jede Zweigstelle speichert die sie betreffenden Daten direkt vor Ort.

nämlich erreichen oft nur Großrechner (Mainframes) eine ausreichend hohe Verarbeitungskapazität. Die Kosten sind dementsprechend hoch. Verteilte Datenbanken dagegen erlauben die Nutzung mikroprozessorbasierter Rechnerknoten mit typischerweise weit geringeren Kosten pro MIPS (million instructions per second) als Großrechner.

Hohe Anforderungen

Um diese Vorteile realisieren zu können, sind jedoch komplexe Änderungen der vorhandenen DB-Software vonnöten. Die wichtigsten Anforderungen, die Verteilte Datenbanksysteme idealerweise erfüllen sollten, formulierte C. J. Date [4] in 12 sogenannten 'Regeln'. Die meisten seiner Postulate betreffen Einzelaspekte hinsichtlich der Verteilungs- und Replikationstransparenz sowie der Systemunabhängigkeit (siehe Kasten). Technische Schwierigkeiten, diesen Anforderungskatalog zu erfüllen, führten dazu, daß trotz intensiver Forschungsarbeiten erst Mitte der achtziger Jahre die ersten brauchbaren Systeme auf den Markt kamen. Mittlerweile ist bereits eine Vielzahl verteilter Datenbanksysteme kommerziell verfügbar (Ingres Star, Tandem NonStop SQL, Oracle etc.), zumeist jedoch ohne den geforderten Funktionsumfang voll abzudecken.

Auch die Administration eines Verteilten Datenbanksystems ist weitaus komplexer als die eines zentralen DBMS. Der Systemverwalter muß den Datenbestand auf die einzelnen Rechner physisch aufteilen. Um den Kommunikationsaufwand möglichst gering zu halten, sollte dabei ein möglichst hoher Anteil lokaler Datenbankzugriffe unterstützt werden. Änderungen etwa der logischen Datenstruktur und der Zugriffsrechte sind aber auf jeden Fall auch global abzustimmen und kön-



12 goldene Regeln für Verteilte Datenbanksysteme

1. Lokale Autonomie: Jeder Rechner sollte ein Maximum an Kontrolle über die auf ihm gespeicherte Daten ausüben. Insbesondere darf der Zugriff auf diese Daten nicht von anderen Rechnern abhängen.

2. Keine zentralen Systemfunktionen: Zur Unterstützung einer hohen Knotenautonomie und Verfügbarkeit sollte die Datenbankverarbeitung nicht von zentralen Systemfunktionen abhängen. Solche Komponenten bilden zudem einen potentiellen Leistungsengpaß.

3. Hohe Verfügbarkeit: Idealerweise unterbrechen Fehler im System (z. B. Rechnerausfall) oder Konfigurationsänderungen (Installation neuer Software oder Hardware) nicht die Datenbankverarbeitung.

4. Ortstransparenz: Der physische Speicherort von Datenbankobjekten sollte für den Benutzer verborgen bleiben. Der Datenzugriff darf sich vom Zugriff auf lokale Objekte nicht unterscheiden.

5. Fragmentierungstransparenz: Eine Relation (Table) der Datenbank sollte verteilt auf mehreren Knoten gespeichert werden können. Die dabei zugrunde liegende (horizontale oder vertikale) Fragmentierung der Relation bleibt für den Datenbankbenutzer transparent, das heißt, unsichtbar.

6. Replikationstransparenz: Die mehrfache Speicherung von Teilen der Datenbank auf unterschiedlichen Rechnern bleibt für den Benutzer unsichtbar; die Wartung der Redundanz obliegt ausschließlich der DB-Software.

7. Verteilte Anfrageverarbeitung: Innerhalb einer DB-Operation (SQL-Anweisung) sollte die Möglichkeit bestehen, auf Daten mehrerer Rechner zuzugreifen. Zur effizienten Bearbeitung sind durch das verteilte DBMS geeignete Techniken bereitzustellen (Query-Optimierung).

8. Verteilte Transaktionsverwaltung: Das DBMS hat die Transaktionseigenschaften auch bei verteilter Bearbeitung einzuhalten. Dazu sollten entsprechende Recovery- und Synchronisationstechniken bereitstehen (verteiltes Commit-Protokoll, Behandlung globaler Deadlocks, u. a.).

9. Hardware-Unabhängigkeit: Die DB-Verarbeitung sollte auf verschiedenen Hardware-Plattformen möglich sein. Sämtliche Hardware-Eigenschaften bleiben dem Benutzer verborgen.

10. Betriebssystemunabhängigkeit: Die DB-Benutzung sollte unabhängig von den eingesetzten Betriebssystemen sein.

11. Netzwerkunabhängigkeit: Die verwendeten Kommunikationsprotokolle und -netzwerke haben keinen Einfluß auf die DB-Verarbeitung.

12. Datenbanksystemunabhängigkeit: Es muß möglich sein, unterschiedliche (heterogene) Datenbanksysteme auf den einzelnen Rechnern einzusetzen, solange sie eine einheitliche Benutzerschnittstelle (z. B. eine gemeinsame SQL-Version) anbieten.

nen daher die lokale Autonomie beeinträchtigen.

Verteilungsprobleme

Die Verteilung gliedert sich in zwei Subprobleme: Fragmentierung und Allokation. Im Rahmen der *Fragmentierung* teilt der Systemverwalter zunächst den Gesamtbestand auf und legt die Einheiten der Datenverteilung (Fragmente) fest. Die *Allokation* bestimmt danach, welcher Rechner (DBS) welches der Fragmente abspeichert. Unterstützt das Verteilte Datenbanksystem eine replizierte Speicherung von Daten, können die Fragmente auch mehreren Rechnern zugeordnet werden.

In relationalen Datenbanksystemen, um die es hier ausschließlich geht, stellen im einfachsten Fall Relationen (Tabellen) die kleinsten Verteilungsgranulate dar. Das ist jedoch häufig zu unflexibel. Daher sollte es möglich sein, auch feinere Fragmente zu definieren. Hierzu kommt eine zeilenweise (horizontale) oder spaltenweise (vertikale) Aufteilung der Datensätze in Betracht. Am weitesten verbreitet ist dabei die *horizontale Fragmentierung*, bei der jeder Satz genau einem Fragment zugeordnet ist. Die Zuordnung von Sätzen zu Fragmenten wird dabei im allgemeinen über Prädikate auf den Attributen der Relation definiert. In der Bankanwendung könnten beispielsweise die Kontodaten nach der zugehörigen Zweigstelle horizontal fragmentiert und jedes der Fragmente dem DBMS der jeweiligen Zweigstelle zugeordnet werden. Die Aufteilung der Daten muß sich an der jeweiligen Anwendung orientieren.

Fragmentierungstransparenz heißt, daß die vorgenommene Verteilung für den Benutzer verborgen bleibt. Er bezieht sich auf die logische Definition der Relation. Eine Abfrage etwa, die im Bankbeispiel die Summe aller Kontostände ermittelt, lautet im zentralen wie auch im verteilten Fall:

```
SELECT SUM (Ktostand) FROM
KONTO
```

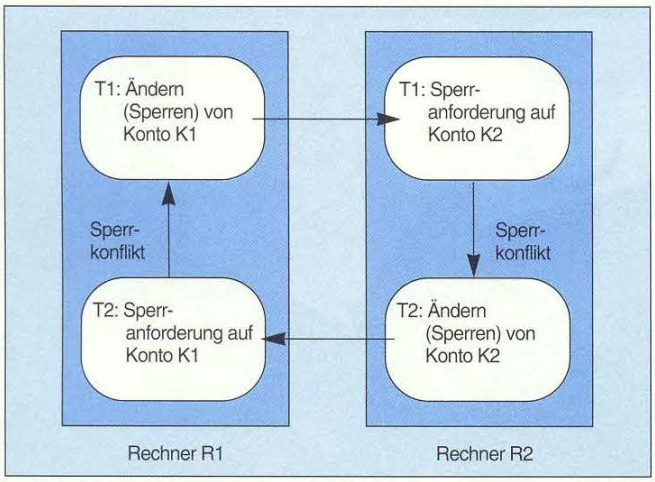
Bei fehlender Fragmentierungstransparenz würde dagegen für jede Zweigstelle die lokale Summe durch eine eigene Anweisung bestimmt, um daraus anschließend die Gesamtsumme manuell zu berechnen. Diese Vorgehensweise wäre auch dann erforderlich, wenn ausschließlich Ortstransparenz vorliegt. Denn Ortstransparenz besagt lediglich, daß der Speicherort eines Fragmentes dem Benutzer unbekannt ist, und impliziert keine Fragmentierungstransparenz.

Anfrageübersetzung

Zielsetzung der Anfrageübersetzung ist es, zu einer relationalen DB-Anfrage einen Ausführungsplan festzulegen, der eine möglichst effiziente Bearbeitung des Auftrags ermöglicht. Diese Aufgabe ist bereits für zentralisierte DBMS komplex, da im allgemeinen eine große Anzahl alternativer Ausführungsstrategien besteht. Noch komplizierter ist die Bestimmung eines optimalen Ausführungsplans im verteilten Fall, da neben CPU- und E/A-bezogenen Ausführungskosten der Kommunikationsaufwand (Anzahl und Umfang von Nachrichten) als weiterer Faktor dazu kommt. Sind Teile der Datenbank repliziert gespeichert, ergeben sich für die Anfrageübersetzung zusätzliche Optimierungsmöglichkeiten. Denn für den Datenzugriff wird dann die Kopie ausgewählt, die die geringsten Kommunikationskosten verursacht. Außerdem besteht die Möglichkeit, eine Anfrage parallel an verschiedenen Rechnern zu bearbeiten, um dadurch die Antwortzeit weiter zu reduzieren.

Vergleichsweise einfach ist die Erstellung eines verteilten Ausführungsplanes für die Beispielanfrage zur Berechnung der Kontostandssumme. Hierzu fordert das DBMS, das die Anfrage zur Bearbeitung erhielt, die DBMS der anderen Rechner auf, ihre Kontostände zu summieren. Das kann natürlich auf allen Rechnern parallel erfolgen. Aus den Teilergebnissen wird dann einfach die Gesamtsumme errechnet und an den Fragesteller zurückgeliefert. Eine alternative Vorgehensweise bestünde darin, die Summen nicht lokal zu berechnen, sondern sämtliche Kontostände einem zentralen Rechner zu übertragen, der dann direkt die Gesamtsumme ermittelt. Dieser Ansatz verursacht offenbar einen weit höheren Kommunikationsaufwand und scheidet daher aus.

Kommunikativer Mehraufwand fällt bei der Anfragebearbeitung weg, wenn Teile der Datenbank repliziert abspeichert vorliegen. Hält beispielsweise ein Rechner eine Kopie sämtlicher Fragmente der Kontorelation, kann die Anfrage auf diesem Knoten vollkommen lokal bearbeitet werden. Allerdings belegen Kopien wieder zusätzlichen Speicherplatz. Vor- und Nachteile sind hier je nach Sachlage gegeneinander aufzuwiegen. Wird die Verteilung der Daten auf unterschiedliche Rechner nachträglich geändert, bleiben zwar



Globaler Deadlock zwischen zwei Überweisungstransaktionen T1 und T2: T1 will von K1 überweisen und sperrt K2, T2 will von K2 überweisen und sperrt K1.

aufgrund der Verteilungstransparenz Anwendungsprogramme davon unberührt, nicht aber die bei der Programmübersetzung erstellten Ausführungspläne. Das System muß daher in der Lage sein, von Umverteilungen betroffene Ausführungspläne automatisch zu erkennen und neu zu bestimmen.

Transaktionen

In zentralisierten wie auch in Verteilten Datenbanksystemen bilden Transaktionen die Einheiten der DB-Verarbeitung. Eine *Transaktion* umfaßt dabei eine oder mehrere DB-Operationen, für die die DB-Software die sogenannten Transaktionseigenschaften gewährleistet. Dazu zählt vor allem die Alles-oder-nichts-Eigenschaft (Atomarität), die besagt, daß Änderungen einer Transaktion entweder vollständig oder überhaupt nicht in die Datenbank übernommen werden. Erfolgreichen Transaktionen wird die Dauerhaftigkeit ihrer Änderungen zugesichert, das heißt, sie gehen trotz möglicher Fehlersituationen (Rechner- oder Plattenausfall, Kommunikationsfehler) nicht mehr verloren. Zusicherungen solcher Art machen spezielle Log-Dateien möglich, die alle Änderungen mitprotokollieren, das DBMS stellt für Ausfälle geeignete Recovery-Funktionen bereit.

Transaktionen beinhalten aber noch mehr. Eine weitere Aufgabe betrifft die Isolation (Synchronisation) von gleichzeitig auf der Datenbank arbeitenden Benutzern, um Inkonsistenzen zu vermeiden. Dazu wird vom DBMS vor dem Zugriff auf ein DB-Objekt eine entsprechende Sperre (Locking) angefordert. Dadurch wird verhindert, daß andere Transaktionen dasselbe Objekt zur gleichen Zeit in unverträglicher Weise referenzieren.

Auch bei verteilten Systemen ist die Einhaltung dieser Transaktionseigenschaften erklärtes Ziel aller Beteiligten. Da in Verteilten Datenbanken eine Transaktion Änderungen an mehreren Rechnern vornehmen kann, hat zur Sicherstellung der Atomarität die Koordinierung der beteiligten DBMS Vorrang. Hierzu wird üblicherweise ein verteiltes *Zweiphasen-Commit-Protokoll* eingesetzt. Dabei fungiert das DBMS des Rechners, auf dem die Transaktion gestartet wurde, als Koordinator. Das Protokoll, das der Koordinator am Ende einer Transaktion startet, umfaßt zwei Phasen:

1. Der Koordinator schickt eine 'Prepare-to-Commit'-Nachricht an alle DBMS, die an der Bearbeitung der Transaktion beteiligt waren. Jedes dieser DBMS trifft eine lokale Entscheidung über den Ausgang der Transaktion (*Commit* für erfolgreiche, *Abort* für erfolglose Beendigung) und sendet das Ergebnis an den Koordinator zurück. Nach Eingang aller Teilergebnisse beim Koordinator ist Phase 1 beendigt.

2. Der Koordinator bestimmt aus den Teilergebnissen den Ausgang der Transaktion und protokolliert das Ergebnis in seiner Log-Datei. Das Gesamtergebnis ist dabei *Commit*, falls alle beteiligten DBMS mit *Commit* gestimmt haben, ansonsten wird die Transaktion zurückgesetzt (Abort). Das Gesamtergebnis teilt der Koordinator allen beteiligten DBMS mit, die daraufhin ihrerseits die Transaktion entweder zurücksetzen oder erfolgreich beenden. Sperrmechanismen werden erst zu diesem Zeitpunkt freigegeben. Danach erfolgt eine Quittierung an den Koordinator, um dort die Beendigung der Transaktion zu ermöglichen.

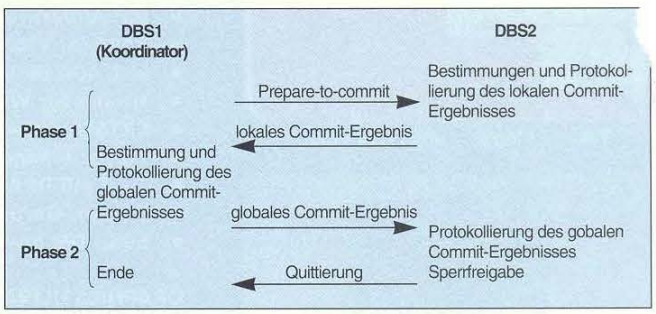
Ein gewisses Problem bei dem skizzierten Protokoll, das soll hier nicht verschwiegen werden, ist die Abhängigkeit vom Koordinator. Erst wenn dieser das Commit-Ergebnis den anderen Rechnern mitgeteilt hat, ist eine Sperrfreigabe möglich. Ein Ausfall des koordinierenden Rechners während des Commit-Protokolls kann daher zu langen Blockierungen führen. In der Literatur wurden zwar – auf Kosten eines erhöhten Nachrichtenaustausches – verschiedene Abhilfen vorgeschlagen, jedoch ist keines dieser Verfahren in kommerziellen Systemen implementiert.

Die Sperrverwaltung ist auch im verteilten Fall relativ einfach möglich: jedes DBMS verwaltet Sperren der ihm zugeordneten Daten selbst. Wartesituationen aufgrund miteinander unverträglicher Sperren können aber zu rechnerübergreifenden Verklemmungen führen. Ein solcher globaler Deadlock tritt zum Beispiel auf, wenn eine Transaktion T1 von Rechner R1 an Rechner R2 auf eine Sperre einer Transaktion T2 warten muß und T2 danach auf Rechner R1 eine Sperre anfordert, die von T1 bereits gehalten wird. In der Bankanwendung wäre dies der Fall, wenn T1 eine Überweisung von einem Konto K1 von Rechner R1 (Filiale) auf Konto K2 von R2 vornimmt und T2 zeitgleich eine Überweisung von K2 auf K1. Die einfachste Methode zur Behandlung globaler Deadlocks ist ein Timeout-Verfahren, das eine Transaktion nach Überschreiten einer maximalen Sperrwartezeit zurücksetzt.

Replizierte Datenbanken

Schwieriger wird es, die Konsistenz und Aktualität der Daten zu gewährleisten, wenn Teile der Datenbank auf mehreren Rechnern gleichzeitig vorliegen. Die replizierte Speicherung erhöht zwar die Verfügbarkeit und bietet mehr Möglichkeiten zur effizienten Bearbeitung lesender DB-Operationen. Auf der anderen Seite vergrößert sich natürlich der Speicherplatzbedarf und Änderungen sind auf sämtliche Kopien anzuwenden, um die Konsistenz der Datenbank zu wahren. Die Forderung nach Replikationstransparenz verlangt außerdem, daß die Replikation für den Benutzer unsichtbar bleibt und die Wartung der Redundanz somit durch das DBMS erfolgt.

Ein Ansatz zur Wahrung der DB-Konsistenz bei replizierten Datenbanken besteht darin, bei der Änderung eines Objektes sämtliche Kopien zu sperren und im Rahmen der Commit-Behandlung zu aktualisieren. Dieses Verfahren hat allerdings



Das Zweiphasen-Commit-Protokoll reglementiert den Ablauf von Transaktionen.

einen offensichtlichen Nachteil: die zum Sperren und Aktualisieren aller Kopien benötigte große Nachrichtenanzahl verlängert in hohem Maße die Antwortzeit für ändernde Transaktionen. Darüber hinaus ergibt sich potentiell gar keine Verbesserung, sondern sogar eine Verschlechterung der Verfügbarkeit, da eine Änderung prinzipiell verlangt, daß freier Zugriff auf sämtliche Kopien besteht.

Ein alternativer Ansatz sähe so aus: für jedes Objekt wird auf einem der Rechner eine sogenannte Primärkopie geführt. Ändernde Personen brauchen nur die Primärkopie zu sperren und zu aktualisieren. Die restlichen Kopien aktualisiert der Primärkopienrechner asynchron, ohne die Antwortzeit für Änderungstransaktionen zu erhöhen. Aber auch hier gibt es einen Nachteil zu verzeichnen. Asynchron aktualisierte Kopien können leicht veraltet sein. Das ist in vielen Anwendungen jedoch tolerabel.

Ausblick

Aus Benutzersicht gibt es zur Nutzung verteilter Datenbestände kaum besseres als Verteilte Datenbanksysteme, da die Systemsoftware sämtliche Aspekte der Verteilung verbirgt. Auf der anderen Seite ist die Administration solcher Systeme sehr komplex, sie lassen aufgrund der erforderlichen Kooperation zwischen den beteiligten Datenbanksystemen auch nur ein begrenztes Maß an lokaler Autonomie zu. Vor allem dieser Punkt, der insbesondere bei heterogenen Datenbanksystemen mit unabhängig voneinander entworfenen Datenbanken von Bedeutung ist, führte dazu, daß in der Praxis andere Systemstrukturen im Einsatz sind, die einen geringeren Grad an Verteilungstransparenz bieten. Zu nennen sind hier vor allem aktuelle Standardisierungsbemühungen wie RDA (Remote Database Access), die es erlauben, in einem Anwendungsprogramm DB-Aufträge an unterschiedliche Datenbanksysteme zu schicken.

Die Existenz verschiedener Datenbanken ist dabei jedoch für den Anwendungsprogrammierer sichtbar; zudem kann innerhalb einer DB-Operation nur auf Daten eines Rechners (DBMS) zugegriffen werden. Einen Kompromiß zwischen dem weniger transparenten Ansatz und Verteilten Systemen stellen sogenannte *Föderative Datenbanksysteme* (federated DBMS) dar, die derzeit im Mittelpunkt vieler Forschungsprojekte stehen. Föderative Ansätze streben eine weitgehende Autonomie der Einzelrechner sowie eine Unterstützung heterogener DBMS an, die sich in Anfragesprache, Datenmodell, Datentypen, Synchronisations- und Recovery-Verfahren unterscheiden können. Trotzdem soll durch eine begrenzte Kooperation der einzelnen DBMS eine verteilte Bearbeitung von DB-Operationen und eine korrekte Transaktionsverwaltung ermöglicht werden.

Einen weiteren Entwicklungstrend markieren *Parallele Datenbanksysteme*, die eine hochgradig parallele Bearbeitung von komplexen und datenintensiven DB-Operationen erlauben. Solche Architekturen weisen zwar viele Gemeinsamkeiten mit Verteilten Datenbanksystemen auf, jedoch steht bei ihnen die effiziente Bearbeitung innerhalb eines lokal verteilten Mehrrechnersystems mit hoher Kommunikationsbandbreite im Vordergrund. (ku)

Der Autor ist Dozent an der Universität Kaiserslautern (FB Informatik).

Literatur

[1] R. Bayer, K. Elhardt, K. Kießling, D. Killar: Verteilte Datenbanksysteme – Eine Übersicht über den heutigen Entwicklungsstand, Informatik-Spektrum 7 (1), S. 1 bis 19, 1984

[2] D. Bell, J. Grimson: Distributed Database Systems, Addison Wesley, 1992

[3] S. Ceri, G. Pelagatti: Distributed Databases. Principles and Systems. McGraw-Hill, 1984

[4] C.J. Date: An Introduction to Database Systems, 5th edition (Chapter 23), Addison Wesley, 1990

[5] M.T. Özsu, P. Valduriez: Principles of Distributed Database Systems, Prentice Hall, 1991

[6] A. Reuter: Verteilte Datenbanksysteme. Stand der Technik und aktuelle Entwicklungen. Proc. 18. GI- Jahrestagung, Informatik-Fachberichte 187, Springer-Verlag 1988, S. 16 bis 33 c't