Parallele Datenbanksysteme

Ein paralleles Datenbanksystem ist eine besondere Spezies verteiltes Datenbanksystem mit

- einem sehr schnellen Netzwerk
- Betonung auf verteilte Ausführung zusätzlich zur verteilten Speicherung von Daten
- dem großen Ziel einer schnellen Antwortzeit für Anfragen
- (Fehlertoleranz, niedrige Kommunikationskosten sind "hausgemachte" Ziele und nicht primäre Motivation)

Überblick

- Einleitung + Strunzerei
- Architekturen
- Datenpartitionierung und Allokation
- Parallele Anfragenbearbeitung
- Sonstiges wie Switch-Over Time, Indexauswahl und gemischte Workloads (gar nicht, obwohl in der Praxis sehr wichtig)

Literatur: DeWitt, Gray, CACM Juni 1992

Die gibt es wirklich!

Software Spezialisten

- Teradata (Spezialist)
- Tandem (Spezialist)
- IBM DB2/2 (besondere parallele Version)
- Oracle (besondere parallele Version)
- Sybase (besondere parallele Version)
- Informix (besondere parallele Version)

Erwartet einen Staatsstreichversuch von Microsofts SQL Server.

Hardware Spezialisten

- Intel
- Sequent
- NCR
- IBM (SP2)

Die gibt es wirklich! (2)

Forschungsprototypen:

- Gamma (ab 1979, Wisconsin)
- Bubba (Mitte der Achtziger, MCC Texas)
- XPRS (ca. 1990, Berkeley)
- ... (Serie von Projekten, die irgendeinen netten Baustein entwickelt haben)

Bemerkung

Auffällig: Die Industrie entwickelt wie wild. Die Forschung macht zur Zeit wenig bis gar nichts. (Das ist normalerweise umgekehrt!)

Ein Anwendungsbeispiel (von 1997)

WWW Search Engines; z.B. Inktomi (HotBot); einige Features aus dem Gedächtnis:

- Anfrage alla "gib mir Adressen der 1000 WWW Seiten, die am besten zu den Begriffen *Karneval, Bläääck Föös, Köln, 1. FC Köln* passen."
- Read-Only Workload. (Updates sind asynchron.)
- Intern: Selektion und Join von "Inverted Lists" (Beispiel eines Joins, wo man ihn nicht vermutet.)
- 64 UltraSparcs = 128 Prozessoren; insgesamt heterogene Hardware
- 256 Festplatten a 4 GB
- Partitionierung nach "Begriffen" (i.e., in Einheiten von Inverted Lists)
- Allokation Round Robin mit Spiegelung zur Erhöhung der Fehlertoleranz.
- Ständig werden Maschinen ausgetauscht und neue hinzugefügt. Es wird immer das beste Preis/Leistungsverhältnis an Hardware gekauft.
- WWW Search Engines sind knallhartes Geschäft. Man muß sehr gut kalkulieren; beste Preis/Performance fürs Gesamtsystem siegt.

Woran erkennt man ein gutes Paralleles DBMS?

Skalierbarkeit, Skalierbarkeit, Skalierbarkeit

1. Linearer Speed-Up
   - Eine Anfrage kann in *halber* Zeit bearbeitet werden, wenn die *doppelte* Anzahl von Prozessoren zur Verfügung steht.
   - Gibt es Beispiele bei Datenbanken für "superlinearen" Speed-Up?
2. Linearer Scale-Up
   - transaction scale-up: Es können *doppelt* soviele Transaktionen in derselben Zeit bearbeitet werden, wenn die *doppelte* Anzahl von Prozessoren zur Verfügung steht.
   - batch scale-up: Eine Anfragen kann auf eine *doppelt* so große Datenbank in derselben Zeit bearbeitet werden, wenn die *doppelte* Anzahl von Prozessoren zur Verfügung steht.

... und ansonsten ist alles genauso wie vorher.

Einige Rekorde

6000 Transaktionen pro Sekunde: Oracle, Anfang der 90er, Hardware habe ich vergessen.

21000 TPC-C Transaktionen pro Minute: NCR, ca. 1997, 110 CPUs, 800 Festplatten.

1 GB in 2,41 Sekunden sortieren: NOW Projekt (Berkeley), 1996, 32 CPUs, 64 Festplatten.

Bemerkung

Rekorde aufstellen hat sich als Sport in der Datenbankszene etabliert. Es gibt sogar Pokale.

Was steht linearem Scale-Up und Speed-Up im Weg?

Startup: Zu Beginn einer Anfrage müssen alle Prozessoren "angeworfen" werden. Dieser Vorgang kann bei vielen Prozessoren teurer sein, als die eigentliche Bearbeitung der Anfrage.

Interference: Die Prozessoren stehen sich gegenseitig auf den Füßen; i.e., Synchronisationsaufwand bei der Benutzung von gemeinsamen Ressourcen wie z.B. dem Netzwerk.

Skew: Es ist schwierig die Arbeit gleichmäßig zu verteilen. Die Gesamtantwortzeit richtet sich immer an der Laufzeit des Langsamsten; i.e., des Prozessors der am meisten zu tun hat. Es hilft nichts, wenn man einem Prozessor, der ohnehin wenig zu tun hat, durch Hinzunahme weiterer Prozessoren Arbeit abnimmt.

Gegenmaßnahmen:

1. gegen Startup: möglichst viel parallelisieren und vorsichtig sein. Ansonsten ist man machtlos.
2. gegen Interference: die richtige Architektur wählen. (Siehe die nächste Folie.)
3. gegen Skew: Sampling der Daten, geschickte und adaptive Auswertungsstrategien. (Wird hoffentlich später klar.)

Architektur von PDBMS

Es gibt grundsätzlich drei verschiedene Architekturen:

shared memory: alle Prozessoren arbeiten auf einem gemeinsamen Hauptspeicher. (CPUs haben aber sehr wohl ihren eigenen Cache.)

shared disk: alle Prozessoren haben ihren eigenen Hauptspeicher; arbeiten aber auf einem gemeinsamen Pool von Festplatten.

shared nothing:
alle Prozessoren bilden eine vollständige Einheit aus CPU (mit Cache), Hauptspeicher und Festplatte(n). Kommunikation zwischen den einzelnen Knoten über ein Hochgeschwindigkeitsnetzwerk.

Bemerkungen:

Hierüber haben sich Leute fast 10 Jahre gestritten. Heute ist man sich einig, daß **shared nothing** der Sieger ist.

- Keine Interferenz $\rightarrow$ sehr gute Skalierbarkeit
- Bessere Preis/Performance durch Commodity Hardware.
  (Stöpsele ein paar Pentium PCs in einem FDDI Ring zusammen.)

Architektur von PDBMS – Beispiele

shared memory: RedBrick auf 9 CPUs

shared disk: Oracle auf 32 CPUs (Sequent)

shared nothing: TeraData, Tandem, IBM, Informix Sybase auf Intel Prozessoren (bis zu 1000 CPUs derzeit)

Bemerkung 1

Oracle und RedBrick werden sich zwangsläufig immer weiter an **shared nothing** annähern.

Bemerkung 2

Es macht keinen Sinn genaue Zahlen und Hardware-Software Partnerschaften zu recherschieren. Das ändert sich täglich, also verzeiht mir meine Fehler. Ich möchte nur einen groben Eindruck vermitteln.

Partitionierung und Allokation
(Jetzt gehts los)

1. **Schritt:** Bestimmung des Verteilgrades einer Relation
   Vorsicht das ist neu und hierdurch wird auch alles Weitere anders!
2. **Schritt:** Partitionierung und Grundallokation anhand des Verteilgrades
3. **Schritt:** Replikation

(Zur Errinnerung: In "klassischen" verteilten DBMS sah es so aus.

1. Partitionierung bei gegebenen Anfragen
2. Allokation inkl. Replikation anhand eines Kostenmodells.

)

Bestimmung des Verteilgrades einer Relation

- hoher Verteilungsgrad gut, da hoher Parallelisierungsgrad (beim Scan) und gute Lastbalancierung
- zu hoher Verteilungsgrad schlecht, da hohe Start-Up Kosten und ggf. hohe Kommunikationskosten
- also machen wir ein Kostenmodell ...

Beispiel

SELECT * FROM Teile WHERE Lieferant = "Strunz"

Teile habe K Tupel. Es kostet a Einheiten pro beteiligter Knoten im Start-Up und dann b Einheiten zur Verarbeitung jedes Tupels. Minimiere für den Verteilgrad p

$$R(p) = a * p + \frac{b * K}{p}$$

Ergebnis (1. Ableitung = 0; 2. Ableitung > 0):

$$p_{opt} = \sqrt{\frac{b * K}{a}}$$

Bemerkung In der Schule habe ich gelernt: "Die Natur macht keine Sprünge." Kostenfunktionen für Datenbanksystem sind allerdings in der Regel nicht differenzierbar wegen Ceiling- und Floorfunktionen.

Partitionierung und Allokation

Noch einmal: Das ist hier ein einziger Schritt.

1. Es wird nur horizontal partitioniert!
   (Versucht mal eine Relation mit 3 Attributen in 1000 vertikale Partitionen zu teilen.)
2. Oberstes Ziel ist Lastbalancierung, Ausnutzung von Parallelität.
   Kostenminimierung durch Partitionierung spielt nur bei OLTP eine Rolle; bei großen Anfragen spielt es eine untergeordnete Rolle.
3. Es gibt folgende Verfahren:
   (a) round-robin (perfekte Lastverteilung)
   (b) hash-basiert (ein Hauch von LH*)
   (c) Bereichsfragmentierung (gut z.B. fürs Sortieren; aber riskant in Punkto Lastbalancierung)
4. **Wichtig:** Das Verteilattribut ist frei wählbar.
   (Hier darf man dann doch mal einen Blick auf typische Anfragen werfen.)
5. Ansonsten hilft wie immer viel gesunder Menschenverstand, und vergeßt nicht alles, was Ihr über verteilte Systeme gelernt habt, denn Kommunikation kann auch in parallelen Systemen zum Problem werden.

Replikation

- Netzwerkpartitionierung, Reduktion von Kommunikationskosten hier nicht wichtig
- **Aber trotzdem wichtig:** Wer viele Festplatten hat, muß auch mit vielen Ausfällen rechnen.
- Weiterer Aspekt: Replikation kann bei der Lastbalancierung helfen.

Im folgenden stellen wir drei Verfahren, die speziell für parallele DBMSe entwickelt wurden, vor. Alle Verfahren haben einen Replikationsgrad von genau 2; alle Verfahren verwenden ROWA zur Konsistenzhaltung.

1. Spiegelplatten
2. Verstreute Replikation
3. Verkettete Replikation

(Als verwandtes Thema siehe auch RAID; Patterson et al. SIGMOD 88.)

Spiegelplatten (1)

Primitive Idee:

Jede Platte wird einmal vollständig repliziert.

1. Bei Ausfall einer Platte kann man problemlos mit dem Zwilling weiterarbeiten.
2. Im Normalbetrieb kann man die Zwillinge auf verschiedene Arten (beim Lesen) ausnutzen.
   (a) Jede zweite Anfrage auf die eine Platte, jede zweite Anfrage auf die Zwillingsplatte. (Perfekte Lastbalancierung.)
   (b) Nehme die Platte, deren Lesekopf gerade am günstigsten für die entsprechende Anfrage stehte. (Geringe Seekkosten; aber evtl. schlechte Lastbalancierung.)
   (c) Mischformen.

Spiegelplatten (2)

3. Freiheitsgrad: Kopplung von Rechnern zum Plattenpaar
   (a) Tandem: Jedes Plattenpaar ist einem Hauptrechner und einem Backuprechner zugeordnet. Im Normalbetrieb werden alle Anfragen durch den Hauptrechner bearbeitet. Falls der Hauptrechner ausfällt, übernimmt der Backuprechner. (Natürlich kann der Backuprechner der Hauptrechner für ein anderes Plattenpaar sein.)
   (b) Inktomi: Rechner 1 ist für Platte 1a zuständig; Rechner 2 ist für Platte 1b (1a Zwilling) und 2a zuständig; Rechner 3 ist für 2b und 3a zuständig, usw.
4. Neben Tandem und Inktomi werden Spiegelplatten in sehr vielen DBMS eingesetzt; z.B. auch in ganzen normalen Einprozessordatenbanken.

Verstreute Replikation (Teradata)

Motivation

Bessere Lastbalancierung nach Ausfall eines Knotens.

Idee

Nach Ausfall eines Knotens werden die Aufgaben dieses Knotens von **mehreren** anderen Knoten (und nicht nur von einem anderen) übernommen. Deswegen werden die Daten eines Knotens "round-robin" (oder so) auf anderen Knoten repliziert.

- bilde Gruppen von Knoten: "alle für einen" innerhalb der Gruppe.
- falls G Knoten in der Gruppe, erhöht sich die Last jedes Knotens um einen Faktor $\frac{G}{G-1}$ bei Ausfall eines Knotens.
- Wahl der Gruppengröße: große Gruppen bewirken gute Lastbalancierung nach dem Ausfall eines Knotens. Kleine Gruppen erhöhen hingegen die Verfügbarkeit des Systems, da pro Gruppe ein Knoten ausfallen darf, ohne die Verfügbarkeit des Systems zu beeinträchtigen.
- Im Extremfall bei $G = 2$ ist man genauso schlau wie bei Spiegelplatten.

Verkettete Replikation (Gamma)

Motivation

Vereine die guten Eigenschaften von Spiegelplatten (hohe Verfügbarkeit) und verstreuter Replikation (gute Lastbalancierung nach Ausfall).

Idee

Wie verstreute Replikation nur mit etwas geschickterer Anordnung der Daten innerhalb der Gruppe, so daß ggf. sogar zwei Knoten innerhalb einer Gruppe ausfallen können, ohne die Verfügbarkeit des Systems zu beeinträchtigen.

Beispiel

Buch von Erhard Rahm Seite 335

Parallele Anfragebearbeitung

Wichtig

Die meisten relationalen Operatoren lassen sich sehr leicht automatisch parallelisieren. (D.h. parallele Datenbanksystem erzielen im Gegensatz zu verteilten Datenbanksystemen fast immer 100% *Transparenz*, und SQL mußte für parallele DBMSe nicht erweitert werden.) Vermutlich sind parallele DBMSe nur deshalb so erfolgreich.

Was wir heute machen: Wir beschränken uns auf "partitioned parallelism."

1. Parallele Scans
2. Parallele Sortierung
3. Parallele Joins
4. Einige Kommentare zu nicht-relationalen Systemen (i.e., objektrelational oder objektorientiert)

Weiterführende Literatur Götz Graefe, "Encapsulating Parallelism" SIGMOD 1990

Parallele Scans

Das ist trivial!

- Tabellen sind ja bereits partitioniert
- etabliere einen `Scan` Iterator für jede Partition. (i.e., Anzahl der Scan Iteratoren = Verteilgrad)
- pipe Ergebnisse der Scan Iteratoren in nachfolgende Iteratoren
- ggf. ist nachfolgender Iterator ein `Merge` Operator, der die Ergebnisse zu einem Tupelstrom zusammenschweißt.

Beispiel

```
SELECT * FROM Teile WHERE Lief = ''Strunz''
```

Plan (Verteilgrad = 3)

$\texttt{Merge(Scan(Teile}_0\texttt{), Scan(Teile}_1\texttt{), Scan(Teile}_2\texttt{))}$

(Hier `Merge` = UNION.)

Paralleles Sortieren (1)

Idee

- Verteile Tupel auf Prozessoren mit einem `Split` Operator.
- Jeder Prozessor sortiert die ihm zugewiesenen Tupel.
- Vereinige Tupelströme (Läufe) der einzelnen Prozessoren mit einem `Merge` Operator.

Übung: Berechne Speed-Up dieses Verfahrens in Abhängigkeit der Anzahl der Prozessoren.

Zwei Varianten

- `Split` ist hashbasiert oder round-robin; `Merge` ist genauso wie `Merge` beim Sortieren in zentralen Datenbanken.
  - Vorteil. Wenig Skew.
  - Nachteil. Relativ hohe Kosten beim Merge.
- `Split` führt range partitioning durch; `Merge` ist eine einfache Concatination.
  - Vorteil. Minimale Kosten beim Merge.
  - Nachteil. Skew, wenn man range partitioning schlecht macht.

Paralleles Sortieren (2)

Wichtige Bemerkung

`Split` und `Merge` verkapseln die Parallelität. D.h. alle anderen Operatoren können wie in einem zentralen System implementiert werden.

Parallele Joins

Strategie 1: Dynamische Replikation

- Ausgangssituation: R ist partitioniert und liegt auf Knoten $K_1, \ldots, K_n$.
- Schicke Kopien von S nach $K_1, \ldots, K_n$
  (falls S partitioniert, dann `Merge` zunächst die einzelnen Partitionen von S)
- Führe lokale Joins auf $K_1, \ldots, K_n$ aus (z.B. durch NLJ).
- `Merge` Ergebnisse durch normales UNION.

Strategie 2: Dynamische Partitionierung durch Hashing

1. Verteile Tupel von R auf Knoten durch Hashing auf die Joincolumn von R.
2. Verteile Tupel von S auf Knoten durch Hashing (mit derselben Funktion wie in 1) auf die Joincolumn von S.
3. Führe lokale Joins der Partitionen von R und S auf jedem Knoten aus (z.B. wieder hashbasiert).
4. `Merge` Ergebnisse durch normales UNION.

Parallelität in nicht-relationalen Systemen

Das ist ein ganz heißes Thema. Wird schwierig durch:

- Mengenwertige Attribute (Skew!)
- Referenzen (Parallele Pointer-Based Joins schwierig!)
- Komplexe Objekte; Arrays und so Zeug (Wiederum Skew!)
- Operationen auf Bildern und Videos (Transparenz, Spatial Partition Skew!)
- ADTs (Wiederum Transparenz!)
- **siehe Vortäge von DeWitt zu diesem Thema**
  `http://www.cs.wisc.edu/ dewitt`