Skew-tolerantes, dynamisches LPT-Scheduling zur Join-Verarbeitung in parallelen Shared-Disk-Datenbanksystemen

Holger Märtens
Institut für Informatik, Abteilung Datenbanken
Universität Leipzig

In parallelen Datenbanken, die für Decision-Support-Aufgaben wie z. B. Data Warehousing eingesetzt werden, spielen hohe Durchsatzraten, kurze Antwortzeiten und damit auch Lastbalancierungsfragen eine entscheidende Rolle. Dies gilt insbesondere für komplexe Operationen wie den relationalen Join. Das größte Problem bei seiner parallelen Ausführung sind nichtuniforme Daten- und Werteverteilungen (Skew), die nur begrenzt vorhersehbar sind und somit zur Laufzeit behandelt werden müssen. Dies ist in den verbreiteten Shared-Nothing-Rechnerarchitekturen jedoch nur schwer zu realisieren, da Datenumverteilungen mit hohem Zusatzaufwand verbunden sind. Wir schlagen daher ein dynamisches Lastbalancierungsverfahren auf Basis einer Shared-Disk-Architektur vor, welches aufgrund der uniformen Zugriffsstruktur weitaus effizienter arbeitet, als dies in Shared-Nothing-Systemen möglich ist. In einer Simulationsstudie zeigt es sich einem herkömmlichen prädiktiven Algorithmus deutlich überlegen.

1 Einleitung

Angesichts stetig steigender Datenvolumina wird auch im Datenbankbereich seit längerem Parallelverarbeitung eingesetzt. Während man sich in frühen parallelen Transaktionssystemen auf die gleichzeitige Verarbeitung mehrerer Anfragen auf jeweils einem Prozessor beschränken konnte (*Inter-Query-Parallelität*), müssen in Decision-Support-Umgebungen auch einzelne komplexe Anfragen (*Intra-Query-Parallelität*) und damit insbesondere deren einzelne Operatoren (*Intra-Operator-Parallelität*) parallelisiert werden (Abb. 1). Dies wird um so wichtiger, je größer die zu verarbeitenden Datenbestände sind und je kürzer die für den Benutzer tolerierbare Antwortzeit ist. Vor diesem Hintergrund gewinnt die Frage der *Lastbalancierung* an Bedeutung.
Die Wahl geeigneter Lastverteilungsverfahren ist unter anderem bestimmt durch die zugrundeliegende Architektur des Systems. Die klassische Unterscheidung ordnet parallele Datenbanksysteme in drei Kategorien [Ra94]:

- *Shared-Nothing*-Architekturen (SN) ordnen jedem Prozessor seinen eigenen Haupt- und Plattenspeicher zu, auf den er exklusiven Zugriff hat.
- *Shared-Disk*-Systeme (SD) verwenden ebenfalls getrennte Hauptspeicher, machen aber jede Platte allen Prozessoren zugänglich.
- *Shared-Everything*-Rechner (SE) teilen sich sowohl Platten- als auch Hauptspeicher.

Obwohl SE-Systeme offensichtlich die größte Flexibilität im Hinblick auf die Lastbalancierung besitzen, scheiden sie für sehr große Datenbanken aus, da sie nicht weit genug skalierbar sind. SD-Architekturen sind ebenfalls recht flexibel, da jeder Prozessor alle Daten verarbeiten kann, so daß die Lastverteilung im Prinzip beliebig erfolgen kann. Allerdings erfordert der gemeinsame Zugriff relativ aufwendige Synchronisationsmechanismen, um die Konsistenz der Datenbestände zu sichern. Trotz erfolgreicher kommerzieller Shared-Disk-Implementierungen (Oracle, DB2/MVS) wird diese Anforderung häufig als zu großes Hindernis angesehen, so daß vor allem im akademischen Bereich oft SN-Systeme als überlegen angesehen werden [DG92]. Diese sind einfacher konzipiert und leichter zu realisieren als SD. Wie wir im Folgenden zeigen werden,

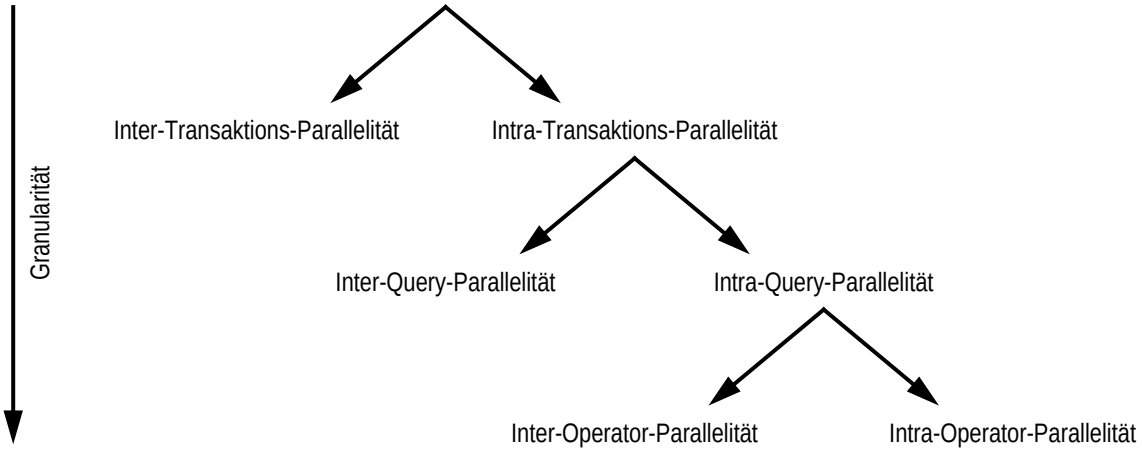
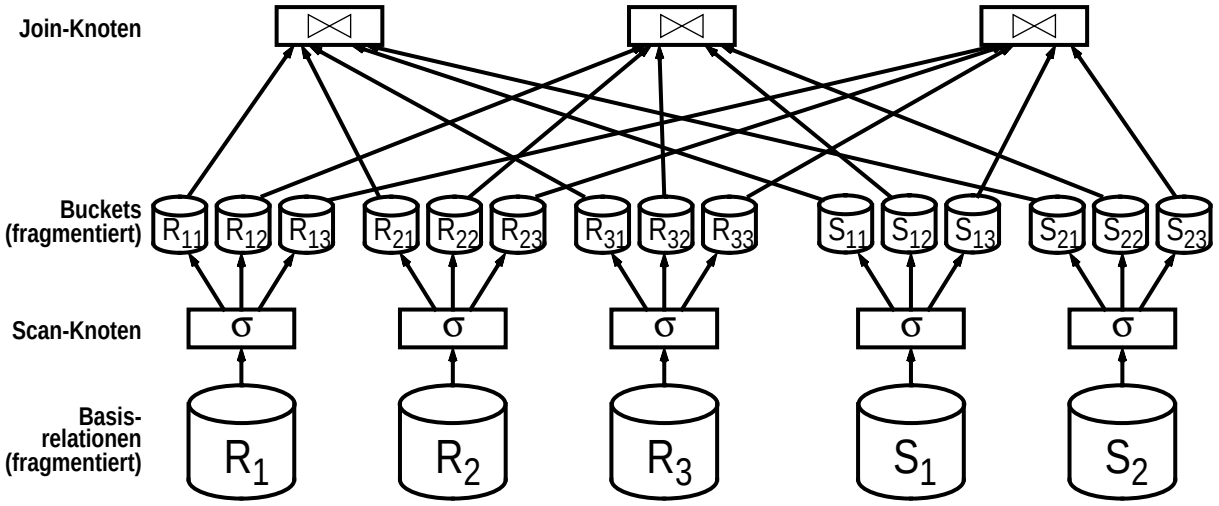


Abb. 1: Arten der Parallelverarbeitung

erschwert das SN-Paradigma jedoch die Lastbalancierung [Ra96].
Ein an der Universität Leipzig laufendes Projekt befaßt sich u. a. mit der Nutzung des speziellen Lastbalancierungspotentials von Shared-Disk-Datenbanksystemen. Eine unserer Studien, deren Ergebnisse wir in diesem Papier präsentieren, untersucht dabei die effiziente Verarbeitung des relationalen Join-Operators. Nachfolgend werden wir daher zunächst den in unserer Arbeit betrachteten parallelen Hash-Join-Operator einführen (Abschnitt 2) und die wichtigsten Aspekte bisheriger Implementierungen in SN-Systemen betrachten (3). Danach stellen wir die beiden auf der LPT-Heuristik basierenden Join-Verfahren für SD-Umgebungen vor (4), die wir in mehreren Simulationsreihen verglichen haben (5). Abschließend machen wir Vorschläge für weitere Untersuchungen (6).

2 Paralleler Hash-Join

Eine der aufwendigsten Operationen in relationalen Datenbanken ist der *Join*. Er verknüpft zwei Relationen, die ein oder mehrere Attribute gemeinsam haben. Jedes Paar von Tupeln, deren Attributwerte übereinstimmen, wird zu einem Ergebnistupel zusammengefaßt (*Equi-Join*). Da bei großen Datenmengen ein paarweiser Vergleich aller Tupel äußerst aufwendig ist, wurden Verfahren entwickelt, um den Verarbeitungsaufwand zu verringern. Das erfolgreichste dieser Verfahren ist der *Hash-Join*, dessen (nichtparallele) Basisvariante folgendermaßen arbeitet: Zunächst wird aus der sogenannten *inneren Relation* (i. a. der kleineren) eine Hash-Tabelle konstruiert; dabei wird die Hash-Funktion auf das Join-Attribut angewandt. Anschließend wird die zweite, *äußere Relation* tupelweise gegen die Hash-Tabelle getestet. Die Arbeitsersparnis beruht darauf, daß ein Tupel der äußeren Relation nur mit jenen der inneren Relation verglichen werden muß, die der gleichen Hash-Klasse angehören; denn nur diese können auch die gleichen Attributwerte besitzen.
Der *parallele Hash-Join* erweitert dieses Prinzip auf die Aufteilung der Arbeit unter mehreren Prozessoren (Abb. 2). Mit einer weiteren, oft ebenfalls hash-basierten Partitionierungsfunktion werden zunächst beide Eingaberelationen in sogenannte *Buckets* zerlegt. Da für beide Relationen dieselbe Funktion verwendet wird, ist es ausreichend, die Buckets paarweise zu verknüpfen, um das gesamte Join-Ergebnis zu erhalten. Folglich werden die Bucket-Paare möglichst gleichmäßig auf die verfügbaren Rechner verteilt und dort mit dem Basisverfahren verarbeitet (*lokaler Join*); anschließend werden ihre Ergebnismengen vereinigt.



Beim parallelen Hash-Join werden die Basisrelationen in Buckets zerlegt. In einer SD-Architektur können die Fragmente der Buckets auf jeder beliebigen Platte gespeichert und von allen Prozessoren gelesen werden, was die Flexibilität des Scheduling gegenüber SN deutlich erhöht.

Abb. 2: Paralleler Hash-Join

2.1 Probleme

Probleme bei der Ausführung des parallelen Hash-Joins erwachsen aus der ungleichen Verteilung von Daten, Attributwerten und Arbeitslast, zusammenfassend als *Skew* bezeichnet [WD91]. Dieses Phänomen ist zum Teil dateninhärent, z. B. als *Attributwert-Skew* (*attribute value skew*, AVS). AVS beschreibt die unterschiedliche Häufigkeit verschiedener Attributwerte innerhalb der Relationen; diese führt selbst bei Verwendung einer uniformen Hash-Funktion zu unterschiedlich großen Buckets. Doch auch bei gleich großen Buckets (die z. B. mit einer geeignet korrigierten Hash-Funktion erreichbar wären), können die Join-Resultate der einzelnen Bukkets aufgrund schiefer Werteverteilungen unterschiedlich groß sein (*Join-Produkt-Skew*).
Aber nicht nur die relative, sondern auch die absolute Größe der Buckets ist zu beachten. Ist bereits der kleinere Bucket eines Paares sehr groß, paßt die Hash-Tabelle nicht vollständig in den Hauptspeicher eines Prozessors. Dadurch wird eine Überlaufbehandlung erforderlich, die i. a. das mehrfache Lesen und Schreiben ein und derselben Datensätze mit sich bringt und daher überdurchschnittlich teuer ist. Zu kleine Buckets nutzen nur einen Teil des verfügbaren Speichers aus, was vor allem im Single-User-Betrieb als nachteilig angesehen wird, da jeder lokale Join einen gewissen, weitgehend konstanten Overhead mit sich bringt (Initialisierung und Ende der Sub-Query, Plattenkopfpositionierung zum Lesen des Buckets etc.). Die Summe dieser Overheads erhöht sich, wenn kleinere und dafür zahlreichere Buckets verarbeitet werden müssen.
Die Berücksichtigung all dieser Faktoren zur Optimierung der Join-Verarbeitung ist nicht praktikabel, da hierfür eine exakte Kenntnis der Werteverteilungen notwendig wäre; diese ist aber i. a. nicht gegeben, insbesondere wenn der Join auf den Resultaten untergeordneter Queries arbeitet. Und selbst wenn perfektes Wissen vorhanden wäre und effizient verarbeitet werden könnte, würden die besten Parallelisierungspläne scheitern, wenn im Mehrbenutzerbetrieb – der für moderne Datenbanken unverzichtbar ist – weitere Queries unvorhergesehen in das System eintreten und das Lastgleichgewicht unter den Prozessoren stören. Somit sind flexiblere, robustere Lastbalancierungsansätze gefragt.

3 Frühere Ansätze zur Join-Lastbalancierung

Da sich der größte Teil der "*database community*" frühzeitig auf Shared-Nothing als vermeint-

lich beste Architektur für parallele Datenbanken festgelegt hat [DG92], sind praktisch alle publizierten parallelen Join-Verfahren auf SN zugeschnitten. Diese Grundannahme wirkt sich massiv auf die Struktur der entwickelten Lösungen aus. Bei sehr großen Join-Anfragen (die für uns von besonderem Interesse sind) müssen die meisten Buckets nach der Partitionierung auf die Plattenspeicher zurückgeschrieben werden, da der Hauptspeicher nicht alle Daten fassen kann. Dabei ist in SN-Systemen die Wahl der richtigen Platte von entscheidender Bedeutung, da mit dem Speicherungsort eines Buckets auch der Ort seiner Verarbeitung bestimmt ist. Falls ein Bucket später auf einem anderen Knoten verarbeitet werden muß als dem, an dem er gespeichert wurde, wird eine kostenträchtige Umverteilung notwendig.
Aus diesem Grunde basieren die meisten parallelen Join-Verfahren auf einem *prädiktiven Scheduling*-Ansatz, bei dem bereits vor der Verarbeitung, oft sogar noch vor der Bucket-Aufteilung entschieden wird, welcher Rechner welchen Teil der Arbeit übernimmt. Die Lastverteilung beruht dabei auf Kostenschätzungen, die jedem Bucket den erwarteten Arbeitsaufwand zuordnen. Diesen Schätzungen wiederum liegen Statistiken über die Attributwertverteilungen in den Eingaberelationen zugrunde.
Es wurden große Anstrengungen unternommen, um diese Art des Join-Scheduling zu optimieren. Diese reichen von *optimalen Histogrammen* [PI96] über *Sampling*-Techniken zur Abschätzung von Werteverteilungen [DN92] bis zu *hierarchischem Hashing* zur optimalen Bucket-Aufteilung [WY93]. Dabei werden einerseits zahlreiche kleine Buckets gebildet, um ein feines Granulat zur Lastverteilung zu erhalten; andererseits werden diese im Rahmen von *Bucket-Tuning*-Strategien zu größeren zusammengefaßt, um die Speicherausnutzung zu verbessern und den Overhead für Sub-Queries zu minimieren [HL91]. Für die Zuordnung von Buckets zu Prozessoren wird meist die einfache LPT-Heuristik (*longest processing time first*) verwendet mit Bezug auf die erwähnte Abschätzung der Verarbeitungskosten pro Bucket. Diese bietet bekanntermaßen eine gute Lösungsqualität [Gr69] bei geringer Zeitkomplexität ($O(n \cdot \log n)$ für die Sortierung der Bucketliste).
Wie bereits in Abschnitt 2.1 angedeutet, ergeben sich bei prädiktivem Scheduling zwei große Probleme:

- Zum einen sind die verwendeten Kostenabschätzungen notorisch schlecht. Sie beruhen meist nur auf der Größe der Buckets, über deren Inhalte zu wenig Information vorliegt; Join-Produkt-Skew kann so nicht berücksichtigt werden. Genauere Schätzungen erfordern einen hohen Aufwand für die Sammlung und Fortschreibung von Statistiken.
- Zum anderen wären selbst perfekte Kosteninformationen und die daraus abgeleiteten Ausführungspläne im Multi-User-Betrieb wertlos, wenn eine optimierte Anfrage während der tatsächlichen Verarbeitung mit anderen, inzwischen hinzugekommenen Queries um Ressourcen konkurrieren muß.

Beide Effekte führen zu *Ausführungs-Skew* und damit zu suboptimalen Antwortzeiten. Während – wie beschrieben – großer Aufwand getrieben wurde, um die Kostenschätzungen zu verbessern, wurden die Probleme des Mehrbenutzerbetriebes in nahezu allen bisherigen Studien vernachlässigt.
Prädiktives Scheduling ist also grundsätzlich als suboptimal anzusehen. Es liegt daher nahe, eine Art von *reaktivem Scheduling* zu entwickeln, das entweder auf die Vorausplanung der Verarbeitung verzichtet oder einen erstellten Plan im Verlauf der Abarbeitung anpaßt. So wurden Verfahren entwickelt, die den Fortschritt der Join-Verarbeitung überwachen und bei signifikanten Abweichungen vom Plan eine *Lastumverteilung* vornehmen [HK95]. Dabei werden z. B. Ergebnistupel auf entfernte Knoten transferiert und dort ausgeschrieben, um die lokalen Platten von Zugriffen zu entlasten (*result redistribution*), oder es werden Teile der Eingaberelationen übertragen und komplett auf dem Zielknoten verarbeitet (*process task migration*). In einem anderen Ansatz [DH94] wird die Verarbeitung gestoppt, sobald der erste Knoten fertig ist, und die verbliebene Arbeit wird neu verteilt. Dabei kommt ein modifiziertes *Weighted-LPT*-Verfahren

zum Einsatz, das die unterschiedliche Verarbeitungsleistung der beteiligten Rechner berücksichtigt.
Der unserer Ansicht nach erfolgversprechendste Ansatz stammt von Zhou und Orlowska [ZO93, ZO95]. Sie verzichten auf die Erstellung eines prädiktiven Plans und ordnen statt dessen jedem Rechner zunächst nur einen Bucket zu. Ist dieser verarbeitet, erhält der Rechner den nächsten Bucket usw. Da die Verteilung in absteigender Reihenfolge nach den geschätzten Kosten stattfindet, stellt auch diese Methode ein LPT-Verfahren dar. Es arbeitet jedoch dynamisch und kann damit auf Ausfürungs-Skew reagieren. Da es nur die Reihenfolge, nicht aber die tatsächliche Höhe der voraussichtlichen Kosten auswertet, ist es zudem weitgehend robust gegenüber Fehlern in der Kostenfunktion.
Es ist zu beachten, daß auch die letztgenannten Verfahren auf Shared-Nothing-Architekturen entwickelt wurden. Sie leiden damit unter den oben erwähnten Zusatzkosten für die Umverteilung der Last zwischen den Rechnerknoten. Diese Kosten umfassen sowohl CPU-Aufwand für die Analyse der Lastsituation und die Koordination der Balancierungsmaßnahmen als auch Netzwerkbelastungen durch den eigentlichen Datenaustausch. Gegebenenfalls werden auch die Platten für die Zwischenspeicherung der umverteilten Daten in Anspruch genommen, so daß letztlich alle Systemkomponenten betroffen sind.
Dasselbe Problem besteht – wenn auch in abgeschwächter Form – für *Distributed-Shared-Memory*-Systeme. Für diese wurde ein dynamischer Algorithmus vorgelegt, der durch die Hardware-Unterstützung beim gemeinsamen Hauptspeicherzugriff zumindest den Lastumverteilungsaufwand verringert [BF96]. Das einzige bekannte dynamische Spezialverfahren für Shared-Disk-Architekturen [LT92] benötigt einen globalen erweiterten Hauptspeicher, der nicht nur zusätzliche Kosten verursacht, sondern auch leicht zum Engpaß wird [Ra94].

4 LPT-Scheduling-Verfahren für Shared-Disk-Systeme

Die in Abschnitt 3 angeführten Lösungen für die parallele Join-Verarbeitung sind als unbefriedigend anzusehen, weil sie entweder nur eine unzureichende Lastbalancierung bieten oder aber die Balancierung mit hohen Umverteilungskosten erkaufen. Da dieses Problem auf die jeweils zugrundeliegende Shared-Nothing-Architektur zurückzuführen ist, haben wir ein Verfahren entwickelt, das sich die Eigenschaften einer Shared-Disk-Umgebung zunutze macht. Es ähnelt dem Ansatz von Zhou und Orlowska, kommt aber ohne explizite Datenumverteilung aus, weil im SD-System jeder Prozessor direkt auf alle Daten zugreifen kann. Als Vergleichsbasis verwenden wir einen prädiktiven Algorithmus, der ebenfalls an die SD-Architektur angepaßt wurde. Da beide Verfahren – auch aus Gründen der Vergleichbarkeit – bis auf das eigentliche Join-Scheduling identisch arbeiten, beschreiben wir im Folgenden zunächst das gemeinsame Verarbeitungsmodell, bevor wir die Unterschiede zwischen den Algorithmen darlegen:
Wenn eine neue Join-Anfrage an einem beliebigen Rechnerknoten ankommt, wird dieser zum Koordinator für diese Query erklärt; er ist von nun an für die Steuerung ihrer Verarbeitung zuständig. Der Koordinator ermittelt zunächst die für die Verarbeitung notwendigen Parallelitätsgrade sowohl für den (dem Join zugrundeliegenden) Scan der Basisrelationen als auch für den Join selbst. Diese Berechnung basiert auf dem jeweils maximalen Datendurchsatz für Platten und Prozessoren und soll die Bearbeitungszeit minimieren, ohne unnötig hohe Parallelitätsgrade zu wählen. Außerdem bestimmt der Koordinator, der über grobe Informationen über die Attributwertverteilung in den Relationen verfügt, die Anzahl der Buckets so, daß jedes Bucket-Paar mit Sicherheit ohne Überlaufbehandlung verarbeitet werden kann (d. h. daß aus dem jeweils kleineren Bucket eine Hash-Tabelle im Hauptspeicher erstellt werden kann)[1]. Für die

1. Bei starkem Attributwert-Skew ist eine solche Aufteilung u. U. unmöglich; sehr große Buckets werden dann gesondert behandelt. Diese Situation soll jedoch hier nicht weiter erörtert werden.

Ausführung des Scans bzw. Joins werden jeweils die am wenigsten belasteten Prozessorknoten ausgewählt. Der Scan wird auf beiden Relationen simultan abgearbeitet und die Buckets auf die Platten zurückgeschrieben.
Nun folgt die eigentliche Join-Verarbeitung. Der herkömmliche, prädiktive Algorithmus (*predictive scheduling*, PS) erstellt zunächst eine Liste, in der die Buckets absteigend nach der verwendeten Kostenfunktion sortiert sind. Dann ordnet er die Buckets nach der LPT-Heuristik den beteiligten Prozessoren zu und übermittelt jedem Knoten eine Liste von Arbeitsaufträgen. Wenn ein Rechner seine Buckets verarbeitet und die Ergebnisse abermals auf die Platte ausgeschrieben hat, meldet er dem Koordinator die Fertigstellung; dieser beendet die Query, sobald alle Erfolgsmeldungen eingetroffen sind.
Nach unserem neuen, bedarfsorientierten Verfahren (*on-demand scheduling*, ODS) werden die Buckets ebenfalls entsprechend den Kostenschätzungen geordnet. Der Koordinator vergibt jedoch zunächst nur einen Bucket an jeden Rechner. Ist dieser verarbeitet und das Resultat ausgeschrieben, fordert der Prozessor beim Koordinator einen weiteren Bucket an, der ihm vom Kopf der Liste zugewiesen wird. Wenn alle Buckets verarbeitet sind, beendet der Koordinator die Query.
Die Funktion zur Kostenabschätzung lautet:

$$c(i) = |R_i| \cdot |S_i| \cdot \left(1 + \frac{|R_i|}{|R|} \cdot n\right) \cdot \left(1 + \frac{|S_i|}{|S|} \cdot n\right).$$

Dabei sind R und S die Eingaberelationen, R_i und S_i die jeweils i-ten Buckets und n die Anzahl der Buckets. Dies ist die beste Näherung, die in der Studie von Zhou und Orlowska gefunden wurde [ZO95]. Da PS eine möglichst gute Abschätzung benötigt, während ODS lediglich die Reihenfolge der Buckets nutzt, stellt die Wahl einer guten Kostenfunktion (die in der Praxis nicht immer zur Verfügung steht) einen Vorteil für das prädiktive Verfahren dar.

5 Simulationsstudie

Um die beiden in Abschnitt 4 beschriebenen Join-Scheduling-Algorithmen miteinander zu vergleichen, haben wir ein *Simulationssystem* entwickelt, in dem insbesondere die Eigenheiten des Mehrbenutzerbetriebes detailliert modelliert sind. So hängt z. B. die Plattenzugriffszeit von den unterschiedlichen Speicherungsorten der verschiedenen Datenfragmente ab. Auch die Konkurrenz unter Queries bezüglich Platten und Prozessoren wurde berücksichtigt. Der Attributwert-Skew in den am Join beteiligten Relationen wurde explizit modelliert und folgt der allgemein als realistisch anerkannten sogenannten *Zipf-Verteilung*. Sie beziffert die Häufigkeit des i-ten Attributwertes[1] als

$$h(i) = \frac{|R|}{i^s \cdot \sum_{j=1}^{|D|} \frac{1}{j^s}}.$$

Dabei ist D die Domäne des Attributes und $s \geq 0$ der Parameter, der das Ausmaß des Skew beschreibt. Für $s = 0$ besteht eine Gleichverteilung, $s = 1$ wird als schwerer Skew angesehen. Die wichtigsten Simulationsparameter sind in Tabelle 1 aufgeführt.

5.1 Single Skew

In unseren ersten Simulationsreihen haben wir Fälle untersucht, in denen nur eine der beiden Eingaberelationen von Skew betroffen ist (*single skew*). Abb. 3 zeigt die durchschnittlichen Verarbeitungszeiten für den Join der Relationen A und B bei variiertem Skew auf B. Dabei wur-

1. Gemeint ist der i-te Wert in der Reihenfolge der Häufigkeit. Ein Zusammenhang zwischen dem Wert selbst und der Häufigkeit seines Auftretens wird nicht vorausgesetzt.

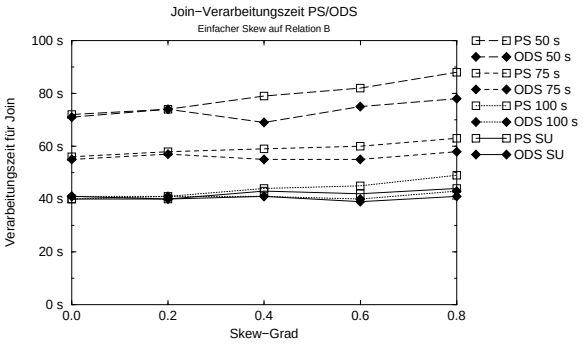


Abb. 3: Einfacher Skew

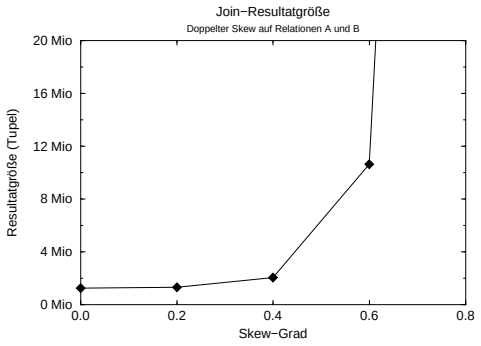


Abb. 4: Join-Resultat bei doppeltem Skew

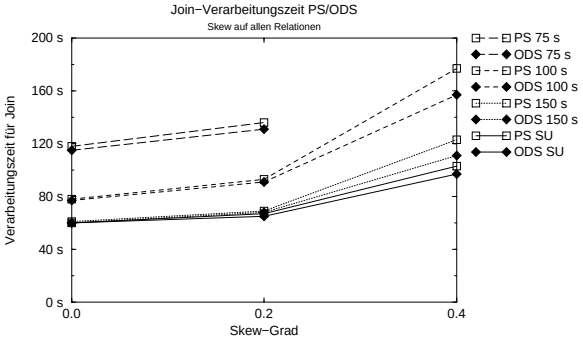


Abb. 5: Geringer doppelter Skew

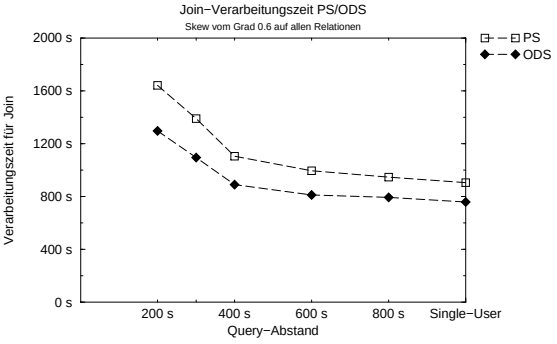


Abb. 6: Hoher doppelter Skew

Relationen		***A***	***B***	***C***
Anzahl Tupel		500 000	1 000 000	2 000 000
Tupelgröße (Bytes)		200	200	200
Gesamtgröße (MBytes)		100	200	400
Verteilungsgrad		10	20	30
Domänengröße		100 000	100 000	200 000
Queries		**Scan**	**Join**	
Selektivität (Prozent)		50	100	
Ausgabe-Tupelgröße (Bytes)		200	300	
Prozessoren		**Platten**		
Anzahl	10	Anzahl		30
Leistung (MIPS)	20	∅ Zugriffszeit (ms)		14
Hauptspeicher (MBytes)	32	Prefetching-Faktor		8

Tabelle 1: Simulationsparameter

den jeweils 100 gleiche Anfragen mit unterschiedlichen Ankunftsraten ausgeführt.
Als Referenz dient der Single-User-Fall (SU), bei dem die Queries strikt sequentiell ausgeführt werden. Da keine Konkurrenz zwischen Queries auftritt, werden hier einerseits die kürzesten Zeiten erzielt; andererseits ist auch der Unterschied zwischen beiden Verfahren am geringsten, da das prädiktive Scheduling nicht unter Ausführungs-Skew leitet.
Im Mehrbenutzerbetrieb nehmen die Konflikte zwischen Queries mit der Ankunftsrate zu. Der größte Unterschied zwischen PS und ODS tritt auf bei einem Abstand von 50 s zwischen je zwei

Queries und einem Skew von 0,4 bzw. 0,8. Die Differenz ist aber mit ca. 14 % (bezogen auf die Dauer der Joinphase bei ODS) noch moderat.
Für einfachen Skew auf Relation *A* ergeben sich vergleichbare Resultate, die hier jedoch aus Platzgründen nicht aufgeführt sind; der Unterschied beträgt dort bis zu 13 %.

5.2 Double Skew

In einer weiteren Testreihe betrachten wir das gleichzeitige Auftreten von Skew in beiden Eingaberelationen des Joins (*double skew*). Wir unterstellen dabei eine *strikte Korrelation* zwischen den Werteverteilungen der Relationen, d. h. die jeweils *i*-ten Werte auf beiden Seiten (in der Reihenfolge der Häufigkeit) stimmen überein. Dies führt zu den größtmöglichen Join-Resultaten sowie zum maximalen Join-Produkt-Skew und verschärft somit die Problematik des Scheduling. Darüber hinaus variieren wir nun die Queries, indem wir aus allen drei Relationen jeweils zwei zufällig als Join-Eingaben auswählen. Wie im vorigen Experiment betrachten wir die durchschnittliche Bearbeitungszeit aus einer Folge von 100 Queries.
Da der Umfang des Join-Produkts – wie in Abb. 4 zu sehen – sehr stark mit dem Skew-Grad steigt, präsentieren wir die Resultate in zwei getrennten Darstellungen. Abb. 5 zeigt die Verarbeitungszeiten für einen Skew bis 0,4. Der Unterschied zwischen beiden Verfahren beträgt bis zu 13 %. Für einen Skew-Grad von 0,6 (Abb. 6) wächst die Differenz auf bis zu 27 %. Bereits im Single-User-Fall liegt sie bei 19 %.

5.3 Interpretation

Diese und weitere Experimente [Mä98] zeigen, daß bedarfsorientiertes Scheduling dem herkömmlichen prädiktiven Verfahren um bis zu 27 % überlegen ist. Derartig große Unterschiede treten insbesondere bei starkem Skew in beiden Eingaberelationen und hohen Query-Ankunftsraten auf. Bei niedrigen Raten und/oder geringem Skew sind die Antwortzeiten für ODS und PS in etwa gleich.
Hieraus läßt sich folgern, daß die von Zhou und Orlowska vorgeschlagene Kostenfunktion realistisch genug ist, um im Einbenutzerbetrieb bzw. bei gleichmäßiger Attributwertverteilung eine gute Lastbalancierung zu ermöglichen. Die von beiden Verfahren erstellten Verarbeitungspläne sind dann häufig identisch. Erst bei hohem Skew wird die Kostenabschätzung so ungenau, daß sie für etwa 70 % des Performance-Unterschiedes (19 von 27 Prozentpunkten) verantwortlich ist. Berücksichtigt man, daß es sich hierbei um die beste bisher gefundene Formel handelt (die zudem speziell auf Zipf-Verteilungen zugeschnitten ist), so ist zu befürchten, daß die Leistung von prädiktivem Scheduling weiter nachläßt bei Verwendung realer Daten, für die weniger genaue Schätzungen vorliegen bzw. deren Werteverteilungen weniger regelmäßig und vorhersehbar sind.
Die verbleibenden 30 % des Leistungsunterschiedes sind auf die mangelnde Flexibilität von PS bei Ausführungs-Skew im Mehrbenutzerbetrieb zurückzuführen. Auch unter diesem Aspekt erwarten wir ein noch größeres Gefälle, wenn das System realistischere, weniger gleichförmige Lasten zu verarbeiten hat. Kleinere Queries oder gar schreibende Transaktionen mit daraus folgenden Sperrkonflikten können die Lastverhältnisse im System noch schneller und unvorhersehbarer schwanken lassen, was zu noch größeren Fluktuationen in den Ausführungszeiten führen kann.
Alles in allem ist davon auszugehen, daß die von uns gemessenen Werte eher eine untere Schranke für das Leistungsgefälle zwischen bedarfsorientiertem und prädiktivem Join-Scheduling darstellen.

6 Fazit

Im Rahmen eines Projekts zur dynamischen Lastbalancierung in parallelen Datenbanksystemen

haben wir prädiktives und bedarfsorientiertes Join-Scheduling in Shared-Disk-Umgebungen verglichen. Unser ODS-Algorithmus erwies sich dabei als robuster bezüglich Fehlern in der Kostenfunktion und flexibler im Hinblick auf Schwankungen in den Verarbeitungszeiten für Teilanfragen. Dieser Erfolg belegt auch das höhere Lastbalancierungspotential von SD-Architekturen im Vergleich zu SN-Systemen, das in der uniformen Zugriffsstruktur von SD begründet ist.

Trotz der ermutigenden Ergebnisse ist noch eine Reihe offener Fragen zu bearbeiten. So konnte unsere Studie erst einen kleinen Ausschnitt aus dem breiten Spektrum möglicher Arbeitslasten untersuchen. Leider steht uns zur Zeit kein paralleles Datenbanksystem zur Verfügung, mit dem ODS unter realen Bedingungen getestet werden könnte, so daß wir bis auf weiteres auf Simulationen angewiesen sind. Noch zu beurteilen ist beispielsweise die Leistung unseres Verfahrens unter Mischlasten mit einem erhöhten Anteil an kurzen Transaktionen und einer größeren Zahl von Zugriffskonflikten. Interessant wäre auch die Übertragung auf komplexere Hardware-Strukturen wie hybride Systeme oder NUMA-Architekturen. Zu guter Letzt sollte auch die Anwendung des ODS-Ansatzes auf andere Anfragetypen wie Scan-, Sortier- oder Aggregatoperatoren geprüft werden.

Literatur

[BF96] L. Bouganim, D. Florescu, P. Valduriez: *Dynamic Load Balancing in Hierarchical Parallel Database Systems*. Proc. 22nd VLDB, Bombay, 1996.

[DG92] D. J. DeWitt, J. Gray: *Parallel database systems: the future of high performance database systems*. Communications of the ACM 35 (6), 1992.

[DH94] H. M. Dewan, M. Hernández, K. W. Mok, S. J. Stolfo: *Predictive Dynamic Load Balancing of Parallel Hash-Joins over Heterogeneous Processors in the Presence of Data Skew*. Proc. 3rd PDIS, Austin, 1994.

[DN92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, S. Seshadri: *Practical Skew Handling in Parallel Joins*. Proc. 18th VLDB, Vancouver, 1992.

[Gr69] R. L. Graham: *Bounds on Multiprocessing Timing Anomalies*. SIAM Journal of Applied Mathematics 17 (3), 1969.

[HK95] L. Harada, M. Kitsuregawa: *Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems*. Proc. DASFAA '95, Singapur, 1995.

[HL91] K. A. Hua, C. Lee: *Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning*. Proc. 17th VLDB, Barcelona, 1991.

[LT92] H. Lu, K.-L. Tan: Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems. Proc. 3rd EDBT, Wien, 1992.

[Mä98] H. Märtens: *Skew-Insensitive Join Processing in Shared-Disk Database Systems*. Proc. IADT '98, Berlin, 1998.

[PI96] V. Poosala, Y. E. Ioannidis: *Estimation of Query-Result Distribution and Its Application in Parallel-Join Load Balancing*. Proc. 22nd VLDB, Bombay, 1996.

[Ra94] E. Rahm: *Mehrrechner-Datenbanksysteme – Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, Bonn, 1994.

[Ra96] E. Rahm: *Dynamic Load Balancing in Parallel Database Systems*. Proc. EURO-PAR 96, Lyon, 1996.

[WD91] C. B. Walton, A. G. Dale, R. M. Jenevein: *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins*. Proc. 17th VLDB, Barcelona, 1991.

[WY93] J. L. Wolf, P. S. Yu, J. Turek, D. M. Dias: *A Parallel Hash Join Algorithm for Managing Data Skew*. IEEE Transactions on Parallel and Distributed Systems 4(12), 1993.

[ZO93] X. Zhou, M. E. Orlowska: *A Dynamic Approach for Handling Data Skew Problems in Parallel Hash Join Computation*. Proc. IEEE TENCON '93, Peking, 1993.

[ZO95] X. Zhou, M. E. Orlowska: *Handling Data Skew in Parallel Hash Join Computation Using Two-Phase Scheduling*. Proc. ICA^{3}PP-95, Brisbane, 1995.