Parallelisierung einer Query Engine

Thomas Blum

19. Dezember 2017

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Fragestellung, wie man mittels Parallelisierung Datenbanksuchen in einer Query Engine effizienter gestalten kann. Sie gibt außerdem einen Überblick über die Evolution in diesem Bereich, da mit *Volcano* und *Morsel-driven parallelism* zwei Konzepte zur Veranschaulichung gewählt wurden, die die Entstehungszeit dieses Forschungsgebietes und den gegenwärtigen Stand der Forschung repräsentieren. Anschließend werden Implementierungen dieser Konzepte vorgestellt und eine Evaluation dieser im Vergleich vorgenommen.

1 Einleitung

1.1 Bedeutung der Parallelisierung in Zeiten von Big Data

Ein zentrales Problem in Zeiten von Big Data ist die Handhabung derart großer Datenmengen verbunden mit großem Speicherbedarf und hohen Zugriffszeiten. Die Entwicklung der Hardware schreitet nicht schnell genug voran, um diese Herausforderung zu kompensieren. Des Weiteren wird allmählich ein Punkt erreicht, an dem die reine Hardwareoptimierung zumindest im Bereich der Digitalrechner an die Grenzen des physisch Machbaren stößt. Darum wird es immer wichtiger die Optimierung der Software in den Mittelpunkt zu stellen. Einen großen Ansatzpunkt bietet hierbei die Parallelisierung. Mithilfe von parallel geschalteten Prozessorkernen können Prozesse und ihre Threads nebenläufig ausgeführt werden. Die zentrale Aufgabe besteht nun darin, die Möglichkeiten, die sich hieraus ergeben, vollständig auszunutzen, um eine möglichst hohe Leistungssteigerung der Software zu erzielen.

1.2 Notwendigkeit einer Effizienzsteigerung im Bereich der Datenbanken

Neben den vielen anderen Gebieten der Informationstechnik, auf die Big Data massive Auswirkung hat, rückt ein Bereich besonders in den Vordergrund. Die Rede ist von der Datenhaltung, welche von Datenbanken übernommen wird. Um Daten mittels einer Datenbanksuchanfrage zu finden, müssen sämtliche Einträge einer Tabelle anhand unterschiedlicher Methoden überprüft werden. Je mehr Einträge eine Datenbanktabelle aufweist, desto komplexer ist es einen von ihnen zu finden. Da derartige Suchanfragen überaus häufig gestellt werden, ist es sinnvoll deren Bearbeitung mittels Parallelisierung zu optimieren. Im Folgenden wird auf diese Parallelisierung näher eingegangen und es werden Ausarbeitungen präsentiert, die diese Ansätze umsetzen.

2 Grundlagen

2.1 Formen der Parallelisierung

In Datenbankmanagementsystemen sind Query Engines auf drei verschiedene Weisen parallelisiert. Man unterteilt grob zwischen *inter-operator* und *intra-operator* oder auch *partitioned* parallelism, wobei man bei *inter-operator-parallelism* weiter zwischen *independent* und *pipelined* unterscheiden kann. Bei *independent inter-operator parallelism* werden zwei Operatoren unabhängig voneinander nebenläufig ausgeführt. Wohingegen bei pipelined *inter-operator parallelism* eine Producer-Comsumer Beziehung existiert, welche sich dadurch kennzeichnet, dass ein Consumer Operator parallel zu einem Producer Operator in unterschiedlichen Prozessen läuft und der Producer Operator mit den Datensätzen arbeitet, die der Consumer Operator für ihn produziert. Spricht man von *partitioned parallelism*, werden mehrere Prozessorkerne benutzt um einen Operator auszuführen, indem die Eingabedaten in Pakete partitioniert werden und eine Kopie des Operator auf jedem dieser Prozessorkerne auf eines dieser Pakete angewendet wird.[1]

2.2 Operator Tree

Ob eine Suchanfrage nach einer von diesen Methoden parallelisierbar ist, lässt sich anhand des Operatorbaums erkennen. Er ist ein Tupel bestehend aus den Knoten und Kanten. Die Knoten repräsentieren die Operatoren, während die Kanten den Datenfluss charakterisieren. Ein Knoten nimmt einen oder mehrere Eingabedatensätze und produziert daraus einen Ausgabedatensatz. Es lassen sich zwei Arten von Kanten unterscheiden. Eine Pipelining Kante tritt zwischen zwei Operatoren auf, wenn der eine Operator warten muss bis der andere Operator seinen Ausgabedatensatz vollständig berechnet hat. Ob eine Datenbankanfrage parallelisierbar ist, lässt sich nun mit einem Blick auf den Operatorbaum feststellen. *pipelined inter-operator parallelism* kann zwischen zwei Operatoren angewendet werden, wenn zwischen den zugehörigen Knoten eine Pipelining Kante existiert. Wann immer ein Operator auf mehrere Eingabedatensätze zugreift, kann *independent inter-operator parallelism* angewendet und die Unterbäume des dazugehörigen Knotens nebenläufig bearbeitet werden. *Intra-operator parallelism* wird gegebenfalls auf jeden Operator angewendet.[1]

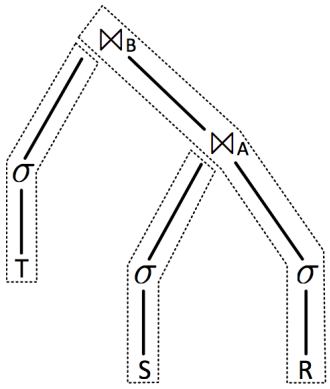

Abbildung 1: Quelle: [1]

2.3 NUMA

Besitzt ein System mehrere Prozessoren, liegt häufig eine *non-uniform memory architecture* (NUMA) zugrunde. Bestehende Prozessoren haben häufig schon diese NUMA, da die Kerne auf einem Prozessor sich einen gemeinsamen Cache teilen. Um die Systemspeicherbandbreite zu erhöhen, werden in modernen Prozessoren die Speicher in den Chip integriert und an einen sogenannten *socket* gebunden, der aus einer Gruppe von Prozessorkernen besteht. Dies wird durch Grafik 2 ersichtlich. So wird der physikalische Adressraum unter den Sockets aufgeteilt und jeder Kern hat einen lokalen Speicherbereich zur Verfügung, den er über den On-Chip Memory Controller des Sockets, dem er zugehört, erreichen kann. Nun muss aber in Shared Memory Multiprocessors jeder Prozessorkern eines Sockets fähig sein auf den lokalen Speicher eines anderen Sockets zuzugreifen. Diese Speicherzugriffe werden *remote memory accesses* genannt und finden auf den Verbindungen zwischen den Sockets statt. Muss nun ein solcher Speicherzugriff getätigt werden resultieren daraus mehrere Nachteile im Gegensatz zu einem Zugriff auf den lokalen Speicherbereich. Es kommt zu einer Verzögerung um bis zu einem Faktor 2 und zu einer reduzierten Bandbreite. Eine gute Datenlokalität ist also sehr wünschenswert und kann nur gewährleistet werden, indem Memory Management und Process Mapping miteinander kombiniert werden.[2]

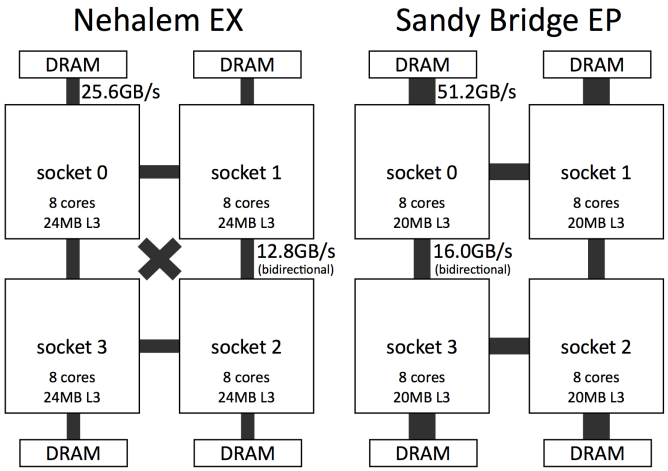

Figure 10: NUMA topologies, theoretical bandwidth

Abbildung 2: Quelle: [4]

3 Ansätze

Im Laufe der Zeit wurden immer wieder neue Ansätze veröffentlicht, welche versuchten die aus der Parallelisierung hervorgehenden Möglichkeiten möglichst effizient auszunutzen. Bereits 1987 wurde mit *Volcano* ein Konzept vorgestellt, welches alle Formen der Parallelisierung umsetzt. Im Zeitalter von Big Data und Hauptspeicherdatenbanken kommt der Parallelisierung allerdings eine größere Rolle zu, was die Entwicklung von noch effizienteren Konzepten vorantreibt. Dies führte unter anderem zur Entwicklung des Konzepts *Morsel-driven parallelism*. Im Folgenden wird ein Überblick über beide Konzepte geschaffen.

3.1 Volcano

Das Parallelisierungskonzept Volcano stammt aus der Anfangszeit dieses Forschungsgebietes und vereint bereits alle Formen der Parallelisierung. Es besitzt eine große Skalierbarkeit, da es alle Parallelisierungsaufgaben auf den sogenannten exchange operator bündelt.

3.1.1 Operatoren

Alle Operatoren sind nach dem Iterator-Konzept implementiert, was das *open-next-close Protokoll* erfüllt. Dies ermöglicht die iterative Bearbeitung eines beliebigen Datensatzes im Vorbild von Streams der konventionellen Filescans. Verbunden mit jedem Operator ist ein State Record, welcher alle Status-Informationen des Operators speichert. Erzeugt man mehrere dieser State Records für eine Suche, können Operatoren gegebenfalls mehrmals verwendet werden. Besitzt eine Suche mehr als einen Operator werden die zugehörigen state records kaskadierend aneinandergefügt, indem die Input Pointer eines Operators, welche auf die Funktionen open, next, close und den State Record verweisen, im State Record des vorherigen Operators in der Kette gehalten werden. Es lassen sich beliebig viele Operatoren aus der relationellen Algebra in einem Implementierungsansatz abbilden. Dem Exchange Operator, der als driver für den Operatorbaum unter ihm fungiert, kommt eine Hauptrolle zu. Im Gegensatz zu allen anderen Operatoren ist er ausschließlich zur Umsetzung der verschiedenen Parallelisierungsformen zuständig. Er lässt sich wie jeder andere Operator im Operatorbaum einfügen, ist aber zudem für die Erzeugung neuer Threads, die Verteilung der Tasks an diese und das Mergen der Ergebnisse zuständig.[3]

3.1.2 Ablauf

Eine Suche wird im Vorbild des *demand-driven dataflows* ausgeführt. Ein Operator muss nicht wissen, woher die Eingabedatensätze kommen, die er bearbeitet. Die Suche wird gestartet, indem im Wurzelknoten des Operatorbaums open() aufgerufen wird. Dies hat zur Folge, dass rekursiv in allen Kind-Knoten ebenfalls open() aufgerufen wird. Bei den meisten Operatoren führt das dazu, dass der State Record initialisiert wird. Der Exchange Operator allerdings erzeugt zusätzlich neue Threads für seinen Kindknoten und reserviert einen Ausschnitt des gemeinsamen Speicherbereich als Port, über den er die Eingabedatensätze erhält. Um die Suche auszuführen, wird next() wiederholend aufgerufen bis ein *end-of-stream indicator* zurückgegeben wird. In der Zwischenzeit ist der Rückgabewert von next() neben dem Statusindikators eine sogenannte NEXT-RECORD Struktur, welche aus einem record identifier und einer record address im buffer pool besteht. Dieser record hat eine feste Adresse im Buffer und dessen Speicherplatz muss vom consumer Operator wieder freigegeben werden. Am Schluss wird wiederum im Wurzelknoten des Operatorbaums close() aufgerufen, was zur Folge hat, dass rekursiv alle Iteratoren in der Kette abgeschaltet werden.[3]

3.2 Morsel-Driven Parallelism

Das Volcano Modell ist immer noch eines der meist verwendeten Konzepte zur Implementierung von Nebenläufigkeit in Suchanfragen. Jedoch hat es sich während den 30 Jahren nicht verändert und sich insbesondere nicht an die Entwicklung der Computerarchitekturen angepasst.

3.2.1 Auswirkungen wachsender Arbeitsspeicher

Gleichzeitig zum Aufkommen von many-core Architekturen mit mehreren zehntausend Kernen wuchsen auch die Arbeitsspeicher auf mehrere Terabyte an. Dies hatte zum Einen die Entwicklung von Hauptspeicherdatenbanken zur Folge. Zum Anderen wird in modernen Prozessoren eine non-uniform memory architecture verwendet. Eine wichtige Aufgabe bei der Verwendung einer solchen Architekur ist es die remote accesses möglichst gering zu halten. Dass das Volcano Modell eine solche Optimierung nicht miteinschließt, war eine Motivation das Morsel-Driven Parallelism Konzept zu entwickeln.[4]

3.2.2 Motivation

Dieser Ansatz verfolgt drei Hauptziele:

1. Bewahren der (NUMA-)Lokalität durch Zuweisen von Morsels an Kerne auf deren Sockets sie liegen.
2. Volle Elastizität des Grades der Parallelisierung.
3. Anpassung der Lastverteilung, sodass alle Kerne die an einer Suche beteiligt sind ihre Arbeit gleichzeitig beenden.

[4]

3.2.3 Aufbau

Das beschriebene System besteht aus verschiedenen Komponenten. Das QEPobject verwaltet den Ausführungsplan und leitet Pipelines an den Dispatcher weiter, deren Voraussetungen erfüllt sind und ausgeführt werden können. Die Aufgabe des Dispatcher ist es, die verfügbaren Ressourcen an die Pipelines zu verteilen, indem er speziell für jeden Kern eine Liste aller ausführbaren Pipeline-Jobs verwaltet und sie bei Bedarf den Worker-Threads zuweist. Worker-Threads werden bei Systemstart für jeden Hardware-Threads erzeugt und an diesen persistent gebunden, sodass kein Overhead durch erzeugen und zerstören von Threads entsteht, und arbeiten iterativ alle Pipeline-Jobs ab. Ein Pipeline-Job besteht aus einer Pipeline und einem dazugehörigen Morsel. Als Morsel bezeichnet man eine Unterteilungseinheit des von einer Pipeline zu verarbeitenden Speicherbereichs, auf dem die zu verwertenden Daten liegen, dessen Größe anpassbar an die Computerarchitektur aber typerischerweise 100 000 Tupel beträgt.[4]

3.2.4 Ablauf

Um eine Datenbanksuchanfrage abzuarbeiten, wird diese zunächst in einen Operatorbaum abgebildet. Dieser Baum wird nach sogenannten Pipelines analysiert, die sich voneinander durch Pipelinebreaker abtrennen. Jede Pipeline wird in ein Codefragment übersetzt sodass die Operatoren Zwischenergebnisse nicht materialisieren müssen. Der Speicherbereich wird in Morsels zerlegt, welche zusammen mit der Pipeline Pipeline-Jobs ergeben. Diese Pipeline-Jobs werden dem Dispatcher übergeben und nacheinander einzeln von einem Worker-Thread verarbeitet. Nachdem eine komplette pipeline abgearbeitet wurde, liegt das Ergebnis in temporär allozierten Speicherbereichen, welche zu gleich großen morsels zurückfragmentiert werden. Wie viele Worker-Threads an einer pipeline arbeiten ist von der Anzahl der Hardware-Threads abhängig. Nachdem ein Worker-Thread ein Morsel einer Suche abgearbeitet hat, kann er entweder ein anderes Morsel dieser Suche verarbeiten oder einer anderen Suche zugewiesen werden, sodass der Grad der Parallelisierung

an jedem Punkt der Suche erhöht oder verringert werden kann und der Plan nach dem die Threads abgearbeitet werden vollständig elastisch ist.[4]

4 Evaluation

In diesem Kapitel werden Implementierungen beider Ansätze getestet. Das Testsystem beinhaltet einen Intel Core i7 Prozessor mit 4 Kernen und einen 8 Gigabyte DDR3 Hauptspeicher. Als Tests wurden zwei Suchanfragen gewählt, deren Operatorbaum in Grafik 4 abgebildet ist. Führt man die Query Engines beider Kon-

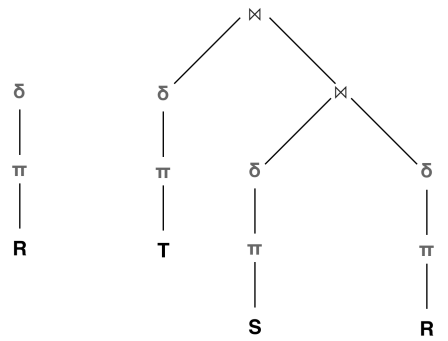

Abbildung 3

zepte und eine Unparallelisierte mit beiden Suchanfragen aus, so ergeben sich die Zugriffszeiten, welche in Grafik 3 abgebildet sind. Deutlich zu erkennen ist, dass

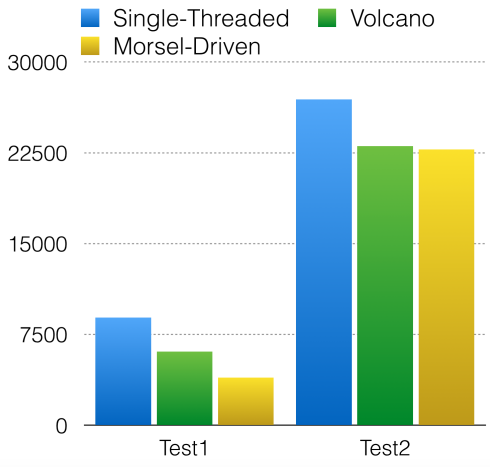

Abbildung 4: links: Test 1, rechts: Test 2

die unparallelisierte Query Engine in beiden Tests am längsten benötigt um die Suchanfrage auszuwerten. Eine deutlich Verbesserung zeigt sich bei der Ausführung auf der Query Engine, die nach dem Volcano Konzept implementiert wurde. Mit der morsel-driven parallelism Implementierung zeigt sich nur noch eine kleine Verbesserung. Jedoch wurde eine wichtige Eigenschaft dieses Konzepts nicht umgesetzt. Die Implementierung weist bezüglich NUMA keine Optimierungsmaßnahmen auf, da das Testsystem keine NUMA aufweist.

5 Fazit

Wie man im Vergleich sehen konnte, bringt Parallelisierung immer eine Zeitersparnis mit sich und ist somit, aus modernen Implementierungen von Query Engines nicht wegzudenken. Außerdem zeigte sich, dass eine Volcano Implementierung auf uniform memory architectures trotz des Alters eine erstaunliche Effizienz aufweist und in diesem Moment noch ausreichend ist. Es sollte jedoch zukünftig der *morsel-driven parallelism Ansatz* verwendet werden, da er wichtige Optimierungsansätze auch im Hinblick auf moderne Computerarchitekturen miteinschließt.

Literatur

[1] Chekuri, Chandra, et al. Scheduling Problems in Parallel Query Optimization. 1995.

[2] Majo, Zoltan, et al. Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. (2011).

[3] Graefe, Goetz. Encapsulation of parallelism in the volcano query processing system. (1989).

[4] Leis, Viktor, et al. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. (2015).