Vergleichende Untersuchung von Datenbanksystemen mit In-Memory-Technologien

Robert Pietzschmann,.....

06.01.2017

Inhaltsverzeichnis

1 Einleitung

Während unseres Projektseminars war es unsere Aufgabe Datenbanksysteme mit In-Memory Technologie vergleichend zu untersuchen. Betreuer bzw. Auftraggeber dessen war Prof. Dr. oec. Gunter Gräfe.
Wir hatten dabei mehrere Schwerpunkte. Es musste untersucht werden, wie Hochverfügbarkeit und Performancesicherung bei In-Memory-Technologien umgesetzt wurden. Es mussten Strategien erarbeitet werden, die zeigen wie Anforderungen an Verfügbarkeit, Ausfalltoleranz, Performance und Konsistenz in den unterschiedlichen Systemen, um transaktionale Daten zu verwalten und auf der anderen Seite die Analyse von großen Daten umgesetzt wurden. Dazu mussten wir uns in die Technologie der In-Memory Datenbank SAP Hana Express und dem MS SQL Server 2016 einarbeiten. Neben diesen beiden Hauptsystemen war noch gefordert selbiges bei NO-SQL Systemen zu überprüfen. Dort wählten wir Cassandra, sowie eine Maria DB mit vorgestelltem Memcache aus. Weiterhin sollten wir auf mehreren Systemen die verschiedenen Technologien umsetzen und in einem selbstgewählten Beispielszenario mit Daten füllen. Diese wurden für die Vergleiche zwischen den Systemen benötigt. Ein möglicher Zusatz dazu sollte der Entwurf einer Übungsaufgabe für das Modul „Erweiterte Datenbanksysteme“.

2 Gruppenmitglieder

Unsere Gruppe bestand während des Projektes aus sechs Mitgliedern. Marcel Kunz, Robin Arnoldt, Clemens Köhler, Phillipp Winkler, Moritz Buchwälder und Robert Pietzschmann.

3 Theorie

3.1 Grundlagen In-Memory Technologie

Der wichtigste Unterschied von einem In-Memory Datenbanksystem gegenüber einem herkömmlichen ist, dass die Datenhaltung im Hauptspeicher des Rechners erfolgt. Das führt zu erheblichen Vorteilen bei der Performance, da die Zugriffzeit so enorm verkürzt wird und Zugriffsalgorithmen sind einfacher. Nachteil ist vor allen der wesentlich höhere Preis für Arbeitsspeicher als für Festplatten. Da aber auch die Preise für jene mit ausreichender Speicherkapazität inzwischen nicht mehr unbezahlbar sind, werden In-Memory Datenbanken immer beliebter. Erst mit der Entwicklung der Kapazität der Hauptspeicher im letzten Jahrzehnt wurde es möglich In-Memory basierte Systeme zu schaffen. Wichtigste Vorraussetzung dafür war die Entwicklung von 64 Bit Systemen. Durch diese wurde es möglich ausreichend großen Arbeitsspeicher zu entwickeln. Im allgemeinen gilt für Speicherarchitekturen, dass je langsamer sie sind, desto billiger werden sie. Zuletzt veränderte sich zwar auch der Rest der Speicher grundlegend durch den Einzug von SSD Festplatten (Flash-Speichern), die wesentlich schneller sind als die Hard Disk. Allerdings sind diese aus Sicht der Software immer noch eine Platte wegen der Persistenz und der Nutzungseigenschaften. Dies hat zur Folge, dass für den Zugriff auf diese immer noch die gleichen Methoden verwendet werden wie für die Hard Disks. Zur vollen Ausnutzung der Geschwindigkeit des Flash Speichermediums müssen diese Algorithmen ständig ernuert werden. Der Hauptspeicher hat dagegen den Vorteil, dass ein direkter Zugriff möglich ist. So dauert das sequenziellen Lesen von einem MB hier nur 250000 ns gegenüber 1851851,9 ns bei einer aktuell schnellen SSD und 30000000 ns bei einer Hard Disk. Alle normalen Operationen der Datenbank werden deshalb im Hauptspeicher augeführt und die Festplatte wird nur für Backups bzw. als Archi genutzt. Als Problem ergibt sich, trotz der wesentlich besseren Performance ein Bottleneck beim Lader der Daten vom RAM in den Cache.
Einhergehend mit der Datenhaltung im Hauptspeicher gibt es einer weitere wesentliche Änderung gegenüber konventionellen Datenbanksystemen. So erfolgt die Datenhaltung spaltenorientiert. Dabei werden alle Reihen (Tupel) in angrenzenden Blöcken gespeichert. Dadurch eignen sie sich perfekt für einen schnellen lesenden Zugriff, was beispielsweise für die Aggregation sinnvoll ist. Zur Reduzierung der Datenmenge werden bei In-Memory Datenbanksystemen verschiedenen Kompressionsverfahren genutzt.
Aufgrund der Datenhaltung im Hauptspeicher ergibt sich noch ein Nachteil aus ökologischer und ökonomischer Sicht. So ist der Energieverbrauch einer von Main-Memory

Datenbankmanagementsystemen erheblich viel höher als der von konventionellen. So verbraucht ein „System“ ganze 1,5 Megawatt.

3.2 In-Memory Technologie im Vergleich zur konventionellen Datenhaltung

3.3 Kompressionsverfahren

Aus dem neuen Performance Bottleneck zwischen Hauptspeicher und Cache ergibt sich die Notwendigkeit Komprimierungsverfahren zu nutzen um eine schnellere Arbeit zu gewährleisten. Weiterhin sind diese notwendig, da trotz der aktuell relativ großen Arbeitsspeicherkapazitäten die Datenbanken oft noch größere Mengen an Daten enthalten, weshalb ohne Komprimierung eine Datenhaltung im Hauptspeicher nicht möglich wäre.

3.3.1 Kompressionsverfahren bei SAP HANA

Basis für die sehr effiziente Kompression stellt hier die spalten-orientierte Speicherung dar. Durch die Verringerung von Bits die zur Darstellung der Daten genutzt werden kann sowohl die Zugriffszeit, als auch der Speicherverbrauch verringert werden.
Grundlage stellt in der HANA das „Dictionary Encoding“ dar. Dabei werden Werte mit einer großen Länge (wie Texte) als Integer Wert gespeichert. Dabei ist es einfach zu verstehen und nicht schwer zu implementieren, was zu höheren Vorteilen in der Performance führt. Es arbeitet dabei spaltenweise. Hat die Tabelle zum Beispiel eine Spalte fName, so wird hier jedem Vornamen eine Positionsnummer zugeordnet. In dem sogenannten „Dictionary“ werden nun die Namen jeweils einzeln eingetragen und ebenso mit IDs versehen. Die beiden IDs werden dann einfach ein einem „Attribut Vector“ verknüpft. Je mehr Dopplungen von Namen in der Spalte fname sind, desto höher ist die Komprimierung. Angenommen jeder Vorname besteht aus 49 Byte und es gibt acht Milliarden Menschen auf der Erde, dann werden rund 365,1GB benötigt um alle zu speichern. Dahingegen braucht der „Attribut Vector“ 23 Bit pro Eintrag multipliziert mit den 8 Millionen Einträgen also 184 Milliarden Bit, was 21,4GByte entspricht. Das „Dictionary“ verbraucht bei 49 Byte pro Eintrag und 5 Millionen Vornamen nur 0,23GByte. Dividiert man nun die Menge ohne Kompression von 365,1GByte mit den 21,63GByte nach Kompression, sieht man, dass man der Speicherverbrauch um den Faktor 17 gesunken ist, was 6 Prozent des ursprünglichen Speicherbedarfs darstellt. Nutzt man dieses Prinzip bei den Geschlechtern,über ebenfalls 8 Milliarden Menschen, werden nur noch 0,93GB anstelle von 7,45 GB benötigt (Kompressionsfaktor 8). Der Kompressionsfaktor hängt dabei von der größe der Daten sowie der Tabelle ab, also der Spaltenkardinalität und der Tabellenkardinalität (Entropie - Maß für Informationsgehalt). Ein weiterer Geschwindigkeitsvorteil lässt sich durch sortierte „Dictionarys“ errreichen, da dann Binärsuche angewendet werden kann. Nachteil dieser Sortierung ist, das sie immer wieder neu durchgeführt werden muss, sobald ein Wert in dem „Drictionary“ angefügt wird.

Dies verursacht bei großen, sich ständig ändernden Datenmengen, aber wieder erhebliche Kosten.
Ein sehr großer Vorteil dieser Methode ist außerdem, dass alle Operationen auf den Daten der Tabelle in Attributvektoren mit Integer Werten ausgeführt werden. Diese kann der Prozessor wesentlich schneller verarbeiten als Zeichenketten. Der Prozessor muss so zwar eine Zusatzlast stemmen, aber da der Arbeitsspeicher hier der Bottleneck ist, ist dies vertretbar und fällt nicht sehr ins Gewicht. Da in der Regel die Ergebnismenge kleiner als die komplette Tabelle ist und bei vielen Operationen wie z.B. „Count“ nicht die echten Werte benötigt werden, ergibt sich auch kein Nachteil aus dieser Methode.
Da die Datenbanken großer Unternehmen oft mehrere Terabyte groß sind und die Kapazität der Arbeitsspeicher noch weit darunter ist, werden zusätzlich noch Kompressionsverfahren genutzt um ganze Datenbank im Hauptspeicher halten zu können. Dies hat auch den Vorteil, dass weniger Daten zwischen dem Prozessor und dem Speicher fließen müssen, was wiederum die Performance verbessert. Wichtig ist hier das Verhältnis von Kompression und den zusätzlich benötigten Operationen des Prozessors.
Heutigte Datenbanken enthalten oft „tonangebende“ Werte, wodurch derselbe Wert ohne Komprimierung sehr oft gespeichert werden muss. Um dieses Problem zu beheben und Speicher einsparen zu können wird das sogenannte „Prefix Encoding“ genutzt. Dafür müssen die Daten nach besagtem Wert sortiert werden und der Attributvektor mit diesem starten. Bei dem Vefahren wird nun nur eine Instanz des Wertes und die Häufigkeit dessen im „Attribut Vector“ gespeichert, anstelle des Wertes in doppelter und dreifacher Ausführung. Der Vektor besteht dann aus der Häufigkeit des Wertes, der ID aus dem „Dictionary“ und der ID's der folgenden Werte. Nimmt man nun zum Beipsiel die Tabelle mit der Bervölkerungsmenge nach Ländern sortiert, so befindete sich der Wert für China ungefähr 1,4 Milliarden mal in dem „Attribut Vector“. Daraus ergibt sich ein Speicherbedarf von 1,4 Milliarden mal 8 Bit. Mit „prefix Encoding“ lässt sich dies auf nur 39 Bit verringern (8 Bit für das einmalige Speichern und 31 Bit für die Häufigkeit). Darauf ergibt sich eine Ersparnis von immerhin 17 Prozent (1,3 GByte in diesem Beispiel). Zusätzlich dazu ermöglicht es direkten Zugriff über „row number claculation“ (Dabei determiniert die Datenbank, dass nur die bestimmte Anzahl an Zeilennummern benötigt (im Beispiel 1 - 1,4 Milliarden), was zu schnelleren Ergebnissen führt.
Eine weitere Möglichkeit der Kompression bietet das „Run Length Encoding“. Es funktionier am besten, wenn der „Attribut Vector“ einige sich unterscheidende Werte enthält, die sich sehr oft wiederholen. Zur Erreichung besonders guter Ergebnisse, muss die Spalte (des Vektors) sortiert werden. Die beieinaderliegenden gleichen Werte werden nun wie-

der zu einem zusammengefasst mit entweder der Häufigkeit oder der Startposition als Abstände. Letzteres hat dabei den Vorteil, dass ein schnellerer Zugriff ermöglicht wird, da direkt die Adresse eines Wertes gelesen werden kann, anstatt das diese erst berechnet werden muss. Bezogen auf das obige Beispiel lässt sich bei der Länderspalte wieder sehr viel einsparen. Dafür werden zwei Vektoren benötigt. Einer mit allen Ländern (200 Einträge) und einer mit den Startpositionen der einzelnen Länder. Acht Bit benötigt jeder Eintrag der Länder und 33 Bit jede Startposition (Log_2 8.000.000.000) plus ein Extra Feld (33 Bit) am Ende, welches die Häufigkeit des letzten Wertes beinhaltet (da diesem ja kein neuer Wert folgt). Alles in allem ergibt sich eine Größe des Vektors von nur rund 1 KB gegenüber 7,45 GB [200*(33Bit + 8 Bit)+ 33Bit]. Würde man die Häufigkeit anstelle der Startposition nehmen, könnte man zwar nochmal 33 Bit einsparen, aber es wäre kein direkter Zugriff mehr möglich, was zu einer viel höheren Antwortzeit führen würde und daher keine gute Option für IN Memory Datenbanken ist.
Cluster Encoding ist ein weiteres Interessantes KOmpressionsverfahren. Hier wird der „Attribut Vector“ in Blöcke der gleichen Größe unterteilt (oft 1024 Werte). Befinden sich in einem Block nur gleiche Einträge, so werden diese zu einem zusammengefasst. Ist dies nicht gegeben, bleibt der Block unverändert. Nun wird noch ein zweiter Vektor benötigt in dem steht, welcher Block durch einen einzelnen Wert ersetzt wurde (wenn Block unverändert 0, wenn zusammengefasst dann 1). Um hier auf die einzelnen Werte zuzugreifen, wird der Index mit Hilfe von Integer Division berechnet (Zeilennummer durch Größe des Blockes). Der wohl größte Nachteil ist hier, dass kein direkter Zugriff auf die Werte möglich ist. Es lässt sich aber eine große Menge Speicher einsparen. In Bezug auf das obige Beispiel, benötigt man bei der Spalte mit den Städten nur noch rund 2,4 GB, anstatt 18,6 GB (87 Prozent Kompressionsrate unter der Annahme, dass es 1 Million Städte gibt).
Bei „Indirect Encoding“ wird der Vektor ebenso in eine Vielzahl gleich großer Blöcke unterteilt (Größe meist wieder 1024). Als sehr effektiv erweist sich diese Methode, wenn ein Block einige sich unterscheidende Werte enthält, was oft gegeben ist, wenn eine Abhängigkeit zwischen zwei Spalten existiert und die Tabelle nach einer sortiert wurde (z.B. nach Land sortiert, wenn man die Spalte Vorname betrachtet wird). Hier wird neben dem globalen „Dictionary“ noch ein lokales benötigt. Wenn man nun die Spalte Vorname betrachtet und annimmt, dass es im Schnitt 200 Vornamen in einem Land gibt, wird in dem lokalen Wörterbuch jedem Wert für einem Namen ein Wert von 0 bis 199 zugeordnet, dieser dann in einem Vektor gespeichert. Die Einsparung an Speicherplatz resultiert hier daraus, dass bei nur 200 verschiedenen Werten nur noch acht statt 23 Bit

pro Eintrag belegt werden (Log_2(5 Millionen verschiedenen Namen Global) ergibt 23 Bit pro Eintrag ohne Kompression), unter der Voraussetzung das vorher nach den Ländern sortiert wurde. Hier erreicht man z.B. immerhin eine Kompressionsrate von 44 Prozent (nur rund 11,8 GB und nicht 21,4 GB) und ein direkter Zugriff ist im Gegensatz zum „Cluster Encoding“ weiterhin möglich.
Neben der Verringerung der Größe des „Attribut Vectors“ gibt es noch die Möglichkeit das „Dictionary“ zu optimieren. Das wird mit „Delta Encoding“ erreicht. Ist es nun alphanumerisch sortiert gibt es oft viele Werte mit der derselben Vorsilbe. Diese Tatsache macht man sich hier zu Nutze indem gemeinsame Vorsilben nur ein Mal gespeichert werden. Dabei wird wieder Blockweise gearbeitet (oft mit 16 Strings pro Block). es speichert die Länge der ersten Zeichenkette, gefolgt von dieser am Anfang eines Blocks. Bei den darauffolgenden Werten wird nun die Anzahl an Zeichen die gleich dem Vorgänger sind gespeichtert plus der Anzahl an folgenden, gefolgt von eben diesen Zeichen. Nimmt man zum Beispiel die Städte im „Dictionary“, der erste Wert des Blocks ist Aach gefolgt von Aachen, dann wird im komprimierten Vektor die Länge 4 und die Zeichen von „Aach“ festgehalten. Für „Aachen“ wird die Länge der gleichen Teils „Aach“ mit der 4 ersetzt, dann die Anzahl der folgenden und die Zeichen an sich gespeichert „en“. Dies lässt sich dann immer weiter so aufbauen, was im optimalen Fall immer mehr Einsparung mit sich bringt. Im hier brtrachteten Beispiel mit den Städten wird eine Kompressionsrate von 90 Prozent erreicht (rund 5,4 MB gegenüber 46,7 MB).
Bei al den Vorteilen durch Kompression sind dem Ganzen auch Grenzen gesetzt. So werden für die meisten Verfahrem sortierte Tabellen benötigt um maximale Erfolge zu erzielen, aber es kann in einer Tabelle immer nur nach einer Spalte sortiert werden. Wenn kein direkter Zugriff möglich ist, ergibt sich außerdem ein großer Nachteil in der Performance.

4 Quellen

Einleitung:
- https://de.wikipedia.org/wiki/raussuchenGlobalfoundries (20.12.2016, 14.15 Uhr)
Kompression:
-(07.01.2018 14.51)