Begriff

Die Bezeichnung „**spaltenorientierte Datenbank**“ ist etwas unglücklich gewählt, da hierunter zwei verschiedene Datenmodelle gemeint sein können. Üblicherweise sind mit spaltenorientierten Datenbanken jene gemeint, die ihre Daten, anders als die klassischen RDBMS, spaltenweise im Speicher ablegen, statt zeilenweise.

Im Bereich von NoSQL sind damit aber vor allem Datenmodelle gemeint, bei denen die Datenhaltung ähnlich wie ein Key-Value-System aufgebaut ist und auf Googles BigTable basieren, die sogenannten „**Column-Family-Systeme**“ (oder auch **Wide Column Stores**). Auch im Englischen wird mit „**column-oriented stores**“ oft nicht zwischen den verschiedenen Modellen unterschieden.

Spaltenorientierte Datenbanken

Spaltenorientierte Datenbanken legen ihre Daten, anders als herkömmliche Datenbanksysteme, nicht Zeile für Zeile, sondern spaltenweise im Speicher ab. Das macht sie besonders für analytische Aufgaben interessant.

Datenmodell

Bei den meisten relationalen Datenbanksystemen werden die Daten zeilenweise gelesen und abgespeichert. Das bedeutet, dass alle Zeilen hintereinander im Speicher abgelegt werden (also viele unterschiedliche Spalteneinträge beieinander). Bei spaltenorientierten Datenbanken geschieht dies spaltenweise (also viele gleichartige Spalteneinträge beieinander). Statt die Werte Zeile für Zeile im Speicher abzulegen, werden hierbei die Spalteninhalte hintereinander gespeichert. Das ermöglicht in gewissen Situationen einen deutlichen Geschwindigkeitsgewinn. Etwa bei statistischen Auswertungen, wenn in Tabellen nur die Werte einer Spalte von Interesse sind.

Als Abfragesprache wird nach wie vor auf SQL gesetzt und auch der Aufbau der Datenbank ist im Grunde gleich. Was anders ist, ist die Art wie die Datensätze im Speicher abgelegt werden. Das Problem traditioneller RDBMS ist, dass eine Tabelle oft hunderte Spalten enthalten kann, bei Abfragen jedoch meist nur ein Bruchteil der Spalten benötigt wird. Da die Datensätze nun aber klassischerweise zeilenorientiert im Speicher abgelegt werden, ist die Zeitspanne, die es braucht, die relevanten Spaltenwerte auszulesen gleich der Spanne die es benötigt, alle Records auszulesen. Die Idee hinter spaltenorientierten Systemen ist es daher, die Tabellen spaltenweise abzuspeichern, da so nun nicht mehr abertausende Records eingelesen werden müssen, nur um einige wenige interessante Werte einer Spalte zu erhalten, sondern alle relevanten Werte hintereinander gespeichert werden. Dies geschieht in der Reihenfolge der Records, sodass der n-te Eintrag aus Spalte X zum n-ten Eintrag aus Spalte Y passt (Vgl. Rouse 2010).

Ein Beispiel für eine simple Tabelle „personal“:

Personalnummer	Name	Abteilung	Gehalt
1	Müller	Schadensregulierung	65000
2	Maier	Schadensregulierung	50000

Personalnummer	Name	Abteilung	Gehalt
3	Huber	Personalabteilung	55000

Bei einer solchen Tabelle würden die Datensätze in einem zeilenorientierten System etwa so auf der Festplatte einfach hintereinander abgelegt werden:

```
1, Müller, Schadensregulierung, 65000; 2, Maier, Schadensregulierung, 50000;
3, Huber, Personalabteilung, 55000;
```

Beim spaltenorientierten Ansatz werden nun nicht die Reihen, sondern eben die Spalten hintereinander abgelegt:

```
1, 2, 3; Müller, Maier, Huber; Schadensregulierung, Schadensregulierung,
Personalabteilung; 65000, 50000, 55000;
```

Vorteile

Der offensichtlichste Vorteil ist die Geschwindigkeit beim Lesen von Spalten. Dabei können die interessanten Daten sofort gelesen werden, ohne erst durch etliche Spalten einer Tabellenzeile in klassischen Systemen springen zu müssen. Würde man nun etwa das durchschnittliche Gehalt der Angestellten berechnen wollen (`SELECT SUM(Gehalt) FROM personal`), ist schon auf den ersten Blick ersichtlich, dass eine spaltenorientierte Speicherweise hier schneller arbeiten würde, da alle relevanten Daten im Speicher hintereinanderstehen und mit einem einzigen Disk-Zugriff gelesen werden können. Ähnliches gilt für ein `UPDATE` auf Spaltenwerte. Auch das Hinzufügen neuer Spalten zu einer Tabelle ist bei diesen Systemen besonders einfach im Vergleich zu den traditionellen Datenbanken, da hier die Spalten hintereinander im Speicher abgelegt werden, wohingegen bei relationalen Systemen reihenweise vorgegangen werden müsste, was letztlich bedeutet, dass jeder Datensatz der betroffenen Tabelle einmal gelesen und so letztlich der komplette Datenbestand neu geschrieben werden muss.

Ein weiterer Vorteil ergibt sich daraus, dass bei einer spaltenorientierten Datenhaltung immer Daten desselben Typs und ähnlichen Eigenschaften hintereinander weg gespeichert werden. Das ermöglicht einfache und effiziente Komprimierungsmöglichkeiten und hilft, Datenredundanzen zu minimieren und gleichzeitig den Speicherbedarf zu verringern. Zudem kann so mehr Information auf weniger Platz gehalten werden, was zugleich auch weniger Plattenzugriffe bedeutet.

Nachteile

Ihre spaltenorientierte Datenhaltung, die ihnen beim Anfügen neuer Spalten zum Vorteil gereicht, wird den spaltenorientierten Datenbanksystemen beim Ergänzen oder Lesen von Zeilen jedoch zum Nachteil, denn dann müssen zunächst erst alle entsprechenden Spalten gesucht werden. Ähnlich aufwendig ist das Suchen von Attributen einer Zeile, da hier durch alle Spalten gesprungen werden muss, um die gewünschten Werte zu finden.

Kompression von Daten hilft zwar, Plattenplatz einzusparen, dies geschieht jedoch zu Kosten der Lese- und Schreibgeschwindigkeit, da Daten nun zunächst komprimiert, bzw. dekomprimiert werden müssen.

Verwendung

Typischerweise werden spaltenorientierte Systeme für Data-Warehouse- oder Business-Intelligence-Anwendungen oder generell bei analytische Aufgaben eingesetzt (OLAP), wo hauptsächlich lesend auf wenige Spalten einer Tabelle zugegriffen wird. Sie liefern schnelle Auswertungen trotz enormer Datenmengen, da sie, im Gegensatz zu reihenorientierten DBMS sofort die relevanten Blöcke lesen können, ohne den kompletten Datensatz lesen zu müssen. Da sich eine spaltenorientierte Speicherform für den Einsatz von Kompressionstechniken besonders gut eignet, werden sie oft auch zusammen mit der In-Memory-Technik verwendet.

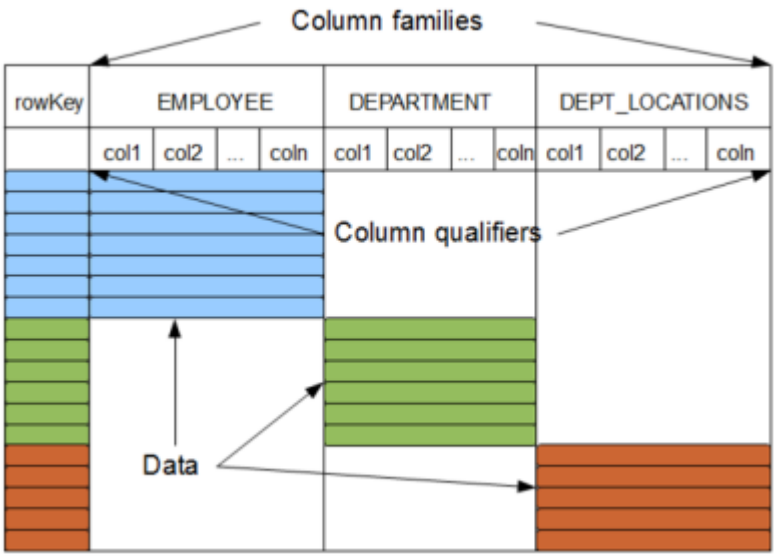
Findet u.a. Verwendung in: Sybase IQ, C-Store, SAP HANA, Oracle 12c, IBM DB2 BLU, in den Programmiersprachen S und R.

Column-Family-Systeme

Column-Family-Systeme (auch **Extensible Record Stores** oder **Wide Column Stores** - wobei diese Bezeichnung gelegentlich auch für spaltenorientierte Datenbanken benutzt wird) sind eine Art zweidimensionale Variante des Key-Value-Konzepts und basieren im Grunde auf Googles BigTable (Chang et al. 2006). Sie speichern Daten über Schlüssel, die mit Werten verknüpft sind und in „Familien“ aggregiert werden, auf die gemeinsam zugegriffen werden soll.

Oft findet man sie mit der Bezeichnung „spaltenorientierte Datenbank“ übersetzt, was nur bedingt richtig ist, da dieser Begriff eigentlich jene relationalen Datenbanksysteme meint, die ihre Daten spaltenweise (statt zeilenweise) abspeichern (die oft aber auch im Zusammenhang mit NoSQL Erwähnung und Verwendung finden).

Datenmodell



Sie werden gelegentlich verglichen mit relationalen Tabellen, bei denen Zeilen eine dynamische Anzahl an Attributen haben. Sie ordnen Daten nach einem Zeilenschlüssel, der mit beliebig vielen

Spalten verknüpft ist. Dabei wurden sie für eine verteilte Architektur entwickelt, damit sie hochverfügbar und –skalierbar sind. Grundsätzlich besteht das Modell aus **Spaltenfamilien** (**Column Familie**s), **Zeilen-Schlüssel** und den **Spalten** (**Columns**).

Die kleinste Einheit dieses Systems ist eine Spalte. Sie besteht aus einem Namen mit einem Wert (Key-Value-Paar) und einem Zeitstempel, der zur Versionsverwaltung dient (tatsächlich können, mehrere Versionen der Daten gespeichert werden, auf die man dann über den Zeitstempel).

Tabellen im klassischen Sinne gibt es nicht. Spalten, die als zusammenhängend betrachtet und auf die auch gemeinsam zugegriffen werden sollen, lassen sich als Sammlung zu **Spaltenfamilien** aggregieren und stellen so etwas Ähnliches wie die Tabellen im relationalen System dar.

Ihre Sortierung und das Referenzieren erfolgt über die **Zeilen-Schlüssel**. Dabei können solche “Zeilen“ beliebig viele Attribute/Spalten besitzen und sind an kein Schema gebunden. Jede Zeile kann einen anderen Aufbau haben, als die nächste. Sie sind in etwa vergleichbar mit den Dokumenten der dokumentenorientierten Datenbanken (Cattell 2010: S. 8).

Atomare Lese-/Schreib-Zugriffe finden über die Zeilen statt. Bevor nun aber Daten gespeichert werden können, müssen die entsprechenden Spaltenfamilien angelegt werden. Jede Spalte muss einer Spaltenfamilie zugeordnet sein. Nach dem Anlegen einer Familie kann dann auf die Spaltenschlüssel in ihr zugegriffen werden. Zudem bilden Spaltenfamilien die Einheit für die Zugriffskontrolle und die Speicherverwaltung (sowohl für RAM, als auch für Disk). (Chang et al. 2006: S. 2)

Darüber hinaus bieten manche Systeme (wie z.B. Cassandra) zusätzlich noch **Super-Columns** an, die wieder aus einem Schlüssel-Wert-Paar bestehen, wobei ihr Schlüssel der Name der Super-Column und der Wert eine Liste aus Spalten ist. Da Super-Columns also nicht direkt Werte enthalten, haben und benötigen sie (im Gegensatz zu regulären Columns) auch keinen Zeitstempel. Entsprechend dazu findet man in **Super-Column-Families** eine weitere Ergänzung, die die Super-Columns aufnimmt und ebenfalls über Zeilen-Keys referenziert werden.

Eine Datenbank im relationalen Sinne ergibt sich aus einer Sammlung von Spaltenfamilien, die meist als **Keyspaces** bezeichnet werden.

(Grafik-Quelle: JackHare 2012)

Beispiel

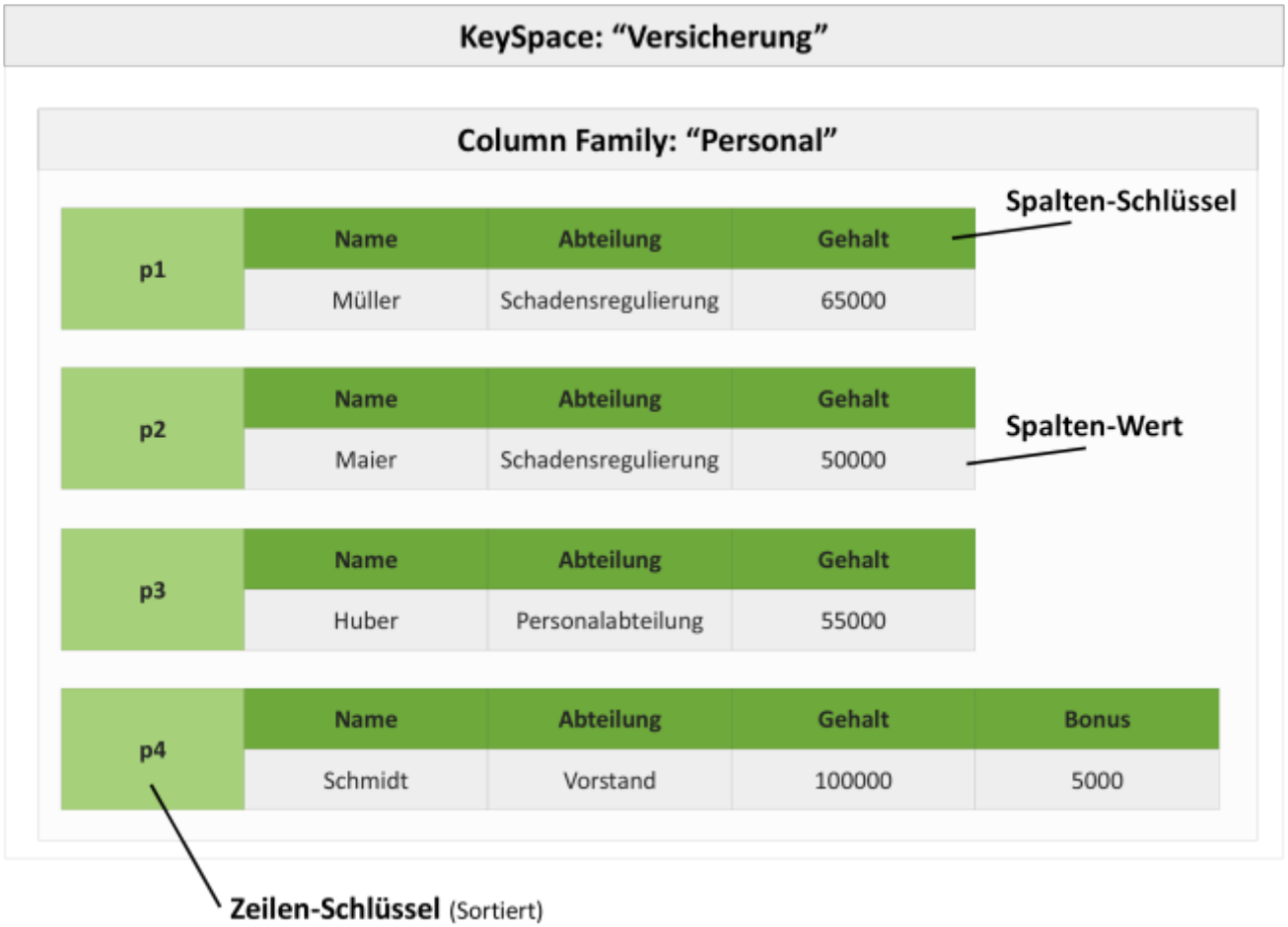
Eine Spalte besteht aus den drei Komponenten Key, Value und einem Timestamp und sieht allgemein so aus:

```
{
    key: "Spaltenname",
    value: "Spaltenwert",
    timestamp: 123456789
}
```

Das Beispiel der spaltenorientierten Datenbank sähe in Cassandra im JSON-Format etwa so aus: Eine Spaltenfamilie „Personal“ könnte im konkreten Beispiel in Cassandra etwa so umgesetzt werden (ohne Beachtung der Zeitstempel):

```
Personal = {
    p1: {
        Name: { name: "Name",value: "Müller" }
        Abteilung: { name: "Abteilung", value: "Schadensregulierung" }
        Gehalt: { name: "Gehalt", value: 65000 }
    },
    p2: ...
    p4: {
        Name: { name: "Name", value: "Schmidt" }
        Abteilung: { name: "Abteilung", value: "Vorstand" }
        Gehalt: { name: "Gehalt", value: 100000 }
        Bonus: { name: "Bonus", value: 5000 }
    }
}
```

Anzumerken ist, dass nun eine weitere Reihe mit der ID „p4“ hinzugekommen ist, die ein Attribut mehr aufweist als die übrigen.



Vorteile

Wie die meisten NoSQL-Datenbanken wurde auch dieses Modell für die Skalierung und Verteilung in großen Clustern konzipiert, um riesige Datenmengen im Petabyte-Bereich performant verarbeiten zu können. Ebenfalls gemein mit den anderen Systemen ist ihre Schemafreiheit, die einen flexiblen Umgang mit unstrukturierten Daten erlaubt.

Anders als die einfachen Key-Value-Stores erlauben sie auch Zugriffe auf einzelne Attribute über Zeilen- und Spalten-Schlüssel (z.B. `get(Zeilen-Key, Spalten-Key);`) und Bereichsabfragen. Diese erfolgen jedoch nicht über Werte, sondern über die Schlüssel. Hierzu werden die Elemente einer Spaltenfamilie nach ihrem Schlüssel sortiert gespeichert (Sadalage/Fowler 2012: S. 23). Da Wide-Column-Stores auf verteilter Architektur arbeiten, sind solche Bereichsabfragen nur dann wirklich performant, wenn die Daten auch auf demselben Server liegen.

Nachteile

Weniger geeignet sind die Systeme für komplexe Abfragen mit Beziehungen, da weder Inhalt noch Datentyp der Attribute von der Datenbank interpretiert werden können und es somit auch keine JOINs oder Fremdschlüssel gibt. Zudem sind keine Abfragen nach Wert möglich. Sie sind beschränkt darauf, nach den (Primär-)Keys vorzugehen. Das liegt auch daran, dass die Keys für die Verteilungsstrategie genutzt werden. Möchte man z.B. eine effiziente Bereichsabfrage, ist es wichtig zu wissen, dass die entsprechenden Werte (bzw. eigentlich Schlüssel) auch auf demselben Server liegen und nicht verstreut sind (Steemann 2013). Sie bieten daher auch Funktionen zur manuellen Verteilung der Daten.

Insgesamt erlauben Column-Family-Systeme viele Einstellungs- und Tuning-Optionen, was zwar für die Effizienzoptimierung sehr nützlich ist, aber die Administration unter Umständen erschwert und genaue Kenntnisse über die Architektur und Infrastruktur des Systems verlangt.

Verwendung

Ursprung des Systems (durch Googles BigTable) war das Bedürfnis nach einem flexiblen System mit hoher Performanz und Verfügbarkeit beim Umgang mit Daten im Petabyte-Bereich, verstreut auf tausenden Cluster-Knoten. Sie kommen vor in Content-Management-Systemen oder auf sozialen Netzwerken und Blog-Plattformen zum Einsatz. Die Datenbank Cassandra wurde bspw. ursprünglich von Facebook für die Posteingangssuche entwickelt. BigTable kommt intern in vielen Google-Diensten zum Einsatz, wie etwa in Google Maps, Google Earth oder YouTube.

Datenbanken die dieses Modell umsetzten: Google BigTable, HBase, Cassandra.