AG DBIS

Seminar: Data Streams
Thema: Stream-Verarbeitung mit konventionellen DBMS

von Christian Rensch

Juni 2005

Inhaltsverzeichnis

1 Einleitung

Die Anforderungen an ein Datenverwaltungssystem waren bisher auf die Verarbeitung von dauerhaft gespeicherten Datensätzen beschränkt. Jedoch wuchs im letzten Jahrzehnt das Interesse an Anwendungen, die nicht nur solche persistenten Daten verwalten können, sondern auch Daten in Form von kontinuierlichen Strömen. Quelle dieser Datenströme können die unterschiedlichsten Anwendungen sein, so z.B.

- Börsen-Ticker, die den Verlauf einer oder mehrerer Aktien mitzeichnen und an das Verwaltungssystem senden, welches dann diese Daten nutzt, um Prognosen für den weiteren Verlauf der Aktien zu stellen.
- Sensornetzwerke, wie sie z.B. in Kernkraftwerken benötigt werden, um Unregelmäßigkeiten direkt zu erkennen und so einen sicheren Betrieb zu garantieren.
- die Sensoren eines Straßenmautsystems.
- Wetterdaten, die von verschiedenen Wetterstationen gesammelt werden und dann von einem Rechenzentrum zur Wetterprognose genutzt werden.

Diese Art von Anwendungen erfordern weitere Eigenschaften, wie sie von bisherigen Nutzungsbereichen von Datenbanken noch nicht verlangt wurden, unter anderem auch die Tatsache, dass diese Datenströme potentiell unbeschränkte Mengen an Daten liefern.
Außerdem können Datenströme viele unabhängige Quellen besitzen, und dies führt dazu, dass es starke Schwankungen bei der Übertragungsrate geben kann, und dass die Reihenfolge des Eingangs der Daten nicht festgelegt ist.
Als einen der wichtigsten Punkte ist die Echtzeitfähigkeit bei der Beantwortung von Anfragen zu gewährleisten: eine zu lange Antwortzeit kann in gewissen kritischen Anwendungsbereichen, wie beispielsweise in Atomkraftwerken, sehr gefährlich werden.

Aus diesen Eigenheiten lassen sich weitere Anforderungen an ein Datenstromsystem stellen:
So kann nicht mehr jeder eingehende Datensatz wie bisher gespeichert werden, da selbst größte Sekundärspeicher bei potentiell unbegrenzten Mengen an kontinuierlichen Daten irgendwann volllaufen. Außerdem ist die ankommende Datenmenge unter Umständen so groß, dass sie nicht schnell genug gespeichert werden kann.
Hinzu kommt jedoch auch, dass nicht wirklich jeder Datensatz wichtige Informationen enthält und seine Werte möglicherweise verworfen werden.
Trotzdem soll es möglich sein, Anfragen auf Werte zu beziehen, die in der Vergangenheit liegen. Da Anfragen nun unter Umständen nicht mehr exakt beantwortet werden können, muss die Möglichkeit gegeben sein, approximative Antworten zu geben.
Zu den Anfragen kann man außerdem feststellen, dass in den Anwendungsgebieten, die Datenströme als Datenquellen besitzen, meist die gleichen Anfragen kontinuierlich gegen die ankommenden Ströme ausgewertet werden, wo hingegen in konventionellen Datenbanken veränderte Anfragen auftreten, und diese auch nur einmalig ausgewertet werden.
Datenstromsysteme sind also datengetrieben:
Daten kommen an, werden verarbeitet und anschließend verworfen. Dies muss das Verwaltungssystem eigenständig leisten, es wird nicht vom Menschen entschieden, welche Daten wie Einfluss auf Auswertungen haben. Man spricht daher auch von einem *DBMS-Active-Human-Passive*-Modell (DAHP) im Gegensatz zum *Human-Active-DBMS-Passive*-Modell (HADP), das die üblichen Datenverwaltungssysteme charakterisiert.

Eine Möglichkeit, diesen neuen Anforderungen zu begegnen, ist die Entwicklung speziell für diese Aufgaben ausgelegter Software, eines so genannten *Data-Stream-Management-Systems*,

kurz DSMS. Hier wurden mehrere Projekte gestartet, die meist auch verschiedene Ansätze verfolgen. Beispiele für DSMS sind

- *Aurora* [1], das von der Brown Universität gemeinsam mit der Brandeis Universität und dem MIT entwickelt wurde und schon kommerziell vertrieben wird.
- *TelegraphCQ* [2] von der Universität in Berkeley
- *STREAM* [3] (STanford stREam datA Management), eine Entwicklung der Universität Stanford, die mittlerweile implementiert ist und sich im Test-Stadium befindet. Interessanter Aspekt dieser Entwicklung ist die *Continuous Query Language* (CQL), die zum Zugriff auf das System verwendet wird. Sie ist eine Weiterentwicklung von SQL mit speziellen Befehlen zum Behandeln von Streams.

Ich möchte hier jedoch eine andere Herangehensweise an die Problemstellung präsentieren und auf ihre Möglichkeiten eingehen: die Verwaltung und Verarbeitung von Datenströmen mit herkömmlichen DBMS. Natürlich müssen auch hier Veränderungen vorgenommen werden, es soll jedoch ein konventionelles Datenbank-Verwaltungssystem als Grundlage dienen.

Die weitere Ausarbeitung gliedert sich wie folgt:
Zunächst möchte ich nochmals genauer auf einige Konzepte der Datenstromverwaltung eingehen und darauf, wie bisher die Anforderungen dieser Systeme gelöst und ihre Probleme umgangen wurden. Anschließend werde ich mich mit der Umwandlung von passiven Datenbanken in aktive Datenbanken beschäftigen, und wie diese die ihnen gestellten Aufgaben lösen. Dann möchte ich einige Projekte vorstellen, die bisherige Systeme nutzen und um eigene Systemteile erweitern, u.a. *Chronicles* und *STREAM*. Weiterhin möchte ich darauf eingehen auf welche Art sich große Anbieter von DBMS (IBM, Microsoft, Oracle) den neuen Aufgaben stellen, unter anderem werde ich dabei auf die Eigenschaften der *MS SQL Version 2005* eingehen, die sich momentan unter dem Codenamen "Yukon" in der letzten Entwicklungsphase befindet und bald auf den Markt kommen soll.

2 Grundlagen

2.1 Datenströme

Als einen Datenstrom bezeichnet man eine kontinuierliche Folge von Datensätzen, die von einem beliebigen, aber festen Typ sind. Ihr Ende ist im Allgemeinen nicht im Voraus bekannt. Die Rate, in der die Datensätze das verwaltende System erreichen, kann sehr stark variieren und möglicherweise so hoch sein, dass nicht jeder Satz verarbeitet werden kann. Daher ist es bei Datenströmen im Gegensatz zu anderen Datenquellen nicht möglich, sie als Ganzes zu verarbeiten, sondern sie müssen fortlaufend verarbeitet werden. Insbesondere ist kein wahlfreier Zugriff möglich, wie er z.B. bei der Nutzung von Arrays gewährt ist, sondern nur ein sequentielles Auslesen der Datensätze [4].

2.2 Anfragen

Anfragen lassen sich durch zwei Merkmale klassifizieren:

a) Wann wird die Anfrage erstellt?
b) Wann wird die Anfrage ausgewertet?

Bei a) unterscheidet man zwischen *vordefinierten* Anfragen, die bereits bekannt sind, bevor Daten eingetroffen sind und *ad-hoc* Anfragen, also Anfragen, die zu einem beliebigen Zeitpunkt an ein System gestellt werden können. Diese können sich jedoch auf Daten beziehen, die dem System nicht mehr zur Verfügung stehen und stellen daher ein Problem für die Verwaltung von Streams dar.

Die Frage b) unterscheidet dann zwischen *einmaligen* und *kontinuierlichen* Anfragen. Einmalige Anfragen, wie sie in konventionellen DBS vorkommen, werten eine Anfrage nur gegen den aktuellen Zustand des Systems aus. Im Gegensatz dazu verwenden kontinuierliche Abfragen ankommende Daten, um über einen längeren Zeitraum immer aktualisierte Antworten zurückzuliefern. Dieser Anfragetyp ist für stream-verwaltende Systeme relevanter. So kann diese Anfrageart dazu genutzt werden, Meldungen beim Verzeichnen gewisser Werte auszugeben. Ein mögliches Anwendungsbeispiel ist die Überwachung mehrerer Temperatursensoren eines Kernreaktors und das Erkennen von kritischen Veränderungen der Temperaturdaten.

2.3 Bisherige Lösungen

Um die Probleme mit kontinuierlichen Datenströmen zu umgehen, mussten bisher stets für jeden Anwendungsbereich eigene Systeme entworfen werden. Ein Top-Level Management-System für Datenströme existierte nicht. Diese Systeme mussten also programmierte Funktionen nutzen, um die ankommenden Daten auf die relevanten Eigenschaften zu untersuchen und gegebenenfalls entsprechende Maßnahmen einzuleiten.
Diese Lösung hat jedoch zwei große Nachteile:

a) Sie ist in der Herstellung sehr teuer, da sie domänenspezifisch angefertigt werden muss.
b) Die Wartung ist sehr aufwendig, da nicht nur ein Datenbankadministrator zur Sicherstellung des Dienstes nötig ist, sondern auch ein Software-Team, das bei Problemen und Anforderungsänderungen die Software umschreiben muss, und dabei exakte Kenntnisse der Programmstruktur besitzt.

Daher wurden solche Systeme jedem Kunden "maßgeschneidert". Als Beispiel hierfür seien Bankensysteme genannt.

3 Ströme in traditionellen DBMS

Die Behandlung von Strömen wie normale Datensätze führt Datenbank-Management-Systeme an ihre Grenzen:

- Obwohl heutige Datenbanksysteme wie *IBM DB2, MS SQL* oder *Oracle Database* mehrere Terrabyte an Daten verwalten können [5], ist es unmöglich, selbst wenn man die Nutzung von Tertiärspeichern mit einbezieht, einen potentiell unbegrenzten Datenstrom auf unbegrenzte Zeit mitzuspeichern, der auch noch in unbegrenzt schneller Folge auf das System eingeht.
- Auch wenn dies möglich wäre, so würden damit die Echtzeitanforderungen an ein solches System gesprengt. Diese Anforderung ist schon bei der reinen Nutzung von Primär- und Sekundärspeichern nicht mehr einzuhalten, wenn das System gewisse Grenzen überschreitet.

- Außerdem müssen diese Systeme im Ernstfall nach einem Absturz schnellstmöglich wieder einsatzbereit sein, um Gefahren abzuwenden. Bei den bisherigen System kann bei einer gewissen Größe eine Recovery-Time von mehreren Stunden vorliegen, was in bestimmten Anwendungsbereichen absolut inakzeptabel ist.

Um die Verarbeitung von Strömen in üblichen Systemen möglich zu machen, wurden bereits verschiedene Ansätze überprüft, die im Folgenden vorgestellt werden. Es bleibt jedoch festzuhalten, dass diese Ansätze meist nur einen Teil des Problems betrachten und zu lösen versuchen.

3.1 Trigger und Assertions

Die Verwendung von *Triggern* in einem DBS soll die Anforderung der kontinuierlichen Anfragen an das System realisieren.
Mit ihrer Hilfe soll es möglich sein, die bisherigen passiven Datenbanken in aktive, also selbstständig auf Ereignisse reagierende, Systeme umzuwandeln. Somit könnten z.B. Warnmeldungen beim Eintreffen kritischer Werte erzeugt werden.
Trigger sind inzwischen in allen großen relationalen DBMS realisiert.
So sind sie in Microsofts SQL Server implementiert, ebenso in Oracle. In IBMs DB2 sind Trigger seit der Version 6 von 1999 [6] vertreten, und auch das kostenlose PostgreSQL besitzt diese Funktionaltität dieser im ANSI-SQL-Standard von 1999 festgelegten Funktionalität.
Trigger besitzen die Form einer ECA - Regel (für Event - Condition - Action).

Dabei überwacht ein Trigger immer genau eine Tabelle. Änderungen an deren Datenbestand sind die Events. Es gibt Trigger für INSERT (wird beim Einfügen von Daten ausgelöst), UPDATE (wird beim Aktualisieren ausgelöst) und DELETE (wird beim Löschen von Datensätzen ausgelöst). Dann kann man die Daten gegen eine Bedingung (Condition) prüfen und kann je nach Antwort Aktionen (Actions) ausführen.

Man kann also einen Trigger als eine spezielle Anfrage auffassen, die dauerhaft im System besteht und fortlaufend ausgewertet wird. Hier wird die Möglichkeit der Nutzung von Triggern zur Realisierung von kontinuierlichen Anfragen offensichtlich. Ebenso kann man nun ein ganzes Datenstromsystem als eine Ansammlung von Triggern verstehen:
Das Ankommen eines neuen Datensatzes stellt ein Ereignis dar, das Verarbeiten innerhalb des Systems die Überprüfung der Bedingung. Schließlich kann nun noch als Resultat der Anfrage eine Aktion, sowohl auf den Daten als auch in der Anwendung selbst, gestartet werden.

Jedoch muss auch in Betracht gezogen werden, dass kontinuierliche Anfragen möglicherweise gegen mehrere Basistabellen ausgewertet werden sollen. Dies kann mit Triggern schon nicht mehr realisiert werden, da sie immer nur Ereignisse in einer Tabelle überwachen.

Ebenfalls hilfreich zur Verarbeitung von Datenströmen könnte die bereits im ersten SQL-Standard festgelegte Funktionalität der *Assertions* sein. Assertions werden komplett separiert von Ereignissen praktisch ständig überwacht. Sie sind an keine Tabelle gebunden und stellen allgemeine Regeln dar, die jederzeit im System gelten müssen. Mit ihnen kann aber nur Konsistenz sichergestellt werden, d.h. im Gegensatz zu den Triggern können keine Aktionen ausgeführt werden – werden die enthaltenen Bedingungen durch Änderungen einer Transaktion verletzt, wird diese zurückgesetzt.

Jedoch hat noch keines der heute auf dem Markt befindlichen DBMS das SQL-Konstrukt der Assertions umgesetzt.

Als Argumentation, weshalb dieses Konstrukt noch nicht oder nur unvollständig eingebaut wurde, gibt die Industrie an, dass eine Implementierung zu aufwändig und zu kompliziert wäre und jede Realisierung zu ineffizient [7].
Auch die im Vergleich dazu schwächere CHECK-Bedingung bei der CREATE TABLE Anweisung besitzt Einschränkungen, die so manche sinnvolle Verwendung verhindern. So können in ihr nur Felder der Tabelle, auf der die CHECK-Bedingung erstellt wurde, zum Test herangezogen werden, und die Verwendung von Aggregatfunktionen ist komplett untersagt. Bedingungen der Form "es müssen exakt n Sätze vorhanden sein" oder "die Summe der Werte des Feldes x muss immer n sein" können also nicht realisiert werden, obwohl dies oftmals interessante Integritätsprüfungen sein könnten.

Manche Professoren vertreten die Ansicht, dass allein die Umsetzung dieser im SQL-Standard festgelegten Regel-Konstrukte ausreichen würde, um aus passiven Datenbanken aktive zu gestalten. So sieht Prof. Manthey von der Universität Bonn unter der Voraussetzung, dass SQL noch an gewissen Stellen ergänzt wird, und dass kommerzielle SQL-Systeme diese Ergänzungen sowie die bisher noch fehlenden Teile des Standards integrieren, die Möglichkeit gegeben, intelligente DBMS zu erschaffen – also DBMS, die selbstständig auf Ereignisse reagieren und die nötigen Reaktionen vollziehen [7].

3.2 Materialisierte Sichten

Ein weiteres Konzept, das die kontinuierlichen Anfragen ermöglichen soll, sind materialisierte Sichten [8]. Sie sollen außerdem der Leistungssteigerung herkömmlicher DBMS dienen.
Normalerweise werden Sichten bei jedem Zugriff von den zugrunde liegenden Tabellen abgeleitet. Dadurch wird automatisch der aktuelle Zustand des Systems ausgewertet. Man nennt die zurückgegebenen Sichten auch virtuelle Sichten, da sie nicht im System als tatsächlich gespeicherte Daten vorliegen. Dies kann jedoch insbesondere bei komplexen Sichten, die sich auf mehrere Basistabellen (und unter Umständen sogar wieder auf Sichten) beziehen, sehr aufwendig sein. Auch die Größe der referenzierten Daten kann zu einer Verlangsamung bei Zugriffen auf die Sicht führen.
Um auch in solchen Fällen lesenden Operationen eine schnelle Antwort zur Verfügung zu stellen, versucht man, *materialisierte Sichten* einzusetzen. Dabei werden die Daten der Sicht tatsächlich "materialisiert", d.h. die Sicht wird gespeichert und muss nicht jedes Mal neu berechnet werden. Jedoch müssen auch diese Sichten immer den aktuellen Zustand des Systems repräsentieren, und so ist es zwingend notwendig, dass Änderungen in den Basisrelationen erkannt werden und die Sichten aktuell gehalten werden. Dabei setzt man auf Verfahren, die nicht jedes Mal die Sicht neu berechnen, sondern nur die wirklich betroffenen Sätze der materialisierten Sicht korrigieren. Auch hier erkennt man wieder eine Verwandtschaft des Konzeptes zu dem der kontinuierlichen Anfragen: Die stetige Auswertung der Daten gegen Abfrageausdrücke.

Sowohl IBM DB2, Oracle als auch Microsoft unterstützen dieses Konzept, wobei Oracle zunächst die Bezeichnung "SNAPSHOT" verwendete und erst mit der Version 8 auf den Begriff "MATERIALIZED VIEW" umstieg [9].

Oracle kennt dabei zwei verschiede Arten, um eine Aktualisierung der Sicht zu gestalten [10]:

- COMPLETE: erzeugt die Sicht komplett neu
- FAST: es werden nur veränderte Daten aktualisiert

Zur Nutzung des FAST-Modus muss jedoch ein "MATERIALIZED VIEW LOG" auf den zugrunde liegenden Relationen erzeugt werden, der die entscheidenden Änderungen mitzeichnet.
Oracle bietet außerdem die Möglichkeit, die bereits im System vorhandenen materialisierten Sichten zu nutzen, um neue Anfragen schneller zu beantworten. Dazu muss man beim Erstellen der neuen Abfrage nur die Option "ENABLE QUERY REWRITE" angeben.
Im Gegensatz zu früheren Versionen, bei denen diese Optimierung nur dann genutzt wurde, wenn durch Vergleich der einschränkenden Prädikate erkannt wurde, dass die Daten der neuen Anfrage bereits in einer vorhandenen materialisierten Sicht vorliegen, geschieht dies in der neuesten Version 10g sogar dann, wenn in den vorhandenen Sichten nicht alle verlangten Daten enthalten sind. Der Optimierer erkennt durch Prädikatenvergleich, welche Teile der Daten bereits materialisiert sind, und sucht nur noch die fehlenden Teile aus den Basisrelationen.

Ähnlich verhält es sich mit den in IBM DB2 erzeugbaren "Materialized Query Tables". Sie stehen dem Entwickler seit Version 8 zur Verfügung. Dabei werden sie wie normale Sichten erstellt, jedoch behandelt IBM ihre Erstellung wie die einer Tabelle – nur dass dem CREATE TABLE-Befehl eine AS-Klausel folgt, und diese dann die zu materialisierende Sicht beinhaltet. Zusätzlich kann man auch in DB2-Datenbanken einstellen, ob bei einer Aktualisierung die komplette Sicht neu aus den Basisrelationen erstellt werden soll, oder ob sie nur aus dem aktuellen Zustand und den in den Basisrelationenen aufgetretenen Änderungen abgeleitet werden soll. IBM nutzt hier im Gegensatz zu Oracle keine LOG-Datei, sondern eine Zwischenspeichertabelle.
Auch der Anfrageoptimierer von IBM nutzt die erstellten materialisierten Sichten zur Auswertung anderer Anfragen aus. Dazu muss die Option "ENABLE QUERY OPTIMIZATION" bei der Erstellung der materialisierten Sicht genutzt werden.

Microsoft führte mit seiner 2000er Version des SQL-Servers die "INDEXED VIEWS" ein [11], die die gleichen Vorteile wie materialisierten Sichten der Konkurrenten haben sollen.
Jedoch wählte man einen anderen Weg, um diese Vorteile zu erreichen:
Man stellt dem Nutzer die Möglichkeit, auf Sichten einen Index zu erstellen. Mussten bis dahin bei Operationen auf Sichten immer alle Sätze durchlaufen werden, um Antworten geben zu können, ist es nun möglich, wie bei echten Relationen mehrere Indexe auf eine solche Sicht zu legen. Um dies möglich zu machen wird aber auch hier die Sicht materialisiert. Dies geschieht, sobald der erste Index auf dem View mit "CREATE UNIQUE CLUSTERED INDEX" erzeugt wird - wobei es sich eben bei diesem ersten auch tatsächlich um einen eindeutigen Index mit Clustered-Eigenschaft handeln muss. Jeder weitere Index kann beliebige Eigenschaften besitzen.
Microsoft gibt aber noch weitere Einschränkungen vor, wobei hauptsächlich das Verbot der folgenden Konstrukte die Verwendbarkeit stark einschränkt:

- Top, Min, Max, Count
- das Benutzen eines anderen Views (vgl. Oracle - dort ist dies möglich)
- UNION, Unterabfragen und OUTER JOINS

Hiermit sind doch einige mögliche Sichten verboten, die man eigentlich sich optimiert wünscht. Gerade die Nutzbarkeit von Aggregatfunktionen macht materialisierte Sichten für Datenströme interessant:
Eine einmal erstellte Anfrage über den höchsten jemals aufgetretenen Wert – als Beispiel sei hier die höchste Verkehrsdichte an einem bestimmte Autobahnabschnitt genannt – müsste nur einmal gestellt, und dann regelmäßig aktualisiert werden. Bei Oracle und DB2 kann dies mittels materialisierten Sichten geschehen, hier bleibt Microsofts SQL-Server hinter der Konkur-

renz zurück – es müssen alle Werte der Basisrelation gesichtet werden, um das Maximum zu bestimmen.

Es sei noch erwähnt, dass auch Microsoft seinen Anfrageoptimierer dahingehend verbessert hat, dass er diese existierenden "Indexed Views" bei neuen Anfragen heranzieht, um die Leistung zu steigern - hier steht das Microsoft Produkt also den Konkurrenten Oracle und IBM nicht nach.
Zum Freeware System PostgreSQL sei gesagt, dass dieses bisher noch keine materialized Views unterstützt. Jedoch stellt Jonathan Gardner auf seiner Homepage eine einfache Möglichkeit vor, materialisierte Sichten in PostgreSQL mit Hilfe der vorhandenen Konstrukte zu erstellen [12]. Die Erzeugung mittels "STORED PROCEDURES" und "TRIGGER" erweist sich jedoch als sehr aufwendig: Will man eine automatische Aktualisierung bei Änderungen in den Basisrelationen realisieren, müssen die Trigger für jede materialisierte Sicht individuell auf den Basisrelationen der Sicht erstellt werden. Außerdem bleibt hier festzuhalten, dass es sich faktisch nur um eine temporäre Kopie einer konventionellen Sicht handelt. Es können dann also Indexe auf dieser Kopie erstellt werden, und so der Zugriff beschleunigt werden – andere Anfragen, deren Antworten aus dieser Kopie schneller abgeleitet werden könnten, werden nicht optimiert.

Auf dem Gebiet der materialisierten Sichten gibt es einige interessante Erkenntnisse, welche sich teilweise auf die Problematik der kontinuierlichen Anfragen übertragen lassen. So wurde untersucht, unter welchen Bedingungen es möglich ist, diese Sichten zu warten, falls deren zugrunde liegenden Relationen nicht oder nur teilweise zur Verfügung stehen [13]. Diese Situation tritt bei der Verwaltung von Datenströmen häufig auf, da hier wie bereits mehrfach erwähnt die Möglichkeit nicht besteht, alle Daten zu speichern.

3.3 Selbst-aktualisierende, materialisierte Sichten

Bei Sichten, die sobald sie einmal erstellt wurden fortan komplett ohne Zugriff auf die Basisrelationen gewartet werden können und nur durch die eintreffenden Daten auf dem neuesten Stand gehalten werden können, spricht man von *self-maintainable*, also sich eigenständig aktualisierenden Views.

Wie komplexe, auf mehreren Relationen basierende materialisierte Sichten selbst-aktualisierend erstellt werden können, haben D. Quass et al. an der Stanford University untersucht [14]. Ihre Grundlage war ein Data Warehouse System, bei dem ähnlich wie bei Strömen allein schon aus Leistungsgründen nicht kontinuierlich auf die Basisrelationen zugegriffen werden kann. Sie haben die Möglichkeiten untersucht, Views, die auf mehrere Relationen zugreifen, auf der Client-Seite aktuell zu halten, ohne dabei ständig über den Flaschenhals Netzwerk auf den Server zugreifen zu müssen. Dabei verwenden sie so genannte "Auxiliary Views", zu deutsch Behelfssichten. Diese Sichten werden auf der Client-Seite materialisiert und natürlich müssen sie wiederum selbst-aktualisierend sein. In ihrem Paper "Making Views Self-Maintainable for Data Warehousing" stellen Quass und seine Kollegen einen Algorithmus vor, der beliebige Sichten in solche selbst-aktualisierende Behelfssichten zerlegt.
Es kann ein großer Teil der möglichen Sichten eines Systems mit Hilfe dieses Algorithmus zerlegt werden; komplexe Abfragen mit mehreren Selektionen, Projektionen und Joins sind möglich.

Die einfachste Form einer Menge von Behelfssichten ist die Menge der Basisrelationen selbst. Wenn sie materialisiert beim Client vorliegen, kann durch sie kann ganz offensichtlich die materialisierte Sicht selbst-aktualisierend gehalten werden. Jedoch bleibt so das Problem, dass

der beim Client benötigte Speicherplatz alle Grenzen sprengen kann, bestehen - außerdem müssen dann trotzdem zunächst alle Daten über das Netzwerk gezogen werden.
Quass et al. stellen in ihrer Arbeit Möglichkeiten vor, die Datenmenge, die tatsächlich materialisiert werden muss, stark zu reduzieren, dabei bedienen sie sich unter anderem dem Umstand, dass durch Fremdschlüssel und den daraus resultierenden referentiellen Integritäten oftmals schon erkannt werden kann, welche Änderungen keine Auswirkungen auf die materialisierte Sicht haben werden. Außerdem erstellen sie ihre Hilfssichten nur mit Feldern, die tatsächlich relevant sind, d.h. entweder in der Sicht angezeigt oder zum Verknüpfen von Relationen genutzt werden.
Die entstehende Datenmenge ist nur noch ein Bruchteil der Gesamtdatenmenge, sodass das Verfahren gut einsetzbar erscheint.

3.4 Sliding Windows

Sliding Windows ist ein Ansatz, der direkt zur Verwaltung von Strömen entwickelt wurde. Mit ihm soll eine Möglichkeit gegeben werden, approximative Antworten zu geben. Das Prinzip der Sliding Windows beruht darauf, immer nur eine bestimmte Menge von aktuellsten Daten zu betrachten. Damit wird es möglich, auch Funktionen, die bisher auf Strömen nicht möglich waren, zu realisieren. So konnte das kartesische Produkt zweier unbegrenzter Ströme nicht in begrenztem Speicher gebildet werden. Auch Durchschnittsbildungen einzelner Werte war auf unbegrenzten Strömen nicht möglich. Beschränkt man jedoch die Anzahl der betrachteten und zur Verknüpfung herangezogenen Datensätze in ausreichendem Maß, so ist eine Behandlung wie bei Relationen möglich. Man unterscheidet zwei Arten von Sliding Windows (SW):

- count-based (anzahlbasierte) SW geben die letzten N Tupel eines Stromes zurück
- time-based (zeitbasierte) SW geben die Tupel zurück, die in den letzten T Zeiteinheiten eingetroffen sind

Für letzteres ist ein geeignetes *Timestamp*-Verfahren unverzichtbar.
Auch dieses Verfahren eröffnet wieder neue Forschungsgebiete. So müssen diese Fenster wie materialisierte Sichten auf dem aktuellen Stand gehalten werden; man möchte ja immer die neuesten N Tupel in diesem Fenster halten. Eine Übersicht über die Optimierungen bei der Einbringungs- und Verdrängungsstrategie wird in [15] behandelt.
Vorteile dieses Verfahrens liegen auf der Hand: Es ist wohl-definiert und leicht zu verstehen.
So ist offensichtlich, wie die Schätzung geschieht, außerdem ist das Verfahren deterministisch. Und schließlich hat es noch einen weiteren Vorteil, denn das Verfahren operiert auf den aktuellsten Daten. Diese sind für die meisten Anwendungen in der Realität viel relevanter als ältere Daten.
Es gibt jedoch weder im ANSI-SQL-Standard, noch in irgendeiner komerziellen Datenbank eine direkte Implementierung dieser Herangehensweise. Einzig im DSMS STREAM (siehe Abschnitt 4.2) wird dieses Konzept umgesetzt – jedoch bieten Datenbanksysteme wie Oracle, DB2 und MS SQL ähnliche Herangehensweisen, um die zu verarbeitende Datenmenge bei Anfrage zu reduzieren, z.B. Partitionen auf Tabellen.

3.5 Partitionen

Das Konzept der Partitionierung ist mit dem Sliding Window-Ansatz verwandt. Oftmals werden die Begriffe als Synonyme gebraucht. Es muss bei Partitionen jedoch angemerkt werden, dass diese meist schon bei der Erstellung der Datenbank implementiert werden, und ankommende Daten direkt einer Partition zugeordnet werden, während echte Sliding Windows den

Datenbestand zur Laufzeit beschränken. Der Datenbestand einer Tabelle wird praktisch nochmals auf mehrere Partitionen verteilt, wobei der Nutzer selbst festlegen kann, wie diese Unterteilung aussehen soll.
Auch bei der Partitionierung geht es darum, die Anzahl der zu betrachtenden Datensätze bei Anfragen zu reduzieren, um so schnellere Antworten bei sehr großen Datenbeständen geben zu können – also auch bei Systemen, die Ströme verwalten. Ein weiterer Vorteil des Konzepts ist, dass Auswertungen, die sperrende Operationen verlangen, nur die Partitionen sperren müssen, für die die Daten ausgewertet werden. So kann auch eine höhere Verfügbarkeit sichergestellt werden. Und schließlich lassen sich durch sie auch Auswahlkriterien realisieren, die entscheiden, welche Daten weiter für das System interessant sein können und welche verworfen oder ausgelagert werden können.

3.5.1 Oracle

Oracle 10g bietet hier wohl die bisher umfangreichste Integration des Konzepts [16]. Ein Beispiel:

```
CREATE TABLE Bestellung
     (BestellNr NUMBER, Datum DATE, ArtikelNr NUMBER,
     Menge NUMBER, Preis NUMBER, KundenNr NUMBER,
     KundeStaatsangehoerigkeit VARCHAR2(2))
PARTITION BY RANGE(Datum)
     (PARTITION vor_jahr_2000 VALUES LESS THAN
          TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
     PARTITION  vor_jahr_2004 VALUES LESS THAN
          TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
     PARTITION nach_jahr_2004 VALUES LESS THAN
          (MAXVALUE) )
```

Der Befehl erzeugt eine Tabelle, die in drei Partitionen unterteilt ist – die erste ("vor_jahr_2000") beinhaltet alle Bestellungen, die vor dem 01.01.2000 eingingen, die zweite ("vor_jahr_2004") alle zwischen dem 01.01.2000 und dem 31.12.2003 erhaltenen und schließlich die dritte, die alle Bestellungen seit dem 01.01.2004 bis zum aktuellen Zeitpunkt beinhaltet.
Auch eine Einteilung in feste Kategorien statt in Bereiche ist möglich. So kann mittels

```
PARTITION BY LIST (KundeStaatsangehoerigkeit)
     (PARTITION best_europa VALUES ('BRD','FR','ESP'),
     PARTITION best_asien VALUES('JAP','TW'),
     PARTITION best_null VALUES (NULL),
     PARTITION best_unbekannt VALUES (DEFAULT))
```

eine exakte Zuteilung von Datensätzen mit bestimmten Eigenschaften zu konkreten Partitionen erfolgen.
Oracle bietet außerdem noch eine dritte Möglichkeit an, Partitionen zu erstellen an:

```
PARTITION BY HASH (KundeStaatsangehoerigkeit)
     (PARTITION best_1,
     PARTITION best_2)
```

unterteilt die Tabelle in zwei Partitionen. Ankommende Datensätze werden durch eine Hash-Funktion operierend auf der Spalte `KundeStaatsangehoerigkeit` in die entsprechende Partition eingeteilt.

Zusätzlich zu den Partitionen können nochmals Subpartitionen erstellt werden. Dies geschieht durch eine weitere Klausel des CREATE TABLE-Statements. Es können für jede Partition eigene Subpartitionen erzeugt werden, oder aber auch ein Subpartitions-Schema, das für alle Partitionen gilt.

Oracle bietet die Möglichkeit, diese Partitionen getrennt zu speichern. Somit ist es möglich, uninteressante Daten auf einem langsameren Datenträger auszulagern, und wichtige Daten, die oft für Berechnungen benötigt werden, schnell zur Verfügung zu haben. Es wird hier also die Selektivität gefördert, Data Streams können so gefiltert, und je nach Grad der Wichtigkeit ihres Inhalts gespeichert werden.

Anfragen, die jetzt auf eine in dieser Weise optimierte Tabelle erfolgen, werden vom Anfrageoptimierer so ausgewertet, dass das Wissen über die Partitionierung Geschwindigkeitsvorteile bringt. Dazu werden die Prädikate der Anfrage mit denen verglichen, die die Partitionen bestimmen.
Außerdem wird dem Nutzer die Möglichkeit gegeben, nicht mehr nur auf die gesamte Tabelle Anfragen zu stellen, sondern direkt auf ihre Partitionen zuzugreifen. Dabei erfolgen diese Zugriffe so schnell wie die Zugriffe auf eine Tabelle.

Mit

```
SELECT BestellNr FROM Bestellung PARTITION (vor_jahr_2000)
```

erhält man also alle Bestellnummern von Bestellungen vor dem 01.01.2000. Somit lässt sich ein time-based Sliding-Window-Verfahren leicht implementieren. Dem Nutzer ist somit auch die Möglichkeit gegeben, Approximationen zu erstellen, die nur die neuesten Daten zur Berechnung heranziehen.

Zur Umstrukturierung von vorhandenen Partitionen und Subpartitionen, wie z.B. das Zusammenführen zweier Partitionen, das Löschen bestimmter Partitionen usw., sowie zum nachträglichen Erstellen neuer Partitionen, wurden dem Nutzer in Oracle 10g ebenfalls Befehle bereit gestellt, mit deren Hilfe diese Aufgaben einfach zu realisieren sind.

3.5.2 DB2

Seit Version 8 unterstützt auch DB2 das Konzept der partitionierteten Tabellen [16]. Zwar war es schon in den Vorgängerversionen möglich, zumindest den über den Index den Inhalt einer Tabelle aufzuteilen, aber nun kann man wie bei der Konkurrenz aus dem Hause Oracle Tabellen einfach in Partitionen einteilen, und sie ebenfalls auf verschiedene Datenträger auslagern. Die Syntax weicht jedoch leicht von der bei Oracle verwendeten ab und besitzt auch nicht den vollen Funktionsumfang. So beherrscht DB2 nicht Oracles Partion-By-List und Subpartitionen werden ebenfalls noch nicht unterstützt. Aber auch IBM bietet den Zugriff direkt auf einzelne Partitionen an, sodass die Vorteile der Partitionierung genutzt werden können.

3.5.3 Microsoft

Microsofts SQL Server 2000 unterstützt Partitionierung nicht. Stattdessen wird den Datenbankadministratoren ein Workaround vorgeschlagen. Microsofts Support empfiehlt die Erstellung mehrere Tabellen, die jeweils eine Partition repräsentieren sollen [17]. Dann soll ein View erzeugt werden, der mittels UNION ALL die "Partitionen" zu einer Relation zusammenfasst – er bildet dann die eigentliche Tabelle, die alle Partitionen besitzt, und auf der Operationen stattfinden können sollen. Es gibt zwei Arten des Views, einmal die *updatable* Version, die das Einfügen und Ändern von Datensätzen erlaubt, jedoch starke Einschränkungen die Schlüssel betreffend mit sich bringt, und einmal die *queryable* Version, an die nur Anfragen gestellt werden können, jedoch keine Einfügungen möglich sind. Bei der zweiten Version muss also die Applikation die Daten direkt in die Tabellen, die als Partitionen dienen, einbringen.
Es ist offensichtlich, dass beide Arten nicht den Funktionsumfang liefern, wie sie die direkte Implementierung des Konzepts bei Oracle und IBM mit sich bringt. Eine starke Einschränkung ist der Umstand, das der UNION ALL-View nur 256 Tabellen (und somit Partitionen) zusammenführen kann. Immerhin werden Anfragen die den UNION ALL-View als Basisrelation beinhalten und die nur bestimmte Partitionen betreffen dahingehend optimiert, dass sie tatsächlich nur auf den betroffenen Partitionen operieren.

3.6 Sampling

Eine weitere Möglichkeit um Geschwindigkeit bei Anfrage zu gewinnen und approximative Antworten zu geben ist das so genannte *Sampling*. Dabei werden nicht wie bei den Partitionen oder den Sliding Windows Datensätze mit bestimmten Eigenschaften betrachtet, sondern eine beliebige Auswahl der Datensätze. Entscheidend für die Anzahl der betrachteten Datensätze ist die *Sampling-Rate*. Je höher die Rate, desto mehr Datensätze werden betrachtet und umso genauer werden die Approximationen – jedoch auf Kosten der Geschwindigkeit.

Bei der Einbindung dieses Konzepts steht Oracle allein. Als einziger Datenbankhersteller hat Oracle dem Nutzer die Möglichkeit zur Nutzung von Sampling direkt in ihren SQL-Dialekt verbaut [16]. Mit

```
SELECT AVG(Bestellmenge) FROM Bestellungen SAMPLE (10)
```

wird jeder Datensatz der Tabelle Bestellungen mit einer Wahrscheinlichkeit von 10% zur Berechnung der durchschnittlichen Bestellmenge herangezogen. Damit werden also insgesamt durchschnittlich etwa 10% der Datensätze betrachtet, wenn die Anfrage mehrmals gestellt wird.

Bei bestimmten Funktionen, wie z.B. bei Summenbildungen ist natürlich zu beachten, dass nur 10% der Datensätze betrachtet werden. So liefert nur

```
SELECT SUM(Bestellmenge) * 10 FROM Bestellungen SAMPLE (10)
```

eine korrekte Abschätzung der Summe der Bestellmengen.

Zusätzlich kann noch ein Seed-Wert beim Sampling in Oracle angegeben werden. Wird der gleiche Seed-Wert auf einer Relation verwendet, so werden auch die gleichen Datensätze zur Approximation verwendet. Ein Fehlen des Seed-Wertes erzwingt bei jeder Ausführung eine neue Abschätzung. Bei

```
SELECT AVG(Bestellmenge) FROM Bestellungen SAMPLE (10)
     SEED (1);
SELECT AVG(Bestellmenge) FROM Bestellungen SAMPLE (10)
     SEED (4);
SELECT AVG(Bestellmenge) FROM Bestellungen SAMPLE (10)
     SEED (1);
```

liefern die erste und die dritte Anfrage die gleiche Abschätzung.

3.7 Analytische Funktionen

Um für große Datenmengen, wie sie Ströme liefern, komplexe Analyseanfragen beantworten zu können, mussten bisher oftmals spezielle Anwendungsprozeduren geschrieben werden, da insbesondere Aggregatfunktionen bisher nur auf mit GROUP BY zusammengefassten Daten arbeiten konnten und auch dort nur wirklich einfache Funktionen ausführen konnten. Zwar bieten die großen DBMS-Hersteller, allen voran IBM, Erweiterungen für ihre System an, die solche Funktionen schnell und einfach zur Verfügung stellen, jedoch müssen diese teuer gekauft werden.
Der SQL-99-Standard führt analytische Funktionen ein, die diese Problematik beseitigen. So ist es seither möglich, Durchschnittsbildungen oder andere Aggregatoperationen auf *Windows* durchzuführen [16].

```
FUNCTION_NAME(expr) OVER {window_name|(window_specification)}

window_specification ::= [window_name][partitioning][ordering][framing]

partitioning ::= PARTITION BY value [, value...] [COLLATE collation_name]

ordering ::= ORDER [SIBLINGS] BY rule [, rule...]

rule ::= {value|position|alias} [ASC|DESC] [NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {start|between} [exclusion]

start ::= {UNBOUNDED PRECEDING|unsigned-integer PRECEDING|CURRENT ROW}

between ::= BETWEEN bound AND bound

bound ::= {start|UNBOUNDED FOLLOWING|unsigned-integer FOLLOWING}

exclusion ::= {EXCLUDE CURRENT ROW|EXCLUDE GROUP|EXCLUDE TIES|EXCLUDE NO OTHERS}
```

Abbildung 1: Window Syntax in SQL03

Dabei erlaubt es das Window-Konstrukt, eingeleitet mit OVER, einzelne logische Bereiche innerhalb einer Sicht zu erzeugen - sie entsprechen dann wieder Teilströmen, deren Daten in bestimmten Attributen übereinstimmen, vergleichbar mit Gruppenbildungen durch die GROUP-BY-Klausel. Der Unterschied zu Aggregatfunktionen auf durch GROUP BY gebildete Gruppen, liegt darin, dass Aggregatfunktionen in einer Anfrage auf mehreren verschiedenen Windows operieren können. Außerdem liefern diese analytischen Funktionen zwar aggregierte Werte zurück, jedoch fassen sie die Resultsets nicht zusammen.

Einige Beispiele:
Gegeben sei folgende Tabelle1:

AngestellterNr	AbtNr	Gehalt
1001	10	2500
1002	10	3000
1003	20	1250
1004	20	1700
1005	20	1000
1006	20	1900
1007	30	3600

Die Anfrage

```
SELECT AbtNr, COUNT(*) AbtCount FROM Tabelle1
 GROUP BY AbtNr ORDER BY AbtNr
```

liefert

AbtNr	AbtCount
10	2
20	4
30	1

wohingegen

```
SELECT AngestellterNr, AbtNr,
 COUNT(*) OVER (PARTITION BY AbtNr) AbtCount
 FROM Tabelle1 ORDER BY AngestellterNr
```

folgendes liefert:

AngestellterNr	AbtNr	AbtCount
1001	10	2
1002	10	2
1003	20	4
1004	20	4
1005	20	4
1006	20	4
1007	30	1

Die Daten einer Partition werden also nicht zusammengefasst, sondern bleiben einzelne Sätze. Dadurch muss auch nicht mehr über jedes Attribut, das im Ergebnis erhalten bleiben soll, gruppiert werden (vgl. AngestellterNr).

Man kann hier auch über die ganze Relation aggregieren, dazu muss nur die PARTITIONING-Klausel ausbleiben.

```
SELECT AngestellterNr, AbtNr, COUNT(*) OVER () GesCount
 FROM Tabelle1 ORDER BY AngestellterNr
```

liefert

AngestellterNr	AbtNr	GesCount
1001	10	7
1002	10	7
1003	20	7
1004	20	7
1005	20	7
1006	20	7
1007	30	7

Und wie bereits erwähnt ist es auch möglich, in einer Anfrage über mehrere verschiedene Bereiche zu aggregieren:

```
SELECT AngestellterNr, AbtNr,
 COUNT(*) OVER () GesCount,
 COUNT(*) OVER (PARTITION BY AbtNr) AbtCount
 FROM Tabelle1 ORDER BY AngestellterNr
```

liefert

AngestellterNr	AbtNr	GesCount	AbtCount
1001	10	7	2
1002	10	7	2
1003	20	7	4
1004	20	7	4
1005	20	7	4
1006	20	7	4
1007	30	7	1

Analytische Funktionen bieten noch mehr, so kann zum Beispiel innerhalb eines logischen Bereiches nochmals eingeschränkt werden, welche Zellen, ausgehend von der Zeile für die die Berechnung gerade stattfindet, betrachtet werden sollen. Man kann hierzu die logischen Bereiche mit ORDER BY sortieren und dann mit den Schlüsselwörten ROWS oder RANGE einen zu betrachtenden Zeilenbereich festlegen, der zur Aggregation betrachtet wird.

Die Anfrage

```
SELECT AngestellterNr, AbtNr,
 COUNT(*) OVER () GesCount,
 COUNT(*) OVER (PARTITION BY AbtNr) AbtCount
 COUNT(*) OVER (PARTITION BY AbtNr ORDER BY Gehalt
  ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
  BesseresGehalt
 FROM Tabelle1 ORDER BY AngestellterNr
```

liefert

AngestellterNr	AbtNr	GesCount	AbtCount	BesseresGehalt
1001	10	7	2	1
1002	10	7	2	0
1003	20	7	4	2
1004	20	7	4	1
1005	20	7	4	3
1006	20	7	4	0
1007	30	7	1	0

Die Tabelle wird für das Feld "BesseresGehalt" also nach Abteilungen partitioniert. Dann werden die einzelnen Partitionen nach Gehalt sortiert, und schließlich werden für jeden Datensatz zur Berechnung von "BesseresGehalt" folgende Sätze gezählt:
Alle Datensätze in der gleichen Partition wie der aktuelle Datensatz und zwischen dem Ersten, auf den aktuellen Datensatz folgenden Datensatz ("1 FOLLOWING") und dem letzten Datensatz ("UNBOUNDED FOLLOWING"). Das Feld "BesseresGehalt" liefert also die Anzahl der Mitarbeiter, die in der gleichen Abteilung arbeiten und mehr verdienen als der gerade betrachtete Mitarbeiter.

Analytische Funktionen bilden ein sehr mächtiges, neues Konzept, das Analyseanfragen an sehr große Datenbanken stark vereinfacht. Auch hier muss man Oracle wieder eine führende Rolle eingestehen. Eine sehr gute Zusammenfassung über die Möglichkeiten, die Oracle hier bietet, findet sich in [18].

IBM hat ebenfalls das Window-Konstrukt implementiert und stellt dem User eine große Anzahl analytischer Funktionen zur Verfügung. Microsofts SQL Server 2000 und PostgreSQL besitzen noch keinerlei Umsetzung des Konzepts.

4 Stream-verwaltende Systeme

4.1 Das Chronicle Data Modell

Das Chronicle Data Modell [19] ist eine Entwicklung von AT&T mit der Zielsetzung, ein System zu erstellen, das mit Strömen umgehen kann. Dabei besteht eine Chronicle Datenbank aus Relationen, Chronicles und materialisierten Sichten.
Relationen sind aus den traditionellen Systemen übernommen. Chronicles sind vergleichbar mit Relationen, nur dass Chronicles eine Sequenz von Datentupeln anstatt einer ungeordneten Menge eben dieser sind. Eigentlich ist ein Chronicle eine Relation mit einem zusätzlichen, die Reihenfolge festlegenden Attribut. Jedoch sind auf ihnen nur Anfügungen erlaubt, Chronicles sind also append-only. Einzige Bedingung für die Einfügung ist, dass die Sequenznummer höher ist als die aller im System vorhandenen Chronicles.
Vor allem basiert dieses System jedoch auf den bereits vorgestellten materialisierten Sichten. Ihre Fähigkeit zur Selbst-Aktualisierung stellt auch hier die hauptsächliche Aufgabe.

Als mögliches Anwendungsbeispiel seien Telefongesellschaften genannt, wobei jeder Anruf eines bestimmten Kunden zu einer bestimmten Zeit ein Datentupel darstellt. Die Datenmengen eines Chronicles können unendlich groß werden, daher kann der gesamte Bestand nicht gespeichert werden. Es sei außerdem erwähnt, dass diese Art von Anwendung die ursprüngliche Motivation zur Entwicklung des Chronicles Modells beinhaltet:

Die kontinuierliche Anfrage an Summen. So kann es für ein Telefonunternehmen wichtig sein, zu prüfen, ob dem Kunden noch Freiminuten zur Verfügung stehen, oder aber der Kunde fragt an, wie hoch seine Rechnung des laufenden Monats ist. Diese Summenanfragen können als Sicht auf die Datenbank aufgefasst werden, die sinnvollerweise von Anfang an materialisiert vorliegen und durch geeignete Methoden aktuell gehalten werden.

4.2 STREAM

Das STREAM (für STanford stREam datA Management) Projekt [3] realisiert ein integriertes System zur parallelen Verarbeitung von persisitenten Relationen (wie in traditionellen Systemen üblich) und Datenströmen. Anfragen an Datenströme können daher leicht mit in normalen Tabellen gespeicherten Daten verknüpft werden, z.B. über Joins.

Die Besonderheit von STREAM liegt darin, dass speziell eine deskriptive Sprache erschaffen wurde, um den neuen Anforderungen zu genügen: CQL, die Continous Query Language ist eine erweiterte Version von SQL, die auch Anfrage an Ströme ermöglicht. Ströme können auf gleiche Weise referenziert werden wie normale Relationen - der Aufbau einer Anfrage enthält die gleiche Syntax wie in SQL. Um aber zwischen Datenströmen und Relationen Verknüpfungen zu erlauben, führt CQL Operatoren ein, die jeweils eines der beiden Datenmodelle in das andere überführen:

- *Stream-to-relation*-Operator:

  Mit seiner Hilfe können Datenströme wie Relationen behandelt werden. Die Umsetzung erfolgt über das bereits vorgestellte Sliding-Windows-Verfahren. CQL bietet hier die Möglichkeit, diese Sliding-Windows wie richtige Relationen zu behandeln.

  - `SELECT Ort, Temperatur FROM Wetter [ROWS 10]`
    nimmt die letzten 10 Einträge des Stroms "Wetter" und erzeugt daraus eine Relation mit den Feldern Ort und Temperatur.

  - `SELECT Ort, Temperatur FROM Wetter [RANGE 1 Day]`
    erzeugt die gleiche Relation mit den Daten der letzten 24 Stunden.

  Es besteht zusätzlich noch die Möglichkeit, die Sampling-Rate durch das Schlüsselwort SAMPLE festzulegen, um nicht fest die ersten n Zeilen zurückzuerhalten, sondern eine zufällige Auswahl (vgl. Abschnitt 3.7).

- *Relation-to-stream*-Operator:

  Hier kennt STREAM drei verschiedene Operatoren, die aus Relationen wieder Ströme erstellen: RSTREAM, ISTREAM und DSTREAM.

  - *RSTREAM* (für Relation-Stream) nimmt einfach alle in der Relation vorhandenen Tupel und erzeugt aus ihnen wieder einen Datenstrom.

  - *ISTREAM* (für Insert-Stream) erstellt einen Datenstrom, der jedes Mal eine Kopie des neuen Tupels erhält, sobald in der zugrunde liegenden Relation ein Tupel eingefügt wird - jedoch sendet ISTREAM die Tupel nicht, wenn sie keinen neuen Werte enthalten.

- *DSTREAM* (für Delete-Stream) schließlich erhält die jeweils aus einer Relation gelöschten Sätze als Kopie.

Außerdem beherrscht CQL *Relation-to-relation*-Operatoren. Da CQL als reine Erweiterung von SQL entwickelt wurde, liegen grundsätzlich keine Einschränkungen vor. Es ist jedoch anzumerken, dass die aktuelle Implementierung von STREAM noch nicht den vollen Sprachumfang unterstützt.
Auf *Stream-to-stream*-Operatoren wurde verzichtet, zum einen, um die relationale Semantik, die ein Großteil der Nutzer von SQL kennt, nicht zu verletzen, zum anderen natürlich auch, weil sich diese Funktionalität über den Umweg stream-to-relation → relation-to-stream realisieren lässt.

Es bleibt festzuhalten, dass das STREAM-Projekt bereits sehr weit fortgeschritten ist, und dem Nutzer einen relativ intuitiven Umgang mit Strömen ermöglicht. Die Nähe der Anfragesprache CQL zum weit verbreiteten SQL-Standard ermöglicht wohl auch eine rasche Ausbreitung. Ob jedoch CQL tatsächlich einmal Standardsprache für den Umgang mit Strömen wird bleibt fraglich. Denn auch der SQL-Standard wird stetig erweitert. Eine Einflussnahme von CQL auf die Entwicklung von SQL ist jedoch wahrscheinlich.

5 Microsoft, IBM, Oracle

Auch die großen Anbieter relationaler Datenbanken müssen sich den neuen Anforderungen stellen. Zwar waren sie es auch, die bisher an den speziell für bestimmte Anwender zugeschnittenen Systemen verdient haben (vor allem IBM mit Banken als Kunden), jedoch will man natürlich den Anschluss nicht verpassen. Auch sie sehen den Vorteil einer leicht integrierbaren deklarativen Sprache zur Behandlung von Strömen und dem Verzicht auf prozedurale Lösungen.

Oracle hat bereits in seinem Datenbanksystem 8i die in SQL-99 eingebrachte Funktionalität der Analytischen Funktionen (Windowing) umgesetzt. Damit wird zumindest die Möglichkeit gegeben approximative Antworten auf Grundlage aktueller Daten zu geben. Dass die Firma auch die Erkenntnisse im Bereich der materialisierten Sichten einsetzt, wurde bereits weiter oben erwähnt. Außerdem stellt Oracle ihrer neuen 10g Version ihres Datenbankmanagementsystems eine neue Option beiseite: Oracle OLAP.
Die *OLAP* (OnLine Analytic Processing) Funktionalität [20] ermöglicht es schnelle Antworten auf komplexe Anfragen zu liefern. Mit ihr ist es z.B. möglich, Marktentwicklungen zu prognostizieren. Bisher lieferten spezielle Systeme diese Möglichkeit, ab 10g soll dieses Konzept jedoch vollständig in eine relationale Datenbank integrierbar sein.

IBMs DB2 bietet seit neuestem ebenfalls die Möglichkeit materialisierte Sichten zu nutzen. Außerdem gibt es von IBM eine spezielle DB2 Data Warehouse Edition, die dann auch eine optimierte Nutzung von spezieller analytischer Funktionalität vergleichbar mit Oracles OLAP Add-on zur Verfügung stellt.

Den größten Coup will jedoch Microsoft mit ihrem neuen SQL Server 2005 landen, der sich momentan noch im Beta-Status befindet [21]. Er soll fast alle Probleme, die bei der Abarbeitung von Strömen auftreten, lösen können. Dabei setzt man ebenfalls die bereits genannten Materialized Views ein, die jetzt nicht mehr die Einschränkungen in der Funktionalität besitzen sollen wie noch in der 2000er Server Version. Neu wird jetzt auch die Möglichkeit der Nutzung von echtem Partitionieren sein. Dabei will man die Konkurrenz mit wesentlich schnelleren Berechnungen übertrumpfen.

Außerdem soll die Skalierbarkeit des Systems wesentlich vereinfacht werden, was die Möglichkeiten Systeme auszubauen erweitert. Versprochen wird ein sehr sicheres System, das auch bei der Recovery-Time bisherige Leistungen übertreffen soll: So soll beim neuen Server die Datenbank beim Neustart nicht erst nach Abarbeitung der Undos zur Verfügung stehen, sondern bereits zum Beginn.
Weiteres Highlight soll die Online-Index-Änderung und –Wartung sein, die es erlaubt, Indexe zu erstellen, zu löschen oder ein Rebuild auszuführen, während weiterhin die Daten der zugrunde liegenden Tabelle genutzt werden.
Die Nutzung der OLAP Funktionen soll nicht wie bei der Konkurrenz erst nach Kauf eines teueren Zusatzproduktes zur Verfügung stehen, sondern soll bereits in den Server integriert werden.

6 Zusammenfassung

Ich habe hier die grundlegende Problematik des Umgangs mit Strömen präsentiert und aufgezeigt, welche Methoden eingesetzt werden können, um DBMS so zu erweitern, dass sie Ströme verarbeiten können. Dabei ist jedoch offensichtlich geworden, dass noch kein System existiert, das wirklich alle offenen Fragen zufrieden stellend löst. Zum bisherigen Zeitpunkt muss man Oracle hier eine führende Rolle zugestehen, da sie mit der Integration der Konzepte materialisierte Sichten, Partitionierung und Sampling sowie einer Vielzahl für komplexe Auswertungen nutzbarer analytischen Funktionen aufwarten. Gespannt sein darf man auch auf Microsofts neuen SQL Server 2005, der in den Bereichen Partitionierung und materialisierte Sichten zu Oracle aufschließen will und weitere neue Funktionalität einbringen will.
Man kann mit den heutigen DBMS bereits vieles realisieren, was für den Umgang mit Datenströmen unverzichtbar ist. Würden die vorhandenen Konzepte verbessert und würden die in Abschnitt 3 angesprochenen Konzepte komplett in die vorhandenen relationalen DBMS übernommen werden, so könnten auch die konventionellen Systeme eine Basis stellen, um mit Datenströmen umzugehen. Es bleibt festzustellen, dass in diesem Bereich noch sehr viel Forschung und Weiterentwicklung notwendig sein wird, um den traditionellen Systemen einen einfachen und direkten Umgang mit Datenströmen zu ermöglichen.

Literaturverzeichnis

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. B. Zdonik:
Aurora: a new model and architecture for data stream management.
VLDB J. 12(2): 120-139 (2003)
http://www.cs.brown.edu/research/aurora/

[2] S. Chandrasekaran,O. Cooper,A. Deshpande, M. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah:
TelegraphCQ: Continuous Dataflow Processing for an Uncertain World.
Proc. of the Conf. on Innovative Data Systems Research (CIDR) (2003)
http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/

[3] B. Babcock, S. Babu, M. Data, R. Motwani, J. Widom:
Models and Issues in Data Stream Systems.
Proceedings of 21st ACM Symposium on Principles of Database Systems (2002)
http://www-db.stanford.edu/stream

[4] http://de.wikipedia.org/wiki/Datenstrom

[5] http://www.bw.fh-deggendorf.de/kurse/db/skripten/skript11.pdf

[6] http://www.craigsmullins.com/db2_triggers.htm

[7] R. Manthey:
Wie intelligent können Datenbanken sein?
DB Stammtisch Dresden (2003)
http://www.informatik.htw-dresden.de/~wloka/vortraegestammtisch/Manthey101203.pdf

[8] A. Gupta and I. S. Mumick:
Maintenance of materialized views: Problems, techniques, and applications.
IEEE Data Engineering Bulletin, 18(2):3-18 (June 1995)

[9] http://www.ordix.de/onews2/2_2002/siteengine/artikel/sundd_2.html

[10] http://www.databasejournal.com/features/oracle/article.php/2192071

[11] http://www.sqlteam.com/item.asp?ItemID=1015

[12] http://jonathangardner.net/PostgreSQL/materialized_views/matviews.html

[13] A. Gupta, H.V. Jagadish, I.S. Mumick:
Data Integration Using Self-Maintainable Views

[14] D. Quass, A. Gupta, I.S. Mumick, J. Widom:
Making Views Self-Maintainable for Data Warehousing
PDIS 12 18 - 12 (1996)
http://dbpubs.stanford.edu:8090/pub/1996-29

[15] Lukasz Golab:
Querying Sliding Windows over On-Line Data Streams
http://db.uwaterloo.ca/~lgolab/phd-workshop.pdf

[16] K. E. Kline:
SQL in a Nutshell, 2nd Edition

[17] http://msdn.microsoft.com/library/default.asp?URL=/library/techart/PartitionsInDW.htm

[18] http://www.orafaq.com/articles/archives/000060.htm

[19] H. Jagadish, I. Mumick, and A. Silberschatz:
View maintenance issues for the Chronicle data model
Proc. of the 1995 ACM Symp. on Principles of Database Systems, 113-124 (May 1995)

[20] http://www.ncb.ernet.in/education/modules/dbms/SQL99/OLAP-99-154r2.pdf

[21] http://www.hansevision.de/downloads/SQLServer2005.ppt