PostgreSQL & MySQL – Welche Chance haben Open-Source-Datenbanksysteme in Data-Warehouse-Projekten?

Autoren: Milen Kulev und Peter Welker, Trivadis GmbH

PostgreSQL beziehungsweise dessen Abkömmling Bizgres sowie MySQL bieten in den neuen Versionen einige Features, die speziell im Bereich Business Intelligence interessant sind. Genügt das, um etablierte, kommerzielle Produkte zu ersetzen? Wo liegen momentan die Vorteile kostenpflichtiger Lösungen wie Oracle 10*g*?

Besonders aufgrund der hohen Kosten für Software-Lizenzen und Support bei den kommerziellen Datenbanken-Anbietern erfreuen sich Open-Source-Datenbank-Management-Systeme wie PostgreSQL oder MySQL steigender Beliebtheit. Im OLTP-Bereich können sich die kostengünstigen Alternativen mittlerweile durchaus beweisen [1], im Hoheitsgebiet der Business Intelligence – speziell im Data Warehousing hingegen – steht diese Bewährungsprobe noch aus. Ein zunehmendes Interesse ist jedoch bereits zu verzeichnen [2]. Der Artikel stellt PostgreSQL und MySQL kurz vor und betrachtet einige der für Data Warehousing interessanten Features.

PostgreSQL

Die Story beginnt im Jahre 1986 im Rahmen eines auf die Datenbank Ingres folgenden Datenbank-Forschungsprojekts der Universität Berkeley. Erst 1994 wurde das daraus hervorgegangene DBMS um die Abfragesprache SQL (Structured Query Language) erweitert und zugleich dessen Name auf Postgres95 geändert. Nur ein Jahr später wurde nochmals eine Namensänderung durchgeführt. Seitdem heißt die Software PostgreSQL – die Bezeichnung "Postgres" ist aber immer noch gebräuchlich. Dank des freien Quelltextes und einer großen Nutzer- und Entwickler-Gemeinde (PostgreSQL unterliegt den BSD-Lizenzbestimmungen) wird die Datenbank stetig weiterentwickelt. Inzwischen bietet sie hohe Stabilität, ausgezeichnete Performance, einen enormen Funktionsumfang und exzellente Erweiterbarkeit. PostgreSQL läuft auf den meisten Unix-/Linux-ähnlichen Plattformen (insgesamt 34) sowie auf MacOS und Windows XP.

PostgreSQL-Systemarchitektur

PostgreSQL verwendet ein typisches Client-/Server-Modell (siehe Abbildung 1). Die Systemarchitektur ist jener anderer relationaler Datenbank-Management-Systeme wie Oracle ähnlich. Am Beispiel des Anmeldevorgangs wird dies deutlich: Der Client verbindet sich mit dem Postmaster-Prozess (einer Art Listener), der wiederum einen Postgres-Backend-Prozess kreiert. Der Postmaster-Prozess teilt dem Client-Prozess den neuen Port des gerade erzeugten Postgres-Backend-Prozesses mit und der Client-Prozess verbindet sich dann direkt mit diesem Postgres-Backend-Prozess.

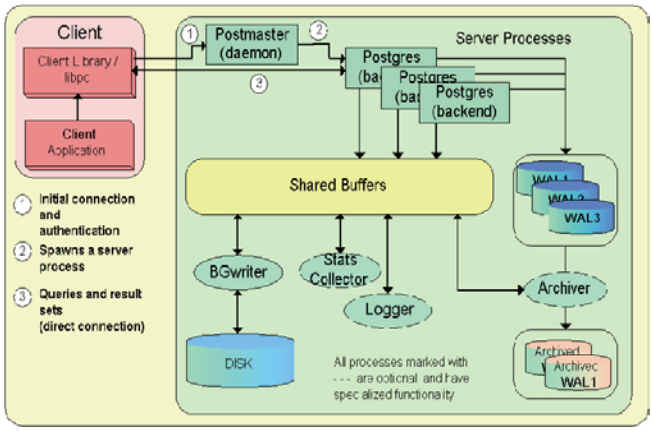

Abbildung 1: Die PostgreSQL-Systemarchitektur

Relevante Features für ein Data Warehouse

PostgreSQL bietet einen breiten Funktionsumfang. In der folgenden Liste finden sich einige Funktionen wieder, die aus unserer Sicht für den Einsatz in Data Warehouses von Bedeutung sind:

- Tabellen-Partitionierung (Constraint Exclusion)
- Eine mächtige Stored-Routines-Sprache namens PL/pgSQL
- Zahlreiche Indextypen
- Eine große Auswahl externer Sprachen wie PL/Ruby, PL/Perl, PL/Python, PL/Java
- Views, RULEs, Triggers, Custom Datatypes
- Schneller Data-Load/Data-Unload
- (Globale) temporäre Tabellen
- Leistungsfähiger, kostenbasierter SQL-Optimizer

Die folgenden Abschnitte beschreiben einige dieser Features.

Partitionierung

PostgreSQL unterstützt eine eingeschränkte Partitionierungsoption mittels Tabellenvererbung und Constraint-Exclusion. Die Idee dahinter:

a) Eine Haupttabelle anlegen, welche die Struktur der vererbten Tabellen bestimmen soll
b) Mehrere Child-Tabellen erben die Struktur der Haupttabelle. Zusätzliche CHECK-Constraints müssen eingerichtet werden, um zu bestimmen, welche Datensätze zu jeder Partition gehören
c) Regeln (Rewrite Macros) oder Trigger werden eingerichtet, um die ankommenden Datensätze auf die richtigen Partitionen zu verteilen
d) Indexe werden auf jede Partition angelegt

Folgende Schritte veranschaulichen diese Vorgehensweise:

```
-- a) create main table
CREATE TABLE orders (ord_date date,
   ord_num   int,    store_id int,
   client_id int,    items    int);

-- b) create as many child tables
--    (=partitions) as you need
CREATE TABLE orders_jan_2006 (
   CHECK(ord_date>='01.01.2006' AND
         ord_date<='31.01.2006'))
   INHERITS(orders);

-- c) create a rule for new rows
CREATE RULE orders_jan06 AS
   ON INSERT TO orders WHERE (
      ord_date>='01.01.2006' AND
      ord_date<='31.01.2006' )
   DO INSTEAD INSERT INTO orders_jan_2006
      VALUES ( NEW.ord_date, NEW.ord_num,
 NEW.store_id, NEW.client_id, NEW.items);

-- d) create index on part key(optional)
CREATE INDEX idx_orders_jan06_1
   ON orders_jan_2006(ord_date);

-- e) example for partition pruning
SELECT count(*) FROM orders
   WHERE ord_date='13.01.2006'
```

Jede dieser "Partitionen" kann weiter (sub-)partitioniert werden:

```
CREATE TABLE orders_jan_2006_store_1
   (CHECK(store_id =1))
   INHERITS(orders_jan_2006)
   TABLEPSPACE slow_tbs;

-- example for sub-partition pruning
SELECT count(*) from orders WHERE
   ord_date='13.01.2006' AND store_id=1;
```

PostgreSQL setzt keine symmetrische (Sub-)Sub-Partitionierung wie Oracle voraus. Das bedeutet, dass die am häufigsten zugegriffenen Child-Tabellen gezielt feiner partitioniert werden können:

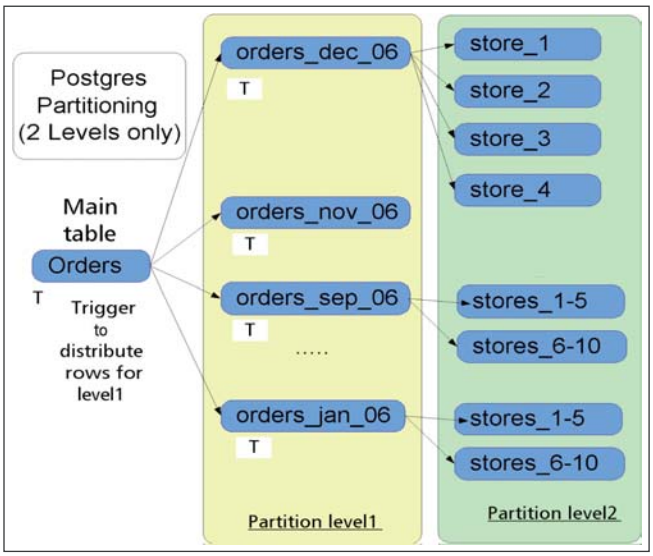

Abbildung 2: Unsymmetrische (Sub-)Partitionierung

PostgreSQL setzt auch keine symmetrische Indizierung der einzelnen Partitionen voraus. Das heißt, die am intensivsten (normalerweise mit den neuesten Daten) verwendeten Sub-Partitionen können aus Performance-Gründen mehr oder andere Indexes haben als die Partitionen, die ältere Daten beinhalten. In der folgenden Tabelle sind kurz die wichtigsten Vor- und Nachteile der PostgreSQL-Partitionierung zusammengefasst:

Vorteile	Nachteile
Partition Elimination* bei Ausführung	Literale müssen verwendet werden wie bspw. WHERE = 100
LIST-, RANGE-Partitioning	Kein Dynamic Partition Pruning*
Sub-Partitioning: Unbegrenzte Partitionierungstiefen und Detailgrade	Keine automatische Datentypenkonvertierung
Partitioning-Syntax ist deklarativ. Transparent für Benutzer und Applikationen	Nur STABLE-Funktionen, die Konstanten zurückliefern, können verwendet werden
Die Child-Tabellen können separat gepflegt werden (Statistiken sammeln, Daten aufräumen, Bulk-DELETE auf Partitionen)	Beträchtlichen Aufwand für das Erstellen notwendiger DDL-Statements (kann aber automatisiert werden)

* Oracle-Terminologie

Spezielle Index-Typen

PostgreSQL verfügt über einige Index-Typen, die auch für den Einsatz im Data Warehouse-Umfeld von Interesse sind:

a) *Partielle Indices*

Ein partieller Index wird über einen Teil einer Tabelle erstellt; dieser Teil wird durch einen Bedingungsausdruck (das so genannte Prädikat des partiellen Indices) bestimmt. Der Index enthält nur Einträge für jene Tabellenzeilen, die das Prädikat erfüllen. Ein Hauptanliegen der partiellen Indices ist es, zu verhindern, häufig vorkommende Werte zu

indizieren. Im Beispiel gibt es vier "Stores" mit folgender Aufteilung (ID = 1 → 1 %, 2 → 49 %, 3 → 49 % und NULL → 1 %):

```
CREATE INDEX idx_orders_store1 ON
    orders(store_id) WHERE store_id=1 ;
```

Dies können auch Multi-Column-Indices sein:

```
CREATE INDEX idx_orders_store1_items ON
   orders(store_id, items)
   WHERE store_id=1 AND items>2;
```

Erstaunlicherweise können auch die NULL-Werte indiziert werden:

```
CREATE INDEX idx_orders_nulls ON
   orders(store_id )
   WHERE store_id IS NULL;
```

Ein partieller Index kann sogar UNIQUE sein:

```
CREATE UNIQUE INDEX idx_uq_ord_store1 ON
   orders(ord_num, client_id)
   WHERE store_id=1 ;
```

b) *Funktionsindices*
Ein Funktionsindex ist – wie bei Oracle – ein Index, der über das Ergebnis einer Funktion, die auf eine oder mehrere Spalten einer Tabelle angewendet wird, definiert ist:

```
CREATE INDEX idx_fbi_cust_l_name ON
   customers(UPPER(l_name));
SELECT f_name || l_name FROM customers
   WHERE UPPER(l_name)='MUSTER';
```

Ein Funktionsindex kann gleichzeitig auch ein partieller Index sein.

Datenverifizierung/Prozedurale Sprachen

Bei PostgreSQL kann man der Datenbank leicht neue Programmiersprachen für Funktionen und Prozeduren hinzufügen. Diese Sprachen werden Prozedurale Sprachen genannt und durch eine Pg/SQL-Funktion gewrapped. Anschließend kann man sie "ganz normal" innerhalb der Datenbank aufrufen. Dabei startet der Postgres-Backend-Prozess im Hintergrund den passenden Interpreter (Perl, Python, Java etc.) und übergibt ihm alle erforderlichen Parameter.

Aus Data-Warehouse-Sicht sind im Bereich Datacleaning ("Reinigen" von Daten) und Verification besonders PL/Python und PL/Perl interessant. Sie bieten exzellente Möglichkeiten der String-Bearbeitung und umfangreiche und ausgereifte Funktionsbibliotheken, die innerhalb PostgreSQL direkt benutzt werden können. Ein Beispiel:

```
CREATE FUNCTION valid_email(text)
   RETURNS BOOLEAN IMMUTABLE
   LANGUAGE plperlu AS $$
# this is PERL!
use Email::Valid;
my $email = shift;
Email::Valid->address($email) or die
   "Invalid email address: $email\n";
return "true";
$$;

CREATE DOMAIN validemail AS text NOT NULL
   CHECK ( valid_email(VALUE) );

CREATE TABLE login (
   username TEXT PRIMARY KEY,
   email VALIDEMAIL );

-- test the verification routine
INSERT INTO login (username, email)
   VALUES('mku', 'milen.kulev@trivadis');

ERROR: error from Perl function: Invalid
email address: milen.kulev@trivadis
```

Materialized Views

PostgreSQL verfügt über keine native Implementierung von Materialized Views. Das PG-Snapshot-Projekt [3] bietet aber eingeschränkte Materialized-View-Funktionalität (FULL- und INCREMENTAL-Refresh, kein automatisches Rewrite), die aber die Installation verschiedener PERL-Module voraussetzt. Ein nettes Feature am Rande: Das PG-Snapshot-Projekt schafft die Möglichkeit, Daten transparent (im Hintergrund, basierend auf Perl-DBI-Modulen) von einer Oracle-Datenbank zu ziehen und synchronisiert in einer PostgreSQL-Datenbank zu halten. Ein Nachteil ist die im Vergleich zu einer nativen Implementierung eher suboptimale Performance.

Das Bizgres-Projekt

Eine speziell für Data-Warehouse-Umgebungen optimierte Version von PostgreSQL liefert das kalifornische Unternehmen Greenplum [4, 5]. Sowohl die kostenfreie Open-Source-Variante als auch der kommerzielle Bruder (Bizgres MPP) bieten zusätzliche, über die bereits erwähnten PostgreSQL-Features hinausreichende Funktionen wie Bitmap-Indices auf Disk, einen verbesserten Sort sowie den KETL-Datenloader.

Bitmap-Indices

Aus Performance-Sicht sind BITMAP-Indices speziell für relationales OLAP (Online Analytical Processing) eine außerordentlich nützliche Erweiterung, die bei der Filterung größerer Datenmengen über die meist riesigen Fakten-Tabellen ihre Vorteile ausspielen. Die Vorteile der Bizgres-BITMAP-Indices gegenüber üblichen BTREE+-Indices lassen sich kurz zusammenfassen:

- CREATE INDEX ist wesentlich schneller
- Die Indices sind bei korrekter Nutzung, sprich geringer Selektivität, viel kleiner
- Die Einzel-Indices können effizient (Bitmap AND/OR) kombiniert werden

Die gegenwärtige Implementierung von BITMAP-Indices lässt aber leider noch keine DML-Operationen zu, das heißt, vor dem Laden der Daten müssen die BITMAP-Indices gelöscht und danach neu erstellt werden. Ein kurzes Beispiel zur Veranschaulichung: Die Tabelle ORDERS hat 100 Millionen Datensätze und ist 6.5 GB groß. Indiziert wird die Spalte Status (mögliche Werte open, closed, in process):

```
CREATE TABLE orders
   (ord_date  date, ord_num int,
    store_id  int,  items   int,
    status    varchar(10));
-- populate table with 100 Mio records...
CREATE INDEX  idx_orders_bmp ON orders
   USING BITMAP (status) ;
```

Index Type / Messungen	BTREE+	BITMAP
Indexaufbauzeit [sec]	2401	504
Indexgröße [MB]	2780	45

KETL-Datenloader

PostgreSQL verfügt über ein COPY-Kommando, das zum ASCII-Importieren/Exportieren von dateibasierten Daten dient. Bizgres bietet zusätzlich einen spezialisierten Datenloader (in Java implementiert) an, der die Defizite des herkömmlichen COPY-Kommandos überwinden soll. Dieser ermöglicht unter anderem:

- Parallele Ausführbarkeit
- Batches/Batch-Memory (Array-Insert aus Buffer)
- Error-Logs und Discard-Logs für gezieltes Fehlerhandling
- Lade-Einstellungen per Konfigurationsfile
- Preview-Mode

Im Großen und Ganzen ähnelt die KETL-Loader-Funktionalität der des SQL*Loaders. Demnach ist KETL nicht – wie der Name vielleicht vermuten lässt – ein komplettes ETL-Werkzeug (Extraktion/Transformation/Load) wie der Oracle Warehouse Builder, sondern eher ein Import-Werkzeug für Dateien. In dieser Kategorie schlägt er sich nicht schlecht und erzielt beispielsweise durch die Nutzung von Parallel-Threads eine optimale Auslastung verfügbarer System-Ressourcen und durch Batches/Batch-Memory (mit häufigeren COMMITs) auch beim Laden großer Datenmengen noch einen guten Durchsatz.

Bizgres MPP und andere Cluster

Das originale PostgreSQL-Projekt bietet selbst keine Möglichkeit, Daten parallel (intra-node oder inter-node) zu verarbeiten. PostgreSQL-Cluster-Lösungen schaffen hier Abhilfe. Sie erlauben die Verteilung der Verarbeitung auf mehrere unabhängige Systeme und dadurch unter anderem die Nutzung höherer Rechenleistung, einen höheren Gesamtdurchsatz auf Datei-Ebene sowie den Einsatz kostengünstiger Hardware (Blades und DirectAcessStorage/JBOD).

Dementsprechend bietet Greenplum für größere Data Warehouses (laut Herstellerangaben ab 200 GB) mit Bizgres MPP (Massive Parallel Processing), auch bekannt als Greenplum Database [6], eine clusterfähige, aber kostenpflichtige Variante an. Sie basiert auf einer verteilten Shared-Nothing-Architektur, bei der jeder Prozessor seinen eigenen Datenspeicher hat und somit jeweils nur für einen Teil der physikalischen Daten verantwortlich ist. Die Daten können dadurch auf viele, weitgehend unabhängige und kostengünstige Maschinen verteilt werden. Die dafür erforderliche Modifikation der Execution-Engine geht in Bizgres MPP aber leider mit Funktionseinschränkungen einher. So gibt es unter anderem weder Global-Temporary-Tables, Foreign-Keys, Triggers CREATE TABLE AS noch korrelierten Unterabfragen (NOT EXISTS, NOT IN). Bizgres MPP unterstützt momentan auch nur die Betriebssysteme RedHat Enterprise Linux 3 Update 4 (x86) und Solaris10 (x86).

Alle PostgreSQL-Cluster-Lösungen, die heute angeboten werden (Extendb [7], Bizgres MPP, PgPool II [8]), setzen im Übrigen auf eine Shared-Nothing-Architektur, da hier keine speziellen und teuren Komponenten wie Clustered-Files-Systeme, Cluster-Manager oder Shared-Storages benötigt werden (siehe Abbildung 3).

Manche dieser Implementierungen (Bizgres MPP, ExtenDB Workgroup Edition) nutzen auch das Konzept von "Agenten". Das sind kleine Programme, die auf jedem DB-Node

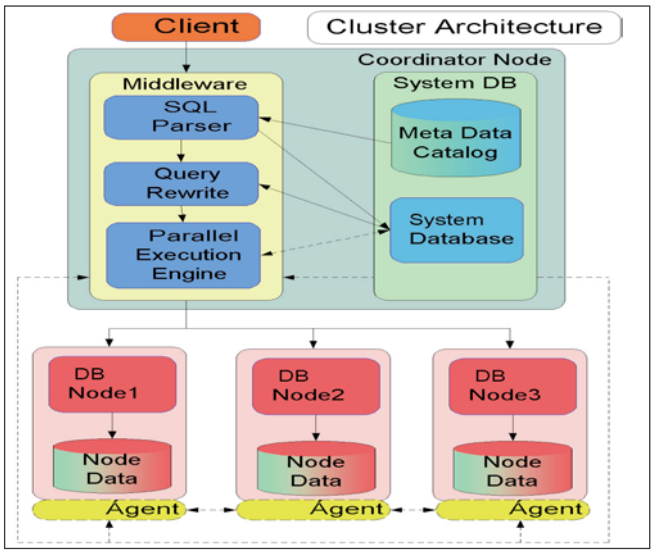

Abbildung 3: Grobe Architektur der PostgreSQL-Cluster-Projekte

laufen und miteinander kommunizieren, um bei Bedarf mit den anderen Knoten des Clusters Daten direkt auszutauschen. Dies dient der Verbesserung des Durchsatzes und der Antwortzeit bei Sortierungen.

Fazit PostgreSQL

Natürlich gilt es in einigen Bereichen durchaus noch aufzuholen. So ist beispielsweise das Fehlen folgender Features gerade für Data Warehouse-Anwendungen nachteilig:

- Dynamic-Partition-Pruning: Werden im Join der Dimensionstabellen mit den partitionierten Fakttabellen die Filter auf Attribute der Dimensionen gelegt, ist kein Partition-Pruning möglich
- Bitmap-Indexe erlauben kein DML
- MERGE-Statement: Dieses kann zwar auch mit INSERTS und UPDATES nachgebildet werden, kostet aber im Vergleich zu einer nativen Implementierung Performance und Implementierungsaufwand
- Keine native Implementierung von Materialized Views
- Keine eingebaute Parallelisierung, was entweder die Verwendung von Cluster-Lösungen nahelegt oder eine manuelle Unterteilung der Arbeitsprozesse außerhalb der DB erfordert
- Jede Cluster-Lösung hat Einschränkungen, die vor einer Auswahl miteinander zu vergleichen sind

Davon abgesehen bieten die Open-Source-Software PostgreSQL und seine – allerdings teilweise kostenpflichtigen – Clones heute neben den üblichen, wichtigen Eigenschaften eines Datenbank-Management-Systems wie hohe Stabilität, Skalierbarkeit und gute Performance auch schon einige Data-Warehouse-relevante Enterprise-Features an, wie sie auch in kommerziellen RDBMS gegen oft recht hohe Kosten verfügbar sind.

MySQL

Das Open-Source-DBMS MySQL wurde 1994 von den Schweden David Axmark und Allan Larsson sowie vom Finnen Michael "Monty" Widenius aus der Taufe gehoben.

1996 kam die erste Version 3.11.1 auf den Markt. Das Produkt wird durch das schwedische Unternehmen MySQL AB [9] entwickelt und im so genannten Dual Licensing Modell vertrieben – auch wenn die Entwicklergemeinde inzwischen nicht mehr nur in Skandinavien sitzt. Benötigt ein Kunde Support und Zertifizierung, erhält er diese mit der zweiten, kommerziellen Variante des MySQL-Licensing-Modells, wobei zwischen der freien und der kommerziellen Variante der Software keinerlei funktionale Unterschiede existieren.

MySQL dürfte inzwischen das am weitesten verbreitete Open-Source-DBMS sein. Laut Herstellerangaben [10] gibt es mittlerweile mehr als zehn Millionen Installationen. Entsprechend umfangreich ist demzufolge die Thematisierung in den gängigen Foren und in der Literatur. Das Produkt gibt es binär für praktisch alle gängigen Linux- und Unix-Derivate (auf unterschiedlichster Hardware), für Windows, MacOS X und NetWare, ja sogar für das Echtzeitbetriebsystem QNX. Zudem ist natürlich der Quellcode verfügbar.

Das aktuelle Produktionsrelease ist die Version 5.0. Die Beta-Version von 5.1 steht jedoch schon seit einigen Monaten zum Download zur Verfügung und bietet speziell mit Partitioning ein sehr wichtiges Feature für Data Warehousing. Darum berücksichtigen wir im Folgenden den Funktionsumfang der Version 5.1.

MySQL-Architektur

Die MySQL-Architektur (siehe Abbildung 4) zeigt einige Besonderheiten. So läuft auf allen unterstützten Plattformen lediglich ein einziger Serverprozess.

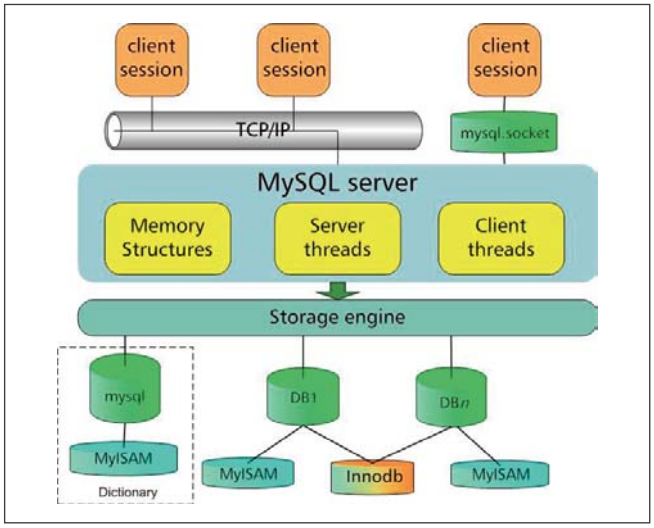

Abbildung 4: Vereinfachte Architektur von MySQL

Die gesamte nebenläufige Verarbeitung (auch der Listener) ist durch Threads innerhalb des Hauptprozesses implementiert. Jeder Client verbindet sich via TCP/IP über den Listener-Thread oder – sofern es sich um einen lokalen Client handelt – effizienter via Sockets (Linux/Unix), Shared-Memory oder Named-Pipes (Windows).

Storage-Engines

Am auffälligsten ist das Konzept der so genannten Storage-Engines. Während andere DBMS lediglich eine einzige Art der Datenspeicherung bieten, kann man mit MySQL je Tabelle eine andere Variante wählen. Storage-Engines beeinflussen aber nicht nur die physische Speicherung von Tabellen und Indices. Sie haben oft auch Auswirkungen auf anderen Gebieten wie Backup, Locking oder Transaktionsverarbeitung. Nachfolgend werden die wichtigsten Storage-Engines genauer betrachtet:

- *MyISAM*
  Dies ist die Default-Storage Engine in MySQL. Nach einer Standardinstallation wird jede neue Tabelle in der MyISAM-Engine erzeugt. MyISAM ist die MySQL-Implementierung der rund 40 Jahre alten ISAM-Methode (Indexed Sequential Access Method) von IBM. ISAM ist eine außerordentlich einfache und daher für bestimmte Operationen auch außerordentlich effiziente Methode und gerade für Data-Warehouse-Fact-Tables somit besonders interessant. Dafür muss man auf der anderen Seite bei Operationen auf Tabellen dieses Typs vollständig auf Transaktionen verzichten. Nur eine schreibende Aktion darf demnach gleichzeitig aktiv sein. Folgerichtig ist nur Table-Locking möglich. Auch die Auswirkung auf ein konsistentes Backup sind offensichtlich: Die Tabelle muss während des gesamten Backup-Vorgangs gelockt bleiben. Oft sind diese Nachteile in Data Warehouses aber durchaus akzeptabel.
- *InnoDB*
  Die gegenwärtig noch wichtigste transaktionsfähige Storage-Engine in MySQL ist Eigentum des finnischen Unternehmens InnoBase Oy. InnoBase wurde zwischenzeitlich allerdings – wie auch die andere transaktionsfähige Engine Berkley DB – von Oracle gekauft. Im Gegensatz zu BDB, dessen Support seitens MySQL ab Version 5.1.12 eingestellt ist, soll InnoDB weiterhin noch "mehrere" Jahre in MySQL verfügbar sein [11]. InnoDB unterstützt ACID mit diversen Isolation Levels. Die Daten liegen in Tablespaces (alternativ geht auch 1 File pro Tabelle), es gibt Logs, Rollbacksegmente, konfigurierbare Caches, Row-Level-Locking, Hot-Backup-Möglichkeiten und auch sonst vieles, was man von einem modernen RDBMS erwarten darf. InnoDB hat aber auch Nachteile. So führt – neben der nicht eben optimalen Verteilbarkeit von Daten-Files und einer inperformanten Freigabe von Speicherplatz in Tablespaces – naturgemäß der durch die Transaktionsfähigkeit verursachte Overhead gegenüber MyISAM zu deutlich schlechteren physikalischen Zugriffszeiten. Für häufig verwendete Daten hingegen, die nicht direkt von Disk gelesen werden müssen, kann InnoDB dank Caching und einiger anderer intelligenter Features (etwa Adaptive-Hash-Indexes) aber durchaus sogar schneller als MyISAM agieren.
- *Memory*
  Die Memory Storage Engine ist – wie der Name schon sagt – eine reine Hauptspeicher-Tabelle. Der Zugriff ist sehr effizient – Transaktionen gibt es aber keine. Wird der Server neu gestartet, sind natürlich alle Daten verschwunden. Diese Engine unterstützt übrigens Hash-Indices. Wie wir noch sehen werden, ist das ein äußerst nützliches Feature – auch im Data-Warehouse-Umfeld.
- *NDB*
  Dies ist die Cluster-Storage-Engine von MySQL. Sie verteilt ihre Tabellendaten (übrigens transparent für alle MySQL-Server-Instanzen) nach einem Hash-Partitioning-Verfahren auf verschiedenen Knoten – falls ge-

wünscht auch bis zu 4-fach redundant. Diese Engine bedingt in Version 5.0 allerdings noch die Haltung aller Daten im Hauptspeicher. Ab Version 5.1 gilt dies nur noch für die Indices.

Es gibt noch unzählige weitere Storage-Engines für MySQL (Federated für den Zugriff auf andere MySQL-Datenbanken, Archive mit Kompression für Daten-Archive, Backhole, CSV, Merge etc.) von MySQL AB selbst oder von Drittherstellern wie SolidDB [12] oder Thinking Networks [13]. Und wer mag und über die entsprechenden Fähigkeiten verfügt, kann sich seine Storage-Engine auch selbst schreiben. Gegenwärtig arbeitet MySQL übrigens an einer eigenen transaktionsfähigen Storage-Engine names Falcon. Ob diese aber schon in Version 5.1 von MySQL enthalten sein wird, ist noch unklar.

MySQL und Data Warehousing

MySQL bietet zudem eine ganze Reihe weiterer Data Warehouse-relevanter Features, von denen wir die ersten vier näher betrachten werden:

- Partitionierung (5.1-Beta)
- INFILE Data-Loader
- Query-Cache
- Scheduler (5.1-Beta)
- Federated-Storage-Engine (für DB-Links)
- Mehrere Index-Typen (auch Fulltext und Spatial)
- Views, Triggers, Stored Routines
- Die Möglichkeit, externe Funktionen einzubinden
- Temporäre Tabellen
- Einen kostenbasierten SQL-Optimizer

Keine Hash-Joins?

Eines fällt sofort auf: MySQL bietet (noch) keine Hash-Joins [14]. Dies führt insbesondere beim Joinen sehr großer Datenmengen und/oder bei Joins ohne Indices zu einem suboptimalen Laufzeitverhalten. Die unterstützten Nested-Loop-Joins sind zwar eine Hilfe – aber kein große. Dennoch sollte man niemals versuchen, große Tabellen ohne Indices zu joinen:

```
-- Keine Indexe (5844 & 1402346 Records)
SELECT COUNT(*) FROM dim JOIN fact
   ON (dim.id = fact.dim_id);
-- ...killed after 1 hour

CREATE INDEX i_dim ON dim(id);

-- Nested Loop möglich - aber langsam
SELECT COUNT(*) FROM dim JOIN fact
   ON (dim.id = fact.dim_id);
...
1 row in set (3.95 sec)
```

Schneller ist da ein manueller Workaround mittels Memory-Storage-Engine. So simuliert man zumindest einen

Hash-Join. Der oben erzeugte Index auf dim kann beim Create Table mit großen Datenmengen übrigens hilfreich sein. Hier ist jedoch noch kein Unterschied sichtbar.

```
CREATE TABLE mem ENGINE=MEMORY AS
   SELECT id FROM dim;
Query OK, 5844 rows affected (0.02 sec)

-- Und jetzt der Join gegen Hash-Indexes
SELECT COUNT(*) FROM mem JOIN fact
   ON (mem.id = fact.dim_id);
...
1 row in set (1.52 sec)
```

Allerdings muss die Memory-Tabelle in den Hauptspeicher passen.

Partitioning

(Symmetrische) Partitionierung ist neu ab Version 5.1. Der implementierte Funktionsumfang ist dafür recht imponierend. Es gibt Range-, List- und Hash-Partitioning (Letzteres in zwei Varianten Hash oder Key) sowie Subpartitioning mit den Haupt-Partitionierungstypen Range oder List und den Sub-Partitionierungstypen Hash oder Key. Nicht nur die Syntax wird dem Leser irgendwie bekannt vorkommen …

Partitioning funktioniert mit den meisten Storage-Engines – auf jeden Fall mit MyISAM und InnoDB. Auch die Partition-Maintenance-Operationen (ADD, DROP, SPLIT etc.) sind verfügbar. Dasselbe gilt auch für Partition-Pruning. Momentan müssen Partitions noch durch Integer-Werte eindeutig definiert sein. Dafür kann man aber leicht Ausdrücke definieren:

```
CREATE TABLE pfact (ord_id  int,
                    ord_dt  date, ...
) PARTITION BY RANGE (YEAR(ord_dt))(
PARTITION p00 VALUES LESS THAN (1992),
PARTITION p00 VALUES LESS THAN (1993),
...
PARTITION p16 VALUES LESS THAN MAXVALUE);
```

Partition Pruning funktioniert dann folgendermaßen:

```
EXPLAIN PARTITIONS
SELECT count(*), sum(price) FROM pfact
WHERE  ord_time_id  BETWEEN  STR_TO_DA-
TE('01.01.1995',     '%d.%m.%Y')     and
STR_TO_DATE('31.12.1995', '%d.%m.%Y');
...
+----+-------+------------+-------+
| id | table | partitions | rows  |
+----+-------+------------+-------+
|  1 | pfact | p04        | 63858 |
+----+-------+------------+-------+
```

Im Gegensatz zu PostgreSQL bietet MySQL sogar eine Art dynamisches Partition-Pruning namens Partition-Selection. Es setzt allerdings die (lokale – eine andere gibt es noch nicht) Indizierung des Foreign-Keys voraus.

INFILE Data-Loader

Dieses Werkzeug erlaubt das Laden von Daten aus Flat-Files in Tabellen direkt auf dem Server. Die Syntax ist sehr einfach und SQL-Funktionen sowie Variablen können leicht genutzt werden. Ein Beispiel (@vord_dt bezeichnet eine Variable, ord_dt ein Feld in stage_orders):

```
LOAD DATA INFILE '/u02/csv/orders.csv'
   INTO TABLE stage_orders
   FIELDS TERMINATED BY '|'
   LINES TERMINATED BY '\n'
   (ord_id, cust_id, @vord_dt, ...)
SET ord_dt =
STR_TO_DATE(@vord_dt,'%d.%m.%Y%H:%i:%s');
Query OK, 250000 rows affected (1.72 sec)
```

Query-Cache

Der Query-Cache ist eine einfache, aber in bestimmten Bereichen sehr effiziente Möglichkeit, auf dem Server mehrfach aufgerufene, identische SQL-Abfragen im Grunde nur einmal auszuführen und stattdessen ein einmal erzeugtes Resultat schlicht mehrfach aus einem Cache zurückzugeben. Dabei dürfen sich die Daten in den zugrunde liegenden Tabellen aber nicht ändern. Die Größe des Caches ist einstellbar:

```
SELECT count(*) from largetab;
1 row in set (58.22 sec)

-- anschließend aus einer anderen Session
SELECT count(*) from largetab;
1 row in set (0.00 sec)
```

Scheduler

Ab Version 5.1 steht in MySQL zudem ein einfacher Scheduler zur Verfügung. Damit erzeugte Events bestehen aus einem Startzeitpunkt, einem optionalen Ende, einem Intervall für Wiederholungen sowie dem auszuführenden Code wie einen Prozeduraufruf:

```
CREATE EVENT e_call_myproc ON SCHEDULE
    EVERY  1 DAY
    STARTS NOW() + INTERVAL 30 MINUTE
    ENDS   NOW() + INTERVAL 4 WEEK
    DO CALL myproc(5, 27);
```

Fazit MySQL

MySQL zeigt sich ab Version 5.1 durchaus für den Einsatz im Data Warehouse-Umfeld gerüstet, wenn auch vor allem das Fehlen von Hash-Joins dem ungetrübten Einsatz mit sehr großen Datenmengen noch gewisse Grenzen setzt und man in einigen Bereichen Dinge wie Bitmap-Indices, Materialized Views, eine inter-node-Parallelisierung oder die fehlende Homogenität beim Backup vermisst. Dafür kann MySQL mit einem flexiblen Storage-Konzept und einigen Enterprise-Features (Partitioning, Clustering) punkten.

Fazit

Die zwei erfolgreichsten Open-Source-DBMS bieten in den neuen (Beta-)Versionen zahlreiche Möglichkeiten, die den Einsatz im Business-Intelligence-Umfeld – in unserer Betrachtung dem Bereich Data Warehousing – rechtfertigen können. So sind beispielsweise beim Thema Partitionierung die Funktionsumfänge der beiden "Freien" so manchem kommerziellen Produkt wie dem MS SQL Server 2005 EE sogar voraus. Der Einsatz von MySQL und PostgreSQL kann demnach zumindest überall dort, wo auf aufwändige Features wie Materialized-Views-Rewrite oder Parallel-SQL verzichtet werden kann, durchaus seine Berechtigung haben. Berücksichtigt man dazu noch den nicht unerheblichen Faktor Lizenzkosten, können die beiden Open-Source-Datenbanken durchaus attraktive Alternativen sein.

Literatur und Links zum Thema

[1] www.computerwoche.de/produkte_technik/software/566847
[2] www.heise.de/open/artikel/73725
[3] pgfoundry.org/projects/snapshot
[4] www.greenplum.com
[5] pgfoundry.org/projects/bizgres
[6] www.greenplum.com/products/bizgresMpp.php
[7] www.extendb.com/
[8] pgfoundry.org/projects/pgpool/
[9] www.mysql.com
[10] www.mysql.de/why-mysql
[11] www.theopenforce.com/2006/04/thank_you_ken_j.html
[12] www.solidtech.com
[13] www.thinking-networks.com
[14] dev.mysql.com/tech-resources/interviews/tkatchaounov.html

Kontakt:
Milen Kulev (PostgreSQL)
milen.kulev@trivadis.com
Peter Welker (MySQL)
peter.welker@trivadis.com