Leistungsoptimierung für das Datenbanksystem MySQL Cluster

Technisches MySQL Whitepaper

Inhalt

1. Einleitung

Unabhängig davon, ob Sie schnell neue Dienste einführen oder eine riesige Datenmenge in Echtzeit verarbeiten möchten, muss Ihre Datenbank skalierbar, schnell und hochverfügbar sein, um den sich ständig wandelnden Marktbedingungen und strikten Service Level Agreements (SLAs) gewachsen zu sein. Die MySQL Cluster Datenbank wurde entwickelt, um genau diese Anforderungen zu erfüllen.

Mit einer verteilten „Shared-Nothing"-Architektur ohne singuläre Fehlerquelle bietet MySQL Cluster 99,999 % Verfügbarkeit, damit Benutzer die Anforderungen ihrer anspruchsvollsten geschäftskritischen Anwendungen erfüllen können.

Das Echtzeit-Design bietet konsistente Antwortlatenzzeiten von wenigen Millisekunden, während gleichzeitig sowohl für lese- als auch für schreibintensive Arbeitslasten zehntausende Transaktionen pro Sekunde verarbeitet werden können. Unterstützung für im Arbeitsspeicher oder auf Festplatte abgelegte Daten, automatische Datenpartitionierung mit Lastverteilung und die Möglichkeit zum Hinzufügen von Knoten zu einem laufenden Cluster ohne Ausfallzeit bieten eine sehr gute Skalierbarkeit der Datenbank, sodass Sie selbst unvorhergesehene Arbeitslasten problemlos verarbeiten können.

Diverse Kunden in der Telekommunikationsbranche, im Bereich der Webanwendungen, im Finanzsektor und im öffentlichen Bereich setzen MySQL Cluster bereits erfolgreich in ihren Datenverwaltungsumgebungen mit höchsten Leistungsanforderungen ein. Dazu zählen u. a. Alcatel-Lucent, Cashpoint, Cisco, Ericsson, Kurier.at, neckermann.de, Telenor und die US-Marine.

Da es sich bei MySQL Cluster um eine verteilte Shared-Nothing-Datenbank handelt, sind bestimmte Anwendungen aufgrund ihrer Merkmale ideal für die Architektur dieser Lösung geeignet. Gleichermaßen gibt es Anwendungen, für die bei Verwendung von MySQL Cluster ein größerer Optimierungsaufwand notwendig ist, um die erforderliche Leistung (gemessen als Transaktionsdurchsatz und Reaktionszeit) zu erzielen.

> *„MySQL Cluster ist für unsere Anwendungen die Datenbank mit der besten Leistung."*
>
> **Peter Eriksson, Manager Network Provisioning**
> **Telenor**

In diesem Whitepaper wird erläutert, wie MySQL Cluster für die Verarbeitung diverser Arbeitslasten optimiert wird. Zunächst werden Muster für den Datenzugriff erläutert und Sie erfahren, wie verteilungsfähige Anwendungen implementiert werden. Anschließend wird beschrieben, wie Schemata, Abfragen und Parameter optimiert werden und wie Sie die aktuellen Innovationen beim Hardware-Design optimal nutzen.

Abschließend werden aktuelle Leistungs-Benchmarks beschrieben, die für die MySQL Cluster Datenbank ausgeführt wurden, und Sie erfahren, wie MySQL Cluster mit anderen MySQL Speicher-Engines kombiniert werden kann. Darüber hinaus umfasst dieses Whitepaper eine Liste weiterer Informationsquellen, um die Leistung von MySQL Cluster zu optimieren.

2. Anwendungen ermitteln, die sich ideal für MySQL Cluster eignen

> *„MySQL Cluster liefert Carrier-Grade-Verfügbarkeit und -Performanz mit linearer Skalierbarkeit auf preisgünstiger Hardware. Es kostet einen Bruchteil der proprietären Alternativen und ermöglicht uns damit einen aggressiven Wettbewerb und den Netzbetreibern die Maximierung ihres durchschnittlichen Erlöses pro Kunde."*
>
> **Jan Martens, Managing Director**
> **SPEECH DESIGN Carrier Systems GmbH**

Um mithilfe von Redundanz und einem schnellen Failover Hochverfügbarkeit zu erzielen, werden die von MySQL Cluster verwalteten Tabellen in Partitionen unterteilt und auf mehrere Knoten innerhalb des Clusters verteilt.

So lässt sich nicht nur eine hochverfügbare, ausfallsichere Umgebung implementieren, sondern gleichzeitig eine Multi-Master-Datenbank mit paralleler Architektur bereitstellen, mit der eine große Anzahl gleichzeitiger Lese- und Schreibvorgänge verarbeitet werden kann. Aktualisierungen sind umgehend für sämtliche Anwendungsknoten (SQL- oder NDB-API) verfügbar, die auf die in den Datenknoten gespeicherten Daten zugreifen.

Da Schreiblasten auf sämtliche Datenknoten verteilt werden, ist mit MySQL Cluster ein sehr hoher Schreibdurchsatz sowie eine umfassende Skalierbarkeit für transaktionale Arbeitslasten möglich. Darüber hinaus kann MySQL Cluster eine Vielzahl an SQL Knoten nutzen, die parallel ausgeführt werden. Da jeder Knoten mit mehreren Verbindungen arbeitet, bietet diese Lösung Unterstützung für transaktionale Anwendungen mit einer großen Anzahl an gleichzeitigen Vorgängen.

Abbildung 1 zeigt die Architektur der MySQL Cluster Datenbank.

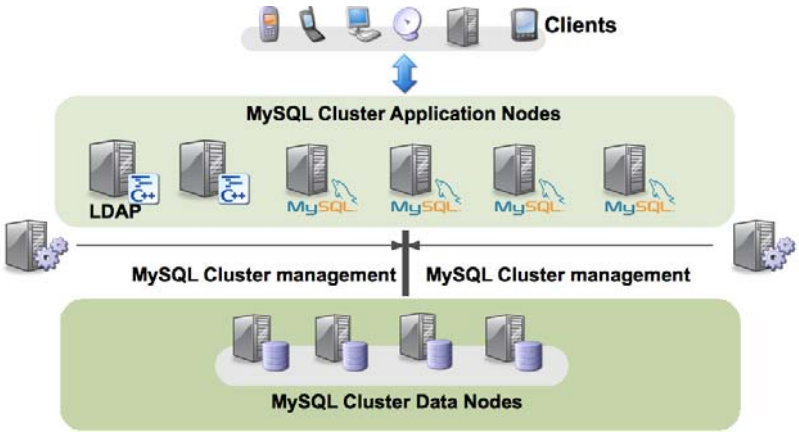


Abbildung 1: ***MySQL Cluster ist eine Multi-Master-Datenbank mit paralleler Architektur***

Weitere Informationen zur MySQL Cluster Architektur finden Sie im MySQL Cluster Architecture and New Features-Whitepaper unter der folgenden Adresse:
http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster7_architecture.php

Aufgrund der verteilten Shared-Nothing-Architektur müssen Sie die Arten von Arbeitslasten bzw. Anforderungen kennen und verstehen, für die MySQL Cluster die ideale Lösung darstellt. In der folgenden Tabelle sind die wichtigsten Aspekte aufgeführt, anhand derer Sie ermitteln können, ob MySQL Cluster Ihre individuellen Anforderungen erfüllt.

Für MySQL Cluster geeignet	Ergänzung durch andere MySQL Speicher-Engines
- Hoher Schreibdurchsatz mit geringer Latenz	- Datenbankgröße von mehr als 2 TB
- 99,999 % Verfügbarkeit mit Failover in Sekundenbruchteilen und automatischer Wiederherstellung	- Zeilen, die mehr als 8 KB Arbeitsspeicher (mit Ausnahme von BLOBs) und 128 Felder erfordern
- Unvorhersagbare Skalierbarkeitsanforderungen	- Notwendigkeit, Objekte mit mehr als 2 MB zu speichern
- Hauptsächlich Primärschlüsselzugriff oder Unterkomponenten von Primärschlüsseln mit mehreren Feldern	- Umfassende Berichterstellung, die vollständige Tabellenscans mit „komplexen“ Joins erfordert
- „Einfache“ Joins	- Fremdschlüsselunterstützung

Abbildung 2: ***Ermitteln der geeigneten Anwendungsbereiche für MySQL Cluster***

Nachfolgend werden verschiedene Beispielanwendungen aufgeführt, welche die oben genannten Punkte erfüllen und für die MySQL Cluster erfolgreich bereitgestellt wurde[1].

Autentifizierung / Authorisierung / Abrechnung	Bereitstellung von mobilen Inhalten
Anwendungsserver	Online-Anwendungsspeicher und -Portale
Persistente Datenhaltung für LDAP / RADIUS / Diameter	Online-Spiele
eCommerce-Dienste (z. B. Warenkorb, Zahlungstransaktionen, Auftragserfüllung usw.)	Zahlungsverarbeitung
IPM-Dienste (IP MultiMedia Subsystem)	Bereitstellungsplattformen für Telekommunikations- & Mehrwertdienste
Intelligente Netzwerkknoten	Teilnehmerdatenbanken (HLR, HSS usw.)
Standortbezogene Dienste und Präsenzverwaltung	VoIP-, IP-Fernseh- und Videoabruf-Dienst
Nachrichtenspeicher/Warteschlangen	Session Management

In den folgenden Abschnitten dieses Whitepapers wird erläutert, wie sich die Leistung der MySQL Cluster Datenbank selbst messen und optimieren lässt, um bei diversen Anwendungsdiensten von den Vorteilen dieser Lösung profitieren zu können.

3. Leistungsmessung und Problemermittlung

Vor der Optimierung sollte eine wiederholbare Methodologie verwendet werden, um die Leistung – sowohl den Transaktionsdurchsatz als auch die Latenz – zu messen. Im Rahmen des Optimierungsprozesses müssen die Änderungen an der Datenbank selbst sowie die Art und Weise berücksichtigt werden, in der die Anwendung die Datenbank verwendet. Daher sollten Leistungsmessungen idealerweise unter Verwendung der tatsächlichen Anwendung mit einem repräsentativen Datenverkehrsaufkommen durchgeführt werden. Die Leistung sollte sowohl vor der Implementierung von Änderungen als auch im Anschluss an die einzelnen Optimierungsschritte gemessen werden, um die durch die Änderung erzielten Verbesserungen (bzw. in einigen Fällen Verschlechterungen) zu überprüfen. In vielen Fällen müssen bestimmte Leistungsanforderungen erfüllt werden. Daher bietet Ihnen ein iterativer Prozess aus Messung, Optimierung und erneuter Messung die Möglichkeit, Ihre Fortschritte beim Erreichen des vorgegebenen Ziels nachzuverfolgen.

Wenn mit der tatsächlichen Anwendung keine Messwerte ermittelt werden können, kann mithilfe des Werkzeugs mysqlslap unter Verwendung mehrerer Verbindungen Datenverkehr generiert werden. Damit diese Tests wiederholbar sind, sollten Sie Dateien mit der anfänglichen Konfiguration (Tabellen und Daten) und eine weitere Datei mit den Abfragen erstellen, die wiederholt ausgeführt werden sollen:

create.sql:

```
CREATE TABLE sub_name (sub_id INT NOT NULL PRIMARY KEY, name VARCHAR(30)) engine=ndb;
CREATE TABLE sub_age (sub_id INT NOT NULL PRIMARY KEY, age INT) engine=ndb;
INSERT                          INTO                              sub_name                          VALUES
(1,'Bill'),(2,'Fred'),(3,'Bill'),(4,'Jane'),(5,'Andrew'),(6,'Anne'),(7,'Juliette'),(8,'Awen'),(
9,'Leo'),(10,'Bill');
INSERT                          INTO                              sub_age                           VALUES
(1,40),(2,23),(3,33),(4,19),(5,21),(6,50),(7,31),(8,65),(9,18),(10,101);
```

query.sql:

```
SELECT sub_age.age FROM sub_name, sub_age where sub_name.sub_id=sub_age.sub_id
AND sub_name.name='Bill' ORDER BY sub_age.age;
```

[1] Klicken Sie hier, um auf eine vollständige Liste mit MySQL Cluster Anwenderberichten und Anwendungen zuzugreifen: http://www.mysql.de/customers/cluster/

Diese Dateien werden dann beim Ausführen von mysqlslap in der Befehlszeile angegeben:

```
shell> mysqlslap --concurrency=5 --iterations=100 --query=query.sql --create=create.sql

Benchmark
    Average number of seconds to run all queries: 0,132 seconds
    Minimum number of seconds to run all queries: 0,037 seconds
    Maximum number of seconds to run all queries: 0.268 seconds
    Number of clients running queries: 5
    Average number of queries per client: 1
```

Wenn eine vorhandene Datenbank verwendet werden soll, geben Sie keine create-Datei an, und qualifizieren Sie Tabellennamen in der query-Datei (z. B. „clusterdb.sub_name“).

Die Liste der mit mysqlslap zu verwendenden Abfragen kann aus dem Log langsamer Abfragen (Slow Query Log: Abfragen, deren Ausführungszeit eine konfigurierbare Zeitdauer überschreitet; Einzelheiten finden Sie weiter unten) oder aus dem allgemeinen Log (alle Abfragen) stammen.

Neben dem Messen der Gesamtleistung Ihrer Datenbankanwendung ist es sinnvoll, auch einzelne Datenbanktransaktionen zu untersuchen. Möglicherweise wissen Sie, dass die Ausführungszeit bestimmter Abfragen zu lang ist, oder Sie möchten Abfragen ermitteln, deren Optimierung einen größtmöglichen finanziellen Nutzen bedeuten würde. In einigen Fällen kann eine Anwendung Mechanismen enthalten, mit deren Hilfe sich Abfragen ermitteln lassen, die am häufigsten ausgeführt werden und/oder die längste Ausführungszeit aufweisen.

Wenn Sie über Zugriff auf MySQL Enterprise Monitor verfügen und für Datenbankabfragen SQL verwenden, können Sie den leistungsfähigen MySQL Query Analyzer nutzen, mit dem sich kostenintensive Transaktionen ermitteln lassen. Bei Verwendung des Query Analyzer befindet sich zwischen der Anwendung und den MySQL Server Knoten ein Proxy, sodass sich der Einsatz dieses Werkzeugs auf die Leistung auswirkt. Eine Testversion von MySQL Enterprise Monitor ist unter der folgenden Adresse verfügbar: http://www.mysql.de/trials/

Der Query Analyzer kann keine Datenbankoperationen untersuchen, die über die NDB-API oder eine Zwischenschicht für den Datenbankzugriff ausgeführt werden (z. B. die JPA-Schnittstelle von MySQL Cluster oder der back-ndb-Treiber für OpenLDAP, der die NDB-API verwendet). Beachten Sie, dass MySQL Enterprise Monitor für MySQL Cluster gegenwärtig von eingeschränktem Nutzen ist, da nur die MySQL Server Knoten, nicht jedoch die Datenknoten überwacht werden können, in denen sich die eigentlich interessanten Ressourcen befinden.

Wenn MySQL Enterprise Monitor nicht verfügbar ist, können die Transaktionen mit langen Ausführungszeiten mithilfe des MySQL Logs langsamer Abfragen (Slow Query Log) ermittelt werden. Beachten Sie, dass die Ermittlung dieser Abfragen nicht immer ausreichend ist, um die Abfragen zu bestimmen, die idealerweise optimiert werden sollten. Sie müssen zudem untersuchen, mit welcher Häufigkeit die verschiedenen Abfragen ausgeführt werden (die Optimierung einer Abfrage mit einer Ausführungszeit von 20 ms, die pro Stunde tausendfach ausgeführt wird, kann sinnvoller sein als die Optimierung einer Abfrage mit einer Ausführungszeit von 10 Sekunden, die pro Tag nur einmal ausgeführt wird).

Sie können mithilfe der Variablen long_query_time festlegen, wie lange die Bearbeitung eine Abfrage dauern muss, um in das Log langsamer Abfragen aufgenommen zu werden. Der Wert dieser Variablen wird in Sekunden angegeben, und bei einem Wert von 0 werden sämtliche Abfragen protokolliert. Bei folgendem Beispiel werden sämtliche Abfragen mit einer Ausführungszeit von mehr als 0,5 Sekunden im Log `/data1/mysql/mysqld-slow.log` erfasst:

```
mysql> set global slow_query_log=1; // Turns on the logging
mysql> set global long_query_time=0.5 // 500 ms
mysql> show global variables like 'slow_query_log_file';
      +---------------------+------------------------------+
      | Variable_name     | Value |
      +---------------------+------------------------------+
      | slow_query_log_file | /data1/mysql/mysqld-slow.log |
      +---------------------+------------------------------+
      1 row in set (0.00 sec)
```

Das folgende Beispiel zeigt einen Eintrag aus der Log-Datei:

```
# Time: 091125 15:05:39
# User@Host: root[root] @ localhost []
# Query_time: 0.017187  Lock_time: 0.000078 Rows_sent: 6  Rows_examined: 22
SET timestamp=1259161539;
select * from sub_name, sub_age where sub_name.sub_id=sub_age.sub_id order by sub_age.age;
```

Hinweis: Änderungen an der Variablen long_query_time (einschließlich der ersten Festlegung, bei der der Standardwert von 10 Sekunden geändert wird) wirken sich nicht auf aktive Clientverbindungen mit dem MySQL Server aus – diese Verbindungen müssen verworfen und erneut hergestellt werden.

Zum Durchsuchen der Log-Inhalte wird ein Werkzeug – mysqldumpslow – bereitgestellt. Es kann jedoch erforderlich sein, die Log-Datei direkt anzuzeigen, um eine ausreichende Genauigkeit für die Zeiten zu erhalten.

Um die Gründe für lange Ausführungszeiten bei Abfragen zu ermitteln, kann als erster Schritt der Befehl EXPLAIN ausgeführt werden, um Details dazu anzuzeigen, wie der MySQL Server die Abfrage ausführt (z. B. welche Indizes gegebenenfalls verwendet werden):

```
mysql> explain select * from sub_name, sub_age where sub_name.sub_id=sub_age.sub_id order by
sub_age.age;

+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------
+
| id | select_type | table | type | possible_keys        | key      | key_len | ref   | rows | Extra       |
+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------
+
|  1 | SIMPLE      | sub_age  | ALL    | PRIMARY,index2 | NULL    | NULL    | NULL                     |    8 | Using filesort
|
|  1 | SIMPLE      | sub_name | eq_ref | PRIMARY,index1 | PRIMARY | 4       | clusterdb.sub_age.sub_id |    1 |
|
+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------
+
2 rows in set (0.07 sec)
```

Das Log langsamer Abfragen (Slow Query Log) und der Befehl EXPLAIN sind nicht auf MySQL Cluster beschränkt und können mit allen MySQL Speicher-Engines verwendet werden.

Wenn das Log langsamer Abfragen nicht ausreicht, kann das allgemeine Log (General Log) aktiviert werden, um eine vollständige Ansicht sämtlicher Abfragen zu generieren, die auf einem MySQL Server ausgeführt werden. Das allgemeine wird wie folgt aktiviert:

```
mysql> set global general_log=1; // Turns on the logging of all queries – only use for a short
period of time as it is expensive!
```

4. Die Leistung von MySQL Cluster optimieren

4.1. Zugriffsmuster

Um für eine MySQL Cluster Bereitstellung eine maximale Leistung zu erzielen, müssen Sie die Datenbankarchitektur verstehen. Im Gegensatz zu den meisten anderen MySQL Speicher-Engines werden die Daten für MySQL Cluster Tabellen nicht durch den MySQL Server gespeichert, sondern partitioniert und auf einen Pool aus Datenknoten verteilt (siehe Abbildung 3). Die Zeilen für eine Tabelle werden in Partitionen unterteilt, und in jedem Datenknoten befindet sich das primäre Fragment für eine Partition sowie ein sekundäres (Backup-) Fragment für eine andere Partition. Wenn zum Abrufen der benötigten Daten bei einer Abfrage eine Vielzahl von Network-Hops vom MySQL Server zu den Datenknoten bzw. zwischen den Datenknoten erforderlich sind, wird die Leistung verringert und die Skalierbarkeit beeinträchtigt.

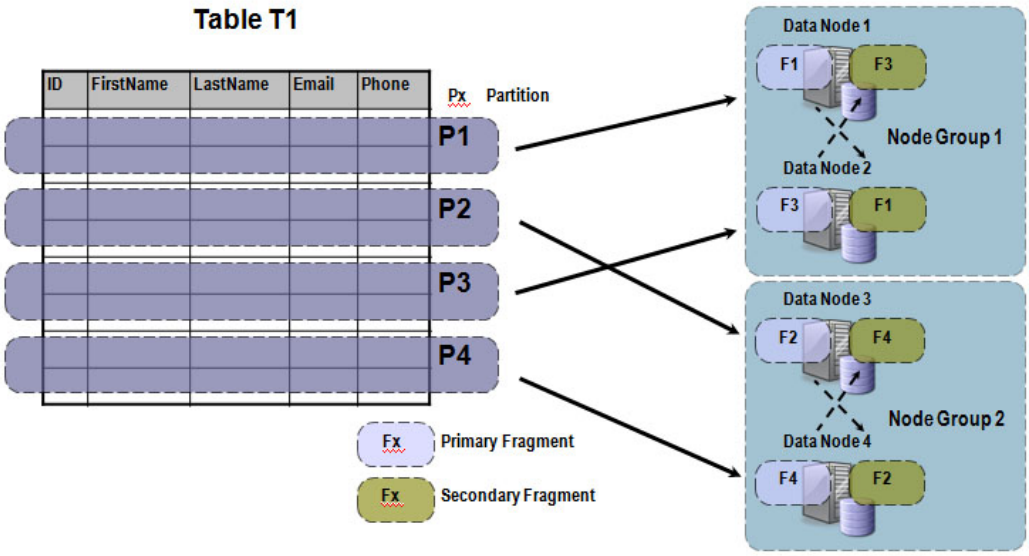


Abbildung 3: ***Partitionierung von MySQL Cluster Daten***

Um für eine MySQL Cluster Datenbank eine bestmögliche Leistung zu erzielen, muss typischerweise die Anzahl an Network-Hops minimiert werden.

Die Partitionierung basiert standardmäßig auf einem Hashing des Primärschlüssels für eine Tabelle. Diese Voreinstellung kann zum Verbessern der Leistung jedoch außer Kraft gesetzt werden, wie im Abschnitt 4.2 „Verteilungsfähige Anwendungen“ beschrieben.

Einfache Zugriffsmuster sind entscheidend, um skalierbare und hochleistungsfähige Lösungen zu erstellen (dies gilt nicht nur für die MySQL Cluster Speicher-Engine, aufgrund der verteilten Architektur ist dies hier jedoch umso wichtiger).

Die beste Leistung lässt sich erzielen, wenn Primärschlüssel-Lookups mit konstanter Dauer ausgeführt werden können (unabhängig von der Datenbankgröße und der Anzahl an Datenknoten). Tabelle 1 zeigt den Zeitaufwand (Zeit in µs) typischer Primärschlüsseloperationen bei Verwendung von 8 Anwendungsthreads, die sich mit einem einzelnen MySQL Server Knoten verbinden.

Primärschlüsseloperationen	Durchschn. Aufwand (µs)	Mindestaufwand (µs)	Normalisiert (skalar)
Insert 4B + 64B	768	580	2,74
read 64B	280	178	1
update 64B	676	491	2,41
Insert 4B + 255B	826	600	2,82
read 255B	293	174	1
update 255B	697	505	2,38
delete	636	202	2,17

Tabelle 1: ***Primärschlüsseloperationen – Kosten***

Diese Ergebnisse zeigen Folgendes:

1. Die Menge der Daten, für die ein Lese- oder Schreibvorgang ausgeführt wird, wirkt sich nur in geringem Maße auf die Ausführungszeit einer Operation aus

2. Operationen, über die Schreibvorgänge ausgeführt werden (einfügen, aktualisieren oder löschen), weisen eine längere Ausführungszeit auf als Lesevorgänge, jedoch nicht in extrem hohem Maße (ca. das 2,2-2,8-Fache). Der Grund für die längere Ausführungszeit bei Schreibvorgängen ist das Zwei-Phasen-Commit-Protokoll, mit dem sichergestellt wird, dass die Daten vor dem Commit der Transaktion sowohl im primären als auch im sekundären Fragment aktualisiert werden.

Indexsuchen dauern bei steigender Anzahl an Tupeln (n) in den Tabellen länger, und eine Latenz von O(log n) ist wahrscheinlich.

Neben der Tabellengröße kann sich auch die Anzahl an Datenknoten auf die Leistung eines Indexscans auswirken. Bei sehr großer Ergebnismenge kann das Verwenden der Verarbeitungsleistung aller Datenknoten zu den schnellsten Ergebnissen führen (dies ist das Standardverhalten). Wenn die Ergebnismenge jedoch verhältnismäßig klein ist, kann das Einschränken des Scans auf einen einzelnen Datenknoten zu einer reduzierten Latenz führen, da weniger Network-Hops benötigt werden. Im Abschnitt 4.2 „Verteilungsfähige Anwendungen“ wird erläutert, wie diese Art von Partition Pruning umgesetzt werden kann.

MySQL Cluster unterstützt Joins, die Leistung kann jedoch geringer sein als bei anderen MySQL Speicher-Engines, wenn die Tiefe des Joins (die Anzahl an Tabellen) zu hoch und die Datensätze zu umfangreich sind.

Wie in Abbildung 4 gezeigt, werden Joins im MySQL Server Prozess (mysqld) verarbeitet, der MySQL Server muss die Daten dann allerdings von den Datenknoten abrufen. Wenn dieser Vorgang nur eine geringe Anzahl an Netzwerk-Paketumläufen umfasst, wird eine hohe Leistung erzielt. Bei einer großen Anzahl an Paketumläufen steigt die Abfragelatenz jedoch.

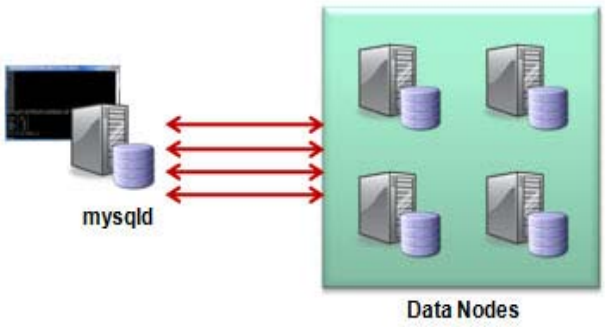


Abbildung 4: ***Network-Hops von Joins***

Die Anzahl der erforderlichen Paketumläufe wird durch die Größe der Ergebnismenge aus jeder Phase des Joins sowie durch die Tiefe der einzelnen Joins beeinflusst. Darüber hinaus steigt bei umfangreichen Tabellen innerhalb eines Joins der Zeitaufwand für die einzelnen Paketumläufe zu den Datenknoten.

Dies lässt sich am besten anhand eines Beispiels veranschaulichen.

Gehen wir von einer sehr einfachen Anwendung für ein soziales Netzwerk aus: es sollen alle Bekannten ermittelt werden, die im aktuellen Monat Geburtstag haben. Zu diesem Zweck wird eine Abfrage ausgeführt, die Daten aus zwei vorhandenen Tabellen verwendet:

```
mysql> describe friends; describe birthday;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| name   | varchar(30) | NO   | PRI | NULL    |       |
| friend | varchar(30) | NO   | PRI |         |       |
+--------+-------------+------+-----+---------+-------+

+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| name  | varchar(30) | NO   | PRI | NULL    |       |
| day   | int(11)     | YES  |     | NULL    |       |
| month | int(11)     | YES  |     | NULL    |       |
| year  | int(11)     | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

Die Abfrage erstellt einen Join zwischen diesen zwei Tabellen. Zunächst werden alle relevanten Zeilen aus der Tabelle „friends“ (bei denen der Name „Andrew“ lautet) ermittelt, anschließend wird für all diese Bekannten in der Geburtstagstabelle überprüft, ob ihr Geburtstag im Monat Juni liegt:

```
mysql> SELECT birthday.name, birthday.day FROM friends, birthday WHERE friend.name='Andrew'
AND friends.friend=birthday.name AND birthday.month=6;
+------+------+
| name | day  |
+------+------+
| Anne |   12 |
| Rob  |   23 |
+------+------+
```

Abbildung 5 zeigt, wie diese Abfrage ausgeführt wird. Der MySQL Server ruft zunächst alle Zeilen aus der Tabelle „friends“ ab, deren Namensfeld mit der Zeichenfolge „Andrew“ übereinstimmt. Anschließend durchläuft der MySQL Server sämtliche Zeilen der Ergebnismenge und sendet für jede Zeile eine neue Anforderung an die Datenknoten, um übereinstimmende Zeilen zu ermitteln, bei denen der Name mit einem bestimmten Bekannten übereinstimmt und der Geburtsmonat Juni lautet (der 6. Monat).

Abbildung 5: *Komplexität von Tabellen-Joins*

Bei diesem Join sollte eine gute Leistung erzielt werden, da die erste Ergebnismenge nur vier Zeilen umfasst und daher insgesamt lediglich fünf Paketumläufe zu den Datenknoten erforderlich sind. Stellen Sie sich jedoch vor, dieser Vorgang würde für einen Benutzer mit äußerst umfangreicher Bekanntenliste ausgeführt, sodass bereits im ersten Teil der Abfrage Tausende Übereinstimmungen ermittelt würden. Wenn die Join-Tiefe zusätzlich um eine weitere Ebene erweitert würde, um in einer Sternzeichentabelle den Geburtstag aller Freunde zu ermitteln, für die eine Übereinstimmung (Geburtstag im Monat Juni) zurückgegeben wurde, würden weitere Netzwerk-Paketumläufe hinzugefügt.

Wenn für Ihre Anwendung Joins ohne Schreibvorgänge erforderlich sind, die bei Verwendung Ihrer primären Datenbank (MySQL Cluster) zu kostenintensiv wären, können Sie die Daten (mithilfe der asynchronen Replikation von MySQL) in eine andere Datenbank replizieren, für die eine besser geeignete Speicher-Engine verwendet wird.

Die Join-Leistung ist ein Bereich, in dem Entwickler gegenwärtig an Neuerungen arbeiten, um in zukünftigen Releases Verbesserungen bereitstellen zu können.

4.2. Verteilungsfähige Anwendungen

Wie bereits erläutert, wird beim Hinzufügen von Zeilen zu einer Tabelle, die MySQL Cluster als Speicher-Engine verwendet, jede Zeile einer Partition zugewiesen, für die ein bestimmter Datenknoten im Cluster verantwortlich ist (siehe Abbildung 3). Die beste Leistung wird häufig erzielt, wenn sich alle für eine Transaktion erforderlichen Daten auf einer einzigen Partition befinden. Das bedeutet, dass die Transaktion mit den Daten innerhalb eines einzelnen Datenknotens abgeschlossen werden kann und kein Wechsel zwischen mehreren Knoten erforderlich ist, der zu zusätzlicher Latenz führen würde.

MySQL Cluster partitioniert die Daten standardmäßig, indem ein Hashing für den Primärschlüssel durchgeführt wird. Wie unten gezeigt, ist dies nicht immer die optimale Vorgehensweise. Gehen wir beispielsweise von zwei Tabellen aus, wobei die erste einen Primärschlüssel mit einer Einzelspalte (sub_id), die zweite einen kombinierten Schlüssel (sub_id, service_name) verwendet:

```
mysql> describe names;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| sub_id | int(11)     | NO   | PRI | NULL    |       |
| name   | varchar(30) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+

mysql> describe services;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| sub_id       | int(11)     | NO   | PRI | 0       |       |
| service_name | varchar(30) | NO   | PRI |         |       |
| service_parm | int(11)     | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
```

Wenn zu diesen (anfänglich leeren) Tabellen nun Daten hinzugefügt werden, kann mithilfe des Befehls EXPLAIN ermittelt werden, welche Partitionen (und damit physischen Hosts) zum Speichern der Daten für diesen einzelnen Nutzer verwendet werden:

```
mysql> insert into names values (1,'Billy');

mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);

mysql> explain partitions select * from names where sub_id=1;
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | names | p3         | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+

mysql> explain partitions select * from services where sub_id=1;
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+------
-+
| id | select_type | table    | partitions  | type | possible_keys | key     | key_len | ref   | rows | Extra
|
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+------
-+
|  1 | SIMPLE      | services | p0,p1,p2,p3 | ref  | PRIMARY       | PRIMARY | 4       | const |   10 |
|
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+------
-+
```

Die Dienstdatensätze für diesen Nutzer (sub_id = 1) werden auf vier verschiedene Partitionen (p0, p1, p2 & p3) verteilt. Die Abfrage hat daher zur Folge, dass Nachrichten zwischen den vier verschiedenen Datenknoten übergeben werden. Und dies wiederum führt dazu, dass mehr CPU-Zeit (Central Processing Unit) verbraucht wird und die Netzwerklatenz steigt.

Das Standardverhalten kann außer Kraft gesetzt werden, indem Sie festlegen, welche Felder des Primärschlüssels MySQL Cluster für den Hash-Algorithmus verwenden soll. In diesem Beispiel kann davon

ausgegangen werden, dass eine Transaktion auf mehrere Datensätze für denselben Nutzer zugreift (diese werden durch ihre sub_id identifiziert). Folglich wird für die Anwendung eine optimale Leistung erzielt, wenn sich sämtliche Zeilen für diese sub_id in derselben Partition befinden:

mysql> drop table services;

```
mysql> create table services (sub_id int, service_name varchar (30), service_parm int, primary
key (sub_id, service_name)) engine = ndb
-> partition by key (sub_id);

mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);

mysql> explain partitions select * from services where sub_id=1;

+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------
+
| id | select_type | table    | partitions | type | possible_keys | key     | key_len | ref   | rows | Extra
|
+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------
+
|  1 | SIMPLE      | services | p3         | ref  | PRIMARY       | PRIMARY | 4       | const |   10 |
|
+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------
+
```

Nun befinden sich alle Zeilen für sub_id=1 aus der Dienstetabelle innerhalb einer einzigen Partition (p3), in der sich auch die Zeile für dieselbe sub_id in der Namentabelle befindet. Beachten Sie, dass das Verwerfen, Neuerstellen und erneute Bereitstellen der Dienstetabelle nicht notwendig war. Mithilfe des folgenden Befehls ließe sich dasselbe Ergebnis erzielen:

```
mysql> alter table services partition by key (sub_id);
```

Dieser Prozess, bei dem ein Indexscan von einem einzelnen Datenknoten ausgeführt werden kann, wird als Partition Pruning bezeichnet. Eine andere Möglichkeit, um zu überprüfen, ob Partition Pruning implementiert wurde, bietet die Statusvariable ndb_pruned_scan_count:

```
mysql> show global status like 'ndb_pruned_scan_count';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Ndb_pruned_scan_count | 12    |
+-----------------------+-------+
select * from services where sub_id=1;
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
|      1 | ssh          |          666 |
|      1 | Video        |          654 |
|      1 | VoIP         |           20 |
+--------+--------------+--------------+

mysql> show global status like 'ndb_pruned_scan_count';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Ndb_pruned_scan_count | 13    |
+-------------------------+-------+
```

Die Erhöhung des Werts um die Zahl 1 bestätigt, dass Partition Pruning möglich war.

Der Nutzen von Partition Pruning hängt von der Anzahl an Datenknoten innerhalb des Clusters und der Größe der Ergebnismenge ab. Die besten Ergebnisse werden bei einer größeren Anzahl an Datenknoten und einer geringeren Anzahl an abzurufenden Datensätzen erzielt.

Abbildung 6 zeigt, wie mithilfe von Partition Pruning (orangefarbene Balken) bei kleineren Ergebnismengen die Latenzzeit reduziert wird, bei größeren Ergebnismengen jedoch längere Latenzzeiten das Ergebnis sein können. Beachten Sie, dass kürzere Balken/geringere Latenzzeiten für eine bessere Leistung stehen.

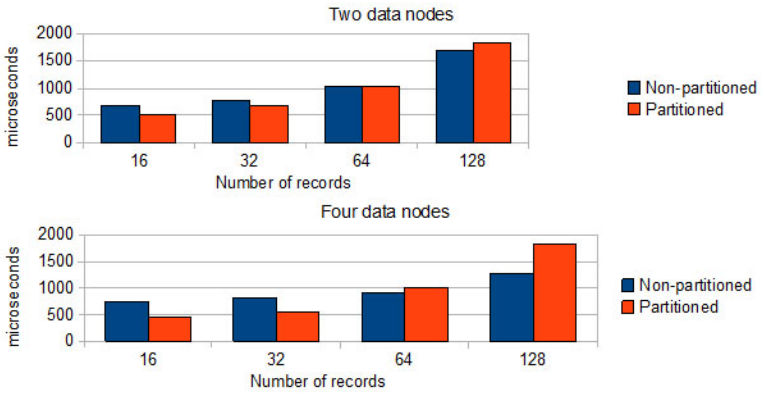


Abbildung 6: ***Auswirkungen von Partition Pruning beim Indexscan***

4.3. Stapelverarbeitung

Stapelverarbeitung kann zum Lesen oder Schreiben mehrerer Zeilen mit einem einzigen Paketumlauf vom MySQL Server zu den Datenknoten eingesetzt werden, um so die Ausführungszeit der Transaktion deutlich zu reduzieren. Es ist möglich, Einfügevorgänge, Indexscans (sofern diese nicht Teil eines Joins sind) sowie Lese-, Lösch- und die meisten Aktualisierungsvorgänge für Primärschlüssel im Stapelmodus auszuführen.

Eine Stapelverarbeitung kann z. B. das Einfügen mehrerer Zeilen unter Verwendung einer einzigen SQL-Anweisung anstelle mehrerer Anweisungen umfassen. Beispiel:

```
mysql> insert into services values (1,'VoIP',20);
mysql> insert into services values (1,'Video',654);
mysql> insert into services values (1,'IM',878);
mysql> insert into services values (1,'ssh',666);
```

Stattdessen kann eine einzige Anweisung verwendet werden:

```
mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);
```

In einem Test, bei dem 1 Million Einfügeanweisungen durch 62.500 Anweisungen zum Einfügen von je 16 Zeilen ersetzt wurden, wurde die Gesamtzeit von 765 auf 50 Sekunden reduziert – eine 15,3-fache Verbesserung! Daher sollte der Stapelmodus stets verwendet werden, wenn dies möglich und sinnvoll ist.

Gleichermaßen können mehrere Select-Anweisungen durch eine einzige Anweisung ersetzt werden:

```
mysql> select * from services where sub_id=1 AND service_name='IM';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
+--------+--------------+--------------+
1 row in set (0.00 sec)

mysql> mysql> select * from services where sub_id=1 AND service_name='ssh';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | ssh          |          666 |
+--------+--------------+--------------+
1 row in set (0.00 sec)

mysql> select * from services where sub_id=1 AND service_name='VoIP';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | VoIP         |           20 |
+--------+--------------+--------------+
```

Diese Anweisungen können durch folgende Einzelanweisung ersetzt werden:

```
mysql> select * from services where sub_id=1 AND service_name IN ('IM','ssh','VoIP');
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
|      1 | ssh          |          666 |
|      1 | VoIP         |           20 |
+--------+--------------+--------------+
```

Operationen für mehrere Tabellen können ebenfalls im Stapelmodus ausgeführt werden, indem Sie den Parameter transaction_allow_batching aktivieren und zwischen den Anweisungen BEGIN und END mehrere Operationen einschließen:

```
mysql> SET transaction_allow_batching=1;
mysql> BEGIN;
mysql> INSERT INTO names values (101, 'Andrew');
mysql> INSERT INTO services VALUES (101, 'VoIP', 33);
mysql> UPDATE services SET service_parm=667 WHERE service_name='ssh';
mysql> COMMIT;
```

transaction_allow_batching kann nicht mit SELECT- oder UPDATE-Anweisungen verwendet werden, durch die Variablen geändert werden.

4.4. Schemaoptimierungen

Es gibt im Wesentlichen zwei Möglichkeiten, um ein Datenschema für eine optimale Leistung zu ändern: Verwendung der effizientesten Typen und gegebenenfalls Denormalisierung des Schemas.

Nur die ersten 255 Byte von BLOB- und TEXT-Spalten werden in der Haupttabelle gespeichert, die übrigen Daten werden in einer ausgeblendeten Tabelle gespeichert. Das bedeutet, dass der von Ihrer Anwendung wahrgenommene einzelne Lesevorgang tatsächlich zwei Lesevorgänge umfasst. Aus diesem Grund lässt sich eine bestmögliche Leistung erzielen, wenn Sie stattdessen die Typen VARBINARY und VARCHAR verwenden. Beachten Sie, dass BLOB- oder TEXT-Spalten weiterhin verwendet werden müssen, wenn die Gesamtgröße einer Zeile anderenfalls 8.052 Byte überschreitet.

Die Anzahl an Lese- und Schreibvorgängen kann durch das Denormalisieren Ihrer Tabellen reduziert werden.

Abbildung 7 zeigt Sprach- und Breitbanddaten, die auf zwei Tabellen verteilt sind, sodass zum Zurückgeben aller Daten des Benutzers mit diesem Join zwei Lesevorgänge erforderlich sind:

```
mysql> SELECT * FROM USER_SVC_BROADBAND AS bb,
USER_SVC_VOIP AS voip WHERE bb.userid=voip.userid
AND bb.userid=1;
```

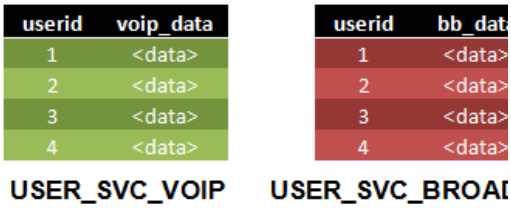


Abbildung 7: ***Normalisierte Tabellen***

Das Ergebnis ist ein Gesamtdurchsatz von 12.623 Transaktionen pro Sekunde (mit einer durchschnittlichen Reaktionszeit von 658 µs) auf einem normalen Linux-basierten Cluster mit zwei Datenknoten (2,33 GHz, Intel E5345 Quad Core – zwei CPUs).

In Abbildung 8 werden diese zwei Tabellen zu einer Tabelle zusammengeführt, sodass der MySQL Server nur einmalig aus den Datenknoten lesen muss:

```
SELECT * FROM USER_SVC_VOIP_BB WHERE userid=1;
```

Das Ergebnis ist ein Gesamtdurchsatz von 21.592 Transaktionen pro Sekunde (mit einer durchschnittlichen Reaktionszeit von 371 µs) – eine 1,71-fache Verbesserung.

userid	voip_data	bb_data
1	<data>	<data>
2	<data>	<data>
3	<data>	<data>
4	<data>	<data>

USER_SVC_VOIP_ BB

Abbildung 8: ***Denormalisierte Tabelle***

4.5. Abfrageoptimierung

Wie im Abschnitt 4.1 “Zugriffsmuster” erläutert, können Joins in MySQL Cluster kostspielig sein. In diesem Abschnitt wird anhand eines Beispiels aufgezeigt, wie ein Join rasch zu sehr hohem Aufwand für MySQL Server führen kann. Anschließend erfahren Sie, wie Sie mithilfe weniger Informationen zur Art und Weise, in der die Anwendung die Daten verwendet, die Leistung signifikant steigern können.

Für unser Beispiel beginnen wir mit 2 Tabellen (siehe Abbildung 9), für die wie dargestellt ein Join ausgeführt wird:

```
SELECT fname, lname, title FROM a,b WHERE b.id=a.id AND a.country='France';
```

authid (PK)	fname	lname	country
1	Albert	Camus	France
2	Sully	Prudhomme	France
3	Johann	Goethe	Germany
4	Junichiro	Tanizaki	Japan

author a

authid (PK)	ISBN (PK)	title
1	1111	La Peste
1	1112	La Chute
1	1113	Caligula
2	2111	La France

authorbook b

Abbildung 9: ***Join mit verschachtelter Schleife***

Diese Abfrage führt für die linke Tabelle einen Indexscan zur Ermittlung von Übereinstimmungen aus und sucht anschließend für jede Übereinstimmung in „a“ nach Übereinstimmungen in „b“. Dies könnte eine äußerst kostenintensive Abfrage zur Folge haben, wenn für `a.country='France'` eine Vielzahl an übereinstimmenden Datensätzen vorhanden sind. Da in diesem Beispiel beide Tabellen über denselben Primärschlüssel verfügen, wäre es vorteilhaft, die Tabellen zu denormalisieren und zu einer einzigen Tabelle zusammenzuführen.

Die Anzahl an Netzwerkanfragen, die der MySQL Server an die Datenknoten sendet, steigt ebenfalls, wenn eine zusätzliche Tabelle zum Join hinzugefügt wird – in diesem Beispiel, um mithilfe der Tabellen in Abbildung 10 zu überprüfen, welche Buchhändler Bücher aus Frankreich auf Lager haben:

```
SELECT c.id FROM a,b,c WHERE c.ISBN=b.ISBN b.id=a.id AND a.country='France' AND c.count>0;
```

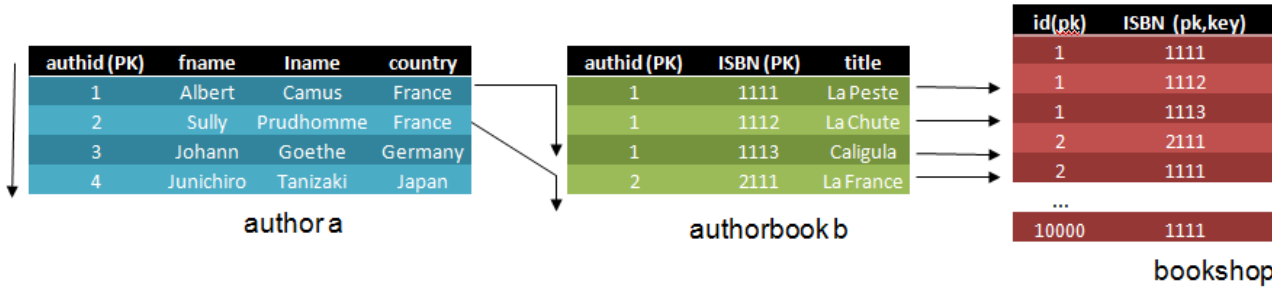


Abbildung 10: *Join mit drei Ebenen*

Für jede ISBN in „authorbook“ ist nun ein Join für die ISBN in „bookshop“ erforderlich. In diesem Beispiel werden in „authorbook“ vier Übereinstimmungen ermittelt, was vier Indexscans für „bookshop“ zur Folge hat – also vier weitere Network-Hops (Kommunikation zwischen Absender und Empfänger). Wenn die Anwendung diese resultierende Latenz tolerieren kann, ist dieses Design akzeptabel.

Ist eine bessere Leistung erforderlich, kann das Schema mithilfe von Informationen zu den eigentlichen Datensätzen geändert werden. Wenn wir z. B. wissen, dass keiner der Autoren mehr als 64 Bücher geschrieben hat, können wir eine Liste mit ISBNs direkt in eine neue Version der Autorentabelle aufnehmen und die Anzahl an Network-Hops mithilfe dieser Pseudo-SQL-Anweisung auf zwei reduzieren (siehe Abbildung 11):

```
SELECT ISBN_LIST FROM a WHERE a.country='France'
SELECT id FROM c WHERE a.isbn in (<ISBN_LIST>);
```

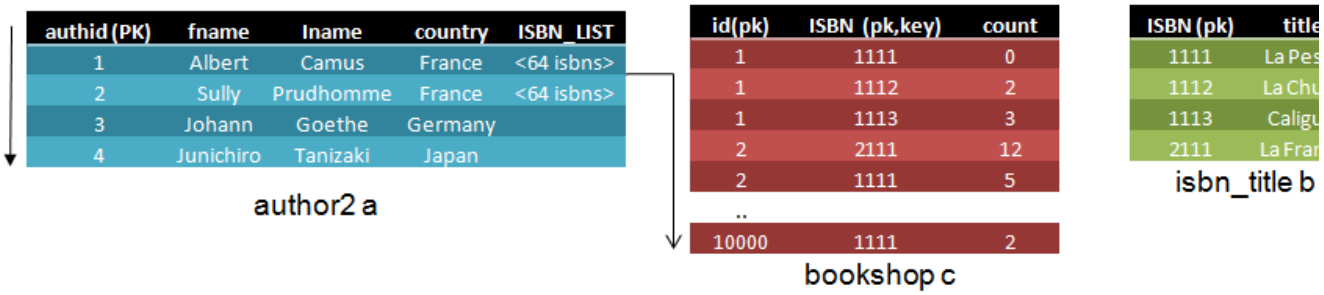


Abbildung 11: *Auf zwei Network-Hops reduzierte Abfrage*

Auf diese Weise werden mit einem Indexscan alle ISBN-Werte für französische Bücher ermittelt, und mithilfe dieser Liste wird anschließend in der „bookshop“-Tabelle ein Einzelscan nach entsprechenden Buchhandlungen ausgeführt.

Als nächster Schritt wird der Pseudo-Code als gespeicherte Prozedur implementiert:

```
CREATE PROCEDURE ab(IN c VARCHAR(255))
BEGIN
  SELECT @list:=GROUP_CONCAT(isbn) FROM author2 WHERE country=c;
  SET @s = CONCAT("SELECT DISTINCT id FROM bookshop WHERE count>0 AND ISBN IN (", @list, ");");
  PREPARE stmt FROM @s;
  EXECUTE stmt;
END
```

Diese gespeicherte Prozedur kann anschließend zum Implementieren des Joins verwendet werden, indem die Zeichenfolge „France“ übergeben wird:

```
mysql> call ab('France');
```

Diese Vorgehensweise führt zu denselben Ergebnissen wie der ursprüngliche Join, ist jedoch 4,5-mal schneller. Da dies eindeutig eine komplexere Lösung mit zusätzlichen Entwicklungskosten ist, entscheiden Sie sich bei akzeptabler Leistung wahrscheinlich für das ursprüngliche Design.

Um zu ermitteln, ob Sie die Anzahl der vom MySQL Server an die Datenknoten gesendeten Anforderungen tatsächlich senken, können Sie prüfen, um welchen Wert die Statusvariable Ndb_execute_count erhöht wird (diese Variable zählt die Anzahl an Anforderungen, die vom MySQL Server an die Datenknoten gesendet werden):

```
mysql> show global status like 'Ndb_execute_count';
mysql> show global status like 'Ndb_execute_count';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Ndb_execute_count | 9     |
+-------------------+-------+
```

Zusätzlich zum Vereinfachen oder Entfernen von Joins können Sie durch die richtige Verwendung von Indizes Abfragen beschleunigen. Messen Sie beim Hinzufügen von Indizes den Nutzen, um zu überprüfen, ob sie gerechtfertigt sind. Mithilfe des Befehls EXPLAIN kann überprüft werden, ob der richtige Index verwendet wird.

Beispielsweise wird eine Tabelle mit zwei Indizes für die Spalte „a“ erstellt. Beim Überprüfen, wie eine SELECT-Anweisung ausgeführt würde, werden beide Indizes als Optionen angegeben. Da bekannt ist, mit welchem Index für diese Abfrage wahrscheinlich die beste Leistung erzielt wird, können wir die Verwendung dieses Indexes erzwingen:

```
mysql> CREATE TABLE t1(id int(11) NOT NULL AUTO_INCREMENT,
                       a bigint(20) DEFAULT NULL,
                       ts timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,

                       PRIMARY KEY (id),
                       KEY idx_t1_a (a),
                       KEY idx_t1_a_ts (a,ts)) ENGINE=ndbcluster DEFAULT CHARSET=latin1;
mysql> explain select * from t1 where a=2 and ts='2009-10-05 14:21:34';
+----+-------------+-------+------+----------------------+----------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys        | key      | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+----------------------+----------+---------+-------+------+-------------+
|  1 | SIMPLE      | t1    | ref  | idx_t1_a,idx_t1_a_ts | idx_t1_a | 9       | const |   10 | Using where |
+----+-------------+-------+------+----------------------+----------+---------+-------+------+-------------+

mysql> explain select * from t1 FORCE INDEX (idx_t1_a_ts) where a=2 and ts='2009-10-05 14:21:34';
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+-------------
--
| id | select_type | table | type | possible_keys | key         | key_len | ref         | rows | Extra
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+-------------
--
|  1 | SIMPLE      | t1    | ref  | idx_t1_a_ts   | idx_t1_a_ts | 13      | const,const |   10 | Using where
wit
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+-------------
--
```

4.6. Parameteroptimierung

Um mit einer bestimmten Konfiguration eine bestmögliche Leistung zu erzielen, können verschiedene Parameter gesetzt werden. Die verschiedenen Parameter sind in der MySQL Cluster Dokumentation beschrieben, als Ausgangspunkt sollten Sie jedoch das unter www.severalnines.com/config verfügbare Werkzeug verwenden, um bekanntermaßen gut funktionierende Werte zu erhalten, die Sie an Ihre spezifischen Anforderungen anpassen können.

4.7. Verbindungspools

Zur Skalierung Ihrer Anwendung verfügen Sie möglicherweise über mehrere Threads, die jeweils mit einem MySQL Server Prozess (mysqld) verbunden sind (siehe Abbildung 12). Für den Zugriff auf die Datenknoten verwendet der mysqld-Prozess standardmäßig eine einzige NDB-API-Verbindung. Da mehrere Threads versuchen, diese einzige Verbindung zu nutzen, tritt für einen Mutex ein Konflikt auf, der eine Verbesserung des Durchsatzes verhindert. Dasselbe Problem kann bei direkter Verwendung der NDB-API auftreten (siehe „Alternative APIs“).

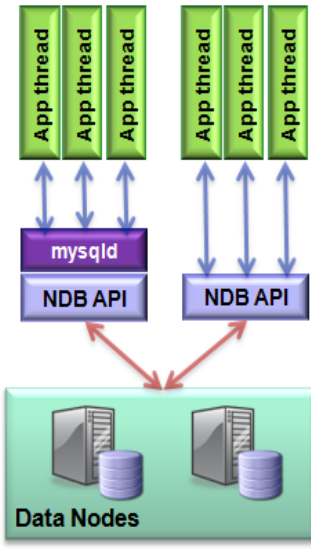


Abbildung 12: ***API-Threadkonflikt***

Eine Umgehungslösung ist die Verwendung eines dedizierten mysqld-Prozesses durch jeden Anwendungsthread. Dies würde jedoch eine Verschwendung von Ressourcen bedeuten und zu einem erhöhten Verwaltungsaufwand führen (beispielsweise müssen für jeden mysqld-Prozess separat Benutzerberechtigungen festgelegt werden).

Eine effektivere Lösung ist die Verwendung mehrerer NDB-API-Verbindungen vom mysqld-Prozess zu den Datenknoten, wie in Abbildung 13 gezeigt.

Zur Verwendung eines Verbindungspools muss jede Verbindung (vom mysqld-Prozess) über einen eigenen [mysqld]- oder [api]-Abschnitt in der Datei config.ini verfügen, und ndb-cluster-connection-pool muss in my.cnf auf einen Wert größer als 1 gesetzt werden:

```
config.ini:
==========
[ndbd default]
noofreplicas=2
datadir=/home/billy/mysql/my_cluster/data

[ndbd]
hostname=localhost
id=3

[ndbd]
hostname=localhost
id=4

[ndb_mgmd]
id = 1
hostname=localhost
datadir=/home/billy/mysql/my_cluster/data

[mysqld]
hostname=localhost
id=101
#optional to define the id

[api]
# do not specify id!
hostname=localhost

[api]
# do not specify id!
hostname=localhost

[api]
# do not specify id!
hostname=localhost

my.cnf:
======

[mysqld]
ndbcluster
datadir=/home/billy/mysql/my_cluster/data
basedir=/usr/local/mysql
ndb-cluster-connection-pool=4
# do not specify an ndb-node-id!
```

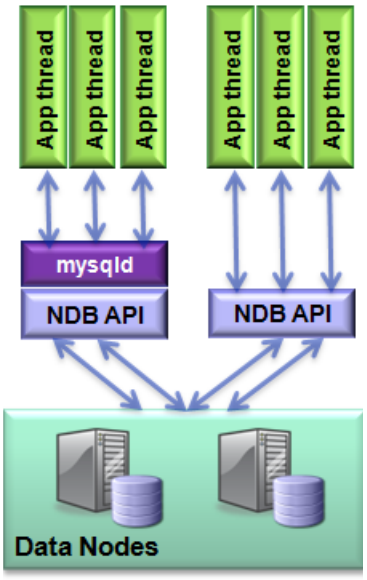


Abbildung 13: ***Pool mit mehreren Verbindungen***

Geben Sie zudem keine ndb-node-id als Befehlszeilenoption an, wenn der mysqld-Prozess gestartet wird. In diesem Beispiel verfügt der einzelne mysqld-Prozess über vier NDB-API-Verbindungen.

Abbildung 14 zeigt die Leistungsvorteile bei Verwendung eines Verbindungspools. Für Anwendungen wird typischerweise eine Verbesserung von 70 % erzielt, in einigen Fällen sogar von mehr als 150 %

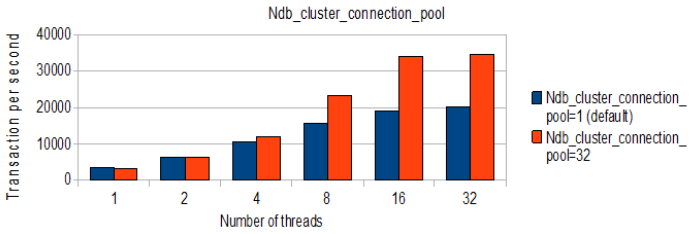


Abbildung 14: ***Vorteile eines Verbindungspools***

4.8. Multithread-Datenknoten

Multithread-Datenknoten bieten die Möglichkeit zur vertikalen Skalierung auf umfangreichere Computerhardware, da die Datenknoten mehrere CPU-Kerne/-Threads in einem einzelnen Serverhost besser nutzen können. Jeder dieser Datenknoten kann bis zu acht CPU-Kerne/-Threads effektiv nutzen. Daten werden partitioniert und auf mehrere Threads innerhalb des Datenknotenprozesses verteilt. Dadurch wird die partitionierte MySQL Cluster Architektur innerhalb des Datenknotens dupliziert. Bei diesem Design wird die Leistung der einzelnen Threads maximiert, und die Kommunikation der Threads wird minimiert.

Um dieses Ziel zu erreichen, wird der lokale Abfrage-Handler (Local Query Handler, LQH) im Multithread-Modus ausgeführt, während andere wichtige Aktivitäten (Transaktionskoordination, Replikation, E/A und Transporter) in separaten Threads ausgeführt werden.

Jeder LQH-Thread ist für eine primäre Unterpartition (ein Teil der Partition, für die ein bestimmter Datenknoten verantwortlich ist) und eine Replikunterpartition verantwortlich (sofern NoOfReplicas auf den Wert 2 gesetzt ist). Der Transaction Coordinator für einen Datenknoten ist für die Weiterleitung von Anforderungen an den richtigen LQH-Thread innerhalb des Datenknotens sowie bei Bedarf an andere Datenknoten verantwortlich.

Abbildung 15 zeigt diesen Vorgang für ein System mit acht Kernen, wobei vier Threads zur Ausführung der Instanzen des lokalen Abfrage-Handlers verwendet werden.

Der Transaction Coordinator koordiniert Transaktionen und Timeouts und wird als Schnittstelle zur NDB-API für Indizes und Scanoperationen genutzt.

Der Access Manager ist für Hash-Indizes von Primärschlüsseln verantwortlich, um einen schnellen Zugriff auf die Datensätze zu bieten.

Der Tuple Manager ist für das Speichern von Tupeln (Datensätzen) verantwortlich und umfasst die zum Filtern von Datensätzen und Attributen eingesetzte Filterungs-Engine.

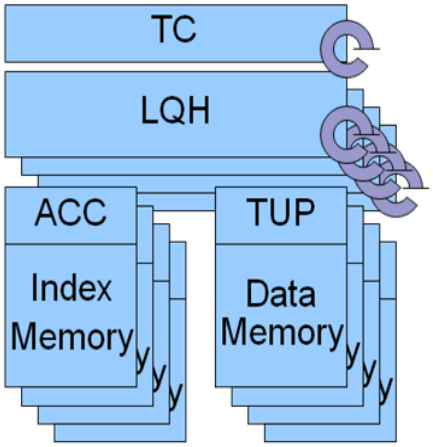


Abbildung 15: ***Multithread-Datenknoten***

Den vier LQH-Threads wiederum werden vom einzelnen TC-Thread (Transaction Coordinator) Aktivitäten zugewiesen. Der Access Manager (ACC) und der Tuple Manager (TUP) werden innerhalb der einzelnen LQH-Threads ausgeführt.

Die Leistung kann bei Hosts mit 8 Kernen um mehr als das 4-Fache gesteigert werden.

ndbmtd ist das Äquivalent von ndbd, wird jedoch für Multithread-Datenknoten eingesetzt. Dieser Prozess verarbeitet sämtliche Tabellendaten unter Verwendung des NDB-Datenknotens von MySQL Cluster. ndbmtd wurde für den Einsatz auf Hostcomputern mit mehreren CPU-Kernen/-Threads entwickelt.

Die Funktionsweise von ndbmtd ist praktisch vollständig mit der Funktionsweise von ndbd identisch, sodass die Änderung für die Anwendung nicht relevant ist. Die mit ndbd verwendeten Befehlszeilenoptionen und Konfigurationsparameter können ebenfalls mit ndbmtd eingesetzt werden. ndbmtd bietet zudem Dateisystemkompatibilität mit ndbd. Mit anderen Worten, Sie können einen Datenknoten, der ndbd ausführt, anhalten, die Binärdatei durch ndbmtd ersetzen und den Datenknoten ohne Datenverlust neu starten. So können Entwickler oder Administratoren völlig problemlos zur Multithread-Version wechseln.

Bei Verwendung von ndbmtd anstelle von ndbd müssen zwei Unterschiede beachtet werden:

- Sie müssen einen geeigneten Wert für den Konfigurationsparameter MaxNoOfExecutionThreads in der Datei config.ini setzen. Anderenfalls wird ndbmtd im Einzelthread-Modus ausgeführt, weist also dasselbe Verhalten wie ndbd auf.
- Ablaufverfolgungsdateien werden für kritische Fehler in ndbmtd-Prozessen etwas anders generiert als durch ndbd-Fehler.

Der Konfigurationsparameter MaxNoOfExecutionThreads wird zum Bestimmen der Anzahl an Ausführungsthreads verwendet, die von ndbmtd erzeugt werden. Wenngleich dieser Parameter im Abschnitt [ndbd] oder [ndbd default] der Datei config.ini gesetzt wird, gilt er nur für ndbmtd und wird für ndbd ignoriert. Für diesen Parameter kann eine Ganzzahl zwischen 2 und 8 angegeben werden. Im Allgemeinen sollten Sie diesen Wert auf die Anzahl an CPU-Kernen/-Threads auf dem Datenknotenhost setzen, wie in Tabelle 2 gezeigt.

Anzahl an Kernen	Empfohlener MaxNoOfExecutionThreads-Wert
2	2
4	4
8	8

Tabelle 2: *Geeignete Einstellungen für MaxNoOfExecutionThreads*

Selbstverständlich hängt die Leistungssteigerung, die mit dieser Funktion erreicht werden kann, von der Anwendung ab. Gehen wir z. B. von einer einzigen Anwendungsinstanz mit Einzelthread aus, die einfache Lese- oder Schreibtransaktionen über eine einzige Verbindung an den Cluster sendet. In diesem Fall ließe sich durch eine parallele Konfiguration für die Datenknoten kein höherer Durchsatz erzielen. Wenn Änderungen an der Anwendung also nicht erforderlich sind, kann eine Überarbeitung dazu beitragen, eine maximale Verbesserung zu erzielen.

4.9. Alternative APIs

Der bisherige Schwerpunkt dieses Whitepapers lag auf der Leistung beim Zugriff auf MySQL Cluster Daten unter Verwendung von SQL über den MySQL Server. Die bestmögliche Leistung kann erzielt werden, indem der Zugriff auf die Datenknoten direkt über die C++-NDB-API erfolgt. Dies erfordert bei Entwicklung und Wartung Ihrer Anwendung eine größere Investition, als Ergebnis lassen sich jedoch ein höherer Durchsatz und deutlich niedrigere Latenzzeiten erzielen. Auch eine kombinierte Verwendung ist möglich, bei der Sie SQL für den Großteil der Anwendung, für bestimmte, leistungskritische Aspekte jedoch die NDB-API verwenden. Beachten Sie, dass MySQL Server die NDB-API intern für den Zugriff auf die Datenknoten verwendet.

Um von den Vorteilen beider Ansätze zu profitieren – einer höheren Leistung und einfachen Wartung –, stehen immer mehr Middleware-Komponenten zur Verfügung, die die NDB-API direkt verwenden. Ein Beispiel ist der back-ndb-Treiber für OpenLDAP, mit dem Sie in MySQL Cluster mithilfe des LDAP-Protokolls Lese- und Schreibvorgänge für Daten ausführen können. MySQL wird eine JPA-kompatible Java-Schnittstelle anbieten, die eine optimale Methode für hochleistungsfähige Java-Anwendungen bietet, um MySQL Cluster zu verwenden.

4.10. Hardwareverbesserungen

Der Schwerpunkt dieses Whitepapers lag hauptsächlich auf der Verringerung der Anzahl an Nachrichten, die zwischen den Knoten eines Clusters gesendet werden. Ein ergänzender Ansatz ist die Verwendung von Netzwerk-Interconnects mit größerer Bandbreite und geringerer Latenz. Die DX-Familie der von Dolphin Interconnect Solutions (www.dolphinics.com) bereitgestellten SCI-Produkte hat sich beim Einsatz mit MySQL Cluster bewährt und ermöglicht ohne weitere Optimierungsaufgaben eine mehr als 2-fache Leistungssteigerung bei netzwerkintensiven Arbeitslasten.

Schnellere CPUs oder CPUs mit mehr Kernen oder Threads können zu einer verbesserten Leistung führen. Wenn Sie Datenknoten auf Hosts mit mehreren CPUs/Kernen/Threads bereitstellen, sollten Sie einen Wechsel auf den Multithread-Datenknoten in Betracht ziehen, der mit MySQL Cluster 7.0 und höher verfügbar ist. Die

Datenträgerleistung kann stets zu Einschränkungen führen (selbst bei Verwendung von speicherresidenten Tabellen). Daher wurden bei Verwendung von SSD-Speichern (Solid State Device) bei steigender Datenträgerleistung bessere Ergebnisse erzielt.

4.11. Verschiedenes

Die Verwendung des MySQL Abfragen-Caches ist für MySQL Cluster selten sinnvoll, sodass seine Deaktivierung empfohlen wird.

Stellen Sie sicher, dass die Datenknoten nicht SWAP-Speicher anstelle des tatsächlichen Arbeitsspeichers verwenden. Dies beeinträchtigt sowohl die Leistung als auch die Systemstabilität.

Schränken Sie Datenknotenthreads bei Verwendung von Ethernet für die Interconnects auf CPUs ein, die nicht für Netzwerk-Interrupts verantwortlich sind. Erstellen Sie bei Verwendung von Sun CMT Hardware (Chip Multi-Threading) Prozessorgruppen, und binden Sie die Interrupt-Verarbeitung an bestimmte CPUs. Threads können mithilfe der Parameter LockExecuteThreadToCPU und LockMaintThreadsToCPU im Abschnitt [ndbd] der Datei config.ini auf einzelne CPUs beschränkt werden.

5. Skalieren von MySQL Cluster durch das Hinzufügen von Knoten

MySQL Cluster kann horizontal skaliert werden. Die Leistung – insbesondere der Transaktionsdurchsatz des Systems – lässt sich häufig steigern, indem Sie einfach zusätzliche MySQL Server oder Datenknoten hinzufügen.

Zum Hinzufügen weiterer MySQL Server Knoten fügen Sie einfach einen zusätzlichen Abschnitt [mysqld] zur Datei config.ini hinzu, führen einen Rolling-Neustart aus und starten den neuen mysqld-Prozess. Daten, die nicht in der MySQL Cluster Speicher-Engine gespeichert sind (z. B. gespeicherte Prozeduren und Benutzerberechtigungen) müssen auf dem neuen Server neu erstellt werden. Während die neuen Server hinzugefügt werden, stellen die anderen Server weiterhin Dienste bereit.

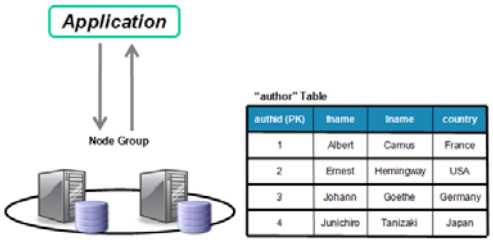


Abbildung 16: ***Ausgangskonfiguration***

Neue Knotengruppen (und damit neue Datenknoten) können zu einem laufenden MySQL Cluster hinzugefügt werden, ohne dass der Cluster heruntergefahren und neu gestartet werden muss. Zudem können die vorhandenen Tabellendaten neu partitioniert und auf alle Datenknoten verteilt werden (eine Untermenge der Daten aus den vorhandenen Knotengruppen wird in die neue Gruppe verschoben).

In diesem Abschnitt werden anhand eines Beispiels die erforderlichen Schritte erläutert, um einen Cluster mit einer einzelnen Knotengruppe zu erweitern, indem eine zweite Knotengruppe hinzugefügt und die Daten neu partitioniert und auf die zwei Knotengruppen verteilt werden. Abbildung 16 zeigt das ursprüngliche System.

In diesem Beispiel wird davon ausgegangen, dass die zwei Server der neuen Knotengruppe bereits in Betrieb genommen wurden. Der Schwerpunkt liegt daher auf dem Hinzufügen der Gruppe zum Cluster.

Schritt 1: Bearbeiten Sie config.ini auf allen Verwaltungsservern, wie in Abbildung 17 gezeigt.

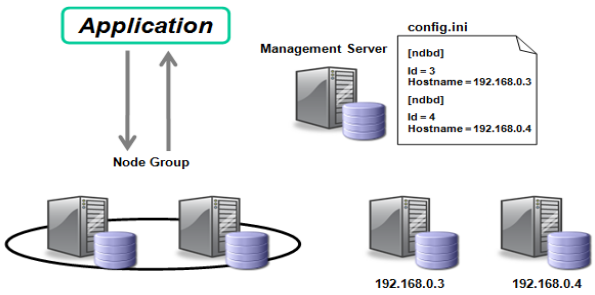


Abbildung 17: ***Aktualisieren von „config.ini“ auf allen Verwaltungsknoten***

Schritt 2: Starten Sie die Verwaltungsknoten, vorhandenen Datenknoten und MySQL Server neu, wie in Abbildung 18 gezeigt. Beachten Sie, dass Sie für jeden Neustart auf eine Meldung zum erfolgreichen Abschluss dieses Vorgangs warten müssen, bevor der nächste Knoten neu gestartet werden kann, damit die Dienste nicht unterbrochen werden.

Zusätzlich sollten alle MySQL Server Knoten neu gestartet werden, die auf den Cluster zugreifen.

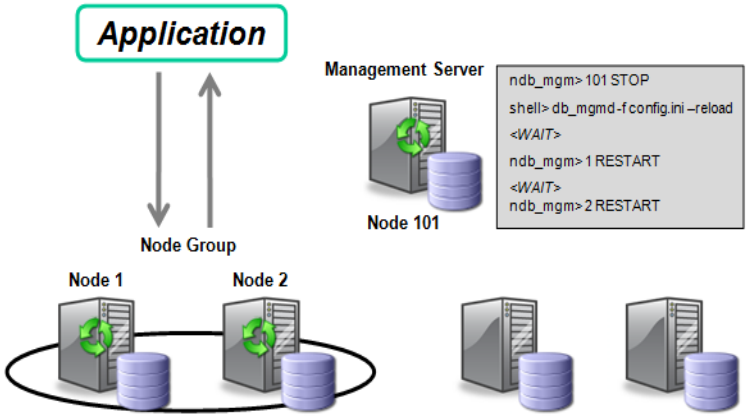


Abbildung 18: ***Durchführen eines Rolling-Neustarts***

Schritt 3: Erstellen Sie die neue Knotengruppe.

Die zwei neuen Datenknoten können nun neu gestartet werden. Anschließend können Sie die Knotengruppe mit diesen Datenknoten erstellen, um diese zum Cluster hinzuzufügen (siehe Abbildung 19).

Beachten Sie, dass in diesem Fall nicht auf den Start von Node 3 gewartet werden muss, um Node 4 zu starten. Sie müssen jedoch auf den abgeschlossenen Start beider Knoten warten, um die Knotengruppe zu erstellen.

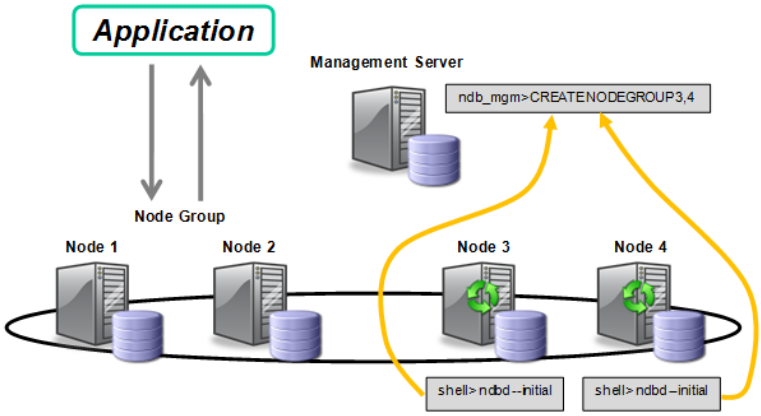


Abbildung 19: ***Starten der neuen Datenknoten und Hinzufügen der Knoten zum Cluster***

Schritt 4: Neupartitionieren der Daten

Zu diesem Zeitpunkt sind die neuen Datenknoten Teil des Clusters, die gesamten Daten befinden sich jedoch noch in der ursprünglichen Knotengruppe (Knoten 1 und Knoten 2). Beachten Sie, dass neue Tabellen automatisch partitioniert und auf alle Knoten verteilt werden, nachdem die neuen Knoten als Teil einer neuen Knotengruppe hinzugefügt wurden.

Abbildung 20 zeigt die Neupartitionierung der Tabellendaten (Festplatte oder Arbeitsspeicher), wenn der Befehl über einen MySQL Server ausgeführt wird. Beachten Sie, dass der Befehl für jede Tabelle wiederholt werden muss, die neu partitioniert werden soll. Als abschließender Schritt kann der SQL-Befehl OPTIMIZE verwendet werden, um ungenutzten Speicherplatz in den neu partitionierten Tabellen erneut nutzen zu können.

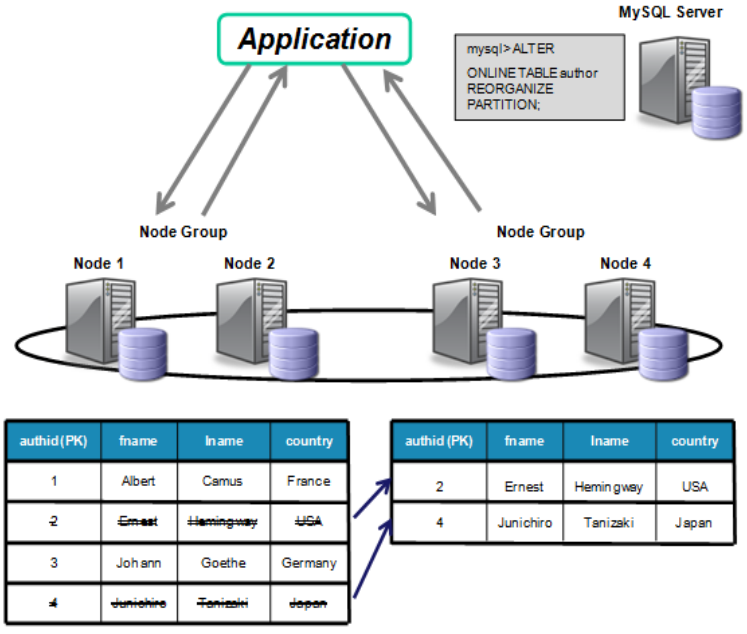


Abbildung 20: ***Neupartitionierung der Daten und Verteilung auf die Knotengruppen***

6. MySQL Cluster DBT2-Leistungs-Benchmark

Wie bereits in diesem Whitepaper erläutert, verwendet MySQL Cluster eine parallele Serverarchitektur mit mehreren aktiven Masterknoten. Auf diese Weise ist für Transaktionen (sowohl Lese- als auch Schreibvorgänge) eine Lastverteilung und eine automatische Skalierung auf mehrere SQL Server gleichzeitig möglich. Dabei kann jeder SQL Knoten auf die Daten aller Knoten innerhalb des Clusters zugreifen und diese aktualisieren.

MySQL Cluster bietet ferner eine flexible Architektur mit der Möglichkeit, Indizes und Daten im Arbeitsspeicher oder Daten auf Festplatte zu speichern. Durch diese Option zum Speichern von Daten im Arbeitsspeicher können mit MySQL Cluster festplattenbasierte E/A-Engpässe eingeschränkt werden, indem Transaktionsprotokolle zum Beibehalten der Echtzeitleistung asynchron auf Festplatte geschrieben werden.

Um die Funktionen des parallelen Echtzeit-Designs zu veranschaulichen, wurden für MySQL Cluster unter Verwendung der DBT2-Testreihe Benchmarks durchgeführt.

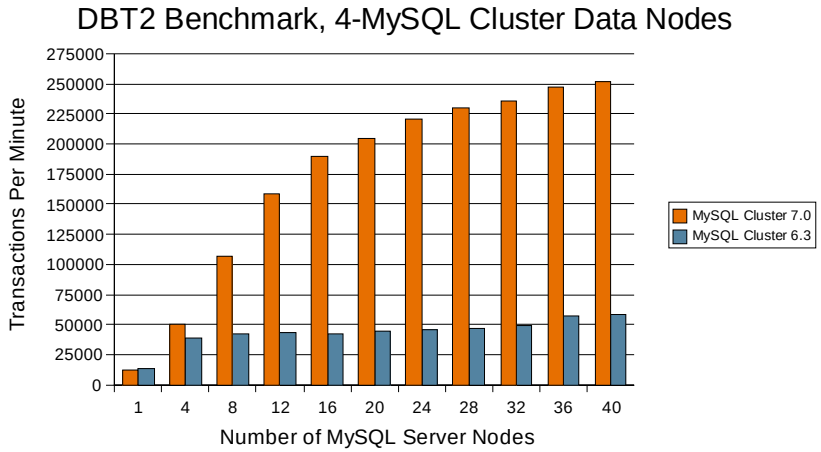


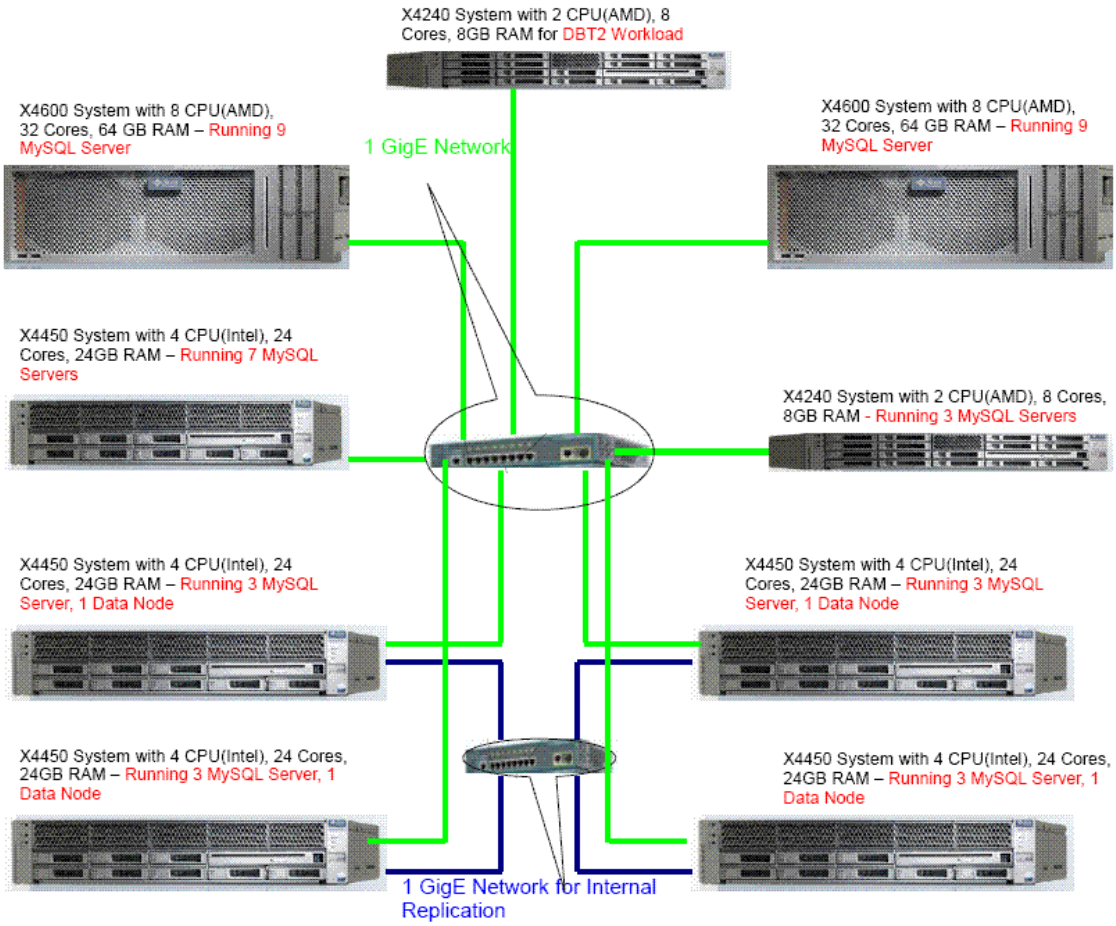
Abbildung 21: ***Mit MySQL Cluster werden über 250.000 Transaktionen pro Minute bzw. 125.000 Operationen pro Sekunde mit einer durchschnittlichen Latenz von nur 1,5 Millisekunden erzielt***

In den DBT2-Tests konnte mit MySQL Cluster bei Verwendung von nur vier Datenknoten ein Ergebnis von 251.000 Transaktionen pro Minute erzielt werden[2]. Jede Transaktion umfasste etwa 30 Datenbankoperationen, sodass mit MySQL Cluster ca. 125.000 Operationen pro Sekunde mit einer durchschnittlichen Reaktionszeit von nur 1,5 Millisekunden erzielt werden konnten. Diese Leistung bedeutet eine vierfache Verbesserung der Skalierbarkeit im Vergleich zu früheren MySQL Cluster Versionen.

Hinweis: *Auf jedem physischen Server wurden mehrere MySQL Server Knoten installiert, wobei einige Serverinstanzen als Lastgeneratoren für die MySQL Cluster Datenbank verwendet wurden. In einer alternativen Bereitstellung könnten mehrere Verbindungen von jedem MySQL Server verwendet werden, was bei gleichbleibender Leistung eine geringere Anzahl an MySQL Server Knoten zur Folge hätte.*

[2] Weitere Informationen zu diesen Benchmark-Tests finden Sie unter: http://blogs.sun.com/hasham/entry/mysql_cluster_7_performance_benchmark

6.1. Benchmark-Topologie



MySQL Cluster Benchmark Topology – 4 Data Nodes, GigE

Hardware Configuration:
x4450 Systems - 4 * Intel Xeon(106D1 family 6 model 29 step 1) Processor at 2.66GHz with total 24 cores, 24 GB RAM, 4 * 1GB GigE, 4 * 146 GB internal HDD, OpenSolaris

X4600 Systems: 8 * AMD (AuthenticAMD 100F42 family 16 model 4 step 2) processor at 2.7 GHz with total 32 cores, 64GB RAM, 4 * 1GB GigE, 4 * 146 GB internal HDD, OpenSolaris

X4240 Systems: 2 * AMD Opteron (100F23 family 16 model 2 step 3) quad core at 2.3 GHz, 8GB RAM, 4 * 1GB Gigabit Ethernet (GigE), 4 * 146 GB internal Hard Disk Drive (HDD), OpenSolaris

Notes: Each MySQL Server is running within a processor set of 3 cores. Total 40 MySQL Servers used.

Abbildung 22: *MySQL Cluster Benchmark-Topologie*

6.2. Database Test 2 (DBT-2)

DBT2 ist ein von OSDL (Open Source Development Labs) entwickelter Open-Source-Benchmark-Test. Um die Ausführung mit einer Clusterdatenbank wie MySQL Cluster Carrier Grade Edition zu vereinfachen, wurde der Benchmark umfassend aktualisiert. DBT2 simuliert eine typische OLTP-Anwendung (Online Transaction Processing), die fünf verschiedene Transaktionstypen ausführt.

DBT2 und MySQL Cluster wurden für den Benchmark als „In-Memory"-Datenbank konfiguriert, um typische „Echtzeit"-Datenbankkonfigurationen zu simulieren. Beachten Sie, dass Leistungsergebnisse als New-Order-Transaktionen pro Minute (TPM) gemessen werden. Die am DBT2-Benchmark vorgenommenen Änderungen

wurden dokumentiert und sind auf der SourceForge-Seite für DBT2-Downloads verfügbar. Ein Download steht auch unter www.iclaustron.com zur Verfügung.

7. Einsatz der modularen MySQL Speicher-Engine-Architektur, um verschiedene Anwendungsanforderungen zu erfüllen

Eine Vielzahl an webbasierten oder Kommunikationsdiensten sind modular aufgebaut und umfassen diverse Module zur Bereitstellung spezifischer Funktionalität. Ein standortbezogener Dienst (Location Based Service - LBS) umfasst beispielsweise verschiedene Module zur Verarbeitung bestimmter Aufgaben:

- Verwaltung von Teilnehmerdaten & Kontaktlisten
- Verarbeitung von Standortaktualisierungen und Präsenzstatus
- Mapping von Datenverwaltung und Entfernungsberechnung (z.B. im Telekommunikationsumfeld)
- Mobile Advertising und Mobile Commerce, einschließlich Empfehlungs-Engines basierend auf Präferenzen
- Aktivitätsprotokollierung für CDRs

Einige der oben beschriebenen Module eignen sich ideal für Datenbank-Engines, die für den Primärschlüsselzugriff optimiert sind. Für andere hingegen sind Bereichsabfragen, Scans und komplexe Joins mit mehreren Tabellen erforderlich. In einigen Fällen ist die Verwendung von MySQL Datenbank-Speicher-Engines, die für jede dieser Anforderungen konzipiert sind, die optimale Lösung, um die Leistungs- und Skalierungsanforderungen der Anwendung zu erfüllen.

MySQL Benutzer profitieren bereits seit langer Zeit von der modularen MySQL Speicher-Engine-Architektur, mit der ein Entwickler oder Datenbankadministrator eine spezialisierte Speicher-Engine für eine bestimmte Anwendungsanforderung auswählen kann. Eine MySQL Speicher-Engine ist eine Low-Level-Engine zur Verwaltung von Datenspeicherung und -abruf. Der Zugriff erfolgt über eine MySQL Server Instanz und in einigen Situationen direkt über eine Anwendung. MySQL Cluster ist beispielsweise als MySQL Speicher-Engine implementiert. Die Verwendung der Speicher-Engine ist sowohl für die Anwendung als auch für den Benutzer transparent.

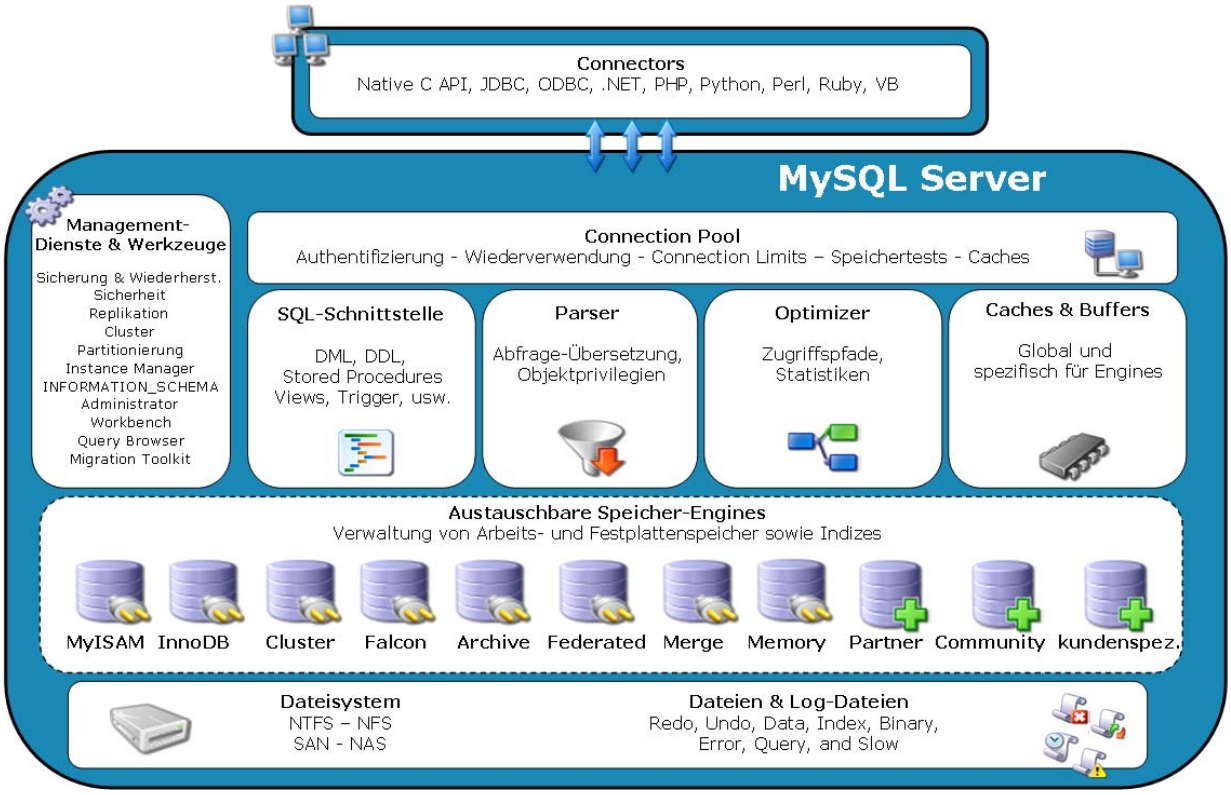


Abbildung 23: ***Die modulare Speicher-Engine-Architektur von MySQL bietet eine beispiellose Datenbankflexibilität***

Die Flexibilität von MySQL geht sogar so weit, dass Sie eine bestimmte Speicher-Engine für jede Tabelle auswählen können, auf die eine Anwendung zugreift. Bei dem oben genannten LBS-Beispiel kann MySQL Cluster daher als Masterdatenbank für die Verwaltung von Benutzerdaten, Standortaktualisierungen, Aktivitätsprotokollierung usw. verwendet werden, während die Verarbeitung von Entfernungsberechnungen und Empfehlungslisten (z.B. Artikelempfehlungen bei Handelsplattformen) auf MySQL Server Slaves verlagert wird, die eine andere MySQL Speicher-Engine verwenden (z. B. MyISAM).

Die MySQL Cluster Funktion zur geografischen Replikation kann verwendet werden, um Echtzeitdaten zwischen Tabellen, die von MySQL Cluster verwaltet werden, und MyISAM Tabellen (die sich möglicherweise auf einem anderen Host oder an einem anderen Standort befinden) zu replizieren (typischerweise innerhalb einer Sekunde nach dem Commit). Im LBS-Beispiel repliziert der MySQL Server Standort- und Präsenzdaten aus der MySQL Cluster Speicher-Engine in eine InnoDB oder MyISAM Speicher-Engine, die als Slave fungiert. Für diese Engine können komplexe Abfragen mit Bereichsscans und Joins mit mehreren Ebenen ausgeführt werden.[3]

Bei Verwendung dieser Funktionalität kann MySQL Cluster auch dann zum Verarbeiten schreibintensiver Arbeitslasten mit Anforderungen an eine maximale Skalierbarkeit und Echtzeitleistung eingesetzt werden, wenn die Module des Diensts selbst das Ausführen komplexer Abfragen mit Operationen erfordern, die sich nicht für MySQL Cluster eignen.

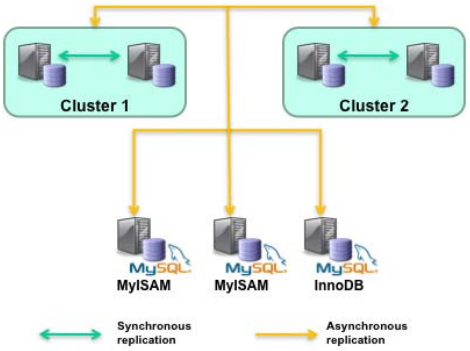


Abbildung 24: ***Mithilfe der geografischen Replikation können Daten in alternative Speicher-Engines und zur Wiederherstellung nach einem Notfall in Remotestandorte repliziert werden***

8. Weitere Informationsquellen

Das Professional Services Team von MySQL verfügt über umfassende Erfahrungen beim Erzielen der bestmöglichen Leistung mit MySQL Cluster. Die Unterstützung durch unsere Mitarbeiter kann im Rahmen eines Beratungspakets oder eines kundenspezifischen Beratungsprojekts in Anspruch genommen werden. Einzelheiten finden Sie unter http://www.mysql.de/consulting/.

MySQL Cluster Gold Support umfasst u.a. die Leistungsoptimierung. Einzelheiten finden Sie unter http://www.mysql.de/products/database/cluster/support.html.

Für die MySQL Cluster Datenbank werden häufig neue Entwicklungen zur Verfügung gestellt, und es sind eine Reihe von Blogs verfügbar, die in Echtzeit Informationen zu diesen Entwicklungen liefern. Eine Liste finden Sie unter http://www.mysql.de/products/database/cluster/resources.html.

[3] Weitere Informationen zum Bereitstellen dieser Art der Replikation finden Sie im Blog.

9. Fazit

Seit der Veröffentlichung im Jahr 2004 wurde MySQL Cluster kontinuierlich weiterentwickelt, um äußerst hohe Anforderungen im Hinblick auf Leistung, Skalierbarkeit und 99,999 % Verfügbarkeit zu erfüllen.

Mit dem verteilten Design ist MySQL Cluster für Arbeitslasten und Anforderungen optimiert, die hauptsächlich einen Primärschlüsselzugriff umfassen. In diesem Whitepaper wurde jedoch aufgezeigt, dass MySQL Cluster noch umfangreichere Anwendungsanforderungen erfüllen kann, wenn Sie Schemata, Abfragen und Parametern optimieren und verteilungsfähige Anwendungen nutzen.

Kontinuierliche Erweiterungen der wesentlichen Datenbanktechnologie sowie Überwachungs- und Berichterstellungsfunktionen in Echtzeit ermöglichen es Datenbankentwicklern und Administratoren, ihre Anwendungsbereiche für MySQL Cluster zu erweitern.

> *„Mit MySQL Cluster erfüllen wir unsere Anforderungen an Skalierbarkeit, Leistung und kontinuierliche Verfügbarkeit bei deutlich geringeren Kosten im Vergleich zu proprietären Technologien. Ohne diese Lösung wären wir aufgeschmissen“.*
>
> **Richard McCluskey, Senior Engineer, go2 Media**

Weitere Informationen zur MySQL Cluster Datenbank finden Sie unter www.mysql.de/cluster.