Large Scale Datenanalyse

SQL für MapReduce

Axel Fischer

Zusammenfassung

Der Begriff *Large Scale Datenanalyse* bezeichnet die Auswertung von sehr großen Mengen an Daten und war bis vor einigen Jahren noch *Massive Parallel Processing DBMS* vorbehalten. Auf der ODSI 2004 (*Operating Systems Design and Implementation*) wurde mit der Vorstellung der *MapReduce*-Plattform allerdings ein alternatives Paradigma mit speziellen Vor- und Nachteilen beschrieben. In seiner ursprünglichen Ausführung unterstützt *MapReduce* die *SQL* nicht, was von Anwendern als Manko angesehen wurde [TEN, HII]. Mittlerweile wurden mit *Hive* und *Tenzing* Erweiterungen der *MapReduce*-Plattform entwickelt, die diese und andere Schwächen von *MapReduce* ausmerzen.

1 MapReduce

1.1 Motivation

Viele Aufgabenstellungen der *Large Scale Datenanalyse* zeichnen sich durch einfache Berechnungsalgorithmen, aber sehr große Datenmengen aus.

Typische Beispiele hierfür sind

- Die Anzahl der gecrawlten Seiten pro Host bestimmen
- Suchmaschinenanfragen nach Tagen gruppieren und sortieren
- Die Worthäufigkeiten in einer großen Menge an Dokumenten ermitteln

Die angeführten Aufgaben haben gemeinsam, dass sich mittels Programmiersprachen wie Java oder C++ mit geringem Aufwand ein einfaches Programm erstellen lässt, welches die gewünschten Berechnungen durchführt. Die Parallelisierung der Verarbeitung, sowie die Anpassung des Programms an die Anforderungen von großen Clustern aus unzuverlässiger Hardware ist hingegen komplex und erfordert tiefgehende Kenntnis der verwendeten Programmiersprache und der Parallelverarbeitung von Daten im Allgemeinen. Derartige Programme sind oft auf die konkrete Aufgabenstellung und die aktuell verwendete Hardware angepasst, so dass sich die Wiederverwendbarkeit des Programmcodes und speziell dessen Abschnitte, welche die parallele Verarbeitung der Daten realisieren, als schwierig oder gar unmöglich gestalten. Die Folge hieraus ist, dass Programmabschnitte, welche für die Parallelisierung verantwortlich sind, für viele Aufgabenstellungen wieder neu entworfen werden müssen - und somit das Rad immer wieder neu erfunden werden muss.

MapReduce soll dem Anwender eben diese Aufgaben abnehmen: die Verteilung der Verarbeitung auf mehrere Prozesse, die Aufteilung der Eingabedaten und die Behandlung

von ausgefallenen Geräten und soll dabei einfach zu benutzen sein. Hierdurch ermöglicht *MapReduce* auch Entwicklern ohne Erfahrungen in den Bereichen Parallelverarbeitung und verteilte Systeme die Nutzung von verteilten Systemen. Weitere Anforderungen wie der zuverlässige Betrieb auf "unzuverlässiger" Standard-Hardware, sowie eine gute Skalierbarkeit sollen die wirtschaftliche Konkurrenzfähigkeit der *MapReduce*-Plattform gewährleisten.

1.2 Programmiermodell

Jegliche Berechnungen auf Grundlage des *MapReduce*-Modells basieren darauf, dass der Algorithmus eine Menge von Eingabepaaren der Form (Schlüssel, Wert) erhält und hieraus eine Menge an Ausgabepaaren der Form (Schlüssel, Wert) berechnet. Unter Verwendung des *MapReduce*-Frameworks beschränkt sich der Aufwand des Anwenders darauf, die beiden Funktionen *map* und *reduce* in einer für die Aufgabenstellung geeigneten Weise zu implementieren.
Map: Als Parameter übernimmt *map* ein Eingabepaar der Form (Schlüssel, Wert) und erzeugt daraus eine ungeordnete Menge an zwischenzeitlichen Ausgabepaaren derselben Form. Die *MapReduce*-library gruppiert diese Ergebnispaare nach ihren zwischenzeitlichen Schlüsseln und reicht die dadurch entstehenden Paare der Form (Schlüssel S, SET[Werte zu Schlüssel S]) an die *reduce*-Funktion weiter.
Reduce: Die Funktion *reduce* führt die Weiterverarbeitung der durch *map* aufbereiteten Eingabedaten durch. Als Parameter übernimmt *reduce* einen zwischenzeitlichen Schlüssel S und die Menge aller zu S gehörenden Werte. Die *reduce*-Funktion führt diese Werte nun in einer geeigneten Weise zusammen und reduziert somit die Anzahl der zu Schlüssel S gehörigen Werte. In vielen Anwendungen kann die Anzahl der Werte direkt auf Eins oder Null verringert werden [MAP].

1.3 Architektur

Ein Verständnis der Architektur der *MapReduce*-Plattform lässt sich am Besten erlangen, indem man die Ausführung der *MapReduce*-Tasks betrachtet. Abbildung 1 veranschaulicht diese. Die Speicherung der Zwischen- und Endergebnisse erfolgt in einem verteilten Dateisystem. Hierfür kommt GFS (*google file system*) zum Einsatz.

Die Ausführung der Aufgaben obliegt grundlegend zwei Arten von Prozessen: Master- und Worker-Prozessen. Bei der Definition neuer *MapReduce*-Tasks legt der Anwender zunächst die Anzahl der einzusetzenden Prozesse fest. Dies geschieht implizit über die Angabe, in wieviele Teile M die Eingabedaten zerlegt werden sollen und in wieviele Teile R die zwischenzeitliche Schlüsselmenge, welche ein Teil der Ausgabedaten der *map*-Worker ist, partitioniert werden soll. Die Partitionierung der zwischenzeitlichen Schlüsselmenge geschieht über eine definierbare Partitionierungsfunktion, deren hashbasierte Standard-Implementierung *hash(key) mod R* berechnet, um die Partition des zwischenzeitlichen Schlüssels zu bestimmen. Wenn das Anwendungsprogramm die *MapReduce*-Funktion aufruft, werden zunächst die Eingabedaten durch die *MapReduce*-Library des Anwenderprogramms in M Teile zerlegt. Die Anzahl der Teile ist abhängig von der durch den Anwender festgelegten Größe der Teile. Nachfolgend werden ein Master- und mehrere Worker-Prozesse gestartet. Verfügbaren Workern wird vom Master einer der M *map*- oder R *reduce*-Tasks zugeteilt. Die *map*-Worker legen auf ihren lokalen Speichergeräten einen Zwischenspeicher, unterteilt in R Partitionen, an und lesen den ihnen zugewiesenen Teil

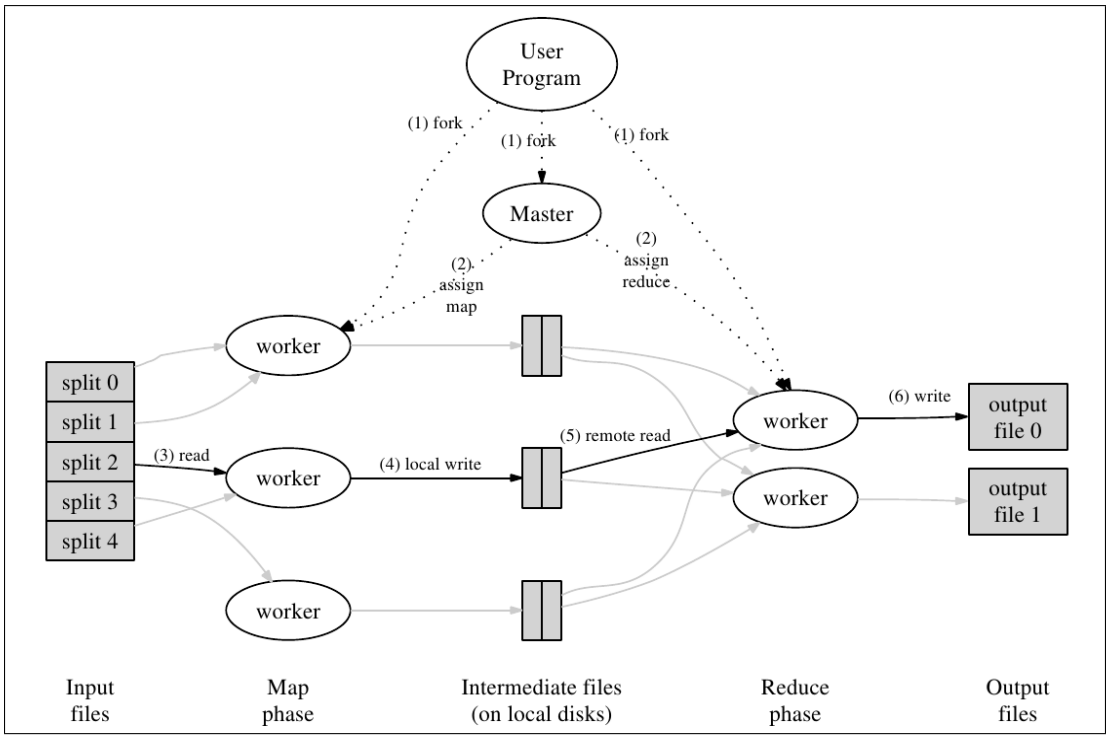

Abbildung 1: Ausführung von *MapReduce*-Tasks

der Eingabedaten und extrahieren hieraus die (Schlüssel, Wert)-Paare, auf welche die vom Anwender definierte *map*-Funktion angewandt wird. Die daraus resultierenden zwischenzeitlichen (Schlüssel, Wert)-Paare werden im Hauptspeicher der *map*-Worker gepuffert und periodisch in ihre, dem zwischenzeitlichen Schlüssel zugehörige, Partition geschrieben. Diese Speicherorte werden dem Master mitgeteilt, welcher sie an die zuständigen *reduce*-Worker weiterreicht. Sobald ein *reduce*-Worker die Speicherorte vom Master erhalten hat, liest er die (Schlüssel, Wert)-Paare von den lokalen Speichergeräten der *map*-Worker und gruppiert diese Paare nach ihren Schlüsseln. Anschließend wendet er auf die resultierenden Paare der Form (Schlüssel S, SET[Werte zu S]) die *reduce*-Funktion an, deren Ergebnis er in eine Ausgabedatei schreibt. Sobald alle *map*- und *reduce*-Aufgaben beendet wurden, benachrichtigt der Master das Anwenderprogramm. Die Ergebnisse liegen nun in R Ausgabedateien vor - einer für jeden *reduce*-Worker. Diese Ergebnisdateien können auch direkt als Eingabedaten für einen anderen *MapReduce*-Task dienen.

Bei dieser Abarbeitung können Fehler auftreten, welche unbehandelt in einem falschen Ergebnis oder gar in einer Blockade des Systems münden können. *MapReduce* soll zudem zuverlässig auf unzuverlässiger Standard-Hardware laufen, somit muss es in der Lage sein, den Ausfall von Geräten erkennen und mit ihm umgehen zu können:
Ausfall von Workern: Der Master pingt alle Worker periodisch an um deren Status zu überprüfen. Sollte ein Worker die Anfrage nicht innerhalb eines bestimmten Zeitraums beantworten, wird dieser vom Master als "ausgefallen" gekennzeichnet. Nicht beendete *map*- oder *reduce*-Aufgaben, sowie bereits beendete *map*-Aufgaben des Workers werden an andere verfügbare Worker verteilt und erneut durchgeführt.
Ausfall des Masters: Der Master erstellt periodisch "Wiederherstellungspunkte", indem er die Informationen zu den laufenden und beendeten Tasks, die vorhandenen Worker und die Speicherorte der Ergebnisdateien sichert. Bei einem Ausfall des Masters muss somit

lediglich ein neuer Master-Prozess mit den gesicherten Daten gestartet werden. Dieser Master beginnt vom Stand des letzten Wiederherstellungspunktes aus.

1.4 Beispiele

1.4.1 Ermittlung von Worthäufigkeiten in Dokumenten

In einer großen Mengen an Dokumenten soll die Häufigkeit des Auftretens der Wörter gruppiert nach den Wörtern ermittelt werden. Gemäß dem beschriebenen Programmiermodell muss der Anwender hierfür die beiden Funktionen *map* und *reduce* in geeigneter Weise implementieren. Um die Berechnung zu beginnen, wird der *MapReduce*-Task an den Master zur Ausführung übergeben. Der Master weist daraufhin den verfügbaren Workern *map*- oder *reduce*-Aufgaben zu.

Input : String dok_name, String dok_inhalt
foreach *Wort w in dok_inhalt* **do**
  ausgeben: (w, 1);

Funktion map(String dok_name, String dok_inhalt)

Input : String wort, list werte
int $summe\_haeufigkeit = 0$;
foreach *Wert in werte* **do**
  $summe\_haeufigkeit = summe\_haeufigkeit + 1$;
ausgeben: (wort, $summe\_haeufigkeit$);

Funktion reduce(String wort, list werte)

Jeder *map*-Worker erhält hierbei eines der auszuwertenden Dokumente - `dok_name` ist der Bezeichner dieses Dokuments, `dok_inhalt` dessen Inhalt als Zeichenkette. Der Worker geht den Inhalt wortweise durch und schreibt für jedes gelesene Wort w das Ergebnis (w, 1) in seinen lokalen Speicher und teilt dem Master die Speicherorte mit. Der Master teilt diese Speicherorte dem zuständigen *reduce*-Worker mit, der die Daten einliest und die vom Anwender definierte *reduce*-Funktion darauf anwendet.

1.4.2 Weitere Anwendungsbeispiele

Auf Grundlage des *MapReduce*-Paradigmas lassen sich viele verschiedene Aufgabenstellungen abdecken. Die Implementierungen der *map*- und *reduce*-Funktionen unterscheiden sich hierbei grundlegend.

Beispielhaft sind

- **Distributed Grep:** Die *map*-Funktion gibt die entsprechende Zeile des Dokuments aus, wenn das gesuchte Muster gefunden wurde. Die *reduce*-Funktion hat dann lediglich alle Werte unverändert auszugeben. Somit ist die Realisierung eines *distributed Greps* auch ohne *reduce*-Task möglich.

- **Invertierter Index:** Die map-Funktion parsed alle Dokumente und gibt für jedes Wort das Paar (wort, dokument_id) aus. Die reduce-Funktion konkateniert die IDs der Dokumente und gibt dann das Paar (wort, list[dokument_id]) aus.

1.5 Analyse

Das *MapReduce*-Framework abstrahiert von der Ebene der Prozesse. Der Anwender muss sich nicht um die Initialisierung und Beendigung der Worker-Prozesse kümmern oder wie die durch ihn definierten *MapReduce*-Tasks auf die zur Verfügung stehenden Prozesse verteilt werden. Dies ermöglicht die Nutzung Verteilter Systeme auch ohne Kenntnisse von Parallelverarbeitung oder Verteilten Systemen.
Skalierbarkeit: *MapReduce* zeichnet sich durch eine gute Skalierbarkeit aus. Durch die Abstraktion von der Prozessebene muss sich der Anwender nicht darum kümmern, wieviele Worker zur Verfügung stehen und wie deren Auslastung ist. Neue Worker werden beim Master angemeldet und dieser verteilt die aktuellen Jobs an die freien Prozesse.
Zuverlässigkeit: Wie in Abschnitt 1.3 beschrieben, erfragt der Master bei jedem Worker periodisch dessen Zustand. Bei Nichtbeantwortung dieser Anfrage wird dieser Worker als ausgefallen gekennzeichnet und seine Zwischenergebnisse verworfen. Die nun wieder offenen Tasks werden nun an verfügbare, andere Worker verteilt.

Als Schwächen von *MapReduce* können vor allem die folgenden Punkte ausgemacht werden:
Format der Zwischenergebnisse: Sämtliche Zwischen- und Endergebnisse werden für eine bessere Fehlertoleranz materialisiert. Somit steht ein Sicherungspunkt bereit wenn ein Prozess ausfällt. Diese Materialisierung macht sich allerdings negativ bemerkbar, wenn die Ergebnisse eines *MapReduce*-Tasks als Eingabedaten für einen weiteren *MapReduce*-Task dienen sollen. Hierfür wäre ein direktes Streaming zwischen den Prozessen eine performantere Lösung, da dann die zeitaufwendigen Schreib- und Leseoperationen nicht mehr erforderlich sind.
Lowlevel Programmiermodell: Das Programmiermodell von *MapReduce* ist nicht hoch entwickelt. Der Anwender muss für Aggregatsfunktionen oder JOINS, welche in SQL einfach zur Verfügung stehen, stets eigene Funktionen schreiben.
Hohe Initialisierungsdauer: Die Initialisierung der Prozesse bei Beginn einer Anfrage kann mehrere Sekunden dauern, da bei MapReduce die Worker-Prozesse nicht permament am Laufen gehalten werden.

Von *Pavlo et al.* [CMP] wird zwar die Meinung, dass *MapReduce* gut skalierbar ist geteilt ("*While MR may indeed be capable of scaling up to 1000s of nodes ...*"), jedoch wird vor allem die mangelnde Geschwindigkeit und das lowlevel Programmiermodell von *MapReduce* im Vergleich zu *Massive Parallel Processing DBMS* kritisiert. Dem ist entgegenzuhalten, dass *MapReduce* dafür kostenlos ist, "out of the box" funktioniert [DCA] und durch die zunehmende Verbreitung von Open-Source-Implementierungen auch leicht erweitert werden kann. Durch die Zuverlässigkeit auf unzuverlässiger Standard-Hardware ist MapReduce auch kostengünstiger einzusetzen als *MPP DBMS*.

2 Hive

2.1 Motivation

Die Größen der zu sammelnden und auszuwertenden Daten in der Industrie wachsen schnell, was traditionelle Warehousing-Lösungen unbezahlbar macht [HII]. MapReduce hingegen bietet eine gute Skalierbarkeit und Zuverlässigkeit auf unzuverlässiger Hardware "von der Stange", was es auch aus wirtschaftlichen Gründen interessant macht. Aus diesen Gründen basiert *Hive* auf Hadoop, einer Open-Source-Implementierung des MapReduce-Paradigmas. Das Programmiermodell von MapReduce ist allerdings, wie in Abschnitt 1.2 beschrieben, lowlevel und fordert von seinen Anwendern zumindest grundlegende Kenntnisse in Datenverarbeitung und einer Programmiersprache wie Java oder C++, was es möglicherweise unbenutzbar für Gelegenheitsnutzer macht. Ein weiterer Nachteil dieser Schwäche ist, dass nach dem MapReduce-Paradigma geschriebene Programme nur schlecht wiederverwendbar sind und selbst für ähnliche Problemstellungen komplett neue Programme erstellt werden müssen. Hive bietet mit *HiveQL* ein einfacher zu benutzendes Interface als MapReduce an. Über HiveQL können Anfragen durch eine SQL-ähnliche, deklarative Anfragesprache formuliert werden. Der Anwender kann MapReduce-Skripte weiterhin in HiveQL-Anfragen einbauen.

2.2 Datenmodell

Hive verwendet zu Speicherung der Daten HDFS (*Hadoop Distributed File System*). Die Daten werden hierbei hierarchisch anhand Tabellen, Partitionen und Buckets organisiert, wobei Tabellen in Hive analog den Tabellen in relationen Datenbanksystemen sind. Jeder Tabelle entspricht ein HDFS-Verzeichnis, in welchem ihre Daten in Form von Dateien gespeichert werden. Tabellen können mehrere Partitionen beinhalten, die festlegen, ob und wie die Daten auf Unterverzeichnisse verteilt werden. Wenn eine Tabelle `mitarbeiter` in der Datenbank `unileipzig` anhand ihrer Spalte `abtnr` partitioniert wird, werden beispielsweise Daten mit `abtnr=10` im Verzeichnis `/unileipzig/mitarbeiter/abtnr=10` gespeichert. Innerhalb der Partitionen besteht durch sogenannte Buckets eine weitere Unterteilungsmöglichkeit.

2.3 Architektur

Hive setzt auf Hadoop, einer verbreiteten Open-Source-Implementierung des MapReduce-Frameworks, auf und erweitert es um SQL-Funktionalitäten. Hive lässt sich grundlegend in sechs Hauptkomponenten gliedern, welche mit dem zugrundeliegenden Hadoop in Abbildung 2 dargestellt sind:

- Compiler
- Execution Engine
- Driver
- Metastore
- Thrift Server
- External Interfaces

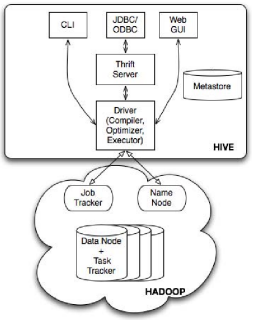

Abbildung 2: Hive Architektur

Der Anwender übermittelt seine HiveQL-Anfragen über die **External Interfaces** an Hive. Hive bietet hierfür kommandozeilenbasierte und webbasierte Interfaces an, sowie auch eine API. Der **Thrift Server** bietet eine einfache API an, um programmiersprachenübergreifend durch JDBC (Java) oder ODBC (C++) auf Hive zugreifen zu können. Der **Compiler** wird vom Driver aufgerufen, sobald eine HiveQL-Anfrage übermittelt wurde. Er erstellt aus der Anfrage einen Ausführungsplan, welcher aus einem oder mehreren MapReduce-Tasks besteht. Nachfolgend übermittelt der Driver die MapReduce-Tasks topologisch sortiert an die **Execution Engine**, welche derzeit Hadoop ist. Der **Metastore** ist der Systemkatalog, mit welchem alle anderen Komponenten interagieren. Er enthält die Namen der verwendeten Datenbanken, welche die namespaces für die Tabellen darstellen, sowie auch Metadaten zu den Tabellen und Partitionen.

Nach Übermittlung der HiveQL-Anfrage ruft der Driver den Compiler auf. Dieser parsed die Query und erstellt den logischen Ausführungsplan in Form eines Operatorbaums. Nachfolgend läuft ein Optimizer über den Code und führt auf Basis eines einfachen und kleinen Regelwerks [HII] Optimierungen durch. Hierzu gehört die Zusammenfassung vieler `JOIN`-Operationen auf den gleichen Schlüssel zu einem Mehrfach-`JOIN`, welcher dann als einzelner MapReduce-Task ausgeführt werden kann. Andere Optimierungen zielen darauf ab, den Datenfluss zu begrenzen, indem man sich die den Daten zugrundeliegende Partitionierungsschemata zunutze macht. Werden in dem Beispiel Abschnitt 2.2 die Namen aller Mitarbeiter der Abteilung mit `abtnr=10` gesucht, müssen nur die Dateien im Verzeichnis `/unileipzig/mitarbeiter/abtnr=10` gelesen werden. Durch eine frühzeitige Erkennung nicht beteiligter Partitionen kann die zu lesende Datenmenge und somit auch die Bearbeitungszeit reduziert werden. Auf Grundlage des optimierten logischen Ausführungsplans erstellt der Compiler den physischen Ausführungsplan in Form eines gerichteten, azyklischen Graphen, dessen Knoten die MapReduce-Tasks sind. Ein Beispiel hierfür wird in Abbildung 3 dargestellt.

2.4 SQL-Features

Mit HiveQL unterstützt Hive eine SQL-ähnliche, deklarative Anfragesprache, welche die gängigen SQL-Anweisungen wie `SELECT`, `PROJECT`, `JOIN`, `UNION ALL`, sowie Aggregatsfunktionen unterstützt [HLM]. DDL (*Data Definition Language*) wird dahingehend unterstützt, dass mittels HiveQL-Anfragen Tabellen einschließlich Partitionsschemata angelegt werden können. Mit Hilfe der Operatoren `LOAD` und `INSERT` können Daten aus externen Quellen oder resultierend aus HiveQL-Anfragen in die Tabellen eingefügt werden. Eine Unterstützung von DML-Operationen (*Data Manipulation Language*) wie `UPDATE` und `DELETE` ist hingegen nicht gegeben, da das zur Datenspeicherung verwendete Dateisystem HDFS innerhalb der Dateien nur Anhängeoperationen erlaubt, so dass die Löschung wie auch Änderung von Zeilen nicht unterstützt wird.

2.5 Beispiel

Nachfolgend soll die Zerlegung von HiveQL-Anfragen anhand eines einfachen Beispiels betrachtet werden.
In einer Tabelle `buecher` seien alle in einer Menge von Büchern vorkommenden Wörter gespeichert. Bei x-fachem Auftreten in den Büchern, so auch x Einträge in der Tabelle. Um die Wörter mit ihren Häufigkeiten absteigend sortiert zu erhalten, formuliert der Anwender eine Anfrage gemäß Listing 1.

Listing 1: Anfrage zur Ermittlung der Häufigkeiten von Wörtern

```
SELECT wort, COUNT(*)
FROM buecher
GROUP BY wort
ORDER BY 2 DESC
```

Der Compiler übersetzt diese HiveQL-Anfrage in einen Ausführungsplan, wie in Abbildung 3 dargestellt, bestehend aus zwei MapReduce-Tasks. Der erste *map*-Task erzeugt für jedes `wort w` in `buecher` ein Paar (`w`, 1). Der erste *reduce*-Task summiert dann analog zu Abschnitt 1.4.1 die Häufigkeiten auf. Der zweite MapReduce-Task führt daraufhin die Sortierung durch. Eine derartige Anfrage wird vom Compiler immer in zwei MapReduce-Tasks übersetzt, da die Ergebnisdaten des ersten map-Tasks bei verschiedenen reduce-Workern eintreffen und somit, für eine korrekte Sortierung, zwei MapReduce-Schritte erforderlich sind. Eine Alternative bietet die Anweisung `SORT BY` [HLM], bei deren Verwendung anstelle von `ORDER BY` die Query in nur einen MapReduce-Task übersetzt wird. Unter Verwendung von `SORT BY` liefert die Anfrage allerdings im Allgemeinen das korrekte Ergebnis nur dann, wenn die Anzahl der *reduce*-Worker auf einen beschränkt wird.

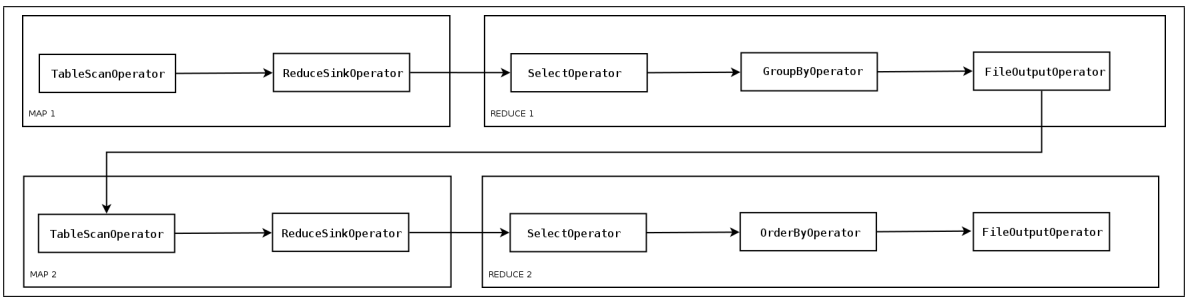

Abbildung 3: Ausführungplan bei Hive

2.6 Analyse

Da Hive auf einer Implementierung des MapReduce-Frameworks aufbaut, erbt es dessen Stärken wie die hohe Skalierbarkeit und die Zuverlässigkeit des Betriebs auch auf unzuverlässiger Hardware. Durch letzteren Punkt bietet sich Hive auch aus wirtschaftlichen Gründen als Alternative zu MPP DBMS an. Mit HiveQL konnte der Nachteil des lowlevel Programmiermodells von MapReduce behoben werden. Weitere Schwächen wie die hohe Initialisierungsdauer bestehen auch bei Hive. Die zukünftige Arbeit an dem Projekt umfasst die Erweiterung von HiveQL, welches bisher nur eine Untermenge von SQL darstellt, sowie das Ersetzen des einfachen, regelbasierten Optimizers durch einen kostenbasierten, weiter entwickelten Optimizer.

3 Tenzing

3.1 Motivation

Das Datawarehouse für die Daten von *Google-Ads* basierte auf einer proprietären Datenbankanwendung, welche nachfolgend mit *DBMS-X* bezeichnet wird. Bei *DBMS-X* handelt es sich um eines der führenden *Massive Parallel Processing DBMS* [TEN]. Die Datenbank war rasch wachsenden Datenvolumina ausgesetzt und die Kosten für die erforderliche Skalierung von *DBMS-X* wurden als unvertretbar hoch angesehen. Die für *ETL* (*Extract, Transform, Load*) erforderlichen Zeiten wuchsen ebenfalls stark an, da *ETL* bei mehreren heterogenen Datenquellen kompliziert und aufwendig sein kann. Eine weitere Einschränkung des bisherigen Systems betraf die Anwender, welche durch die natürlichen Beschränkungen von SQL oftmals dazu gezwungen waren, selbst Programmecode zu schreiben, um die Auswertung komplexer Datensätze zu ermöglichen [TEN]. Es wurde die Entscheidung gefällt, künftig die Infrastruktur von Google und speziell die *MapReduce*-Plattform zu verwenden.

3.1.1 Anforderungsanalyse

Auf Grundlage der ausgemachten Schwächen des momentan eingesetzten Systems wurde ein umfassendes Neu-Design der verwendeten Plattform entschieden. Aus den existierenden Problemen und der zurückliegenden Entwicklung der Datenvolumina wurden für die neue Plattform folgende Anforderungen ermittelt.

Skalierbarkeit: Die neue Plattform musste auf tausende Cores, hunderte Nutzer und Petabytes an Daten skaliert werden können.
Fehlertoleranz: Zuverlässiger Betrieb auf unzuverlässiger Hardware "von der Stange" musste gewährleistet sein.
Performanz: Die neue Plattform musste mindestens die gleiche Performanz wie das aktuelle System bieten.
Datenunterstützung: Direkte Unterstützung der unter dem Google File System abgelegten Daten, um teure und langsame ETL-Operationen zu vermeiden.
Funktionsumfang: Unterstützung aller benötigter SQL-Features um die Einarbeitungszeit zu verkürzen, sowie Unterstützung von Experten-Funktionen.

Die ursprüngliche Plattform von *MapReduce* erfüllt, wie in Abschnitt 1.5 beschrieben, die Anforderungen Skalierbarkeit, Fehlertoleranz und Datenunterstützung, nicht jedoch alle dieser Anforderungen.

Schwächen von *MapReduce*

- SQL wird nicht unterstützt. Die Anwender sind gezwungen, ihre Anfragen mittels einer Programmiersprache wie Java oder C++ nach dem *MapReduce*-Paradigma zu stellen
- Hohe Initialisierungsdauer der *MapReduce*-Tasks, die mehrere Minuten betragen kann
- Zwischen einzelnen Tasks ist kein Streaming möglich

Nachfolgend wird beschrieben, welche Schritte unternommen wurden, *MapReduce* zu erweitern um allen Anforderungen gerecht zu werden.

3.2 Architektur

Der Aufbau von *Tenzing* lässt sich grob in vier Hauptkomponenten unterteilen und erinnert deutlich an die ihm zugrunde liegende *MapReduce*-Plattform, welche in Abschnitt 1.3 beschrieben ist.

- Workerpool
- Query Server
- Metadata Server
- Client Interfaces

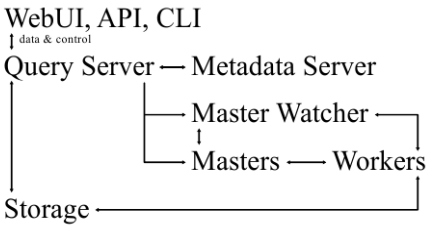

Abbildung 4: Tenzing Architektur

Workerpool: Der Workerpool bildet das Ausführungssystem, welches vom Query Server einen Ausführungsplan übernimmt und die durch diesen definierten *MapReduce*-Tasks ausführt. Um die im Abschnitt 1.5 aufgeführte hohe Initialisierungszeit der Tasks zu verringern, werden die Prozesse nicht ständig neu gespawned, sondern permanent am Laufen gehalten. Der Workerpool besteht aus Workern, Mastern und einem Master Watcher. Die in Abschnitt 1.3 erläuterten Worker und Master wurden um den Master Watcher ergänzt, welcher eingehende *MapReduce*-Tasks an freie Master delegiert.
Query Server: Der Query Server dient als Gateway zwischen den Clients und dem Workerpool. Er parsed die übermittelten Anfragen, führt Optimierungen durch und erstellt einen Ausführungsplan, welcher im Allgemeinen aus mehreren *MapReduce*-Tasks besteht. Dieser Ausführungsplan wird dem Workerpool zur Abarbeitung übergeben. Im Gegensatz zu Hive verfügt der Tenzing-Optimizer neben seinem Regelwerk auch über kostenbasierte Optimierungsstrategien, um eine breitere Menge an Anfragen optimieren zu können. Beispielsweise wird bei einem `JOIN` überprüft, ob die sekundäre Tabelle in den Arbeitsspeicher der Worker passt. Ist dies der Fall, laden die Worker eine Kopie dieser Tabelle in ihren Arbeitsspeicher um die folgenden Leseoperationen zu beschleunigen.
Metadata Server: Der Metadata Server bietet eine Programmierschnittstelle an, über welche man Metadaten wie die Bezeichner von Tabellen und Schemata speichern und auslesen kann.
Client Interfaces: Tenzing bietet dem Anwender verschiedene Schnittstellen an. Unter anderem stehen Kommandozeilen- und Webbasierte Interfaces zur Verfügung, die sich in ihrem Funktionsumfang und im Anspruch ihrer Bedienbarkeit unterscheiden. Neben diesen wird eine API angeboten, die es erlaubt, Anfragen durch MapReduce-Skripte direkt an den Workerpool zu stellen.

Zur Speicherung der Daten werden von Tenzing mit traditionellen relationalen Datenbanksystemen wie MySQL oder verteilten Dateisystemen wie GFS verschiedenartige Backends unterstützt.

3.2.1 Anfragenverabeitung

Im Regelfall - wenn Anfragen nicht direkt an den Workerpool gestellt werden - erfolgt die Verarbeitung der Anfragen nach dem folgenden Muster:

1. Der Anwender oder ein anderer Prozess übermittelt durch eines der Client Interfaces eine Anfrage an den Query Server.

2. Der Query Server parsed die Anfrage und erstellt einen Operatorbaum, welcher die Grundlage für die folgende Zerlegung in MapReduce-Tasks bildet.

3. Der Query Server lädt der Anfrage entsprechend Daten vom Metadata Server und ergänzt die Anfrage um erforderliche Metadaten. Erfolgt die Speicherung der Daten beispielsweise durch HDFS, würde der Tabellenname Mitarbeiter durch den Bezeichner des entsprechenden HDFS-Verzeichnisses ersetzt - im Beispiel aus Abschnitt 2.2 wäre dies das Verzeichnis `/unileipzig/mitarbeiter`.

4. Der Query Server optimiert ergänzte Anfrage regel- und kostenbasiert und erstellt den Ausführungsplan. Eine mögliche Optimierung in unserem Beispiel wäre die Überprüfung, ob vorhandene `WHERE`-Klauseln dem Partitionierungsschema der Daten entsprechen. Unser Beispiel wurde anhand der Abteilungssnummer `abtNr` partitioniert; bei Vorhandensein der Klausel `WHERE abtNr=10` kann der Optimizer die Anfrage frühzeitig auf das Verzeichnis `/unileipzig/mitarbeiter/abtnr=10` einschränken.

5. Der Ausführungsplan besteht aus einem oder mehreren *MapReduce*-Tasks. Für jeden dieser Tasks fordert der Query Server beim Master Watcher einen verfügbaren Master an und übermittelt diesem den Plan. Zu diesem Zeitpunkt ist die ursprüngliche Anfrage des Anwenders bereits in mehrere physisch voneinander getrennte Arbeitsteile zerlegt.

6. Verfügbare Worker fordern bei den Mastern periodisch Arbeit an. Der Master mit der Abarbeitung eines MapReduce-Tasks betraute Master weist ihnen dann *map*- oder *reduce*-Aufgaben zu. Die *reduce*-Worker schreiben ihre Ergebnisse in einen Zwischenspeicherbereich

7. Der Query Server überwacht den Zwischenspeicherbereich und wartet auf die Ergebnisse der *reduce*-Worker. Sobald diese erstellt wurden, sammelt er sie ein und reicht sie an den Upstream-Client weiter.

3.3 SQL-Features

Tenzing unterstützt die gebräuchlichen *SQL92*-Konstrukte, sowie speziell für Analyse-Aufgaben entwickelte Erweiterungen von SQL, wie hashtable-basierte Aggregation und `HASH JOINS`. Alle Erweiterungen wurde so entworfen, dass sie weitestgehend parallelisierbar sind, um die Vorteile des zugrunde liegenden *MapReduce*-Framework auszunutzen. Unterstützte SQL-Features sind Projektions- und Filterungsoperatoren wie `IN`, `LIKE`, `BETWEEN`, `CASE` und arithmetische Operatoren. Tenzing unterstützt alle standardmäßigen Aggregatsfunktionen wie `SUM`, `MAX` und `COUNT`.

Bezüglich DML-Operationen wie `UPDATE` oder `DELETE` stellt sich für Tenzing dasselbe Problem wie für Hive. Das zugrundeliegende Dateisystem unterstützt möglicherweise lediglich Anhängeoperationen. Um dennoch eine entsprechende Funktionalität zur Verfügung stellen zu können, wird bei einer `UPDATE`- oder `DELETE`-Operation ein neuer Datensatz mit den geänderten Werten erstellt und die Referenz auf den alten Datensatz im Metadata-Server aktualisiert.

3.4 Beispiel anhand einer Hash-Aggregation

Um eine perfomante Implementierung der Hash-Aggregation zu erreichen, war eine weitere Änderung an MapReduce erforderlich. In der ursprünglichen Implementierung von MapReduce sortieren *reduce*-Worker ihre Eingabedaten vor der Verarbeitung anhand ihrer Schlüssel. Bei der Hash-Aggregation ist dieser Schritt nicht erforderlich, da lediglich notwendig ist, dass ein *reduce*-Worker alle Werte zu einem Schlüssel erhält - ob er diese mit einem oder mit mehreren Aufrufen der reduce-Funktion verarbeitet, ist hingegen nicht von Bedeutung. Bei Tenzing kann der Anwender dem Compiler mitteilen, dass er eine Hash-Aggregation wünscht und somit auf das Sortieren der Daten verzichtet wird.

In nachfolgendem Beispiel sollen in einem Unternehmen die Anzahl der Mitarbeiter gruppiert nach den Abteilungen bestimmt werden. Der Anfrage zugrunde liegt eine Tabelle der Form Mitarbeiter(name, abtNr). Der Anwender würde hierbei eine Anfrage gemäß Listing 2 stellen.

Listing 2: Ermittlung der Anzahl an Mitarbeitern gruppiert nach Abteilungsnummer

```
SELECT abtNr, COUNT(*)
FROM Mitarbeiter
GROUP BY 1
```

Der Query-Server erstellt hieraus einen Ausführungsplan, welcher aus jeweils einem *map*- und *reduce*-Task besteht. Ein Schlüssel in der *map*-Funktion ist hierbei für die Auswertung nicht erforderlich.

Listing 3: *map*-Task

```
start()
{
  abt_hash = new HashTable();
}

map(dummy, angestellte : list)
{
  fuer jeden Mitarbeiter m in der Liste angestellte
    abt_hash[m.abtNr]++;
}

finish()
{
  fuer jede abtNr in abt_hash
    ausgeben: (abtNr, abt_hash[abtNr])
}
```

Die Funktionen *start* und *finish* werden zu Beginn bzw. am Ende der *map*-Phase einmal aufgerufen. Die Funktion *start* erzeugt hierbei eine HashTable, die der Zwischenspeicherung der Ergebnisse dient. Nach der Initialisierung wird die Funktion *map* in jedem zugeteilten Worker mit einem Teil der zu verarbeitenden Daten aufgerufen. Dieser inkrementiert nun für jeden Mitarbeiter die Summationsvariable der entsprechenden Abteilung.

Am Ende der *map*-Phase gibt die Funktion *finish* die Daten aus.

Listing 4: *reduce*-Task

```
start()
{
  abt_hash_erg = new HashTable();
}

reduce(abtNr, anzahl : list)
{
  fuer jedes Element a in anzahl
    abt_hash_erg[abtNr] += a;
}

finish()
{
  fuer jede abtNr in abt_hash_erg
    ausgeben: (abtNr, abt_hash_erg[abtNr])
}
```

Die Funktion *start* legt auch hier eine HashTable an, welche verwendet wird, um die Zwischenergebnisse der *map*-Phase zu aggregieren. Die Funktion *reduce* erhält eine Menge von Werten zu einem bestimmten Schlüssel, der Abteilungsnummer. Sie summiert diese Werte in der HashTable auf. Bei Ende der *reduce*-Phase werden die Ergebnisse in den Zwischenspeicherbereich geschrieben, vom Query-Server eingesammelt und an den Upstream-Client weitergereicht.

3.5 Analyse

Für einige der in Abschnitt 3.1.1 aufgeführten Anforderungen bringt die *MapReduce*-Plattform bereits gute Voraussetzungen mit. Speziell die Anforderungen Skalierbarkeit und Zuverlässigkeit auf unzuverlässiger Hardware werden durch *MapReduce* abgedeckt. Zudem kann *MapReduce* direkt auf den Daten laufen, welche unter GFS gespeichert sind. Somit werden teure ETL-Operationen vermieden. Die durchschlagenden Nachteile von *MapReduce* sind unter anderem die hohe Initialisierungsdauer der Tasks, das Fehlen von Streaming zwischen den Worker-Prozessen und die fehlende Unterstützung von SQL. Die Initialisierungsdauer der Tasks konnte reduziert werden, indem die Worker-Prozesse nicht immer neu gespawned werden sondern permanent am Laufen gehalten. Hierdurch konnten die Wartezeiten von Minuten auf Sekunden reduziert werden [TEN].

Ein weitere Verbesserung stellt das bei Tenzing implementierte Streaming zwischen *MapReduce*-Tasks dar, was bei Anfragen, die in mehrere *MapReduce*-Tasks zerlegt werden, spürbare Performanz-Vorteile bringt.

Um die Skalierbarkeit bewerten zu können, wurde Tenzing mit einer unterschiedlichen Anzahl an Worker-Prozessen einem Benchmark mittels der Anfrage in Listing 5 unterzogen. In diesem Benchmark waren die Daten im *ColumnIO*-Format in GFS-Dateien abgespeichert. Abbildung 5 führt die Ergebnisse dieses Benchmarks auf, wobei Throughput hierbei die Anzahl der Zeilen pro Sekunde und Worker-Prozess bezeichnet. Anhand dieses Beispiels zeigt sich bei Auftragen der Zeit über der Anzahl der beteiligten Worker-Prozesse ein linearer Verlauf, welcher eine sehr gute Skalierbarkeit von Tenzing bedeutet.

Workers	Time (s)	Throughput
100	188.74	16.74
500	36.12	17.49
1000	19.57	16.14

Abbildung 5: *Tenzing* Skalierbarkeit

Die Verabeitung der Anfragen mittels des MapReduce-Paradigmas erzeugt einen gewissen Overhead, der bei geringen Mengen zu verarbeitender Daten nicht bedeutend kleiner als der eigentliche Berechnungsaufwand ist. Um dem Rechnung zu tragen, wird vom Query Server die Gesamtgröße der zu verarbeitenden Daten bestimmt und, wenn dieser einen voreingestellten Grenzwert unterschreitet, der Task lokal ausgeführt.

Query	DBMS-X (s)	Tenzing (s)	Change
#2	129	93	39% faster
#4	70	69	1.4% faster
#1	155	213	38% slower
#3	9	28	3.1 times slower

Abbildung 6: *Tenzing* gegen *DBMS-X*

Um die Performanz von Tenzing und seinem Vorgänger DBMS-X vergleichen zu können, wurden auf beiden Systemen vier SQL-Anfragen durchgeführt, welche für Analyse-Aufgaben typisch sind [TEN]. Das Tenzing-Setup bestand aus 1000 Prozessen, denen jeweils 1 CPU, 2 GB RAM und 8 GB Festplattenspeicher zur Verfügung standen. Die Daten wurden von Tenzing unter *GFS* gespeichert. Bis auf eine Anfrage weisen Tenzing und DBMS-X im Durchschnitt ähnliche Ergebnisse auf. In Anfrage #3 dominiert die Initialisierungsdauer die Berechnungszeit bei Tenzing [TEN].

3.5.1 Erkenntnisse

Durch die Entwicklung von Tenzing konnte gezeigt werden, dass es möglich ist, auf Grundlage der *MapReduce*-Plattform eine vollumfängliche SQL Engine zu erstellen, welche überdies noch über speziell für Analyse-Aufgaben erstellte Funktionalitäten verfügt. Durch Änderungen konnte *MapReduce* so verbessert werden, dass es in den Punkten Durchsatz und Wartezeit mit kommerziellen Massive Parallel Processing DBMS mithalten kann.

Listing 5: Benchmark-Anfrage zur Bewertung der Skalierbarkeit

```
SELECT a, SUM(b)
FROM T
WHERE c = k
GROUP BY a
```

4 Literatur- und Abbildungsverzeichnis

Literatur

[MAP] Dean J., Ghemawat S.: MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004

[GFS] Ghemawat S. et al.: The Google File System, SOSP 2003

[CMP] Pavlo A. et al.: A comparison of approaches to large-scale data analysis, SIGMOD 2009

[DCA] Dean J.: MapReduce: A Flexible Data Processing Tool, Communications of the ACM vVol. 53 (Januar 2010)

[HII] Thusoo A.: Hive - A Warehousing Solution Over a Map-Reduce Framework, VLDB 2009

[HLM] Hive Language Manual, http://wiki.apache.org/hadoop/Hive/LanguageManual

[TEN] Dean J., Ghemawat S.: Tenzing - A SQL Implementation On The MapReduce Framework, VLDB 2011

Abbildungsverzeichnis