Network Traffic Measurement

of Data-intensive Computing Architectures

Diplomarbeit

Technische Universität Berlin
Fakultät IV - FG Internet Network Architectures

vorgelegt von:	Alexander Kordecki
am:	21. Dezember 2012
Studienfach:	Technische Informatik
Matrikelnummer:	18 10 95
Betreuer:	Georgios Smaragdakis, Ph.D.
Betreuer:	Dr. Bernhard Ager
Erstgutachter:	Prof. Anja Feldmann, Ph.D.
Zweitgutachter:	Prof. Dr. Volker Markl

Zusammenfassung

Der beispiellose Erfolg des World Wide Web und der Übergang von Unternehmen und Regierungen in das digitale Zeitalter verursachen immer größere Datenmengen. Im Jahr 2010 schätzte Eric Schmidt, der ex-CEO von Google, dass die Datenmenge, die innerhalb von zwei Tagen auf der Welt produziert wird, dem gesamten Datenaufkommen bis 2003 entspricht. Dieser Trend wird sich fortsetzen, da von Benutzern erzeugte Inhalte immer mehr zunehmen und soziale Aktivitäten und Interaktionen immer mehr digitalisiert werden. Große Datenmengen müssen gespeichert, abgerufen, verarbeitet und analysiert werden. Diese Anforderungen bedeuten neue Herausforderungen für die Systemdesigner, die Datenverarbeitung und -analyse effizienter zu gestalten und skalierbar zu machen. Hierfür wurde eine neue Familie von Computing-Architekturen für datenintensive Anwendungen entwickelt, bei der die Daten in privaten Rechenzentren gehostet und analysiert werden. Obwohl der verteilten Datenverarbeitung und -replikation in Rechenzentren bereits eine Menge Aufmerksamkeit zuteil wurde, sind die Auswirkungen dieser Architekturen auf die Netzwerklast der Rechenzentren noch unklar.

Der Fokus dieser Diplomarbeit liegt auf dem Vergleich von Analysen strukturierter Daten mittels Big Data-Architekturen in privaten Rechenzentren. Wir evaluieren in einem 5-Knoten Cluster sowohl die Ressourcennutzung als auch den Netzwerkverkehr zwischen den Servern für zwei der populärsten Architekturen, MapReduce und verteilte relationale Datenbanken. Zum Vergleich dienen sowohl synthetische Tests auf Basisfunktionen der Datenverarbeitung als auch Anfragen auf echten Messungen des BitTorrent Protokolls.

Unsere Auswertung zeigt, dass eine verteilte Datenbank auf derselben Hardware dem MapReduce-Ansatz in Bezug auf die Anfragegeschwindigkeit überlegen ist, während die MapReduce-Architektur Vorteile in der Geschwindigkeit beim Laden von Daten, bei der Administration der Systeme und bei der Skalierung auf sehr große Clustern bietet. Das Datenaufkommen im Netzwerk ist zwischen beiden Ansätzen sehr unterschiedlich. MapReduce verursacht aufgrund seiner Architektur eine immense Netzwerklast, die Datenbank hingegen kann durch Informationen über die Verteilung der Daten die Menge der zu übertragenden Daten sehr gering halten.

Abstract

The unprecedented success of the World Wide Web and the migration of businesses and governments to the digital age have created large amount of data. In 2010 Eric Schmidt, the ex-CEO of Google, estimated that the data that the world produced every two days was as much as up to 2003. This trend is expected to grow as user-generated content is booming and the human social activity and interaction is digitalized. Voluminous data has to be stored, accessed, processed and analyzed. The above requirements pose new challenges to the system designers on how to efficiently design and scale data processing and analysis. To cope with these challenges, a new family of data-intensive computing architectures have been proposed where data is hosted and analyzed within private datacenters. Despite the fact that a lot of attention has been paid on the distributed data replication and processing within a datacenter, the implications of these architectures on the network operation of a datacenter are still unclear.

The focus of this thesis is on comparison of analyzes of structured data using big data architectures in private data centers. In a 5-node cluster we evaluate both, system resource utilization as well as network traffic patterns between the servers for two of the most popular architectures, MapReduce and distributed relational database. For benchmarking we use synthetic tests based on basic functions of data processing as well as queries on real measurements of the BitTorrent protocol.

Our evaluation unveils that a distributed database on the same hardware is in matter of query speed superior to the MapReduce approach, while the MapReduce architecture has advantages in speed when loading data, administration of the systems and for scaling to very large clusters. The network performance between the two approaches is very different. MapReduce caused its architecture an immense network load. Due to information about data distribution, the amount of data to be transferred by the database is very low.

Inhaltsverzeichnis

1. Einleitung

Im Fachbereich „Internet Network Architectures“ (FG INET) der TU Berlin werden regelmäßig Netzwerkanalysen auf Basis umfangreicher Statistiken erstellt. Momentan werden diese Daten entweder in einer einzelnen Datenbank oder mit Hilfe selbstgeschriebener Programme auf einfachen Textdateien ausgewertet. Beim Arbeiten mit entsprechend großen Datenmengen ergibt sich dabei das Problem hoher Laufzeiten der Berechnungen. Deshalb wurde überlegt, Techniken einzusetzen, die speziell für den „Big Data“ Bereich entwickelt wurden.

Fragestellung und Abgrenzung

Für verteilte Anfragen auf großen Datenmengen existieren inzwischen verschiedene für den Produktiveinsatz geeignete Systeme. Diese beruhen auf unterschiedlichen Ansätzen, weshalb sich die Frage stellt, welcher Ansatz zur Auswertung der Netzwerkstatistiken für den Fachbereich der geeignetere ist. In dieser Arbeit werden daher für diesen speziellen Einsatz Systeme für verteilte relationale Datenbanken und MapReduce verglichen.

Für Firmen ist es zunehmend erforderlich, große Datenmengen in geringer Zeit auszuwerten. Daher werden in Rechenzentren immer häufiger große Cluster gemietet, auf denen entsprechende Software eingesetzt wird. Da sich der FG INET mit Netzwerken auseinander setzt, stellte sich die Frage nach eventuellen Auffälligkeiten im Netzwerkverkehr dieser Systeme, bzw. was bei größeren Installationen hinsichtlich der Netzwerktechnik zu beachten wäre.

Ziel dieser Arbeit ist es, die folgenden beiden Fragen zu untersuchen:

1. Eignet sich besser eine verteilte relationale Datenbank oder ein MapReduce Framework angesichts der spezifischen Eigenschaften der Systeme zur Auswertung von Netzwerkstatistiken?

2. Wie sieht der Netzwerkverkehr zwischen den Knoten in einem entsprechenden Cluster aus, und was muss man deshalb gegebenenfalls beim Skalieren beachten?

Um diese Arbeit zur Beantwortung dieser Fragestellungen abzugrenzen, wurde „Netzwerkstatistik“ als automatisiert aufgezeichnete Messwerte der immer selben Messwertgeber, die Daten aus einem oder mehreren Netzwerkprotokollen extrahieren, festgelegt. Diese Daten stehen somit in einer strukturierten Form zur Verfügung.

Da es sich beim FG INET um eine Forschungseinrichtung handelt, wird davon ausgegangen, dass es bei der Analyse der Daten darum geht, die Daten interaktiv zu untersuchen und kurzfristig verschiedenste Anfragen auszuführen und nicht darum, automatisiert in regelmäßigen Abständen immer wieder dieselben Anfragen auszuführen. Im Rahmen des Einführungsvortrags zu dieser Arbeit stellte sich zudem heraus, dass es vielen Mitarbeitern des Fachbereiches wichtig ist, die Anfragen in einer abstrakten Anfragesprache formulieren zu können, um sich nicht mit den Interna des MapReduce Frameworks auseinander setzen zu müssen.

Da die Daten strukturiert vorliegen, ist der Einsatz einer Datenbank unproblematisch, auch eine abstrakte Anfragesprache ist mit SQL gegeben und bietet die Möglichkeit, interaktiv mit den Daten zu arbeiten. Alle MapReduce Frameworks sind in der Lage, mit strukturierten Daten umzugehen, ebenso existieren für diese Frameworks inzwischen abstrakte Anfragesprachen.

Herrangehensweise

Dieser Arbeit liegt ein empirischer Ansatz zugrunde. Es werden dieselben Anfragen in einer abstrakten Anfragesprache sowohl auf einer verteilten Datenbank, als auch auf einem MapReduce Cluster auf demselben Datenbestand ausgeführt. Während dieser Anfragen werden verschiedene Parameter zur Ressourcennutzung auf den Knoten und im Netzwerk mitgeschrieben.

Der erste Teil der Arbeit widmet sich Tests zu grundlegenden Funktionen der Datenverarbeitung auf synthetischen Daten. Zur Darstellung der Aussagekräftigkeit dieser Tests werden im zweiten Teil auf Basis dieser Erkenntnisse Prognosen über den Verlauf von realen Anfragen auf realen Daten erstellt. Nach der Messung dieser Anfragen auf den realen Daten werden diese Prognosen überprüft.

Die ermittelten Ergebnisse zeigen, dass eine verteilte Datenbank in Bezug auf die Anfragegeschwindigkeit der MapReduce Lösung überlegen ist. Dagegen bietet das MapReduce Framework sowohl beim Laden der Daten als auch beim Aufsetzen und Administrieren der Systeme Vorteile gegenüber der Datenbank.

Aufbau der Arbeit

Eine Prämisse dieser Arbeit ist, dass der Leser über ein grundlegendes Wissen zu Datenbanken verfügt. Zusätzlich stellt Kapitel 2 die technischen Grundlagen verteilter Datenbanken sowie des MapReduce Verfahrens dar.

Kapitel 3 behandelt sowohl den Versuchsaufbau als auch die Versuchsdurchführung für eine empirische Untersuchung der Fragestellungen. Es wird auf die eingesetzte Hard- und Software eingegangen, ein Unterkapitel widmet sich den Messungen. Die sich anschließende

Versuchsdurchführung gliedert sich in zwei Teile, im ersten Teil werden synthetische Tests auf synthetischen Daten ausgeführt, im zweiten Teil echte Anfragen auf realen Daten.

Kapitel 4 präsentiert zu Beginn die Ergebnisse der synthetischen Tests und erstellt darauf aufbauend Prognosen für die Tests auf den realen Daten. Anschließend folgen die Ergebnisse der echten Tests sowie eine Verifikation der Vorhersagen.

Kapitel 5 gibt schließlich einen zusammenfassenden Überblick über die Ergebnisse dieser Arbeit.

2. Grundlagen

Dieses Kapitel wird mit einer Einführung in Datenarten begonnen, auf die ein Einblick in verteilte Datenbanken mit einer Beschreibung von DB2 folgt. Das nächste Unterkapitel widmet sich dem Big Data Thema. Es klärt über die Funktionsweise von MapReduce auf und stellt Hadoop und Pig vor. Das Kapitel endet mit einer Unterweisung in die Konzepte von verteilten Joins.

2.1. Datenarten

Unstrukturierte Daten waren unter anderem ein entscheidender Punkt in der Entwicklung von BigData Anwendungen. Es folgt daher eine kurze Einführung in die verschiedenen Datenarten.

Man unterscheidet grundsätzlich drei verschiedene Daten-Typen:

Strukturierte Daten

Strukturierte Daten sind Daten, die sich ein einer modellbasierten Struktur ablegen lassen, die sich nicht ändert. Sie entsprechen dem, was in den verschiedenen Programmiersprachen als Datenstruktur abgebildet ist. Wolff [Wol12] sagt: „Als strukturierte Daten werden Daten in Datenbanken bezeichnet, da diese, z.B. in relationalen Datenbanktabellen, ein einheitliches Format aufweisen. Wichtig ist hierbei, dass alle Daten spezielle, in einem Schema spezifizierte, Strukturen aufzuweisen haben und spezielle Einschränkungen erfüllt werden müssen.“ Für diese Arbeit wird davon ausgegangen, dass die Daten aus den Netzwerkprotokollen als strukturierte Daten zur Verfügung stehen.

Semistrukturierte Daten

Als semi- oder teilstrukturierte Daten bezeichnet man Informationen, die keiner allgemeinen Struktur unterliegen, sondern einen Teil der Strukturinformation mit sich tragen. Ihnen liegt kein Datenmodell zugrunde, wodurch sich eine Datensammlung aus semistrukturierten Daten beliebig erweitern kann. Ein Strukturmodell kann nachfolgend impliziert werden.

Nach Wolff [Wol12] können semistrukturierte Daten mit Hilfe von Grammatik und Lexik in eine Form gebracht werden, die folgende Charakteristika aufweist:

E1 Die Datensammlung besteht aus einer oder mehreren Folgen von Objekten.

E2 Objekte können entweder in Attribute zerlegt werden (komplexe Objekte) oder sie sind atomare Objekte.

E3 Atomare Objekte enthalten Werte eines bekannten, elementaren Datentyps.

Ein Beispiel für semistrukturierte Daten sind XML Dokumente.

Unstrukturierte Daten

In der Informatik sind unstrukturierte Daten digitalisierte Informationen, die in einer nicht formalisierten Struktur vorliegen und auf die dadurch von Computerprogrammen nicht über eine einzelne Schnittstelle aggregiert zugegriffen werden kann. Das Problem bei diesen Daten sind fehlende Hinweise auf den Typ dieser Daten. Ein typisches Beispiel für einen solchen Datentyp sind Textdokumente bzw. Ton- oder Bildaufnahmen.

Im Alltag ist normalerweise der Typ der unstrukturierten Daten anzutreffen, da die meisten Daten nicht in sorgfältig strukturierten Datenbanken gesammelt werden, sondern zumeist ad hoc erfasst werden und somit unterschiedliche oder gar keine Strukturen aufweisen. Da sich aber in genau diesen Daten sehr viele Informationen „verbergen“, musste eine Lösung geschaffen werden um auch mit diesen Daten arbeiten zu können.

2.2. Verteilte Datenbanken

Sowohl Rechenleistung als auch RAM oder Festplattenspeicher sind bei allen Rechnern begrenzt. Sobald eine lokal auf nur einem Rechner laufende Datenbank eine dieser Grenzen sprengt, wird auf verteilte Datenbanken ausgewichen, die ein weiteres Wachstum ermöglichen.

Nach C.F. Date [Dat87] ist „eine verteilte Datenbank eine virtuelle (logische) Datenbank deren Teile an unterschiedlichen Stellen in einer Menge unterschiedlicher, echter Datenbanken gespeichert sind“. Er legte 1987 zwölf „Regeln“ für verteilte Datenbanken fest:

Lokale Autonomie Jeder Knoten agiert als ein unabhängiges Datenbank Management System (DBMS) und wird lokal verwaltet.

Kein zentraler Knoten Um Engpässe im Verarbeitungsdurchsatz zu vermeiden, sollte es keinen zentralen Knoten geben und kein Knoten von einem anderen abhängig sein.

Unterbrechungsfreier Betrieb Das System wird durch den Ausfall von Knoten nicht beeinflusst. Angefallene Arbeiten für ausgefallene Knoten müssten nachträglich nach Wiederinbetriebnahme durch das Verteilte Datenbank Management System (VDBMS) erfolgen.

Standortunabhängigkeit Der Nutzer braucht nicht zu wissen, wo die Daten gespeichert sind, um sie abzurufen.

Fragmentierungsunabhängigkeit Ein verteiltes Datenbanksystem muss die Aufteilung von Relationen in Fragmente transparent für den Nutzer unterstützen. Fragmentierungen können sowohl horizontal (auf Zeilen) als auch vertikal (auf Spalten) definiert sein.

Replikationsunabhängigkeit Das VDBMS muss die Replikation von Tabellen bzw. Tabellenfragmenten transparent für den Nutzer unterstützten.

Verteilte Anfragen Eine verteilte Anfrage kann auf verschiedenen Knoten ausgeführt werden. Die Anfrageoptimierung wird transparent vom VDBMS umgesetzt.

Verteiltes Transaktionsmanagement Eine Transaktion kann Daten auf mehreren Knoten ändern, Rücksetzlogik und Parallelitätskontrolle sind über den gesamten Datenbankverbund verteilt und transparent für den Nutzer.

Hardwareunabhängigkeit Das System muss auf jeder Hardware laufen.

Betriebssystemunabhängigkeit Das System muss auf jedem Betriebssystem laufen.

Netzwerkunabhängigkeit Das System muss unabhängig vom Typ des genutzten Netzwerks laufen.

Herstellerunabhängigkeit Das System muss mit den Datenbanken aller Hersteller funktionieren.

Auch wenn heutzutage kein Produkt alle Regeln erfüllt, so sind sie doch eine gute Grundlage für das Verständnis dafür, was eine verteilte Architektur alles leisten muss. Beispiele für aktuell auf dem Markt erhältliche verteilte relationale Datenbanken sind:

- DB2 ESE
- Oracle mit Partitioning Option
- Vertica
- Teradata
- Greenplum
- Postgres-XC
- MySQL Cluster CGE

Für die Verteilung von Datenbanken existieren, wie nachfolgend beschrieben, verschiedene Ansätze.

2.2.1. Shared-Memory Architektur

Bei diesem Ansatz greifen mehrere Prozessoren auf einen gemeinsamen Haupt- oder Sekundärspeicher zu. Die einzelnen Prozessoren können über den gemeinsamen Speicher miteinander kommunizieren. In Abbildung 2.1 wird die Shared-Memory Architektur symbolisiert dargestellt.

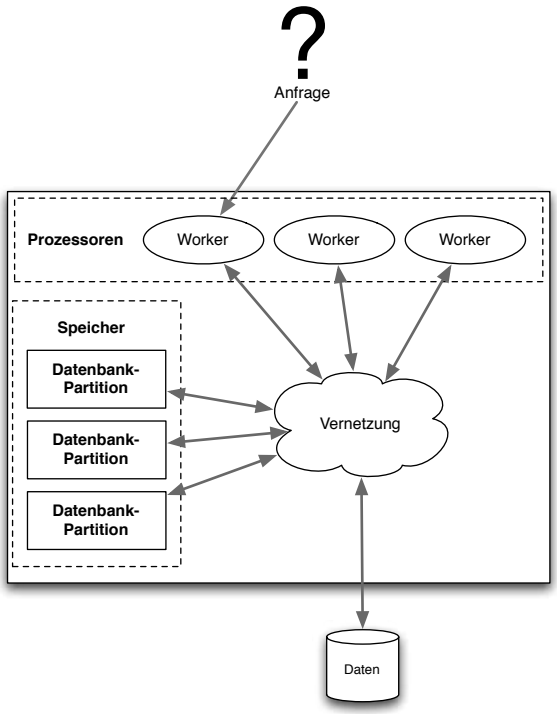

Abbildung 2.1.: Shared-Memory Architektur

Der Vorteil einer solchen Architektur ist, dass sie einfach zu programmieren ist und Loadbalancing effizient umgesetzt werden kann. Ihr Nachteil ist die begrenzte Ausbaumöglichkeit. Aus Geschwindigkeitsgründen können nicht beliebig viele Prozessoren gleichzeitig auf denselben Speicher zugreifen.

2.2.2. Shared-Disk Architektur

Jeder Knoten hat in der Shared-Disk Architektur seinen eigenen Hauptspeicher. Die Festplatten, auf denen die Daten liegen, können jedoch von allen Knoten gemeinsam genutzt werden. Auf jedem Knoten läuft die Datenbanksoftware, erweitert um Funktionen zur Synchronisation. Die Kommunikation läuft über ein Netzwerk. Eine vereinfachte Darstellung zur Shared-Disk Architektur ist in Abbildung 2.2 zu sehen.

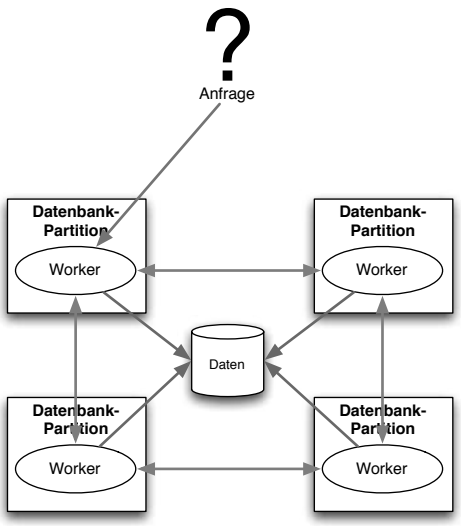

Abbildung 2.2.: Shared-Disk Architektur

Der Vorteil dieser Architektur ist, dass die Redundanz gleich eingebaut ist, denn jeder Knoten kann auf alle Daten zugreifen. Der limitierende Faktor ist hier der gemeinsam genutzte Plattenspeicher. Durch die notwendige Synchronisation bei dem Zugriff auf Daten entsteht hoher zusätzlicher Aufwand. Auch die Bandbreite der Schnittstelle, über die die Knoten auf den gemeinsamen Plattenspeicher zugreifen, kann bei hohen Knotenzahlen zum Engpass werden.

2.2.3. Shared-Nothing Architektur

Auch bei der in Abbildung 2.3 gezeigten Shared-Nothing Architektur hat jeder Knoten seinen eigenen Speicher, verfügt nun aber auch über eine eigene Festplatte und kann als eigenständige Einheit betrachtet werden. Die Daten im VDBMS sind über alle Knoten verteilt. Sowohl horizontale als auch vertikale Fragmentierung der Tabellen ist möglich. Die Knoten kommunizieren über ein Netzwerk miteinander. Für diese Architektur ist es üblich einen Knoten zur Koordination abzustellen, denn die eingehenden Anfragen müssen vom VDBMS in einzelne Anfragen für die unterschiedlichen Knoten aufgeteilt und die Teilergebnisse am Ende wieder zusammengefasst werden.

Diese Architektur lässt sich recht gut skalieren, da die einzigen gemeinsam genutzten Ressourcen das Netzwerk und der Koordinationsknoten sind. Der Nachteil ist, dass bei Operationen auf Daten, die sich auf unterschiedlichen Knoten befinden, zusätzlicher Kommunikationsaufwand nötig ist um die Daten auf einem Knoten zusammenzufassen. Die Bandbreite im Zugriff auf Daten ist durch den jeweiligen Knoten begrenzt, auf dem diese Daten liegen. Redundanz kann durch Replikation der Daten auf weitere Knoten ermöglicht werden.

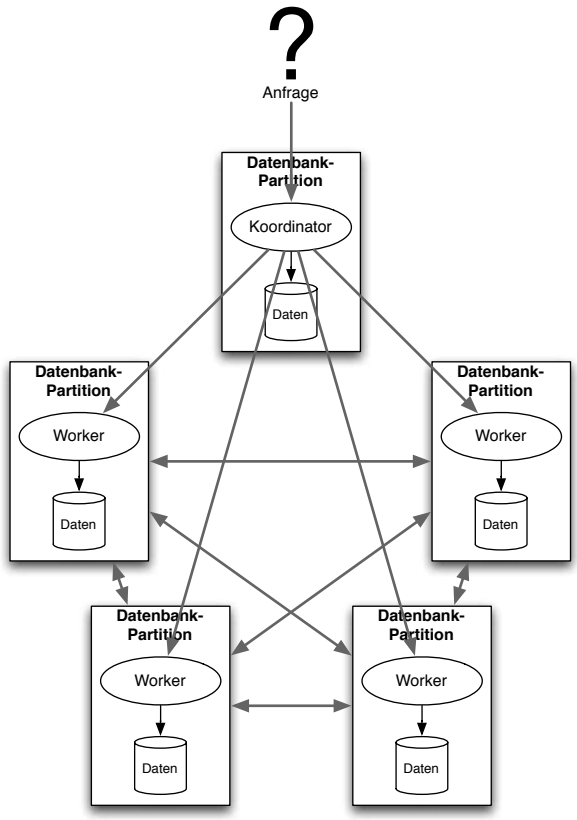

Abbildung 2.3.: Shared-Nothing Architektur

2.2.4. DB2

Eines der getesteten Produkte ist DB2 von der Firma IBM in der Enterprise Server Edition (ESE). DB2 ist ein relationales Datenbank Management System (RDBMS), das in der ESE das Erstellen einer verteilten Datenbank auf Basis einer Shared-Nothing Architektur ermöglicht. IBM nennt diese Möglichkeit der Verteilung der Daten Data Partitioning Feature (DPF) und unterteilt die Datenbank dafür in maximal 1000 Partitionen. Jede Partition agiert als eigenständiges RDBMS, besitzt eigenen Speicher und eigenen Festplattenplatz. Die Partitionen können sowohl auf Einzelprozessorsystemen als auch symmetrischen Multiprozessorsystemen (SMP) laufen. Auf jedem Rechner im Cluster können mehrere Partitionen parallel eingesetzt werden. Die Partitionen, die auf demselben Rechner laufen, kommunizieren über Inter Process Communication (IPC) miteinander, die Kommunikation mit anderen Partitionen läuft über TCP-Verbindungen. Um Redundanz zu ermöglichen, wird mittels des High Availability Desaster Recovery (HADR) Verfahrens zwischen jeweils zwei Partitionen eine aktiv-passiv Replikation unterstützt.

Die Partition „0“ beinhaltet eine besondere Rolle, auf ihr ist zusätzlich der „System Catalog“ installiert, in dem die globalen Tabellenstrukturen, Indizes, Statistiken, etc. definiert

sind. Auch übernimmt sie die Funktion des Koordinators, der die SQL-Anfragen übersetzt, optimiert und an die Einzeldatenbanken der anderen Partitionen weiterreicht, sowie die Teilergebnisse zum Abschluss einer Anfrage zusammenfügt und an den anfragenden Client zurücksendet.

Verteilte Datenbanken werden mit DB2 aufgesetzt, in dem im ersten Schritt Tabellenbereiche (Tablespace) definiert werden, die sich über mehrere Partitionen erstrecken. Im nächsten Schritt werden Tabellen erstellt, die mind. ein Feld als „distribution Key“ enthalten, nach dem die Daten über die im Tablespace enthaltenen Partitionen verteilt werden. Die Verteilung kann entweder anhand eines auf Basis des distribution Key berechneten Hashwertes oder eines Bereichsfilters stattfinden. Indizes werden, wenn sie den distribution Key enthalten, automatisch auf jeder Partition lokal angelegt.

DB2 zerlegt im „Query Plan“ eine SQL Anfrage in Operatoren, die auf mehreren Partitionen parallel laufen können. Diese Operatoren sind über ein „Pipelining“ genanntes Streamingkonzept miteinander verbunden. Wenn ein Operator Daten an den nächsten Operator senden muss, werden die Daten per „Push“ Operation weitergeleitet, Zwischenergebnisse werden, sofern genügend RAM zur Verfügung steht, nicht auf der Festplatte gesichert. Abbildung 2.4 zeigt einen vereinfachen Ausführungsplan des im Kapitel 3.5.6 vorgestellten Tests „BitTorrent Job 1“. Abbildung 2.5 stellt den dazu gehörenden vereinfachten zeitlichen Ablauf der Anfrage mit parallel laufenden Operationen dar.

Wie diese Ausführungspläne gelesen werden ist in Kapitel 4.1.1 beschrieben. Zu erkennen ist, dass die Teilergebnisse eines Operators noch während der weiteren Berechnung bereits im nächsten Operator zur Weiterverarbeitung eingesetzt werden um so zügig wie möglich die ersten Ergebnisse an den anfragenden Clienten zurück liefern zu können.

2.3. Big Data

„Big Data“ ist das aktuelle Schlagwort, wenn es um die Verarbeitung von Massendaten geht. Als Big Data werden besonders große Datenmengen bezeichnet, die mit Hilfe von Standard-Datenbanken und Daten-Management-Tools nicht oder nur unzureichend verarbeitet werden können. Drei Attribute beschreiben hierbei die Problematik für Big Data:

Volume Die Datenmengen gehen bis in die Peta- und Exabytes, bei denen alleine schon das Speichern auf Festplatten keine triviale Aufgabe ist.

Velocity Die Daten müssen schnell ausgewertet werden. Die Dauer der Anfragen sollte eher im Bereich von Minuten oder Stunden liegen und nicht in Jahren.

Variety Es gibt keine festen Datenstrukturen, es müssen auch unstrukturierte Daten verarbeitet werden können.

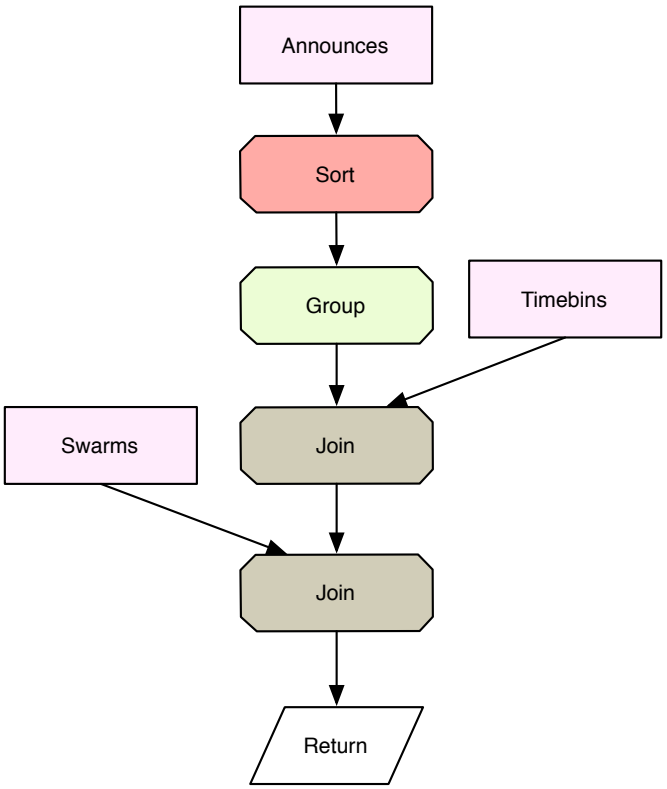

Abbildung 2.4.: Vereinfacher DB2 Ausführungsplan am Beispiel von BitTorrent Job 1

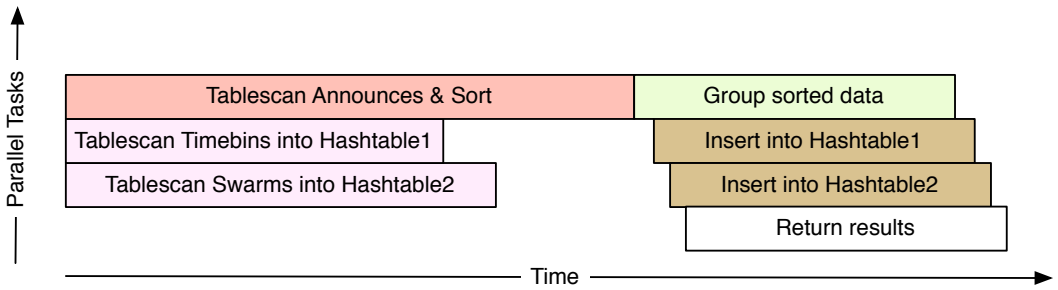

Abbildung 2.5.: Vereinfachtes DB2 Scheduling am Beispiel von BitTorrent Job 1

2.3.1. MapReduce

Die Grundlagen für die sich heute als Quasi-Standard im Big Data Bereich herausgebildete MapReduce Technik wurden von Google entwickelt. Bei Dean und Ghemawat [DG04] ist nachzulesen, dass Google diese Technik hauptsächlich entwickelte um die Erstellung der Indizes für die Suchmaschine zu vereinfachen und besser skalieren zu können. Sie wurde durch die MAP- und REDUCE-Funktionen funktionaler Programmiersprachen inspiriert.

Während die MAP-Funktion ein Tupel aus Schlüssel und Wert auf mehrere andere Tupel aus anderen Schlüsseln und Werten abbilden kann, bildet die REDUCE-Funktion einen Schlüssel mit einer Menge von Werten auf eine Menge von Werten ab.

In Anlehnung an Wikipedia [Wik12] sind die MAP- und REDUCE-Funktionen folgendermaßen definiert:

Es gilt für alle folgenden Abbildungen:

Die Mengen K und L enthalten Schlüssel, die Mengen V und W enthalten Werte.

Alle Schlüssel $k \in K$ sind vom gleichen Typ, z. B. Strings.

Alle Schlüssel $l \in L$ sind vom gleichen Typ, z. B. ganze Zahlen.

Alle Werte $v \in V$ sind vom gleichen Typ, z. B. Atome.

Alle Werte $w \in W$ sind vom gleichen Typ, z. B. Gleitkommazahlen.

Wenn A und B Mengen sind, so ist mit $A \times B$ die Menge aller Paare (a, b) gemeint, wobei $a \in A$ und $b \in B$ (Kartesisches Produkt).

Wenn M eine Menge ist, so ist mit M^* die Menge aller endlichen Listen mit Elementen aus M gemeint (angelehnt an den Kleene-Stern) - die Liste kann auch leer sein.

$$\text{Map} : K \times V \to (L \times W)^* \tag{2.1}$$

$$(k, v) \mapsto [(l_1, x_1), \ldots, (l_{r_k}, x_{r_k})] \tag{2.2}$$

$$\text{Reduce} : L \times W^* \to W^* \tag{2.3}$$

$$(l, [y_1, \ldots, y_{s_l}]) \mapsto [w_1, \ldots, w_{m_l}] \tag{2.4}$$

Folgen diese Funktionen aufeinander lässt sich eine Liste von Schlüssel-Wert-Paaren auf eine neue Liste von Schlüssel-Wert-Paaren abbilden:

$$\text{MapReduce} : (K \times V)^* \to (L \times W)^* \tag{2.5}$$

$$[(k_1, v_1), \ldots, (k_n, v_n)] \mapsto [(l_1, w_1), \ldots, (l_m, w_m)] \tag{2.6}$$

Die MAP Funktion kann auf mehreren Rechnern parallel auf unterschiedlichen Eingangsdaten arbeiten und die REDUCE-Funktion auf mehreren Rechnern parallel auf Mengen unterschiedlicher Schlüssel. Allerdings müssen zwischen diesen beiden Funktionen die Ergebnisse der MAP-Funktion umsortiert und bei der Verteilung über mehrere Rechner auch zwischen diesen Systemen kopiert werden. Die Phase, in der die Daten kopiert werden, nennt sich Shuffle-Phase. Abbildung 2.6 stellt den Datenfluss zwischen den MAP- und REDUCE-Funktionen im MapReduce Framework symbolisiert dar.

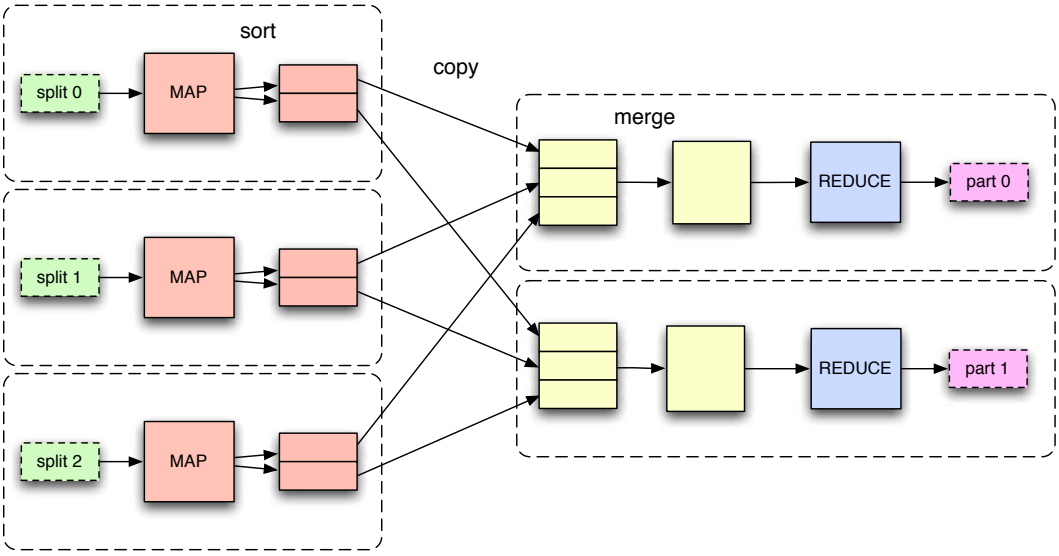

Abbildung 2.6.: MapReduce Dataflow

Die Berechnung einer beliebigen Funktion auf den Ursprungsdaten erfolgt wie in Abbildung 2.7 gezeigt, indem mehrere MapReduce Jobs hintereinander ausgeführt werden.

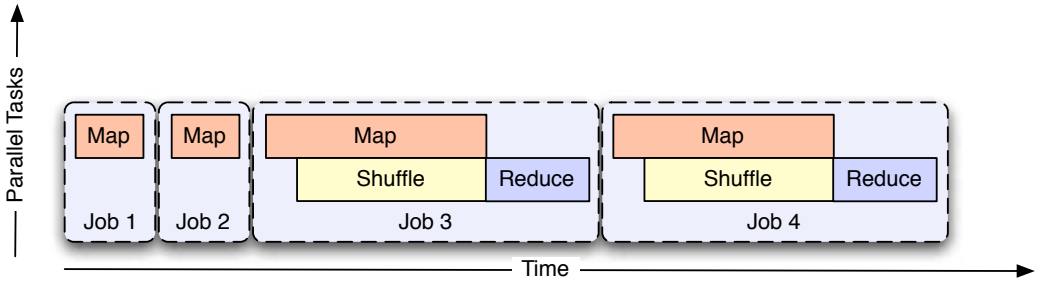

Abbildung 2.7.: MapReduce Batchprocessing

Ein wichtiger Punkt bei MapReduce ist, den Programmcode zu den Daten zu transferieren. Die Berechnungen werden deshalb bevorzugt auf den selben Knoten ausgeführt, auf denen bereits die Daten zur Verfügung stehen.

2.3.2. Hadoop

Hadoop ist ein in Java geschriebenes OpenSource Framework, dass auf den von Google entwickelten Ansätzen zu MapReduce sowie dem verteilten Google Filesystem beruht. Es besteht aus zwei Teilen, dem verteilten Dateisystem Hadoop Distributed File System (HDFS) und einem Framework, dass die Koordination der MapReduce Jobs übernimmt.

HDFS

Das HDFS besteht aus einer Namenode, einer secondary Namenode sowie mehreren Datanodes. Die Metadaten der gespeicherten Dateien werden auf der Namenode gespeichert. Die secondary Namenode ist ein Replikat der Namenode zur Ausfallsicherheit. Der Inhalt der Dateien wird in Blöcke, sogenannte „Splits“, von 128 MB Größe aufgeteilt und gleichmäßig über alle Datanodes verteilt. Jeder Split kann zur Sicherheit auf mehrere andere Maschinen repliziert werden. Die Replikation der Splits findet asynchron statt. Das HDFS bildet eine Shared-Disk bei der jeder Knoten auf jeden Split zugreifen kann. Mittels des Features „Rack Awareness“ kann sichergestellt werden, dass sich aus Sicherheitsgründen mindestens ein Replikat jedes Splits sogar in einem weiteren Rack befindet, während versucht wird bei der Ausführung eines Jobs die Tasks so über die Maschinen zu verteilen, dass ein Großteil des nötigen Netzwerkverkehrs lokal innerhalb des Racks bleibt.

MapReduce

Das MapReduce Framework von Hadoop besteht aus einem Jobtracker und mehreren Tasktrackern. Da Hadoop den Ansatz verfolgt, den Code zu den Daten zu bringen, empfiehlt es sich, die Tasktracker auf den selben Knoten ausführen zu lassen, auf denen auch die Datanodes laufen. Der Jobtracker verwaltet alle MapReduce Jobs und teilt den Tasktrackern einzelne Map- oder Reducetasks zu, die dort jeweils lokal ausgeführt werden.

Bei der Vergabe der Maptasks auf die Tasktracker orientiert sich der Jobtracker daran, auf welchen Knoten sich bereits welche Splits im HDFS befinden um den Kommunikationsaufwand zu reduzieren. Die Ergebnisse eines Map- oder Reducetasks werden im HDFS abgelegt. Ein Reducetask beginnt mit der Shuffle-Phase, in der die für den Reduce benötigten Zwischenergebnisse der verschiedenen Mapper auf den lokalen Knoten kopiert und zusammengeführt werden. Erst wenn alle Zwischenergebnisse fertig sortiert sind beginnt der eigentliche Reduce-Prozess.

Sollte ein Knoten während der Ausführung eines Map- oder Reducetasks ausfallen, wird dieser Task automatisch auf einem weiteren Knoten nachgeholt.

2.3.3. Pig

Pig wurde seit 2006 ursprünglich von Yahoo entwickelt. Es ist ein Framework um ad-hoc Analysen auf großen Datenmengen mittels Hadoop durchzuführen. Die dazu gehörige Sprache nennt sich Pig Latin. Sie ermöglicht es, die Anfragen auf die Daten in einer abstrakten Notation, ähnlich wie in SQL, zu formulieren. Es können, falls nötig, zusätzlich einzelne Funktionen in Java, Python oder JavaScript erstellt werden. Pig Latin ist im Gegensatz zu SQL eine imperative Sprache. Nach Olston u. a. [Ols+08] ist „Pig Latin zu programmieren vergleichbar mit dem Spezifizieren eines Ausführungsplans für SQL, was es dem Entwickler erleichtert, den Fluß seiner Daten während der Ausführung explizit zu kontrollieren“. Pig setzt die in Pig Latin geschriebenen Anfragen in MapReduce Jobs um, die in einem weiteren Schritt mit Hadoop ausgeführt werden.

Weitere Unterschiede zwischen Pig und SQL sind unter anderem die Möglichkeit von Pig Daten nach jeder Operation speichern zu können oder Datenströme zu teilen. Während SQL darauf ausgelegt ist, am Ende eines „Ausführungsbaums“ ein einzelnes Ergebnis zu erhalten, beschreibt Pig Latin einen gerichteten azyklischen Graphen, der Datenströme teilen und auf den jeweiligen Teilströmen unterschiedliche Operationen ausführen kann.

2.4. Verteilte Joins

Bei Shared-Nothing Architekturen, insbesondere bei BigData-Anwendungen, liegen aus Speicherplatzgründen meist nur Tabellenfragmente auf einzelnen Knoten. Kommt es zu einer Join Operation zwischen verteilten Tabellen, gibt es je nach Verteilung und Größe der beteiligten Tabellen unterschiedliche Konzepte, sie zusammenzuführen. Da die Join Operation eine häufige aber auch komplexe Operationen auf den Daten ist, werden im Versuch in den nächsten Kapiteln unter anderem diese Konzepte zwischen DB2 und Hadoop verglichen. Für ein Verständnis, wie diese Operationen auf den Systemen abgebildet werden, sind sie nachfolgend erläutert. Die Beschriftung der Partitionen in den Abbildungen ist äquivalent mit Knoten im MapReduce Cluster.

2.4.1. Collocated Join

Ein „Collocated Join“ findet dann statt, wenn sich die zusammenzuführenden Fragmente der Tabellen, auf denen der Join ausgeführt wird, auf derselben Partition befinden. Die Datenbank besitzt im Gegensatz zu Hadoop aufgrund der Tabellendefinition die Information,

welche Bereiche eines Schüsselfeldes, über das „gejoined“ wird, sich auf welcher Partition befinden. Deshalb kann nur sie diesen Join ausführen, Hadoop, bzw. Pig nutzt für diese Aufgabe den Directed Join. Die Abbildung 2.8 zeigt das Verfahren eines Collocated Joins zwischen der Tabelle A und der Tabelle B.

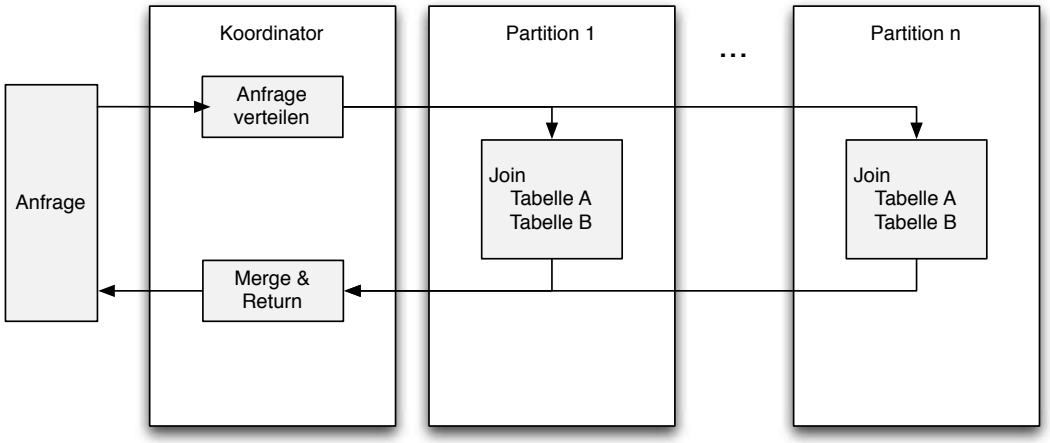

Abbildung 2.8.: Vereinfachte Darstellung eines collocated Join

Sowohl Tabelle A als auch Tabelle B sind so auf den Partitionen verteilt, dass die Tabellenfragmente, die „gejoined“ werden müssen, in der selben Partition liegen. Die koordinierende Partition teilt die Anfrage für den Join so zwischen den Partitionen auf, dass jede Partition einen lokalen Join zwischen den Tabellenfragmenten durchführen kann. Anschließend werden die Ergebnisse auf der koordinierende Partition zusammengefasst. Diese Join-Variante ist zu bevorzugen, da bis auf das Endergebnis keine Daten kopiert werden müssen.

2.4.2. Replicated / Broadcast Join

Zu einem „Replicated Join“ oder „Broadcast Join“ kann es kommen, wenn eine sehr große, über mehrere Partitionen bzw. Knoten verteilte Tabelle mit einer im Verhältnis dazu sehr kleinen Tabelle zusammengeführt wird. Die Abbildung 2.9 zeigt den Replicated Join zwischen der kleinen Tabelle A und der großen Tabelle B.

Im ersten Schritt werden alle Fragmente der Tabelle A auf alle Partitionen repliziert, auf denen Fragmente der Tabelle B liegen. Im zweiten Schritt werden auf jeder Partition die kopierten Fragmente zusammengesetzt, so dass im Endeffekt jede Partition eine komplette Kopie der Tabelle A besitzt. Im dritten Schritt wird auf jeder Partition eine Join-Operation zwischen den lokal liegenden Fragmenten der Tabelle B und der replizierten Tabelle A durchgeführt. Die Ergebnisse werden auf der koordinierenden Partition zusammengefügt.

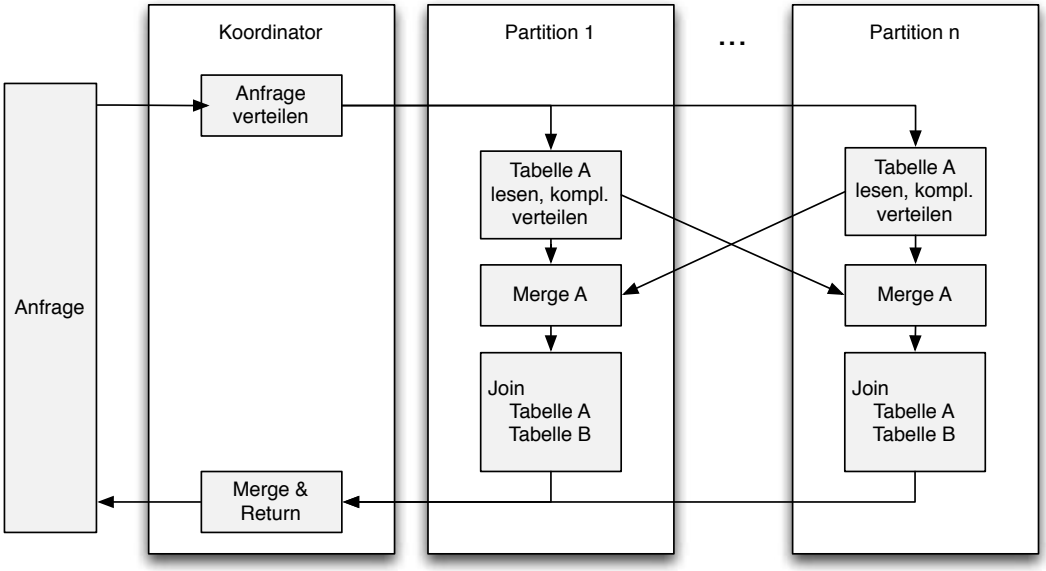

Abbildung 2.9.: Vereinfachte Darstellung eines replicated Join

Da die Tabelle A auf jede Partition kopiert wird, wird deutlich, dass dieses Verfahren nur mit entsprechend kleinen Tabellen funktioniert. Pig geht beim Replicated Join sogar noch einen Schritt weiter, hier erhält jeder Mapper eine Kopie der Tabelle A und hält sie komplett im RAM. Dies ermöglicht dem MapReduce Framework, die Join-Operation bereits in der Map-Phase durchzuführen. Da Pig keine Informationen über die Tabellengrößen besitzt, muss ein Replicated Join explizit durch den Anwender gefordert werden.

2.4.3. Directed Join

Ein „Directed Join“ wird immer dann durchgeführt, wenn es keine Möglichkeit gibt, eine der beiden anderen Join-Verfahren durchzuführen, denn der Directed Join benötigt im Vergleich zu den anderen Varianten mehr Ressourcen. Abbildung 2.10 stellt den Directed Join symbolisiert dar.

Für jeden Datensatz aus Tabelle A und Tabelle B wird auf allen Partitionen nach demselben Verfahren über die Schlüsselfelder der Join-Operation ein Hashwert gebildet. Jedem Hashwert wird eindeutig eine Partition zugeordnet, auf die diese Datensätze anschließend kopiert und dort zu neuen Tabellenfragmenten zusammengesetzt werden. Da die Hashwerte beider Tabellen nach demselben Verfahren aus den Schlüsselfeldern berechnet wurden, können die Tabellenfragmente nun jeweils lokal „gejoined“ werden. Die Ergebnisse werden anschließend auf der koordinierenden Partition zusammengefügt.

Diese Join-Operation wird auch von Pig vergleichbar ausgeführt, allerdings werden die Hashwerte als Schlüssel bereits in der Map-Phase berechnet, das Kopieren der Datensät-

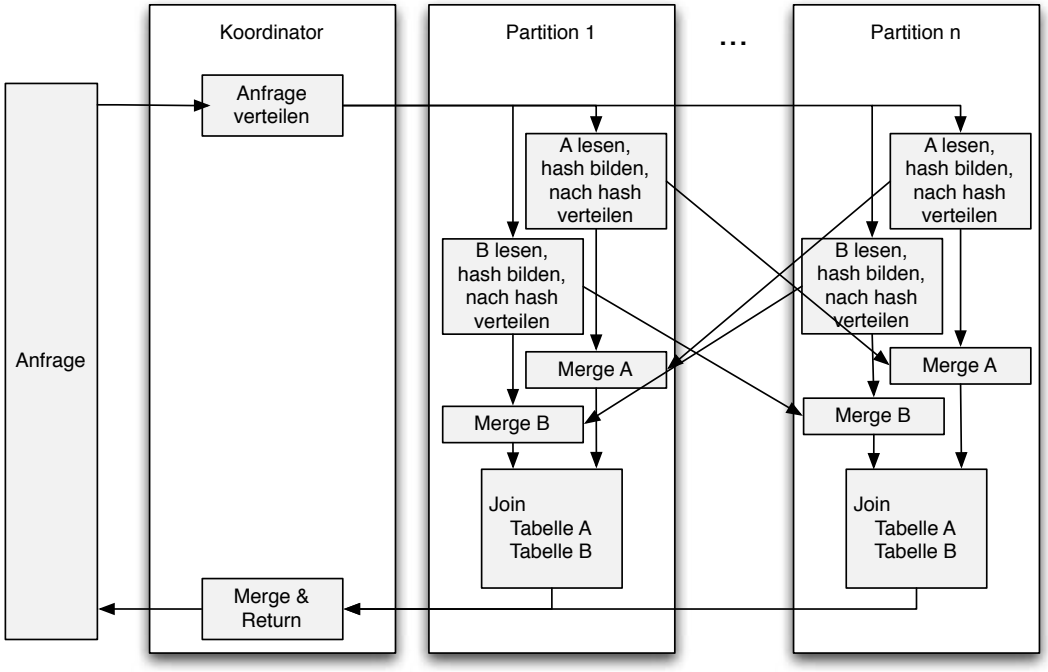

Abbildung 2.10.: Vereinfachte Darstellung eines directed Join

ze erfolgt automatisch innerhalb der Shuffle-Phase und die endgültige Join-Operation wird in der folgenden Reduce-Phase durchgeführt.

3. Versuch

Die erste Hälfte des Kapitels widmet sich dem Versuchsaufbau. Es wird beschrieben, wie die Versuchsanordnung hardwareseitig aufgebaut ist und welche Software eingesetzt wird. Es wird erläutert, wie Hadoop und DB2 für die Versuche aufgesetzt wurden und gezeigt, wie die gemessenen Daten ausgewertet werden. Die zweite Hälfte beschäftigt sich mit der Versuchsdurchführung, wobei sich der Versuch in zwei Bereiche gliedert. Im ersten Teil werden Tests von grundlegenden Funktionen mit künstlichen Daten durchgeführt. Es wird gezeigt, warum diese Tests erstellt werden und wie die dazu gehörigen Daten aufgebaut sind. Die Tests werden sowohl in SQL als auch in Pig dargelegt. Im zweiten Teil werden Tests mit realen Daten durchgeführt. Hier wird eine Einführung in die Daten gegeben und beschrieben, wie sie für die Verarbeitung aufbereitet werden mussten. Im Anschluss erfolgt eine Darstellung von zwei ausgewählten Tests in SQL und Pig.

3.1. Aufbau

Bär u. a. [Bär+11] haben bereits einen ähnlichen Vergleich zur Auswertung von Netzwerkstatistiken zwischen einem MapReduce Framework und einer relationalen Datenbank angestellt. Erschwert wird der Vergleich bei ihnen dadurch, dass dort sehr unterschiedliche Hardware für die Datenbank und das MapReduce Framework eingesetzt wurde. Während die Datenbank auf einem einzelnen System lief, war Hadoop auf einem Cluster installiert. Besser lassen sich die Ansätze vergleichen, wenn beide auf derselben Hardware laufen.

Um die gestellten Fragen zu beantworten, muss also ein Cluster eingesetzt werden, auf dem sowohl eine verteilte Datenbank laufen kann, als auch ein MapReduce Framework.

Um Aussagen über die Nutzung der Hardwareressourcen auf den Systemen treffen zu können, wurden während der Versuchsdurchführung möglichst viele Kennzahlen über die Auslastung aufgezeichnet. Weiterhin wurde die komplette Netzwerkkommunikation zwischen den Knoten protokolliert um sie später genauer auswerten zu können.

Der hardwareseitige Aufbau ist relativ einfach, er besteht aus einem Cisco Catalyst 2960S Gigabit Ethernet Switch mit zwei 10GbE SFP Ports (kurz mit fussnote erläutern was das genau ist) sowie 6 Rechnern: Master,Node1 - Node4 und Sniffer.

Wie in Abbildung 3.1 zu sehen, sind Master, Node1 - Node4 per Gigabit Ethernet (GbE) am Switch angeschlossen während Sniffer am 10 GbE Port angeschlossen ist. Der 10 GbE Port wird im Monitormodus betrieben und spiegelt somit den gesamten Netzwerkverkehr an Sniffer.

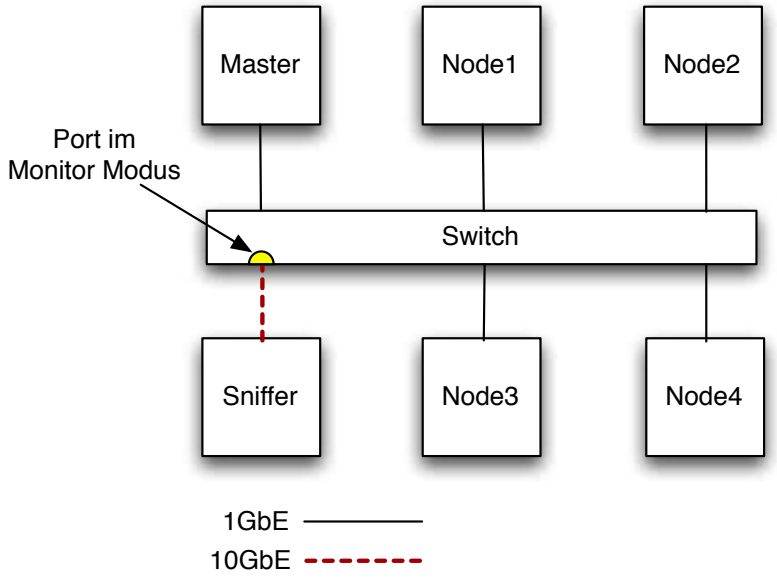

Abbildung 3.1.: Hardware Aufbau

Die eingesetzte Serverhardware ist Tabelle 3.1 zu entnehmen, auf allen Systemen ist Debian Linux 6.0 Squeeze als Betriebssystem installiert. Weiterhin sind auf allen Komponenten zur besseren Ausnutzung des Netzwerkes Jumboframes aktiviert.

Servername	Master, Node1, Node2, Node3, Node4	Sniffer
CPUs	2x 2.5GHz Intel Xeon	4x 1.8GHz AMD Opteron 865
Summe Cores	8	8
Arbeitsspeicher	16 GB	16GB
Plattenspeicher	4x 146 GB im RAID 0	500 GB
Ethernet	Intel E1000e	Myricom Myri10GE

Tabelle 3.1.: Eingesetzte Server-Hardware

3.2. Eingesetze Software

Als MapReduce Framework wird in dieser Arbeit Apache Hadoop (siehe 2.3.2) eingesetzt, da es nach Stonebraker u. a. [Sto+10] das momentan meistbenutzte MapReduce Framework ist und es als Open Source Software kostenlos zur Verfügung steht. Mit Pig steht auch für Hadoop eine Platform zur interaktiven Analyse großer Datenmengen mittels der abstrakten

Sprache Pig Latin zur Verfügung. Es wird mit der Hadoop Version 0.20.2-cdh3u1 sowie Pig Version 0.8.1-cdh3u1 von der Cloudera Distribution getestet.

Als verteile Datenbank kommt IBM DB2 zum Einsatz, da sie mittels des DPF Features eine Möglichkeit zur Verteilung der Daten im Cluster zur Verfügung stellt, vielfältige Optimierungsmöglichkeiten bietet und sich im TPC-H Benchmark[1] des Transaction Processing Performance Council seit Jahren auf den vorderen Plätzen befindet. Es wird hier die Version DB2 ESE 9.7 FP 6 LUW aus dem IBM InfoSphere Warehouse 9.7 eingesetzt.

3.2.1. Konfiguration Hadoop

Wie in Kapitel 2.3.2 beschreiben, besteht Hadoop aus zwei Teilen, dem MapReduce Framework und dem verteilten HDFS-Filesystem. In Abb. 3.2 ist zu sehen, dass auf Master die Prozesse Namenode und secondarynamenode des HDFS sowie der jobtracker von MapReduce laufen. Auf Node1 - Node4 laufen jeweils die HDFS Datanode und der MR Tasktracker.

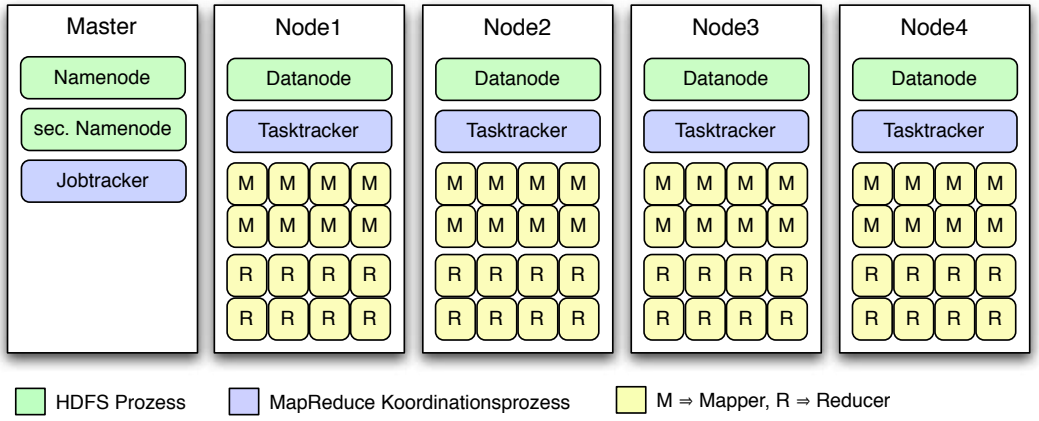

Abbildung 3.2.: Hadoop Prozesse

Um reproduzierbare Zeiten zu erhalten, wird immer nur maximal ein MapReduce Job gleichzeitig auf dem Cluster aktiv. Deshalb konnten pro Knoten sowohl max. 8 Mapper als auch max. 8 Reducer parallel laufen, denn der eigentliche Rechenaufwand in den Reducern findet erst statt, nachdem alle Mapper mit ihrer Arbeit fertig sind. Alle acht CPU-Cores in jedem Knoten sollten maximal ausgelastet werden ohne unnötig viele Task-Wechsel zu erzeugen, bzw. zu viele Plattenzugriffe zu verursachen.

[1] „The TPC Benchmark H (TPC-H) is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.“ Transaction Processing Performance Council [Tra12]

Da aufgrund der relativ kleinen Festplatten Speicherplatz gespart werden muss und der Cluster nicht als Produktivsystem genutzt wird, wird der HDFS Replikationsfaktor auf „1“ gesetzt. Da alle genutzten System gleich schnell sind, die zu ladenen Daten gleichmäßig über alle Systeme verteilt werden und da die Replikation der HDFS-Blöcke ansonsten asynchron laufen würde, hat das keine weiteren Auswirkungen auf die Ausführungszeiten. Alle wichtigen Einstellungen finden sich noch mal im Anhang A.2.

3.2.2. Konfiguration DB2

Wie in Kapitel 2.2 beschrieben, wird eine verteilte Datenbank mit DB2 aufgesetzt, indem man mehrere einzelne Datenbanken auf verschiedenen Systemen als einzelne Partitionen in einem gemeinsamen Datenbank-Kontext zusammenfasst.

Da mit DB2 eine möglichst ähnliche Konfiguration wie mit Hadoop geschaffen werden sollte, wurde Master nur zur Koordination der restlichen Knoten verwendet. Hier liegt, wie in Abb. 3.3 zu sehen ist, nur die Partition ”0” mit dem Systemkatalog und den kleineren Tabellen, bei denen der Aufwand des Verteilens unnötig hoch wäre.

Um auch von der CPU-Belegung eine vergleichbare Konfiguration zu Hadoop zu schaffen, lief DB2 auf Node1-Node4 jeweils acht Mal als ”Single-CPU” Instanz, also jeweils eine Partition pro CPU Core.

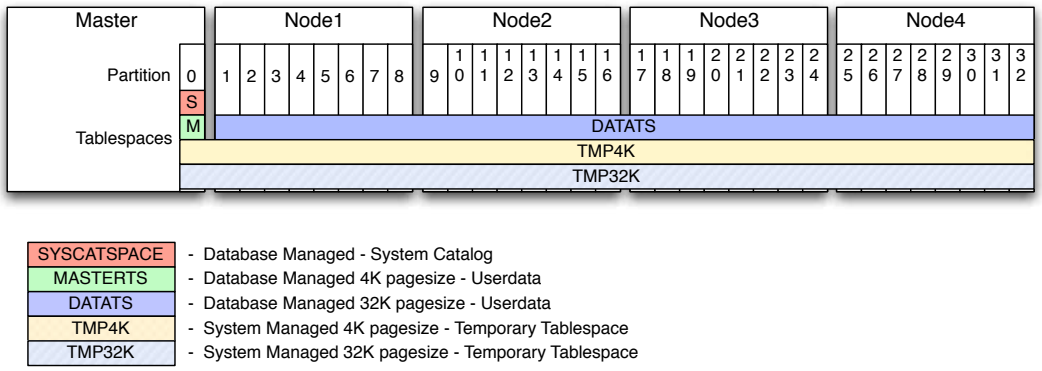

Abbildung 3.3.: DB2 Tablespaces

Es ist absehbar, dass das bestehende Plattensystem voraussichtlich mit den vielen Datenbank-Instanzen pro Rechner etwas überlastet sein werden, worunter die Zugriffsgeschwindigkeit leidet. Allerdings sollen in diesem Vergleich die Konfigurationen so ähnlich wie möglich gestaltet werden und auch eine höhere Belastung des Plattensystems ist eine Aussage.

Wie im HDFS unter Hadoop werden auch bei DB2 die großen Tabellen über Node1-Node4 verteilt. Hierzu wird ein 32K Seitengröße Tablespace namens Datats erstellt, der sich über alle Partitionen auf Node1 - Node4 zieht. Zusätzlich benötigt die Datenbank „Temporary Tablespaces“ auf allen Partitionen um Zwischenergebnisse ablegen zu können.

Auf jedem System wird ausreichend Arbeitsspeicher für das System gelassen, der restliche Speicher jeweils gleichmäßig den Partitionen zugeteilt. Jede Partition bekommt einen kleinen Bufferpool mit 4K Seitengröße, einen recht großen Bufferpool mit 32K Seitengröße sowie etwas Sortierspeicher zugeteilt. Um die Laufzeit der Anfragen soweit wie möglich zu reduzieren wurde der Optimierungsgrad der Datenbank auf das Maximum gestellt. Der Optimierer benötigt zwar zum Vergleichen weiterer, über den Standard hinausgehender Optimierungsmöglichkeiten, mehr Zeit, aufgrund der recht hohen Laufzeit der Anfragen lohnt sich dieser Aufwand beim Umgang mit Massendaten trotzdem. Alle von der Standardkonfiguration abweichenden Parameter sind im Anhang A.1 zu finden.

3.3. Messungen

Es wird gemessen, was auf und zwischen den Systemen während der Ausführung der Tests passiert.

Zur Messung der Systemauslastung setzt diese Arbeit auf die Sysstat Utilities von Godard [God12]. Es werden hier im Sekundenintervall auf Master und Node1 - Node4 während der Abarbeitung der Anfragen alle im Anhang B aufgezählten Parameter protokolliert.

Zum Mitschreiben des Netzwerkverkehrs wird der Rechner Sniffer eingesetzt. Er ist mit einer 10 GbE Netzwerkkarte ausgestattet und bekommt, angeschlossen an einen ebenfalls 10GbE Port des Switches, den gesamten Netzwerkverkehr zwischen Master und Node1 - Node4 gespiegelt. Hier werden die ersten 100Bytes jedes Paketes mittels tcpdump mitgeschrieben.

Weiterhin werden zu jeder Anfrage die Ausführungspläne von DB2 bzw. Pig sowie die MapReduce Ausführungsprotokolle von Hadoop erzeugt und gesichert. Für DB2 wurde dafür innerhalb dieser Arbeit ein Programm entwickelt, das die Ergebnisse einer EXPLAIN Anfrage aus den Systemtabellen extrahiert und in eine „dot“-Datei für das Programm Graphviz konvertiert.

Die Messwerte der Sysstat Utilities werden nach der Ausführung der Tests in eine Sqlite-Datenbank eingelesen. Sqlite ist eine Datenbank, die für den Embedded-Einsatz entworfen wurde, sie benötigt keinen eigenständigen Server und unterstützt ein Großteil der SQL Befehle[2]. Ein weiteres Verarbeiten der Daten aus Skripten wird dadurch erleichtert.

Aus dem aufgenommenen Netzwerkverkehr werden nachträglich nur die benötigten Ports für die Kommunikation zwischen den Knoten von DB2 bzw. Hadoop gefiltert. Diese Kommunikation wird im nächsten Schritt mit Hilfe des Tools tcptrace von Ostermann [Ost12] analysiert. Es werden Datensätze für das TCP-Transfervolumen, geöffnete und geschlossene

[2] Nach SQL-92 Standard

TCP-Verbindungen sowie Paketverluste und Triple DupACKs erzeugt. Die ermittelten Ergebnissen des Transfervolumens wird in einem weiteren Schritt um die Fehler durch Integer Overflows bereinigt.

Ein weiteres Programm aggregiert die übertragenen TCP-Pakete im Sekundentakt einmal nach Quelle und Ziel der Pakete sowie nach ihrer Größe. Alle diese Ergebnisse werden zur weiteren Verarbeitung ebenfalls in die Sqlite-Datenbank geladen.

Die Ausführungspläne werden mit Hilfe von Graphviz visualisiert, die Daten aus der Sqlite-Datenbank mit Hilfe von Gnuplot.

3.4. Synthetische Tests

Im ersten Teil dieser Arbeit werden synthetische Tests durchgeführt, d. h. Tests, in denen bestimmte Basisfunktionen der Datenverarbeitung in einem vorhersagbaren Rahmen analysiert werden können, in dem alle für die Tests notwendigen Daten in einem definierten Schema künstlich erzeugt wurden. Um Anhaltspunkte zur Verarbeitung von großen Datenmengen zu bekommen, wird eine relativ große Tabelle erstellt, auf deren Daten sich die Anfragen beziehen.

Die Tests von Pavlo u. a. [Pav+09] zeigten bereits deutliche Unterschiede zwischen Datenbanken und MapReduce. Diese Diplomarbeit setzt bei den synthetischen Tests auf vergleichbare Funktionen und ergänzt sie um weitere. Allerdings werden die Tests für diese Arbeit nur in einem Cluster und nur mit einer festen Anzahl von Knoten durchgeführt.

Damit möglichst wenige externe Einflüsse, wie z. B. das Übertragen der Daten vom und zum anfragenden Clienten, das Ergebnis verfälschen können, werden bei diesen Tests folgende Maßnahmen ergriffen:

Beim Laden der Daten werden die Daten nicht von der Festplatte gelesen, sondern über eine „Named Pipe“ direkt aus einem Script, das die Daten „on the Fly“ generiert. So fällt die Belastung der Festplatte beim Einlesen der Daten nicht ins Gewicht. Das Script benötigt zum Generieren der Daten ohne parallel laufenden Ladeprozess ca. 3900 Sekunden und belegt nur eine CPU auf Master. Dies ist schneller als alle Ladetests und stellt somit auch CPU-seitig keinen Bottleneck dar.

Vor der Rückgabe der Ergebnisse an den Client werden die Daten aggregiert, um die Datenmenge zu reduzieren. Das hat bei den Tests zwar den Vorteil, dass die für die Übertragung der Daten an den anfragenden Clienten benötigte Zeit reduziert wird, allerdings hat es den Nachteil, dass der Server einen weiteren Verarbeitungsschritt durchführen muss. Gerade bei Hadoop kann diese Methode dafür sorgen, dass ein sonst evtl. unnötiger Reduce - Schritt

durchgeführt werden muss. Da diese Arbeit aber davon ausgeht, dass ein reines Lesen der Daten ohne weitere Verarbeitung unwahrscheinlich ist, wird dies in Kauf genommen.

Folgende Funktionen wurden zum Testen ausgewählt:

Laden der Daten Da das Laden von Daten immer der erste nötige Schritt vor der Auswertung ist, und bei größeren Datenmengen auch einen beträchtlichen Zeitraum einnehmen kann, muss es getestet werden. Besonders Hadoop kann hier seine Stärken zeigen.

Tablescan / Aggregation Interessant ist der Tablescan, also das sequenzielle Einlesen einer gesamten Tabelle. Wegen der bereits im vorherigen Absatz erwähnten Einschränkung folgt jedoch zudem noch eine Aggregation der Daten, um die Rückgabemenge zu reduzieren.

Filter Eine häufige Operation in der Datenverarbeitung ist das Selektieren von Daten, bzw. das Einschränken nach bestimmten Kriterien. Auch hier werden die Daten zur Reduktion der Rückgabemenge in einem weiteren Schritt aggregiert.

Group Beim Gruppieren wird die Datenmenge nach einem bestimmten Kriterium aggregiert, d.h. es wird eine Gruppe für jeden Wert dieses Kriteriums gebildet und über diese Gruppe aggregiert.

Distinct Die Distinct Funktion ermöglicht es, doppelte Werte zu vermeiden, sie ist in diesem Vergleich auch deshalb interessant, da DB2 diese Funktion mit Hilfe einer Sortierung abbildet, während Pig dazu eine Hashtabelle einsetzt.

Collocated Join Wie in 2.4.1 beschrieben.

Replicated Join Wie in 2.4.2 beschrieben.

Directed Join Wie in 2.4.3 beschrieben.

3.4.1. Daten für die synthetischen Tests

Da es in dieser Arbeit um große Datenmengen geht, muss mindestens eine große Tabelle vorhanden sein. Groß bedeutet in diesem Zusammenhang, dass die Tabelle mindestens fünf mal so groß sein sollte wie der gesamte Arbeitsspeicher im Cluster um sicher zu stellen, dass die Anfragen nicht aus dem Speicher beantwortet werden können. Um nicht unnötig viele Daten in die Systeme zu laden, sollen alle Tests auf denselben Daten ausgeführt werden.

Um die Anfragen etwas verständlicher zu gestalten, wurde für die Tests und Daten ein Beispielszenario erarbeitet. In diesem Szenario wird von 10.000 mobilen Endgeräten ausgegangen, die 24 Stunden lang im Minutentakt jeweils 100 Ziele „anpingen“ und die gemessene Round-Trip Zeit speichern. Jedes dieser Endgeräte ist dabei über genau einen Internet-

Provider angebunden. Da sich die Endgeräte bewegen können, wird jede Minute zusätzlich die Position des Gerätes festgehalten.

Um die Datensätze weiter zu vergrößern wurde zusätzlich ein Payload von 192 Bytes als String an jede Zeile angehängt. Der Payload stellt weitere in der Datenbank gespeicherte Felder zu den Datensätzen dar, die für die ausgeführten Anfragen nicht gebraucht werden aber trotzdem Speicher benötigen. Da DB2 eine zeilenbasierte Datenbank ist, „leidet“ sie genauso wie Hadoop unter dieser Maßnahme. Um so größer die Payload ist, um so weniger Datensätze passen bei DB2 auf dieselbe Speicherseite, bzw. bei Hadoop in ein Split. Dadurch werden mehr Leseoperationen zum Lesen der selben Anzahl von Datensätzen nötig. Indizes sind durch diese Maßname nicht betroffen, da sich diese Felder nicht im Index befinden. Da weder bei DB2 noch Hadoop die Kompression der Daten eingeschaltet ist, bringt das Wiederholen ein und derselben Zeichenkette in jedem Datensatz für keines der Systeme Vorteile.

Daraus ergeben sich wie in Abb. 3.4 zu sehen drei Tabellen:

Latency Diese Tabelle enthält neben der Endgeräteidentifizierung, der Zeit, dem Ziel und der gemessenen RTT des Pings noch die eingangs beschriebene Payload.

Position Diese Tabelle beinhaltet das Endgerät, die Zeit und die Position in Längen- und Breitengrad, an der sich das Endgerät zum Messzeitpunkt aufgehalten hat.

Metadata In dieser Tabelle stehen Zusatzinformationen zu den Endgeräten, in diesem Fall der ISP.

Weiterhin erhält die Tabelle Latency drei Indizes:

LAT_TARGET Ein Index nur über die Spalte Target.

LAT_HOUR_UNIT Ein Index über die Spalten Hour und Unit .

LAT_UNIT_HOUR Ein Index über die Spalten Unit und Hour .

Für weiterführende Fragen sind die DDLs im Anhang C abgedruckt, die Größe der Quelldaten für die Tabellen ist der Tabelle 3.2 zu entnehmen.

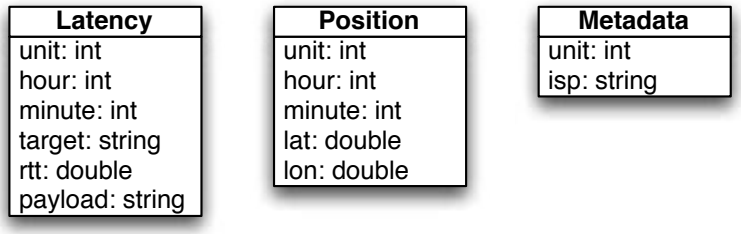

Abbildung 3.4.: Tabellenstruktur synthetische Daten

Tabellenname	Anzahl der Datensätze	Tabellengröße
Latency	1.440.000.000	ca. 330 GB
Position	14.400.000	ca. 434 MB
Metadata	10.000	ca. 450 KB

Tabelle 3.2.: Tabellengröße der synthetischen Daten

Die Daten werden während des Ladeprozesses mittels eines Programms generiert. Während die Daten für UNIT, HOUR, MINUTE und TARGET mit Hilfe eines Zählers durchgezählt werden, werden RTT, LAT, LONG und ISP durch einen Zufallsgenerator gesetzt. Für TARGET und ISP sind jeweils 100 bzw. 10 feste Zeichenketten definiert, PAYLOAD besteht immer aus der selben Zeichenkette.

Als Datenformat für den Transfer der Daten wird CSV genutzt, es kann sowohl vom Ladeprozess der Datenbank als auch von Pig gelesen werden. Für Pig ein anderes Format wie z. B. Hadoop SequenceFiles einzusetzen bringt nach Stonebraker u. a. [Sto+10] keine Vorteile, da die Daten in beiden Fällen vom Maptask dekodiert werden müssen. In den dortigen Tests waren die SequeceFiles sogar langsamer.

Beim Einladen in die Datenbank werden die Daten aus LATENCY nach einem Hashwert aus UNIT und HOUR über die Partitionen verteilt, bei der Tabelle POSITION wird nach einem Hashwert aus LAT und LON verteilt.

3.4.2. Laden der Daten

Wie in 3.4.1 beschrieben, werden die Daten erst beim Laden von einem Programm erzeugt und über eine „Named Pipe“ vom Ladeprozess gelesen.

Getestet wird das Laden sowohl mit Primärschlüssel, Indizes und dem Generieren von Tabellenstatistiken, als auch ohne Primärschlüssel, Indizes und Statistik. Es wird nur das Laden der großen LATENCY Tabelle gemessen, die anderen Tabellen werden nicht betrachtet. Für die Tests ohne Primärschlüssel und Indizes wurden die Tabellen nochmals ohne Primärschlüssel und Indizes angelegt.

Der Ladebefehl für DB2 mit Index und Statistik ist im Listing 3.1 zu sehen, die Befehl ohne Index und Statistik in Listing 3.2.

```
1 LOAD FROM latency.csv OF DEL MODIFIED BY ANYORDER REPLACE INTO latency STATISTICS
      YES AND INDEXES ALL NONRECOVERABLE DISK_PARALLELISM 1 INDEXING MODE REBUILD
      ALLOW NO ACCESS
```

Listing 3.1: DB2 Ladebefehl mit Index und Statistik

```
LOAD FROM latency.csv OF DEL MODIFIED BY ANYORDER REPLACE INTO latency STATISTICS
    NO NONRECOVERABLE DISK_PARALLELISM 1 ALLOW NO ACCESS
```

Listing 3.2: DB2 Ladebefehl ohne Index und Statistik

Da unter Pig keine Indizes existieren, wird hier nur direkt auf den Daten gearbeitet, geladen wird wie in Listing 3.3 zu sehen mittels eine Kopierbefehls auf das HDFS.

```
hadoop fs -put latency.csv /bench/latency.csv
```

Listing 3.3: Hadoop Kopieren in des HDFS

3.4.3. Tablescan / Aggregation

Zuerst erfolgt eine einfache Anfrage, in der alle Datensätze ein Mal gelesen werden müssen und über das Feld RTT der Mittelwert gebildet wird. Listing 3.4 zeigt die SQL Anfrage.

```
select avg(rtt)
from latency
```

Listing 3.4: SQL Aggregation

In einfachen Fällen erkennt der Pig Optimierer automatisch, welche Felder genutzt werden und führt die Projektion selbständig am Anfang durch. Eine Aggregation lässt sich hier nur, wie in 3.5 zu sehen, mittels einer vorher gebildeten Gruppe über alle Datensätze durchführen.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Grp = GROUP Latency ALL;
Result = FOREACH Grp GENERATE AVG(Latency.rtt);

DUMP Result;
```

Listing 3.5: Pig Aggregation

3.4.4. Filter

Als nächstes wird ein Test durchgeführt, in dem die Datensätze der LATENCY Tabelle nach der Stunde gefiltert werden. Am Ende wird wie in der SQL Anfrage in Listing 3.6 zu sehen wieder der Mittelwert gebildet um die Datenmenge, die an den Clienten zurückgeliefert

wird, zu reduzieren. Hier sollte die Datenbank durch den Einsatz eines Indexes sichtbare Vorteile haben.

```
select avg(rtt)
from latency
where hour between 9 and 17
```

Listing 3.6: SQL Filter

Das Listing in 3.7 zeigt die selbe Anfrage in Pig.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Fltr = FILTER Latency BY hour >= 9 and hour <= 17;
Grp = GROUP Fltr ALL;
Result = FOREACH Grp GENERATE AVG(Fltr.rtt);

DUMP Result;
```

Listing 3.7: Pig Filter

3.4.5. Group

Im nächsten Test werden Daten gruppiert. Es soll die durchschnittliche Roundtrip-Time in jeder Stunde gebildet werden. Listing 3.8 zeigt die Anfrage in SQL, Listing 3.9 die Anfrage in Pig.

```
select hour,avg(rtt)
from latency
group by hour
```

Listing 3.8: SQL Group

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Grp = GROUP Latency BY hour;
Result = FOREACH Grp GENERATE group,AVG(Latency.rtt);

DUMP Result;
```

Listing 3.9: Pig Group

3.4.6. Distinct

Auch beim Distinct kann ein Index der Datenbank sehr hilfreich sein, deshalb wird auch dieser Test auf der Datenbank sowohl mit als auch ohne Index durchgeführt. Listing 3.10 zeigt wieder die Anfrage in SQL, Listing 3.11 die Anfrage in Pig.

```
select distinct target
from latency
```

Listing 3.10: SQL Distinct

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Targets = FOREACH Latency GENERATE target;
Result = DISTINCT Targets;

DUMP Result;
```

Listing 3.11: Pig Distinct

3.4.7. Collocated Join

Als Nächstes werden die verschiedenen Join Typen, wie in 2.4 beschrieben, getestet. Listing 3.12 zeigt die Anfrage für einen „collocated join“ in SQL. Als erstes werden für jede Einheit und Stunde die durchschnittliche RTT gebildet und die Ergebnisse wieder mit der Latency Tabelle „gejoined“. Im nächsten Schritt wird durch eine Gruppierung nach der Stunde die Ergebnismenge wieder reduziert. Die Aussage des Ergebnisses dieser Anfrage ist auch hier nicht relevant, die Anfrage wurde so konstruiert, dass die Verteilung der beiden Tabellen gleich ist. In diesem Fall kann ein „collocated Join“ angewandt werden und der Join jeweils lokal auf allen Systemen ausgeführt werden. Erst danach müssen die Daten für die Gruppierung verteilt werden. Um zu verhindern, dass der DB2 Optimierer die Anfrage in eine reine Gruppierung umwandelt, wurde für diese Anfrage der Optimierungsgrad auf den Standardwert „5“ heruntergesetzt.

```
select l.hour, avg( l.rtt/a.average)
from latency l
join (
   select unit,hour,avg(rtt) as average
   from latency
   group by unit,hour
 ) as a on l.unit=a.unit and l.hour=a.hour
group by l.hour
```

Listing 3.12: SQL Collocated Join

Pig kennt keine „collocated Joins“, da Pig keine Informationen darüber besitzt, wie die Daten verteilt sind. Hier wurde wie in Listing 3.13 zu sehen ein normaler Join zum Vergleich eingesetzt.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Proj = FOREACH Latency GENERATE unit, hour, rtt;
HourlyGrp = GROUP Proj BY (unit, hour);
HourlyAverage = FOREACH HourlyGrp GENERATE group,AVG(Proj.rtt) as rtt;
Joined = JOIN HourlyAverage BY (group.unit, group.hour), Proj BY (unit, hour);
Quotient = FOREACH Joined GENERATE Proj::unit, Proj::hour, Proj::rtt /
    HourlyAverage::rtt as rttq;
Grp = GROUP Quotient BY hour;
Result = FOREACH Grp GENERATE group,AVG(Quotient.rttq);

DUMP Result;
```

Listing 3.13: Pig Collocated Join

3.4.8. Replicated Join

Es folgt der „replicated“ oder auch „broadcast“ Join. Hier muss eine Tabelle komplett auf alle Knoten verteilt werden. Das der einfachste Weg unter DB2 ist deshalb, die zu verteilende Tabelle (in diesem Fall die Tabelle Metadata) auf Master zu legen, denn dann muss sie auf jeden Fall als erstes auf Node1 - Node4 kopiert werden, bevor „gejoined“ werden kann. Die SQL Anfrage dazu ist in Listing 3.14 zu sehen.

```
select isp, avg(rtt)
from latency l
join metadata m on l.unit=m.unit
group by isp
```

Listing 3.14: SQL Replicated Join

Pig unterstützt ebenfalls einen „replicated Join“. Man bringt Pig dazu, diesen Join zu nutzen, indem man „USING ’replicated’“ an den Join-Befehl anhängt. Ein „replicated Join“ in Pig hat zusätzlich zum Verteilen der Daten den großen Vorteil, dass schon in der Map-Phase „gejoined“ wird. Allerdings muss dafür sicher gestellt sein, dass die replizierte Tabelle in den RAM Speicher aller Mapper passt. Die Pig Anfrage ist in Listing 3.15 zu sehen.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Metadata = LOAD '/bench/metadata.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, isp:chararray);

Proj = FOREACH Latency GENERATE unit, rtt;
Joined = JOIN Proj BY unit, Metadata BY unit USING 'replicated' ;
Grp = GROUP Joined BY isp;
Result = FOREACH Grp GENERATE group,AVG(Joined.rtt);

DUMP Result;
```

Listing 3.15: Pig Replicated Join

3.4.9. Directed Join

Um einen „directed Join“ zu erreichen, wurde bewusst die Tabelle POSITION nach einem anderen Schlüssel verteilt als die Tabelle LATENCY. In einem produktiven System würde man voraussichtlich versuchen, beide Tabellen nach dem selben Schlüssel zu verteilen, um aus Geschwindigkeitsgründen möglichst einen „collocated Join“ zu nutzen. In der SQL Anfrage in Listing 3.16 muss die kleinere Tabelle POSITION nach UNIT und HOUR umsortiert werden, an die entsprechenden Knoten gesendet werden und kann dann dort mit der Tabelle LATENCY „gejoined“ werden. Am Ende erfolgt wieder eine Aggregation zur Datenreduktion.

```
select 10*int(p.lat/10), 10*int(p.lon/10), avg(l.rtt)
from latency l
join position p on l.unit=p.unit and l.hour=p.hour and l.minute=p.minute
group by 10*int(p.lat/10), 10*int(p.lon/10)
```

Listing 3.16: SQL Directed Join

Der „directed Join“ ist der normale Join in Pig, hier ist wie in Listing 3.17 zu sehen nichts Weiteres zu beachten.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Latency = LOAD '/bench/latency.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, target:chararray, rtt:double, payload:
       chararray);

Position = LOAD '/bench/position.csv'
```

```
9     USING org.apache.pig.piggybank.storage.CSVLoader()
10    AS (unit:int, hour:int, minute:int, lat:double, lon:double);
11
12  Proj = FOREACH Latency GENERATE unit, hour, minute, rtt;
13  Joined = JOIN Position BY (unit, hour, minute), Proj BY (unit, hour, minute);
14  PJoined = FOREACH Joined GENERATE Position::unit, Position::hour, Position::minute
        , 10*(int)(Position::lat/10) as lat, 10*(int)(Position::lon/10) as lon, Proj::
        rtt;
15  Grp = GROUP PJoined BY (lat, lon);
16  Result = FOREACH Grp GENERATE group,AVG(PJoined.rtt);
17
18  DUMP Result;
```

Listing 3.17: Pig Directed Join

3.5. BitTorrent Testszenario

Der zweite Teil der Arbeit befasst sich mit realen Anfragen auf realen Daten. Es wurde ein Datensatz aus einer Messung im BitTorrent (BT) Netzwerk ausgewählt.

Das BitTorrent Netzwerk ist ein Peer-to-Peer Netzwerk zum Austausch von Dateien. Im Gegensatz zu anderen Filesharing-Techniken bildet BitTorrent kein übergreifendes Netzwerk, sondern baut für jede Datei ein separates Verteilnetz auf. Die Menge der Computer oder auch Peers in diesem Verteilnetz wird Schwarm genannt. Jede Datei wird in sogenannte Segmente oder Chunks geteilt, die zwischen den Peers ausgetauscht werden. Die Peers unterteilen sich in zwei Gruppen: Die Seeder, dass sind die Peers, die die gesamte Datei, also alle Segmente, zur Verfügung stehen haben und die Leecher, denen noch Segmente fehlen. Ein Seeder bietet nur noch Segmente an, ein Leecher lädt Segmente von Seedern und anderen Leechern und bietet die bereits erhaltenen Segmente auch wieder anderen Leechern zum Download an.

3.5.1. Aufbau der Daten

Die Daten wurden im Rahmen der Masterarbeit „Measurement and Characterization of Content Distribution in BitTorrent“ von Krenc [Kre12] gesammelt. Stündlich wurde für jeden Peer in jedem „belauschten“ Schwarm ein Eintrag mit allen Informationen zu dem jeweiligen Peer sowie einer Liste aller in dieser Stunde von dem Peer empfangenen Bitfelder abgespeichert. Diese Bitfelder geben Aufschluss darüber, welche Segmente der Datei der Peer zu dem jeweiligen Zeitpunkt bereits empfangen hatte, bzw. welche noch fehlten.

Da die Daten in einem hierarchischen Format abgelegt wurden, mussten sie im ersten Schritt in ein relationales Format konvertiert werden. Für Hadoop hätte man auch Auswertungen in Java schreiben können, die direkt mit diesen Strukturen umgehen können, allerdings war

das nur in Pig, ohne eigene „externe“ Leseroutinen zu entwickeln, nicht möglich. Da für diese Arbeit der der Zugriff auf die Daten in Java vermieden werden sollte, machte das diesen Schritt für beide Systeme notwendig. Zusätzlich ergab sich durch das Laden ein und derselben Dateien und Datenstrukturen in DB2 und Hadoop eine bessere Vergleichbarkeit.

Wie in Abb. 3.5 zu sehen, wurde jede Stunde ein Ordner angelegt, in diesem Fall TIMEBIN genannt, der für jeden Schwarm eine komprimierte Datei mit den Einträgen aller Peers enthielt. Der schematische Aufbau eines Datensatzes für einen Peer ist in Abb. 3.6 dargestellt.

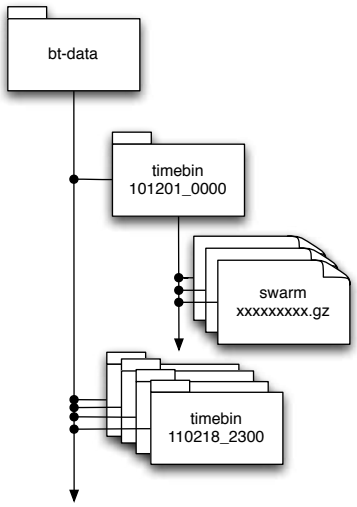

Abbildung 3.5.: BitTorrent Ordner

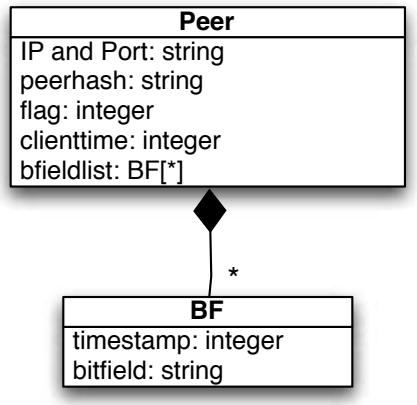

Abbildung 3.6.: BitTorrent Datensatz

3.5.2. Konvertieren der Daten in ein relationales Format

Da der Datensatz mit ca. 3,5 TB im Verhältnis zu der zur Verfügung stehenden Plattenkapazität von ca. 2 TB relativ groß war und so viele Daten wie möglich verarbeitet werden

sollten, wurden nicht nur die Listen der Bitfelder in eine Tabellenstruktur umgewandelt sondern zusätzlich noch eigene Tabellen für die TIMEBINS, SWARMS und PEERS angelegt um Redundanzen vermeiden zu können

Es wurde die in Abb. 3.7 zu sehende Tabellenstruktur erstellt und die Daten mittels eines extra zu diesem Zweck entwickelten Programms in jeweils eine CSV Datei pro Tabelle und Tiembin konvertiert, die direkt in diese Tabellenstruktur geladen werden konnte. Es wurde explizit darauf geachtet, nicht nur die für die gewählten Anfragen nötigen Daten zu übernehmen sondern eine Struktur zu schaffen, in der jedwede Anfrage auf den aufgenommenen Daten ausgeführt werden konnte.

In der Annahme, dass in dem Fall, in dem Anfragen auf den BITVEKTOR gemacht werden, die Anzahl der bereits vorhandenen Segmente eines Peers eine der wahrscheinlichsten Anfragen sein würde, wurde diese Anzahl, im Rahmen der Konvertierung bzw. Kompression (siehe Kapitel 3.5.3) der Bitvektoren, gleich mit berechnet und unter CHUNKS abgelegt. In diesem Fall lässt sich nun, um die Dauer der Anfrage zu verringern, ein Zugriff auf den Bitvektor vermeiden.

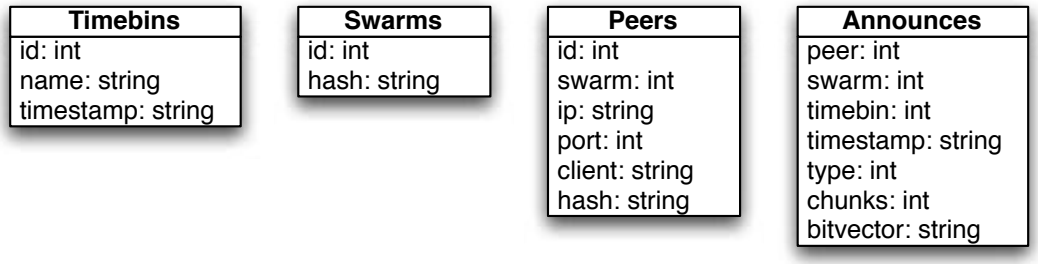

Abbildung 3.7.: Tabellenstruktur BitTorrent Daten

Da die Schlüssel für Timebins, Swarms, und Peers im Zuge der Konvertierung erst erzeugt werden, wurde in der Datenbank absichtlich auf die Nutzung von Primär- und Fremdschlüsseln verzichtet um den Ladevorgang zu beschleunigen. Als Verteilungsschlüssel wurde für die Datenbank sowohl für die PEERS als auch die ANNOUNCES Tabelle die Kombination aus Swarm und Peer genutzt, um eine möglichst gleichmäßige Verteilung zu erreichen und Joins zwischen diesen beiden größten Tabellen als „collocated Joins“ ausführen zu können. Die vergleichsweise kleinen Tabellen TIMEBINS und SWARMS wurden auf MASTER ausgelagert, hier ist bei Bedarf ein „replicated Join“ erforderlich.

3.5.3. Kompression der Bitvektoren

Eine weitere Überlegung zum Einsparen von Speicherplatz war das Komprimieren der Bitvektoren. Sie liegen als eine Zeichenkette von Hexadezimalziffern vor, bei der jedes Bit für das Vorhandensein bzw. Fehlen eines Segments steht. Um diese Zeichenkette weiterhin

einfach mit Anfragen in SQL oder Pig verarbeiten zu können, sollte keine allzu komplizierte Kompression mit Wörterbüchern o.ä. eingesetzt werden, sondern etwas, was sich in den jeweiligen Sprachen auch leicht dekodieren lässt. Der einfachste Ansatz wäre gewesen, jeweils zwei Ziffern beim Speichern zu einem Byte zusammen zu fassen, was einen konstanten Kompressionsfaktor von 2 ergeben hätte. Allerdings wurde in der Hoffnung auf einen noch höheren Kompressionsfaktor ein anderer Ansatz gewählt.

Es wurde davon ausgegangen, dass die häufigsten beiden Zustände der Peers sind, dass sie entweder alle oder fast alle Segmente zur Verfügung haben, also Seeder bzw. fast Seeder sind, oder keine oder nur wenige Segmente haben, also Leecher am Anfang der Übertragung. Im ersten Fall würde der Bitvektor nur aus einer Folge von „F“ mit möglicherweise keinen Unterbrechungen bestehen, im zweiten Fall aus langen Folgen von „0“ mit evtl. kleinen Unterbrechungen. Deshalb lag nahe, eine Lauflängenkompression (RLE) anzuwenden.

Um diese RLE möglichst einfach zu gestalten, wurde sie nicht auf Bit-Ebene sondern auf Ziffern-Ebene abgebildet. Der im Bitvektor genutzte Wortschatz bestand aus den Ziffern 0-9 und den Buchstaben A-F. Der Einfachheit halber wurden für die Längenangabe der Wiederholung nur die Buchstaben G-Z genutzt, wobei „G“ für das dreimalige hintereinander Auftreten eines Zeichens steht und „Z“ entsprechend für das 22-malige Auftreten. Beim doppelten Auftreten eines Zeichens wird, wie im unkomprimierten Bitvektor, das Zeichen zwei mal ausgegeben.

Ein kleines Beispiel für die Kompression ist in Abbildung 3.8 zu sehen.

AAAAAAAAEEEEEEEEEEEEEEEEEEEEAAAAAAAAAADEEEEEEEEEEEEEE

A + 7×A ⇒ AL
E + 19×E ⇒ EX
A + 9×A ⇒ AN
D + 0×D ⇒ D
E + 13×E ⇒ ER

Ergebnis: ALEXANDER

Original: 61 Zeichen Komprimiert: 9 Zeichen Einsparung: ~85%

Abbildung 3.8.: Beispiel zur Kompression der Bitvektoren

Die Kompression der Daten wurde innerhalb des Konvertierungsprogramms durchgeführt. Da in keiner der getesteten Anfragen in SQL oder Pig auf diese Daten zugegriffen werden musste, wurde keine Dekompression für SQL oder Pig entwickelt. Würde man dies tun, wäre der effizienteste Ansatz eine Benutzerdefinierte Funktion in Java, sie könnte sowohl bei DB2 als auch in Pig eingebunden werden.

Im Nachhinein wurde in einer Stichprobe über die Timebins von zwei Tagen nur ein Kompressionsfaktor von im Durchschnitt pro Timebin ca. 1.6 festgestellt. Der Ansatz, jeweils zwei Ziffern zu einem Byte zusammenzufassen wäre effizienter gewesen.

3.5.4. Tests mit den BitTorrent Daten

Ein nötiger Test ist auch hier wieder das **Laden der Daten**, aber es sollten noch weitere nicht ganz triviale Anfragen dazu kommen, die mehrere SQL- oder Pig-Operatoren hintereinander ausführen.

Es wurden zwei Anfragen gewählt, die so auch in der Publikation, bzw. in deren Fortsetzung von Krenc [Kre12] vorkommen.

BitTorrent Job 1 Es wird berechnet, wie viele Seeder, Leecher und Peers, deren Zustand unbekannt ist, sich zu jedem Timebin in jedem Swarm befinden.

BitTorrent Job 2 Es wird berechnet, wie viele Bitfeldänderungen jeder genutzte Client im Durchschnitt pro Timebin mitteilt.

Um die Möglichkeiten der Datenbank auch sinnvoll einzusetzen, wurde ein Index auf der Announces Tabelle angelegt. Er beinhaltet in dieser Reihenfolge die Felder Timebin, Swarm, Peer und Type und kann zur Optimierung beider Anfragen genutzt werden. Um nicht zuviel Zeit beim Laden der Daten mit der Erstellung des Indexes zu verbringen und da die Daten in Blöcken, die jeweils einem Timebin entsprechen, geladen werden, wurde das Feld Timebin als erstes Schlüsselfeld im Index definiert. Wenn nun ein Ladevorgang beendet ist, muss nicht der gesamte Index neu sortiert werden, sondern nur die neu hinzugefügten Datensätze an den bereits bestehenden Index angehängt werden.

Die Anfragen sowohl in SQL als auch in Pig wurden jeweils in mehreren teilweise sehr unterschiedlichen Variationen entworfen und ausprobiert. Hier dargestellt und besprochen werden nur die effizientesten Versionen.

3.5.5. Laden der Daten

Da der zur Verfügung stehende Platz auf dem Speichersystem nicht ausreichte um die gesamten geladenen und zu ladenden Daten vorzuhalten, wurden die Daten in kleinen Teilen geladen. Es wurden der Reihe nach jeweils alle Schwärme eines Timebins entpackt, umkonvertiert und dann geladen. Also erfolgte ein Ladevorgang für jeden Timebin.

Beim Konvertieren eines Timebins wurden auf einer Ramdisk 4 Dateien erstellt, für jede Tabelle eine. Geladen wurde direkt von der Ramdisk um auch hier den Einfluss der Festplatten zu minimieren. Insgesamt wurden 1469 Timebins eingeladen, um noch etwas temporären Speicher für die Datenbank bzw. Hadoop auf den Festplatten vorhalten zu können. Es ergaben sich die in Tabelle 3.3 zu sehenden Tabellengrößen.

Die Timebins Tabelle ist im Vergleich bei DB2 etwas größer. Das liegt daran, dass bei jedem Ladevorgang jeweils nur eine Zeile geladen wurde und der Ladeprozess bei DB2

Tabellenname	Anzahl der Datensätze	Tabellengröße DB2	Tabellengröße Hadoop
Timebins	1.469	ca. 6 MB	ca. 50 KB
Swarms	15.835	ca. 5 MB	ca. 1 MB
Peers	471.280.448	ca. 59 GB	ca. 25 GB
Announces	4.517.558.336	ca. 1.7 TB	ca. 1.5 TB

Tabelle 3.3.: BT Tabellengrößen

jeden Ladevorgang auf einer neuen Speicherseite beginnt (zur Erinnerung: Timebins liegt auf einem Tablespace mit 4K Seitengröße auf Master).

Im Fall der Datenbank wurden, wie in Listing 3.18 zu sehen, in jedem Ladevorgang weitere Daten an die bestehenden Daten angehängt und die Indizes erweitert. Die Statistik wurde nicht jedes Mal neu erstellt, das erfolgte in diesem Fall erst am Ende des gesamten Ladevorgangs und dauerte nur wenige Sekunden.

```
1 load from import/${tb}_tb.csv of DEL MODIFIED BY TIMESTAMPFORMAT="YYYYMMDDHHMMSS"
      INSERT INTO timebins STATISTICS NO NONRECOVERABLE DISK_PARALLELISM 1 INDEXING
      MODE INCREMENTAL ALLOW NO ACCESS
2
3 load from import/${tb}_swarms.csv of DEL MODIFIED BY ANYORDER FASTPARSE INSERT
      INTO swarms STATISTICS NO NONRECOVERABLE CPU_PARALLELISM 1 INDEXING MODE
      INCREMENTAL ALLOW NO ACCESS
4
5 load from import/${tb}_peers.csv of DEL MODIFIED BY ANYORDER FASTPARSE INSERT INTO
       peers STATISTICS NO NONRECOVERABLE DISK_PARALLELISM 1 INDEXING MODE
      INCREMENTAL ALLOW NO ACCESS
6
7 load from import/${tb}_announces.csv of DEL MODIFIED BY ANYORDER FASTPARSE INSERT
      INTO announces STATISTICS NO NONRECOVERABLE DISK_PARALLELISM 1 INDEXING MODE
      INCREMENTAL ALLOW NO ACCESS
```

Listing 3.18: BitTorrent Laden in SQL

Zum Laden der Daten in das HDFS wurde für jede Tabelle ein eigener Ordner angelegt, in den, wie in Listing 3.19 zu sehen, die einzelnen Dateien jeweils hinein kopiert wurden.

```
1 hadoop fs -put import/${tb}_tb.csv /bt/timebins
2 hadoop fs -put import/${tb}_swarms.csv /bt/swarms
3 hadoop fs -put import/${tb}_peers.csv /bt/peers
4 hadoop fs -put import/${tb}_announces.csv /bt/announces
```

Listing 3.19: BitTorrent Laden in Hadoop

Pig ermöglicht dann, die Ordner wie jeweils eine Tabelle zu behandeln und die einzelnen Dateien direkt als Splits für die Mapper einzusetzen.

3.5.6. BitTorrent Job 1

Es gibt meistens mehrere Wege, in SQL eine Anfrage zu formulieren. Allerdings können trotz der selben Ergebnisse die Laufzeiten zwischen diesen unterschiedlichen Wegen erheblich variieren. Da hier mit großen Datenmengen gearbeitet wird, sollten die Mengen so früh wie möglich reduziert werden. Es empfiehlt sich, zuerst zu filtern, dann zu gruppieren und als letztes die Joins anzuwenden.

Deshalb ist der erste Schritt in Listing 3.20 in den Zeilen 1-4, die Daten auf das notwenige Minimum zu reduzieren. Die Anfrage greift nur auf die Felder des Index auf der Announces Tabelle zu, d. h. es wird für diese Tabelle sogar ein Index-Only Zugriff[3] möglich. Als Nächstes wird in Zeile 5-9 nach Timebin und Swarm gruppiert, während für jede Gruppe die Typen der Peers einzeln aufsummiert werden. Als Letztes folgen die Joins mit der Swarms und Timebins Tabelle.

```
1  with sl_list(tb, swarm, peer, type) as (
2    select distinct timebin,swarm,peer,type
3    from announces
4  ),
5  sl_agg(tb, swarm, seeder, leecher, unknown) as (
6    select tb, swarm, sum(case when type=1 then 1 else 0 end), sum(case when type=2
         then 1 else 0 end), sum(case when type=0 then 1 else 0 end)
7    from sl_list
8    group by tb, swarm
9  )
10 select s.hash as swarm, date(t.ts) as tb_date, time(t.ts) as tb_time, seeder,
       leecher, unknown
11 from sl_agg
12 join swarms s on swarm=s.id
13 join timebins t on tb=t.id
```

Listing 3.20: BitTorrent Job1 in SQL

In Abb. 3.9 ist der Vorgang vereinfacht dargestellt. Der Distict Befehl wird von DB2 durch den Sortiervorgang verarbeitet, danach wird gruppiert und in den letzten beiden Schritten „gejoined“. Bei mehreren unabhängigen Joinvorgängen wird der Größe nach mit der kleinsten Tabelle angefangen um nach jedem Joinvorgang die Ergebnisse immer so klein wie möglich zu halten.

In Pig ist die Formulierung der Anfrage etwas aufwendiger. In Zeile 16 von Listing 3.21 wird die Projektion auf die nötigen Spalten durchgeführt um dann in Zeile 18 das Distinct durchzuführen. Die Spalten werden nochmals projiziert und in Zeile 21 nach Timebin uns Swarm gruppiert. In Zeile 22-28 wird die Aggregation durchgeführt, indem gezählt wird, wie viele Einträge jede, in Zeile 23-25 nach dem jeweiligen Typ gefilterte, Menge enthält. Im letzten Schritt folgen wieder die beiden Joins als „replicated Joins“.

[3] Ein Index-Only Zugriff bedeutet, dass alle benötigten Daten bereits im Index vorhanden sind. Statt einem Tablescan findet ein Indexscan statt und es ist kein Fetch nötig um noch weitere Felder aus der Tabelle nachzuladen. Der Index wird quasi wie eine Tabelle genutzt.

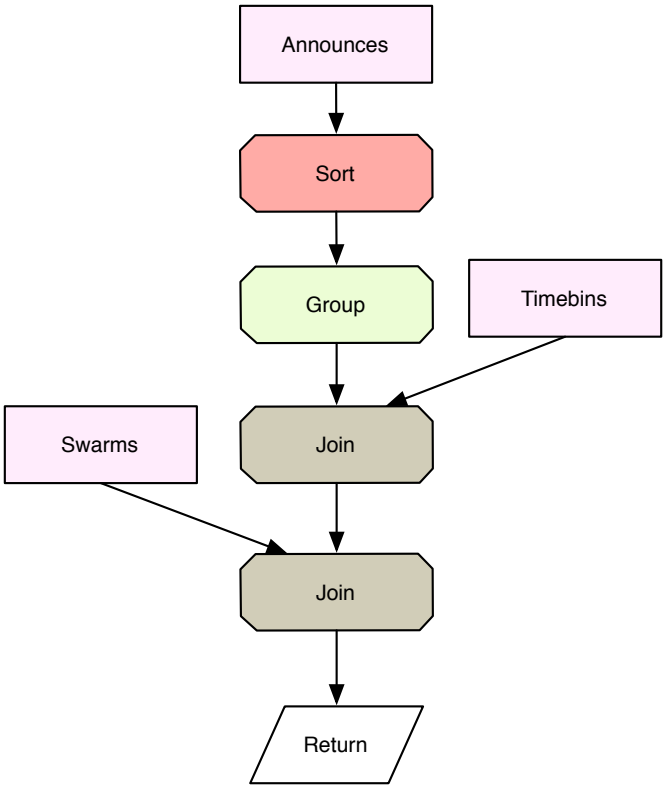

Abbildung 3.9.: Vereinfacher Ausführungsplan BT Job 1 für DB2

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Timebins = LOAD '/bt/timebins'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (id:int, name:chararray, ts:chararray);

Swarms = LOAD '/bt/swarms'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (id:int, hash:chararray);

Announces = LOAD '/bt/announces'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (peer:int, swarm:int, timebin:int, ts:chararray, type:int, chunks:int,
       bitvector:chararray);

A1 = FOREACH Announces GENERATE timebin, swarm, peer, type;

SL_list = distinct A1;
SL_list1 = FOREACH SL_list GENERATE timebin, swarm, type;

SL_group = GROUP SL_list1 by (timebin, swarm);
SL_agg = FOREACH SL_group {
  seeder = FILTER SL_list1 BY type==1;
  leecher = FILTER SL_list1 BY type==2;
  unknown = FILTER SL_list1 BY type==0;
```

```
26
27   GENERATE group, COUNT(seeder) as seeder, COUNT(leecher) as leecher, COUNT(unknown
         ) as unknown;
28 }
29
30
31 SL_TB = JOIN SL_agg BY group.timebin, Timebins BY id USING 'replicated';
32 SL_TB_SW = JOIN SL_TB BY group.swarm, Swarms BY id USING 'replicated';
33
34 Result = FOREACH SL_TB_SW GENERATE hash, ts, seeder, leecher, unknown;
35
36 DUMP Result;
```

Listing 3.21: BitTorrent Job1 in Pig

Für die Pig Ausführung ist in Abb. 3.10 ein vereinfachter Ausführungsplan zu sehen. Pig unterteilt die Anfrage in vier verschiedene Jobs. Während die Jobs 1 und 2 jeweils nur einen Mapper enthalten und für die Projektion der Tabellen Timebins und Swarms auf die benötigten Spalten sorgen, führt Job 3 die Projektion und anschliessend im Reducer das Distinct auf die Announces Tabelle aus. Job 4 sortiert diese Ergebnisse um und führt die Gruppierung und anschließend die Joins im Reducer aus.

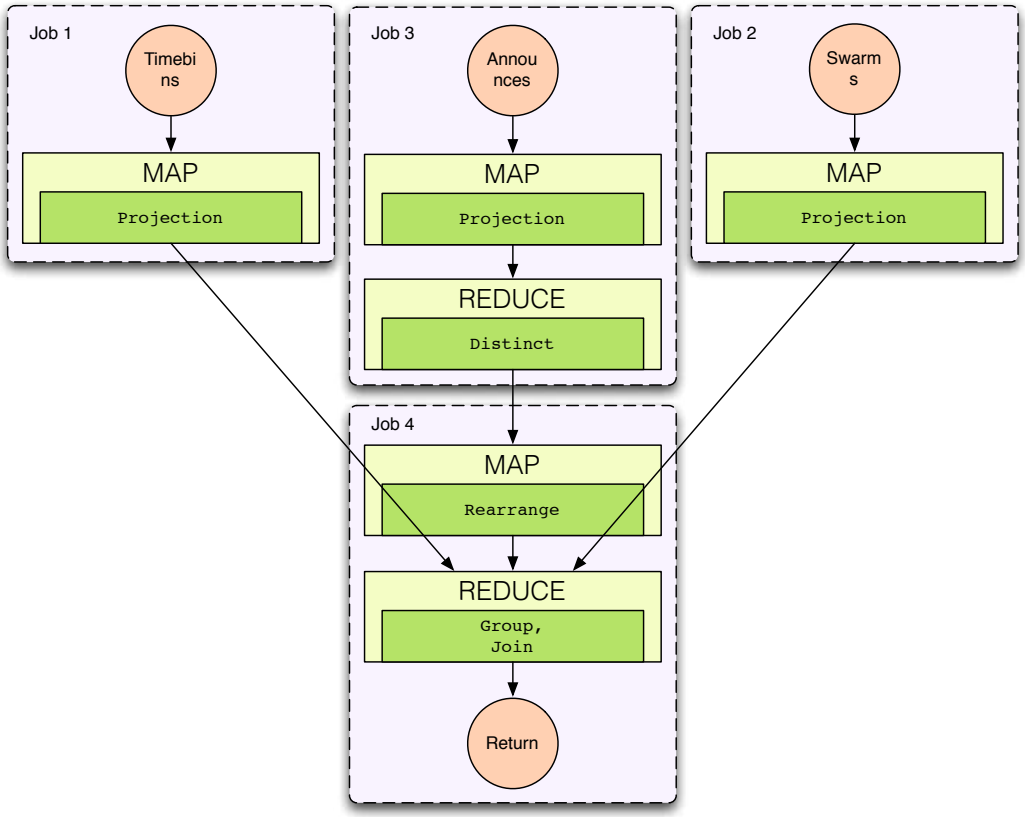

Abbildung 3.10.: Vereinfacher Ausführungsplan BT Job 1 für Hadoop

3.5.7. BitTorrent Job 2

Für BT Job 2 wird ebenfalls nach dem Schema vorgegangen, die Daten so früh wie möglich zu reduzieren und so spät wie möglich zu joinen. Um dieses Verfahren für diese Anfrage möglichst effizient umzusetzen, wird, wie in Listing 3.22 zu sehen, in zwei Phasen aggregiert. In der ersten Phase erfolgt eine Gruppierung und Aggregation nach SWARM und PEER. Auch hier wird ein Index-Only Zugriff möglich, da DB2 den Index auch dann nutzen kann, wenn die erste Schlüsselspalte nicht eingeschränkt wird. Danach erfolgt der Join mit der PEERS Tabelle und es folgt die zweite Phase der Gruppierung. Jetzt wird nach dem Feld CLIENT gruppiert und die bereits in der ersten Phase erstellten Aggregationen aufsummiert. Bei der Ausgabe wird der Quotient gebildet und damit der Durchschnitt berechnet.

```
1 with peer_announces(swarm, peer, announces, appearences) as (
2       select swarm, peer, count_big(*), count_big(distinct timebin)
3       from announces
4       group by swarm, peer
5 )
6 select client, sum(announces) / sum(appearences) as average
7 from peer_announces a
8 join peers p on a.peer=p.id and a.swarm=p.swarm
9 group by client
```

Listing 3.22: BitTorrent Job 2 in SQL

Vereinfacht ist auch hier der Ausführungsplan in Abb. 3.11 zu sehen. Im ersten Schritt werden die Daten in die richtige Reihenfolge für die Gruppierung und das Distinct sortiert, darauf folgt die Gruppierung. Als Nächstes wird mit PEERS gejoined und für das erneute Gruppieren umsortiert.

Als Letztes ist in Listing 3.23 noch einmal Job 2 als Pig Script dargestellt. Im ersten Schritt wird für beide Tabellen eine Projektion durchgeführt. Die ANNOUNCES Tabelle wird nun in Zeile 15 nach den Spalten SWARM und PEER gruppiert. In Zeile 16-21 wird wie im SQL Script die erste Phase der Aggregation vorgenommen und in Zeile 25 „gejoined“. In Zeile 26 kommt die zweite Gruppierung mit der endgültigen Aggregation in Zeile 27.

```
1  SET default_parallel 32
2  REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;
3
4  Announces = LOAD '/bt/announces'
5     USING org.apache.pig.piggybank.storage.CSVLoader()
6     AS (peer:int, swarm:int, timebin:int, ts:chararray, type:int, chunks:int,
          bitvector:chararray);
7
8  Peers = LOAD '/bt/peers'
9     USING org.apache.pig.piggybank.storage.CSVLoader()
10    AS (id:int, swarm:int, ip:chararray, port:int, client:chararray, hash:chararray
          );
11
12 Announces1 = FOREACH Announces GENERATE peer, swarm, timebin;
13 Peers1 = FOREACH Peers GENERATE client, id, swarm;
```

```
14
15 Agrp = GROUP Announces1 BY (swarm,peer);
16 PA = FOREACH Agrp {
17   tb_only = Announces1.timebin;
18   tb_dist = DISTINCT tb_only;
19
20   GENERATE flatten(group), COUNT(tb_only) as announces, COUNT(tb_dist) as
       appearences;
21 }
22
23 Joined = JOIN PA BY (swarm, peer), Peers1 BY (swarm,id);
24 Joined1 = FOREACH Joined GENERATE Peers1::client as client, PA::announces as
     announces, PA::appearences as appearences;
25
26 Grp = GROUP Joined1 BY client;
27 Result = FOREACH Grp GENERATE group, SUM(Joined1.announces) / (double)SUM(Joined1.
     appearences);
28
29 DUMP Result;
```

Listing 3.23: BitTorrent Job 2 in Pig

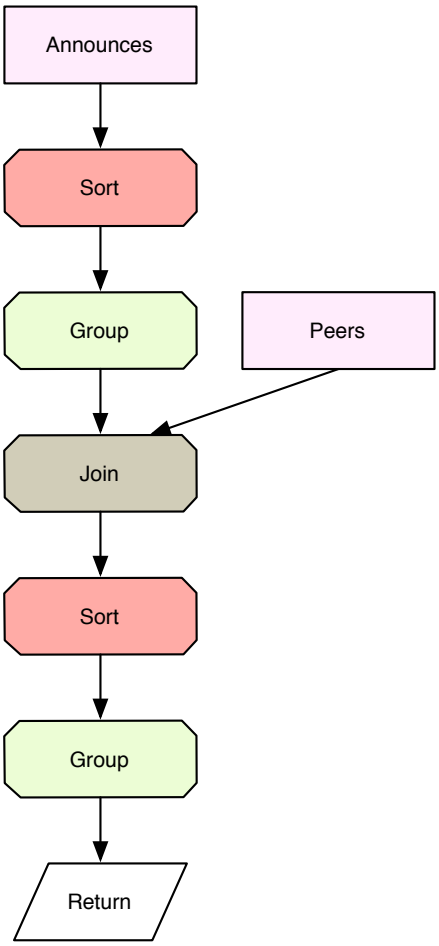

Abbildung 3.11.: Vereinfacher Ausführungsplan BT Job 2 für DB2

In Abb. 3.12 ist ein vereinfachter Ausführungsplan von Pig zu sehen. In MapReduce Job 1 ist die Projektion und Gruppierung der ANNOUNCES Tabelle dargestellt, in MR Job 2 die Projektion der PEERS Tabelle sowie ein UNION mit den gruppierten Ergebnissen aus Job 1. Diese Union wird dann im Reducer von Job 2 „gejoined“. Es schlisst sich Job 3 an, in dem die Zwischenergebnisse aus Job 2 im Mapper umsortiert und anschließend im Reducer gruppiert werden.

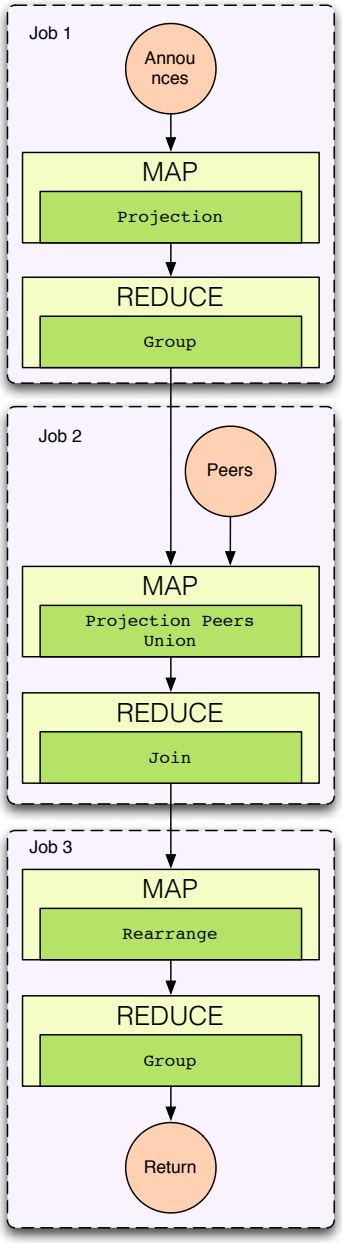

Abbildung 3.12.: Vereinfacher Ausführungsplan BT Job 2 für Hadoop

4. Ergebnisse

Zu Beginn dieses Kapitels werden die Ergebnisse der synthetischen Tests dargestellt. Mit Hilfe dieser Ergebnisse werden im nächsten Schritt Vorhersagen zur Ausführung der Tests mit den echten Daten getroffen. Darauf folgend werden die Ergebnisse der Tests mit den echten Daten gezeigt und am Schluss die Vorhersagen verifiziert.

4.1. Lesen der erzeugten Ausgaben

Im Rahmen dieser Arbeit wurden viele verschiedene Messwerte aufgenommen und verarbeitet. Um diese Werte interpretieren zu können, wurden sie in verschiedenen Graphen visualisiert. Für die am häufigsten vorkommenden Graphen wird im Folgenden eine Einführung gegeben, wie die Daten dargestellt wurden und wie sie gelesen werden.

4.1.1. Ausführungspläne von DB2

Wenn DB2 eine SQL Anfrage verarbeiten muss, wird sie von einem datenbankinternen Optimierer mit Hilfe von Informationen über die Verteilung der Daten zwischen den Datenbankpartitionen, Indizes, Tabellenstatistiken und Hardwarezugriffszeiten im Hinblick auf möglichst geringen Aufwand und damit auch kurze Ausführungszeiten umgeschrieben. Im Zuge dieser Berechnung wird von der Datenbank ein Baum aus Operationen erstellt, die miteinander verkettet am Ende das gewünschte Ergebnis liefern.

Die Darstellung dieses Baumes ergibt einen Überblick darüber, was die Datenbank wann macht, auch wenn man man aufgrund der hohen Parallelisierung der Operationen nicht genau sagen kann, welche Operation in einem bestimmten Moment ausgeführt werden.

Zum besseren Verständnis folgt ein Beispiel in dem die wichtigsten Operationen dargestellt werden. Dieses Beispiel basiert ebenfalls auf den Tabellen zu den synthetischen Tests, allerdings wurde zusätzlich der Index POS_HOUR über die Spalte HOUR auf der Tabelle POSITION angelegt. Die SQL Anfrage zu diesem Beispiel ist in Listing 4.1 zu sehen, der dazu gehörige kommentierte Ausführungsplan ist in Abb. 4.1 dargestellt. Tabelle 4.1 gibt einen Überblick über die verwendeten Symbole im SQL Ausführungsplan.

```
select count(*)
from position,metadata
where position.unit=metadata.unit
  and hour=1
```

Listing 4.1: Beispiel in SQL

Symbol	Farbe	Bedeutung
Quadrat	Pflaume	Tabelle
Raute	Hellblau	Tablequeue
Sechseck	Gelb	Index
Achteck	Pflaume	Operator: Tablescan
Achteck	Grün	Operator: Gruppierung
Achteck	Rot	Operator: Sortierung
Achteck	Gelb	Operator: Operation auf Indexdaten
Achteck	Weiss	Operator: Rückgabe der Ergebnismenge

Tabelle 4.1.: Symbolbeschreibung DB2 Ausführungsplan

Gelesen wird der Ablaufplan im wesentlichen von oben nach unten. Am Anfang stehen die Quelldaten, wie Tabellen und Indizes, am Ende steht das Ergebnis der Anfrage. Begonnen werden die Operationen in der Reihenfolge der Nummer hinter der Operatorbezeichnung, es können mehrere Operationen parallel stattfinden. Die ersten Teilergebnisse einer laufenden Operation werden sofort als Operand der nächsten Operation zugeführt. Eine in dieser Parallelisierung bremsende Operation ist die Sortierung, denn hier stehen die Ergebnisse erst fest, wenn alle eingegangenen Daten komplett sortiert wurden. Wichtig sind auch noch die Tablequeues, hier werden Daten zwischen den Partitionen übertragen. Es gibt bei DB2 vier verschiedene Tablequeues:

Broadcast TQ: Jeder Datensatz wird an alle anderen Partitionen gesendet.

Directed TQ: Jeder Datensatz wird gezielt an bestimmte Partitionen gesendet, z. B. basierend auf der Verteilung der Daten zwischen den Partitionen

Merge Broadcast TQ: Jeder Datensatz wird wieder an alle Partitionen gesendet, allerdings sind die Daten bereits sortiert und können auf der Zielpartition in sortierter Reihenfolge zusammengesetzt werden.

Merge Directed TQ: Er erfolgt eine gezielte Versendung der Daten, aber auch hier sind die Daten sortiert und können auch wieder sortiert auf der Zielpartition weiter verarbeitet werden.

Im Fall des Beispiels wird also als erstes mit einem Tablescan TBSCAN die Tabelle METADATA eingelesen, komplett auf alle Partitionen repliziert und füllt auf jeder Partition bereits Hashtabelle für den künftigen Hashjoin. Parallel dazu wird mit einem Indexscan IXSCAN

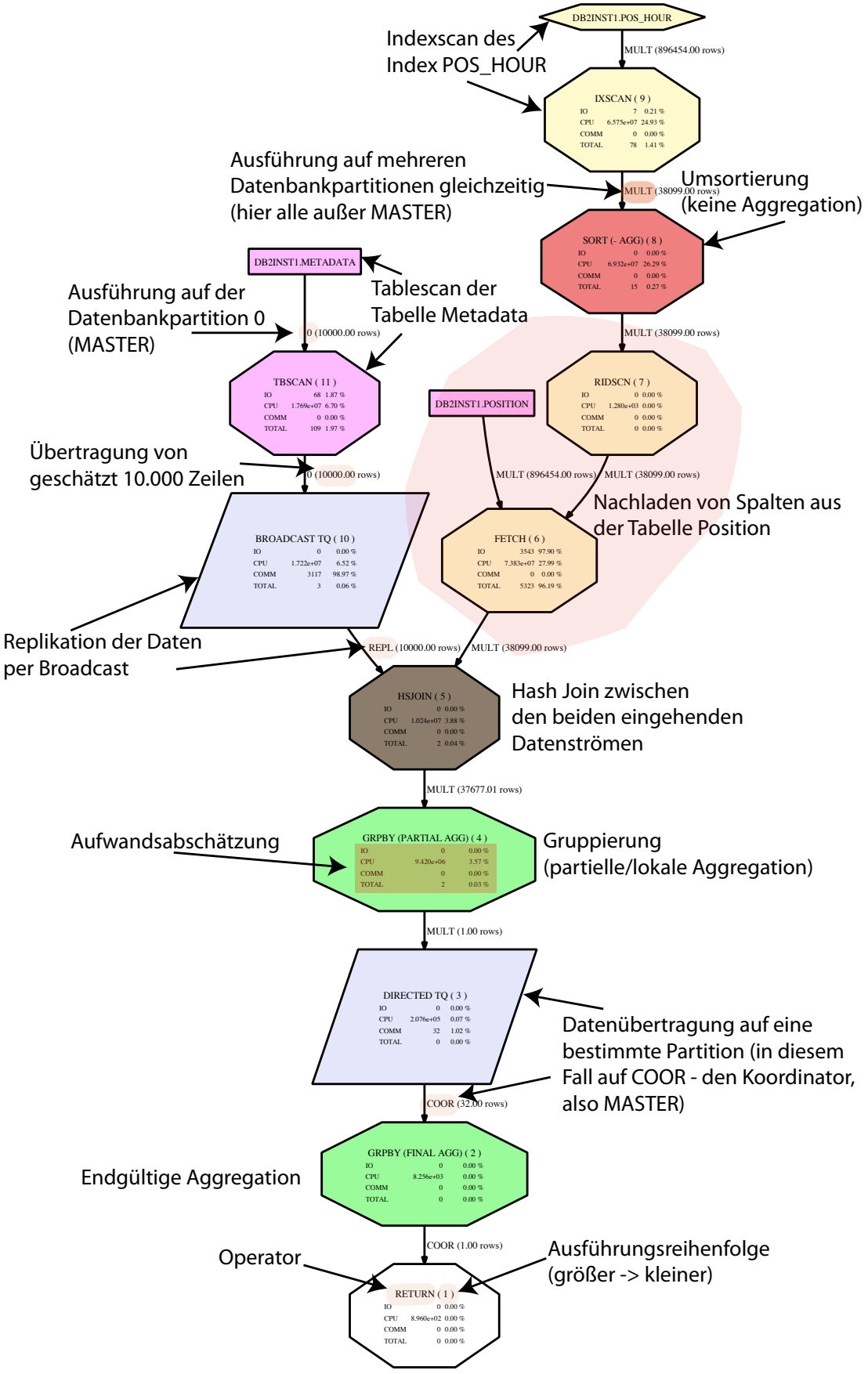

Abbildung 4.1.: Beispiel Ausführungsplan DB2

im Index POS_HOUR nach Zeilen gesucht, bei denen HOUR=1 ist. Die Ergebnisse aus dem Indexscan werden noch ihrer ROWID, die die Zeilen in der Tabelle POSITION adressiert, sortiert. Mittels eines FETCH Operators werden nach der Sortierung Spalten aus der Tabelle POSITION nachgeladen. Die benötigten Zeilen werden über die ROWID im RIDSCN bestimmt. In diesem Fall muss die Spalte UNIT nachgeladen werden, die für den Join benötigt wird. Die Ergebnisse werden im Hashjoin HSJOIN mit Hilfe der bereits angelegten Hashtabelle um die Daten aus der Tabelle METADATA erweitert und der Gruppierung GRPBY zugeführt. Hier werden auf jeder Partition lokal die Zeilen für das COUNT(*) gezählt und über eine weitere Tablequeue DIRECTED TQ an die koordinierende Partition (in diesem Fall der MASTER) weitergeleitet. Die Teilsummen der einzelnen Partitionen werden jetzt nochmals im GRPBY auf dem Koordinator aufsummiert und im letzten Schritt per RETURN an den anfragenden Clienten gesendet.

An den Beschriftungen links der Pfeile zwischen den Operatoren sieht man, an welche Partitionen die Ergebnisse der Operation gesendet werden und rechts, um wie viele Zeilen es sich, basierend auf Tabellenstatistiken, geschätzt handelt. In der kleinen Tabelle im Inneren der Operatoren ist der geschätzte Aufwand für den entsprechenden Operator vermerkt. DB2 rechnet dafür intern mit einer „Timeron“ genannten Maßeinheit. Es werden die folgenden Aufwände berechnet:

IO: Der Aufwand beim Zugriff auf die Datenspeicher.

CPU: Der Aufwand für die CPU durch Berechnungen o.ä..

COMM: Der Aufwand für die Kommunikation mit anderen Partitionen.

TOTAL: Eine Abschätzung für den Gesamtaufwand der Operation.

Jeder Operator enthält die absoluten Zahlen für seinen Aufwand sowie Prozentzahlen für seinen relativen Aufwand im Verhältnis zur gesamten Anfrage. So ist z. B. zu sehen, dass ca. 97.90% des gesamten IO-Aufwands für das Nachladen der UNIT Spalte der POSITION Tabelle benötigt wird, während das Verteilen der METADATA Tabelle ca. 98.97% des gesamten Kommunikationsaufwands verursacht.

Diese Graphen und Werte helfen nun in den weiteren Abschnitten, die Messungen an der Datenbank besser zu verstehen.

4.1.2. Ausführungs- und Taskpläne von Pig und Hadoop

Auch Pig nimmt vor der Ausführung eines Pig Latin Programms einige Optimierungen vor, so werden z. B. Operationen, die die Datenmenge reduzieren, so weit nach oben im Programm geschoben wie möglich. Da Pig keine Kenntnis über die Daten hat, muss der Ent-

wickler meist selbst entscheiden, welche Joins genutzt werden, bzw. in welcher Reihenfolge bestimmte Operationen durchgeführt werden müssen.

Es wird, wie in Listing 4.2 zu sehen dasselbe Beispiel auch in Pig gewählt.

```
SET default_parallel 32
REGISTER /usr/lib/pig/contrib/piggybank/java/piggybank.jar;

Position = LOAD '/bench/position.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, hour:int, minute:int, lat:double, lon:double);

Metadata = LOAD '/bench/metadata.csv'
   USING org.apache.pig.piggybank.storage.CSVLoader()
   AS (unit:int, isp:chararray);


PosP = FOREACH Position GENERATE unit, hour;
MetaP = FOREACH Metadata GENERATE unit;

PosF = FILTER PosP BY hour==1;

Joined = JOIN PosF BY (unit), MetaP BY (unit);
PJoined = FOREACH Joined GENERATE PosF::unit;
Grp = GROUP PJoined ALL;
Result = FOREACH Grp GENERATE COUNT(*);

DUMP Result;
```

Listing 4.2: Beispiel in Pig

Der Verständlichkeit halber ist das Beispiel in Pig nach dem selben Schema aufgebaut wie das SQL-Programm, auch wenn sich das Beispiel in Pig Latin auf andere Weise effizienter lösen ließe. Der von Pig erzeugte Ausführungsplan ist in Abb. 4.2 zu sehen.

Die Gruppierungen der Operationen zu einzelnen MapReduce-Jobs, die später von Hadoop nacheinander ausgeführt werden, sind deutlich zu erkennen. Jeder Job enthält einen Mapper, je nach Bedarf manchmal einen Combiner und häufig einen Reducer. Jeder Operator innerhalb dieser Mapper, Combiner und Reducer ist nach seinem Namen im Pig Programm benannt, was seine Identifizierung und das Lesen des Graphen sehr vereinfacht. Zur Optimierung hat Pig das Lesen beider Tabellen in Job 1 in nur einen Mapper zusammengefasst und übergibt die Ergebnisse dem Reducer zum Joinen als Tupel. Die Gruppierung erfolgt wie im SQL Beispiel auch in Pig in zwei Phasen. Den Mappern in Job 2 folgt ein Combiner, der bereits eine partielle Aggregation, also in diesem Fall das COUNT, auf den lokalen Daten bewerkstelligt. Der folgende Reducer nimmt dann die endgültige Aggregation aller Daten vor.

Eine Hilfe um zu verstehen was in Hadoop passiert, sind die Hadoop Ausführungspläne für die MapReduce-Jobs. In Abb. D.23 im Anhang auf Seite xxviii ist ein aufgenommener

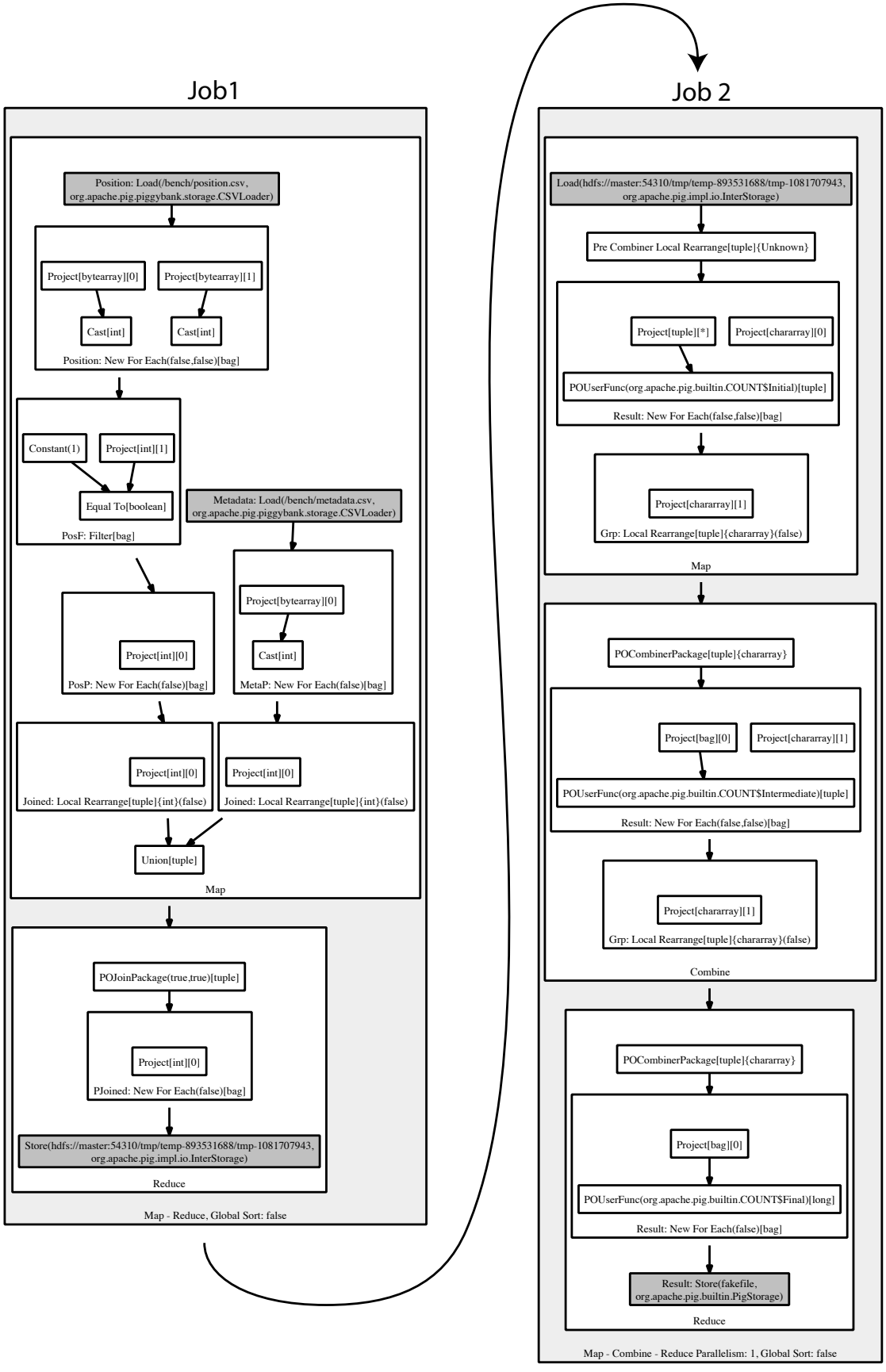

Abbildung 4.2.: Beispiel Ausführungsplan Pig

Hadoop Ausführungsplan zu sehen, der in etwa dem entspricht, der beim Ausführen dieses Beispiels entstehen würde. Allerdings ist im dargestellten Plan die Laufzeit der Tasks deutlich länger.

Senkrecht sind die jeweils acht parallelen Prozesse für Mapper und Reducer auf Node1 - Node4 zu sehen, die horizontal über der relativ zum Beginn abgetragenen Zeit während der Ausführung einer Anfrage dargestellt werden. Mapper sind rot dargestellt, Reducer in blau. Die Shuffle-Phasen, in denen die Reducer die von den Mappern erstellen Datensätze kopieren, sind in gelb dargestellt. Parallel zur Shuffle-Phase sieht man die in grün dargestellten Sortier- und Mergeoperationen der von den Reducern kopierten Daten. Die Combiner als Teil der Map-Phase sind nicht direkt erkennbar. Die Mapper sind segmentiert dargestellt, wodurch die einzelnen „Splits“, in denen Hadoop die Daten an die Mapper liefert, erkennbar werden.

Damit Hadoop schnell auf Ausfälle von Hardware reagieren kann, werden, auch wenn es für die Berechnung nicht notwendig wäre, möglichst immer alle Rechner mit teilweise redundanten Jobs ausgelastet. Das bedeutet, dass wenn ein System bereits alle für dieses System eingeplanten Maps oder Reduces ausgeführt hat, während der MapReduce Job noch nicht beendet wurde, auf diesem System weitere Mapper oder Reducer gestartet werden, die zwar für ein anderes System eingeplant waren, bisher aber nicht fertiggestellt wurden. Für den Fall, dass das andere System ausfällt, hat man hierdurch Ersatz. In dem Moment, in dem ein System einen Mapper oder Reducer beendet, werden alle, aus Redundanzgründen auf weiteren Systemen gestartete, Mapper oder Reducer beendet. Diese weiteren, redundanten Ausführungen von Mappern und Reducern werden als „higher attempt“ für Maps dunkelrot und für Reduces dunkelblau dargestellt.

Das Wissen, welche Operationen in welcher Reihenfolge und in welchem MR Job ausgeführt wird, hilft, die Reports der Tests mit Hadoop zu verstehen.

4.1.3. Testreports

Die aus den Messwerten erstellten Reports stellen den essentiellen Teil dieser Arbeit dar und sind der Einfachheit halber alle gleich aufgebaut. Ein Beispiel für einen solchen Report ist in Abb. 4.3 auf Seite 55 zu sehen. Hier handelt es sich um den Aggregations-Test mit DB2. Die Reports sind vertikal in sechs Graphen unterteilt. Alle Graphen sind zeitsynchron abgebildet und stellen unterschiedliche Themenbereiche der gemessenen Parameter dar, während jeder Graph für sich mehrere inhaltlich zueinander passende Parameter gruppiert. Alle Reports bilden nur die Daten von Node1 - Node4 ab. Master ist nicht mit eingerechnet, da sowohl auf der Datenbank als auch bei Hadoop keine Berechnungen auf Master ausgeführt werden, er wird nur für das Verteilen der Anfragen genutzt. Es sind von oben nach unten folgende Graphen dargestellt:

Avg CPU/MEM und Avg 1min Mov Avg Load

An der linken Achse in Prozent abgetragen als gestapelter Graph aus Flächen ist die Verteilung der CPU Ressourcen erkennbar. Es wird unterteilt zwischen der Zeit, die von den Prozessen durch Systemaufrufe im Kernel verbracht wird (CPU SYS), der Zeit die die CPU mit der Abarbeitung von „normalem“ Programmcode verbringt (CPU USR) und der Zeit, die die CPU auf Ressourcen wie z. B. die Festplatte oder das Netzwerk wartet (CPU IOWAIT). Weiterhin auf der linken Achse abgetragen ist die blaue Linie zu sehen, die als MEM USED darstellt, welcher Anteil des Arbeitsspeichers aktuell genutzt wird. Speicher, der vom Filesystem als Cache verwendet wird, ist hier nicht mit eingerechnet. Alle diese Parameter werden über NODE1 - NODE4 gemittelt dargestellt.

An der rechten Achse abgetragen ist der Durchschnitt von NODE1 - NODE4 des 1MIN MOV AVG LOAD bzw. „System Load“, also der einminütige gleitende Durchschnitt der Länge der Warteschlange der abzuarbeitenden Prozesse. Da alle Systeme 8 Kerne haben, ist bei der Interpretation dieses Wertes zu bedenken, dass auch 8 Prozesse gleichzeitig ausgeführt werden können.

Sum Disk IO

Der Graph „Sum Disk IO“ gibt Informationen über die Plattenzugriffe. Auf der linken Achse abgetragen ist der flächige Graph, der in grün die Bandbreite der lesenden Zugriffe in MB/s und in rot die Bandbreite der schreibenden Zugriffe in MB/s zeigt. Zusätzlich, ebenfalls auf der linken Achse, in grüner, bzw. roter Linie, abgetragen, sind die Bandbreiten in MB/s für das „Pagen“ des virtuellen Speichers, bzw. Zugriffe auf Dateien über ein Memorymapping-Verfahren.

Auf der rechten Achse ist mit dunkelblauer Linie die Gesamtmenge der gelesenen, bzw. mit dunkelroter Linie die Gesamtmenge der geschriebenen Daten seit Beginn der Aufzeichnung in GB abgebildet.

Bandwidth und Packets

Im Bandbreitengraphen sind auf der linken Achse mit Linien die Bandbreiten der Übertragungen in MB/s angegeben. Es werden jeweils die Summen über alle Systeme dargestellt. BW <> CLIENTS steht dabei für die Bandbreite der Übertragungen zwischen NODE1 - NODE4, BW > MASTER für die Bandbreite der Übertragungen von NODE1 - NODE4 an MASTER, BW < MASTER für die Bandbreite der Übertragungen von MASTER an NODE1 - NODE4 und BW LOCAL für die Bandbreite der Übertragungen jeweils lokal über das „Loopback Device“ auf NODE1 - NODE4.

Auf der rechten Achse sind die gesendeten Pakete pro Sekunde in einem gestapelten Graph abgetragen, es stehen hier PKTS <> CLIENTS für die Summe der Pakete zwischen NODE1 - NODE4, PKTS > MASTER für die Summe der Pakete von NODE1 - NODE4 an MASTER,

Pkts < Master für die Summe der Pakete von Master an Node1 - Node4 und Pkts local für die Summe der Pakete über das „Loopback Device“ auf Node1 - Node4.

Transfered Data

Dieser Graph zeigt, auf der rechten Achse abgetragen, die Summe aller seit Beginn der Aufzeichnung übertragenen Daten über das Netzwerk in MB. In der flächigen Kurve in beige ist die Gesamtmenge der übertragenen Daten zu sehen, wogegen die ebenfalls auf der rechten Achse in MB abgetragene dunkelblaue Linie die Datenmenge ohne Paketwiederholungen durch Übertragungsfehler darstellt.

TCP Connections

Hier ist auf der linken Achse abgetragen, wie viele TCP Verbindungen zwischen Master und Node1 - Node4 in jeder Sekunde geöffnet wurden (grüne Linie), geschlossen wurden (rote Linie), bzw. zu dem jeweiligen Zeitpunkt offen waren (blaue Linie). Auf der rechten Achse ist die Summe der geöffneten Verbindungen abgetragen. Da die Datenbank keine geöffneten Verbindungen schließt, sind die Graphen für die aktuell geöffneten Verbindungen und die Summe aller Verbindungen deckungsgleich.

Packetloss

Der letzte Graph zeigt die Paketverluste an. Grüne Impulse stellen die Anzahl der „triple duplicate ACKs“, also Anforderungen zum erneuten Senden von Paketen vom Empfänger in der jeweiligen Sekunde dar, rote Impulse die Anzahl der Retransmits, also Timeouts beim Sender, die zum erneuten Senden von Paketen führen.

4.2. Synthetische Tests

Im Folgenden werden die Ergebnisse der synthetischen Tests dargestellt. Bei Datenbankanfragen, bei denen die gewählten Indizes deutliche Unterschiede verursachen, wurden die Messungen sowohl mit als auch ohne Indizes durchgeführt. Der Einfachheit halber wird in den folgenden Abschnitten immer mit DIR auf den DB2 Report mit Index verwiesen, mit DNR auf den DB2 Report ohne Index und mit HDR auf den Hadoop Report. APD bezeichnet den Ausführungsplan von DB2, APP den Ausführungsplan von Pig und MRT das MapReduce Taskscheduling. Diese Verweise beziehen sich immer auf die zugehörigen Reports dieses Abschnittes. Wenn ein Test sowohl mit als auch ohne Index ausgeführt wurde, zeigt der APD die Version mit Index. Die Abbildungen der APD, APP und MRT befinden sich nur im Anhang dieser Arbeit, da die Beschreibung im Text für das Verständnis durch den Lesers ausreichend ist.

Das DBMS ist so konfiguriert, dass die Datenbank bei der ersten Verbindung eines Clients gestartet wird und heruntergefahren wird, sobald kein Client mehr eine Verbindung zur Datenbank hält. Das hat zwar den Nachteil, dass bei kurzen Anfragen im Verhältnis zur Dauer der eigentlichen Anfrage relativ viel Zeit zum Starten und Stoppen der Datenbank benötigt wird, allerdings hat es in allen anderen Tests den Vorteil, dass zu Beginn der Tests die Bufferpools leer sind und auf diese Weise immer alle Daten komplett geladen werden müssen.

4.2.1. Aggregation

Da sich die Systeme beim Laden etwas anders verhalten als bei Anfragen, wird der Einfachheit halber mit der Aggregation aus Kapitel 3.4.3 begonnen. Die Abbildungen sind an folgenden Stellen zu finden:

APD	Abb. D.1 auf Seite ix
DNR:	Abb. 4.3 auf Seite 55
DIR:	Da die erstellten Indizes für diese Anfrage nicht benutzt werden, würde der DIR genauso aussehen wie der DNR.
APP	Abb. D.8 auf Seite xvi
HDR:	Abb. 4.4 auf Seite 56
MRT	Abb. D.17 auf Seite xxv

Im APD sieht man, dass auf jeder der 32 Partitionen am Anfang per Tablescan auf der Latency Tabelle ca. 44 Millionen Datensätze eingelesen werden, wofür ca. 100% der IO-Zeit dieser Anfrage verbraucht wird. Diese Datensätze werden jeweils lokal aggregiert und als eine Zeile per Tablequeue an den Master geschickt. Dieser bildet den Gesamtdurchschnitt und gibt das Ergebnis zurück. Das Einlesen aller der insgesamt über 1.4 Mrd. Datensätze ist hierfür notwendig, also genau das, was bezweckt werden sollte. Da hierfür 99.94% der gesamten Abarbeitungszeit der Anfrage nötig sind, bekommt man ein recht genaues Bild davon, was bei einem großen Tablescan vorgeht.

Da die Datenbank wie im APD sehen auf keinen Index zugreift, ist der DNR in diesem Fall gleichbedeutend mit dem DIR. Er zeigt im Sum Disk IO Graphen, dass die Systeme mit in Summe bis zu ca. 900 MB/s lesen, während sich die CPU, wie im Avg CPU/MEM Graphen zu sehen, auffällig viel im IOWAIT Zustand befindet. Diese 900 MB/s scheinen der Grenzwert dessen zu sein, was mit der Hardware möglich ist. Der Peak der SYS Zeit der CPU am Anfang der Kurve ist auf das Starten der Datenbank zurückzuführen. Um „Verunreinigungen“ der Bufferpools der Datenbank zu verhindern, also den Fall, dass sich schon Daten im Cache der DB befinden und nicht mehr von der Festplatte gelesen werden müssen, wurde die Datenbank (nicht das DBMS) am Anfang jeder Anfrage neu gestartet. In diesem Graphen ist ebenfalls zu sehen, dass die DB bei dieser Anfrage nur ca. 70% des Speichers nutzt. Hierbei handelt es sich um den Speicher, der für die Bufferpools reserviert ist, der

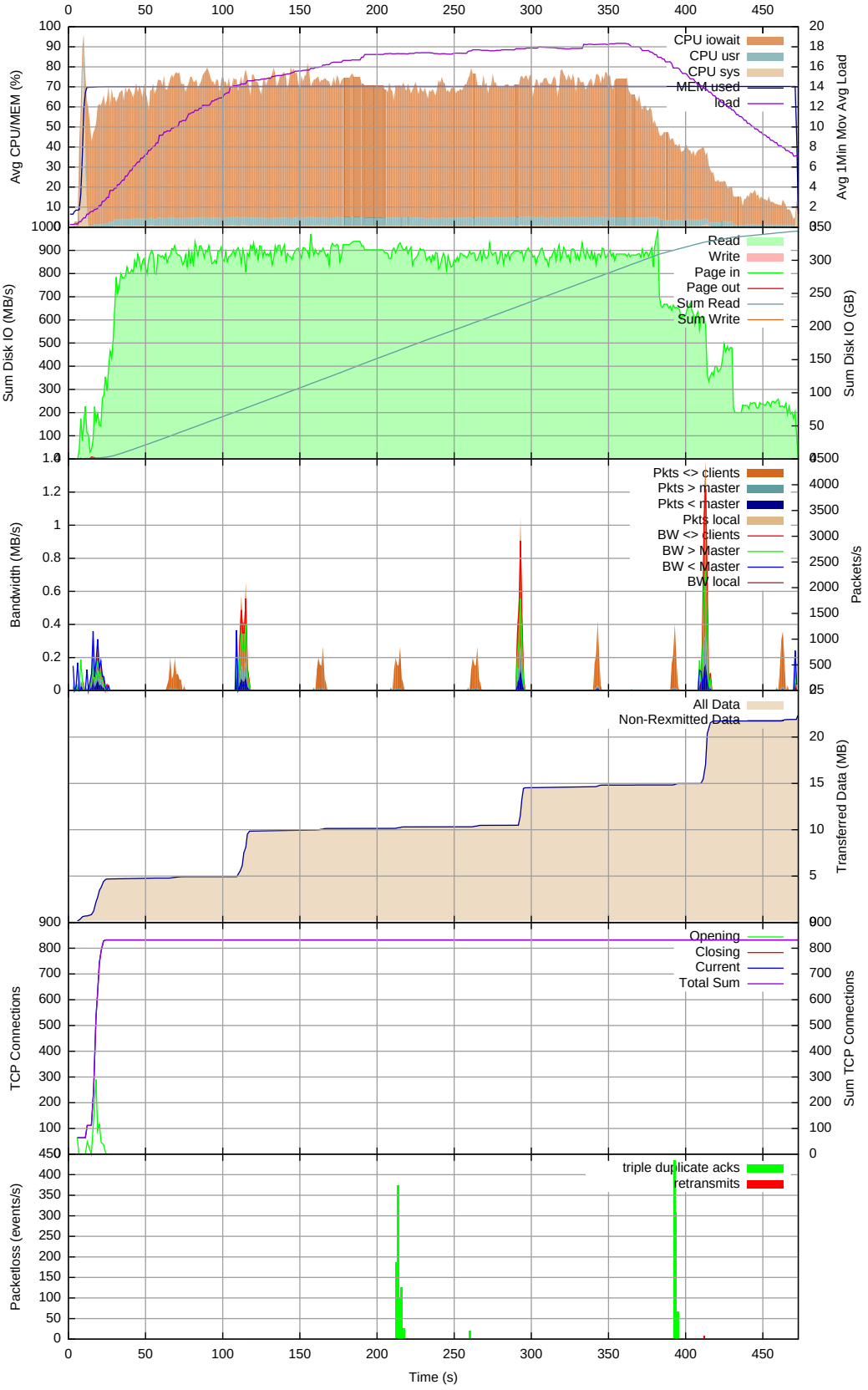

Abbildung 4.3.: DB2 Aggregate Report

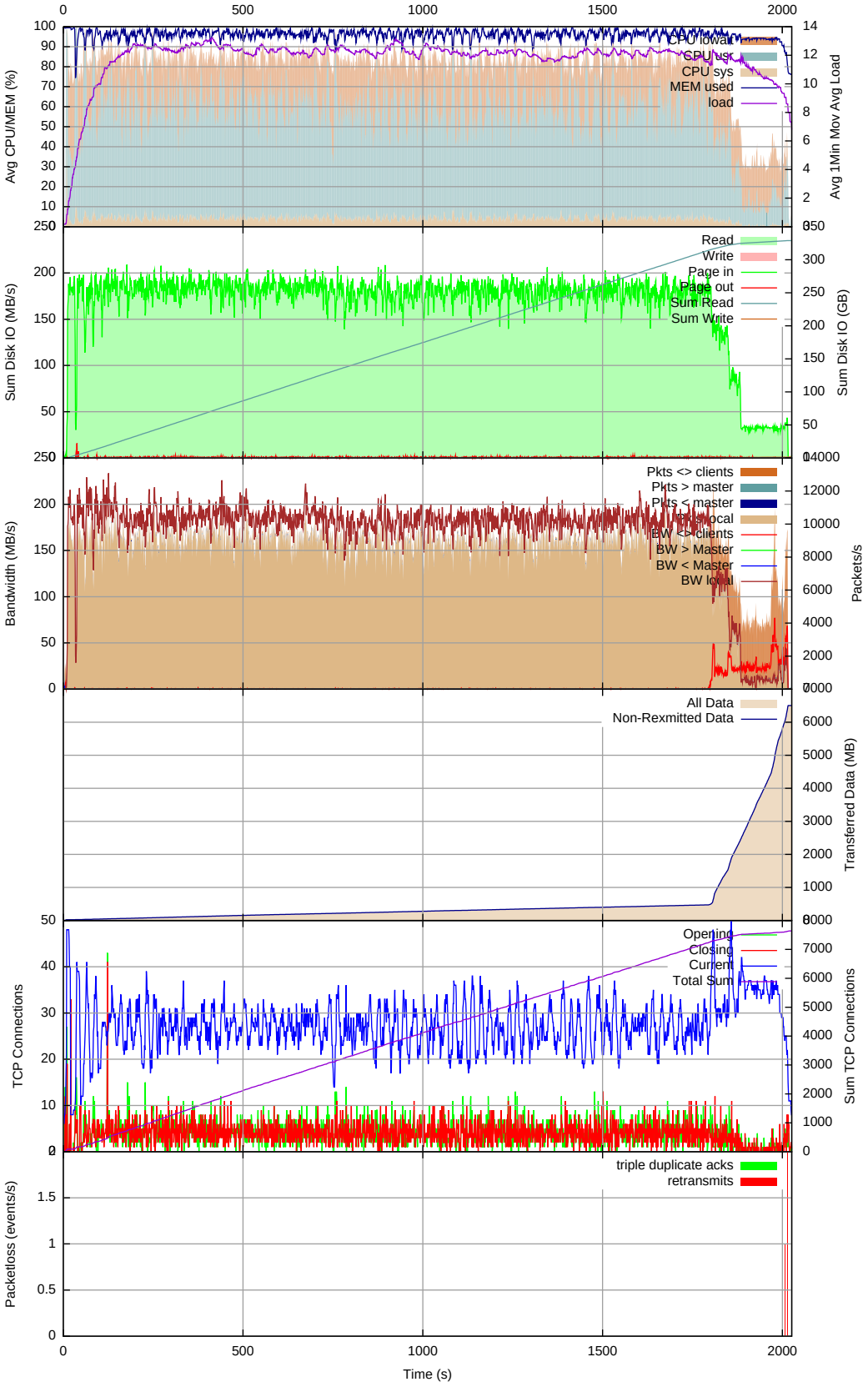

Abbildung 4.4.: Hadoop Aggregate Report

restliche Speicher wird für Sortiervorgänge etc. freigehalten. Die Kurve der IO Bandbreite flacht stufenweise ab, da die einzelnen Partitionen unterschiedlich schnell zu ihrem Teilergebnis kommen. Insgesamt werden, wie in Sum Disk IO zu sehen, ca. 350 GB an Daten eingelesen, es müssen keine temporären Daten geschrieben werden. Aufgrund der sich bei der Berechnung ergebenden recht geringen Datenmenge sind keine größeren Datenübertragungen zu erwarten, und auch der Transfered Data Graph zeigt, dass insgesamt nur ca. 22 MB übertragen wurden. Man kann am Bandwidth und Packets Graphen erkennen, dass der GRPBY Operator bei einer partiellen Aggregation nicht komplett aggregiert, sondern in Blöcken. Sobald eine gewisse Grenze erreicht ist, werden die Zwischenergebnisse an die Tablequeue und damit zum Master übertragen. Auffällig ist im TCP Connection Graphen, dass DB2 gleich am Anfang 832 Verbindungen öffnet und auch bis zum Ende offen hält. Diese Verbindungen zwischen den Partitionen werden zum Austausch der Daten zwischen den Tablequeues genutzt. Jeder Knoten baut zu jedem anderen Knoten jeweils eine Verbindung auf, über die die Daten aller Tablequeues gepushed werden. Es werden 832 TCP Verbindungen aufgebaut, da von den 32 Partitionen auf Node1-Node4 auf jedem Rechner 8 ausgeführt werden, zwischen denen die Kommunikation über IPC läuft. Das heißt, es werden von jedem dieser 32 Partitionen 24 TCP Verbindungen zu anderen Partitionen geöffnet und eine zum Master sowie 32 Verbindungen vom Master zu den anderen Partitionen. Der Packetloss Graph zeigt, dass nur wenig Übertragungsfehler in der Kommunikation aufgetreten sind. Insgesamt benötigt die Datenbank für diese Anfrage ca. 470 Sekunden.

Es folgt dieselbe Anfrage in Pig. Der APP zeigt, dass Pig das Problem mit nur einem MapReduce (MR) Job löst, er enthält einen Mapper, einen Combiner und einen Reducer. Im Mapper wird für jede Zeile ein Tupel aus einem Schlüssel, der für alle Zeilen gleich ist, und der RTT erstellt. Der hin und wieder lokal im Anschluss an einen Mapper ausgeführte Combiner kann im nächsten Schritt nach dem Schlüssel gruppieren und mittels der Summe und Anzahl der RTT Felder ein erstes Teilergebnis der Mittelwertbildung berechnen. Der Combiner stellt einen ”Mini-Reducer” dar, der nur lokal ausgeführt wird und auch nur, wenn die Auslastung es zulässt. Der Reducer kann nun im letzten Schritt aus den Zwischenergebnissen aller Mapper bzw. Combiner den Mittelwert über alle Datensätze berechnen.

Der MRT zeigt, dass verhältnismäßig viele Mapper gestartet werden, allerdings nur ein Reducer. Der Grund dafür ist, dass alle Daten nach nur einem Schlüssel gruppiert wurden, da der Mittelwert über alle Datensätze berechnet wurde. In solch einem Fall würde es keinen Vorteil bringen, mehrere Reducer zu starten, da ansonsten in einem weiteren Job diese Teilergebnisse nochmals zusammengefasst werden müssten. Da allerdings die Combiner die Daten bereits teilweise aggregiert haben benötigt der Reducer kaum noch Zeit für des letzte Zusammenfassen der Daten.

Der HDR zeigt im Avg CPU/MEM, dass CPU und Speicher gut genutzt werden. Auffällig ist im Sum Disk IO Graphen, dass zwar auch die 350 GB gelesen werden, allerdings nur

mit einer Bandbreite von ca. 200 MB/s. Bemerkenswert daran ist, dass quasi deckungsgleich dazu dieselbe Bandbreite im Bandwidth Graphen bei der Übertragung der Daten über das „Loopback Device“ auftaucht. Das heißt, dass die Zugriffe der Mapper auf die Splits im HDFS, auch wenn die Daten lokal auf dem System liegen, über eine mit Netzwerksockets abgebildete Schnittstelle erfolgen und damit die Lesebandbreite reduzieren. Zu dem Zeitpunkt, zu dem die Lesebandbreite nachlässt, sieht man im MRT, dass einige Mapper länger brauchen als bei den vorherigen Splits. Hier werden, auf den Mapper folgend, Combiner gestartet um eine teilweise Aggregation vorzunehmen. Im Transfered Data Graphen erkennt man im Anschluss dieser Combiner die übertragene Datenmenge ansteigen, da die Teilergebnisse in diesem Moment vom Reducer kopiert werden. Die Menge der Zwischenergebnisse beläuft sich auf ca. 6,5 GB. Ein besonderes Merkmal von Hadoop ist, wie in TCP Connections zu sehen, das ständige Öffnen und Schliessen von neuen TCP Verbindungen vor bzw. nach jeder Übertragung. In diesem Fall benötigt die Anfrage 7641 Verbindungen. Jede dieser TCP Verbindungen wird mit einem „slow start“ begonnen, was aufgrund der geringen zu übertragenen Datenmenge pro Verbindung dazu führt, dass die vorhandene Bandbreite nicht optimal ausgenutzt werden kann. Hadoop, bzw. Pig benötigen zum Ausführen dieser Anfrage etwas über 2000 Sekunden.

4.2.2. Filter

Die Ergebnisse des Filter Tests (siehe Kapitel 3.4.4) sind in folgenden Graphen zu sehen:

APD:	Abb. D.2 auf Seite x
DIR:	Abb. 4.5 auf Seite 59
DNR:	Abb. 4.6 auf Seite 60
APP	Abb. D.9 auf Seite xvii
HDR:	Abb. 4.7 auf Seite 61
MRT	Abb. D.18 auf Seite xxv

Im APD ist die Version mit Zugriff auf den Index abgebildet. In der Version ohne Indexzugriff wären die Operatoren 5 bis 8 durch einen Tablescan auf Latency ersetzt, was dazu führen würde, dass der APD identisch mit dem APD der Aggregation aus dem letzten Abschnitt wäre. Die Filter Operation wird in dem Fall während des Tablescans ausgeführt, um so früh wie möglich unnötige Daten zu vermeiden. In der Version mit Index wird der Filter auf den Index angewendet. Da allerdings das RTT Feld nicht mit im Index gespeichert ist, müssen die Ergebnisse aus dem Index-Lookup umsortiert werden und die einzelnen Zeilen, bzw. die RTT Spalte in den Zeilen, nachgeladen werden. Während der DNR nahezu identisch mit dem DIR/DNR bei der Aggregation ist, sind im DIR Unterschiede zum Aggregationstest festzustellen. Im Avg CPU/MEM Graphen ist zu erkennen, dass zusätzlicher Speicher zum Sortieren benötigt wird. Bei Sum Disc IO ist zu sehen, dass die Lesebandbreite am Anfang nur ungefähr 500 MB/s beträgt und ab ca. 200 Sekunden sogar auf ca. 300

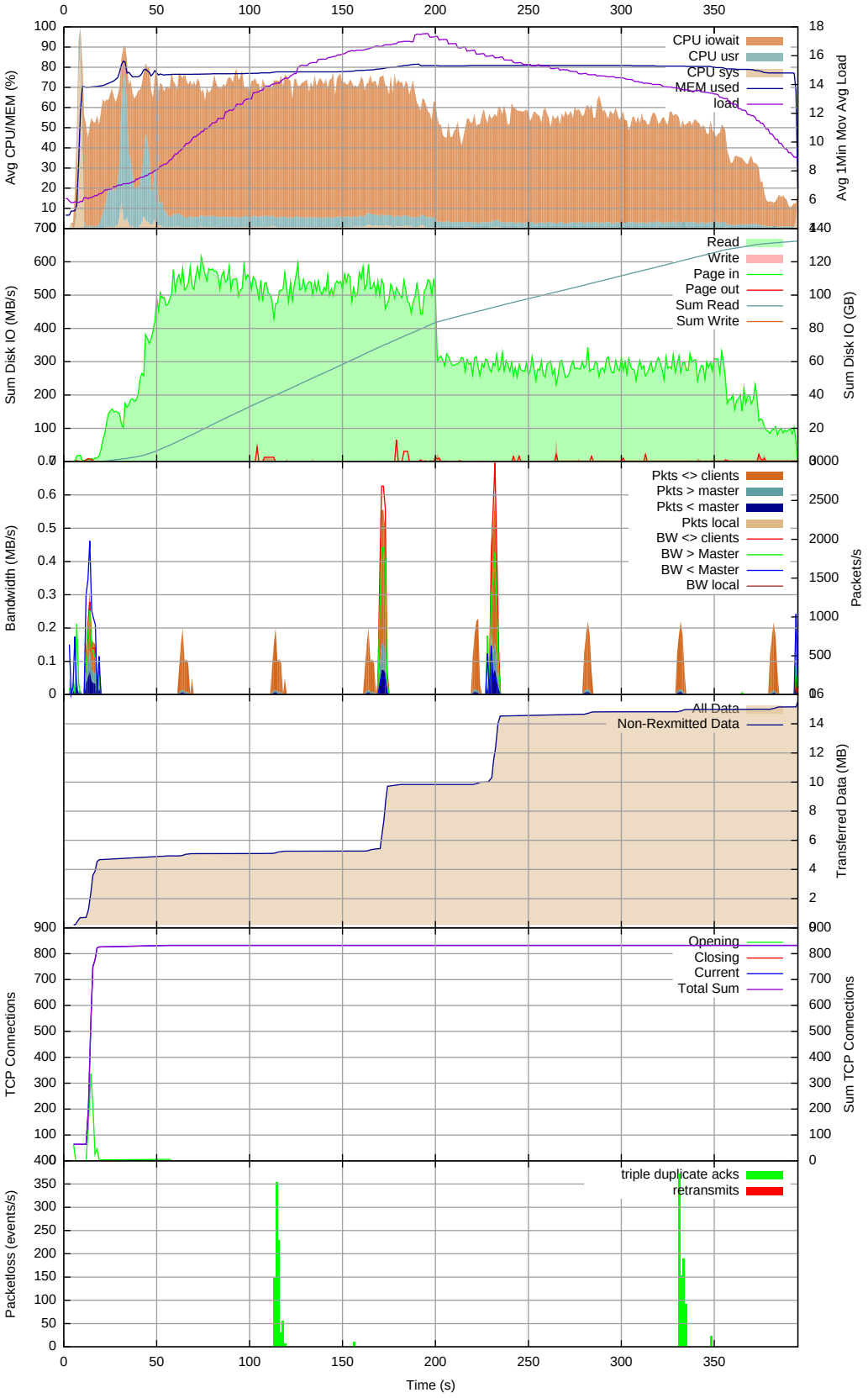

Abbildung 4.5.: DB2 Filter Report mit Index

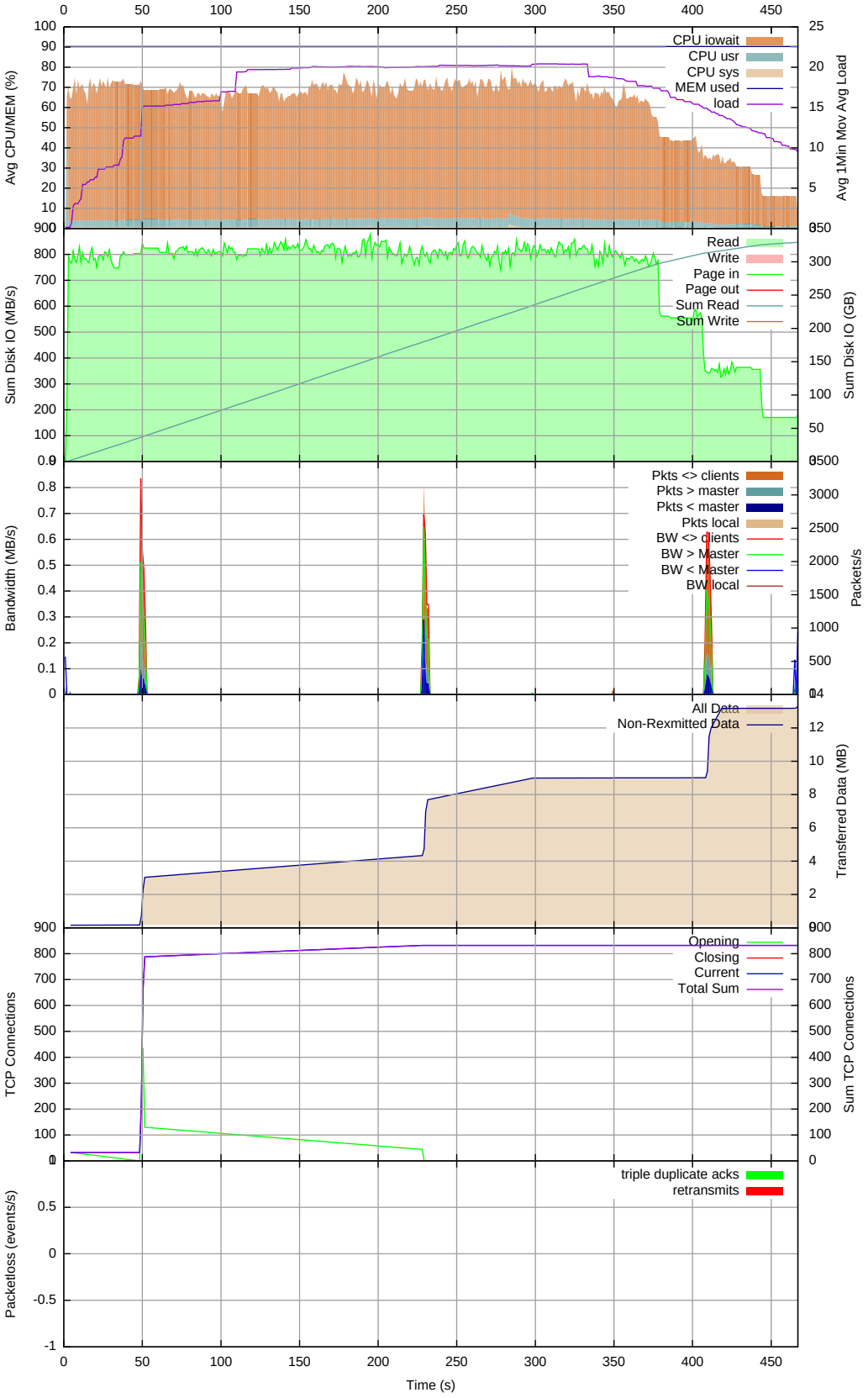

Abbildung 4.6.: DB2 Filter Report ohne Index

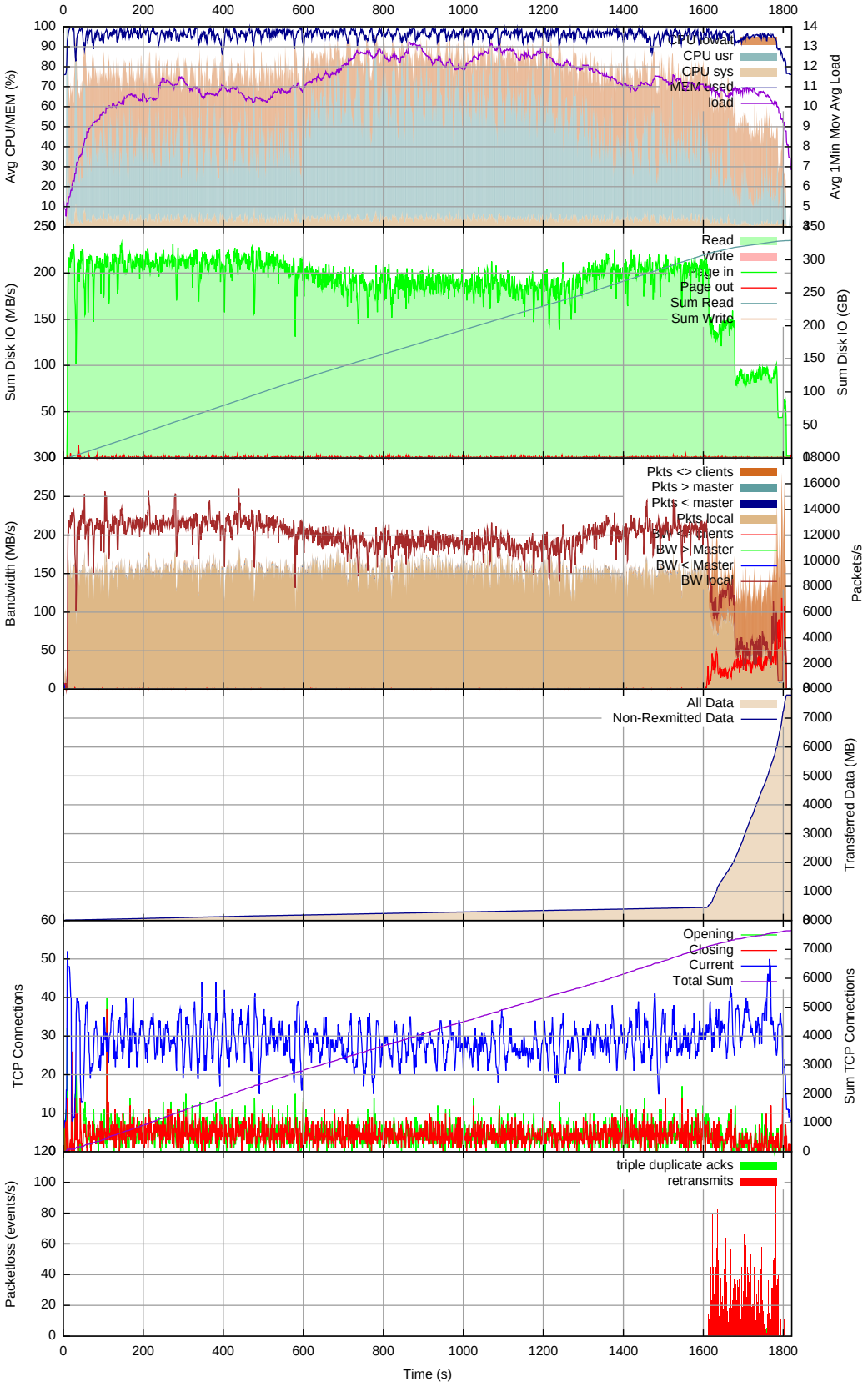

Abbildung 4.7.: Hadoop Filter Report

MB/s einbricht. Am Anfang werden Index und Daten gleichzeitig gelesen, d. h. die Platten müssen an zwei verschiedenen Positionen lesen. Die dafür notwenigen Kopfbewegungen reduzieren die Leserate. Ab ca. 200 Sekunden sieht es so aus, als ob der Index komplett gelesen wurde, anschließend müssen nun die zusätzlichen Daten eingelesen werden. Da nicht alle Datenseiten wie bei einem Tablescan eingelesen werden sondern nur spezifische über den Index referenzierte, reduziert sich hier durch Kopfsprünge der Platte ebenfalls die Bandbreite. Weiterhin auffällig ist, dass in der Version mit Index mehr Daten gelesen werden als in der Version ohne Index. Dies kommt dadurch zustande, dass auf den gelesenen Speicherseiten nicht nur jeweils eine Datenzeile gespeichert ist, sondern mehrere. Dadurch werden Daten eingelesen, die eigentlich nicht benötigt werden. Zusätzlich muss der Index gelesen werden, was ebenfalls Arbeitsspeicher benötigt. Da die Konfiguration der DB während der Tests mit dem Filter ohne Index von den restlichen Tests abwich, was erst nachträglich auffiel, ist bei diesen Tests der Speicherverbrauch leicht erhöht.

APP, HDR und MRT stimmen mit dem Test der Aggregation überein, da weiterhin dieselbe Datenmenge gelesen werden muss. Einzig die Ausführungszeit des Jobs reduziert sich ein wenig, da aufgrund des Filters nicht ganz so viele Daten in den Combinern und Reducern zusammengefasst werden müssen.

4.2.3. Group

Die Ergebnisse des Group Tests (siehe Kapitel 3.4.5) sind in folgenden Abbildungen zu finden:

APD:	Abb. D.3 auf Seite xi
DNR:	Abb. E.1 auf Seite xl
DIR:	Da die erstellten Indizes für diese Anfrage nicht benutzt werden, würde der DIR genauso aussehen wie der DNR.
APP	Abb. D.10 auf Seite xviii
HDR:	Abb. E.2 auf Seite xli
MRT	Abb. D.19 auf Seite xxvi

Die Systeme verhalten sich im Group Test genauso wie im Aggregationstest. Der einzige Unterschied besteht darin, dass nach mehreren Schlüsseln gruppiert wird.

4.2.4. Distinct

Beim Distinct Test aus Kapitel 3.4.6 sind in der Laufzeit recht deutliche Unterschiede zwischen dem Test mit Index und dem ohne Index auf der DB zu sehen.

APD: Abb. D.4 auf Seite xii
DIR: Abb. E.4 auf Seite xliii
DNR: Abb. E.3 auf Seite xlii
APP Abb. D.11 auf Seite xix
HDR: Abb. E.5 auf Seite xliv
MRT Abb. D.20 auf Seite xxvi

Während im Sum Disk IO Graphen des DNR zu erkennen ist, dass wieder ca. 350 GB gelesen werden müssen, liest, wie im DIR zu sehen, die Version mit Index nur ca. 9 GB. Der Index LAT_TARGET beinhaltet bereits alle benötigten Felder. Weiterhin ist in der Version ohne Index zu sehen, dass nun der komplette Sortierspeicher der DB genutzt wird. In Summe benötigt die Version ohne Index etwas über 600 Sekunden, die Version mit Index nur ca. 53 Sekunden.

Der HDR zeigt, dass bei Hadoop die komplette Verarbeitung in nur einem MR Job abläuft, der das Distinct dadurch abbildet, dass in den Sortiervorgängen und in der Shufflephase das Target Feld zum Schlüssel wird. Der APP sieht wieder fast genauso aus wie beim Aggregationstest. Durch die geringere Menge an Targets benötigt allerdings das Sortieren weniger Zeit, was an einer etwas niedrigeren CPU sys Auslastung im Avg CPU/MEM Graphen und an der geringeren Dauer der Anfrage von nur ca. 1700 Sekunden erkennbar wird.

4.2.5. Collocated Join

Der Test des „Collocated Join“ aus Kapitel 3.4.7 unterscheidet sich deutlich von den bisher besprochenen Tests, denn aufgrund der Komplexität der Anfrage ist es für Hadoop nicht mehr möglich, die Anfrage in nur einem MR Job zu durchzuführen.

APD: Abb. D.5 auf Seite xiii
DNR: Abb. 4.8 auf Seite 64
APP Abb. D.12 auf Seite xx
HDR: Abb. 4.9 auf Seite 65
MRT Abb. D.21 auf Seite xxvii

Dass es sich bei dem Join im Test um einen collocated Join handelt, ist im APP zu erkennen. Der Join Operator, in diesem Fall ein Hash Join (Operator HSJOIN), greift auf Daten zu, die lokal auf den Partitionen liegen. Erst weit nach dem Join werden die Daten an den koordinierenden Knoten übertragen.

Dass die Latency Tabelle zur Ausführung der Anfrage doppelt gelesen werden muss, ist im Sum Disk IO Graphen des DNR zu sehen. Es werden von der DB fast 700 GB eingelesen, also genau das Doppelte von dem, was in den bisherigen Tests gelesen wurde. Im Avg CPU/MEM Graphen ist zu erkennen, dass CPU usr nur noch ca. 10% beträgt, denn

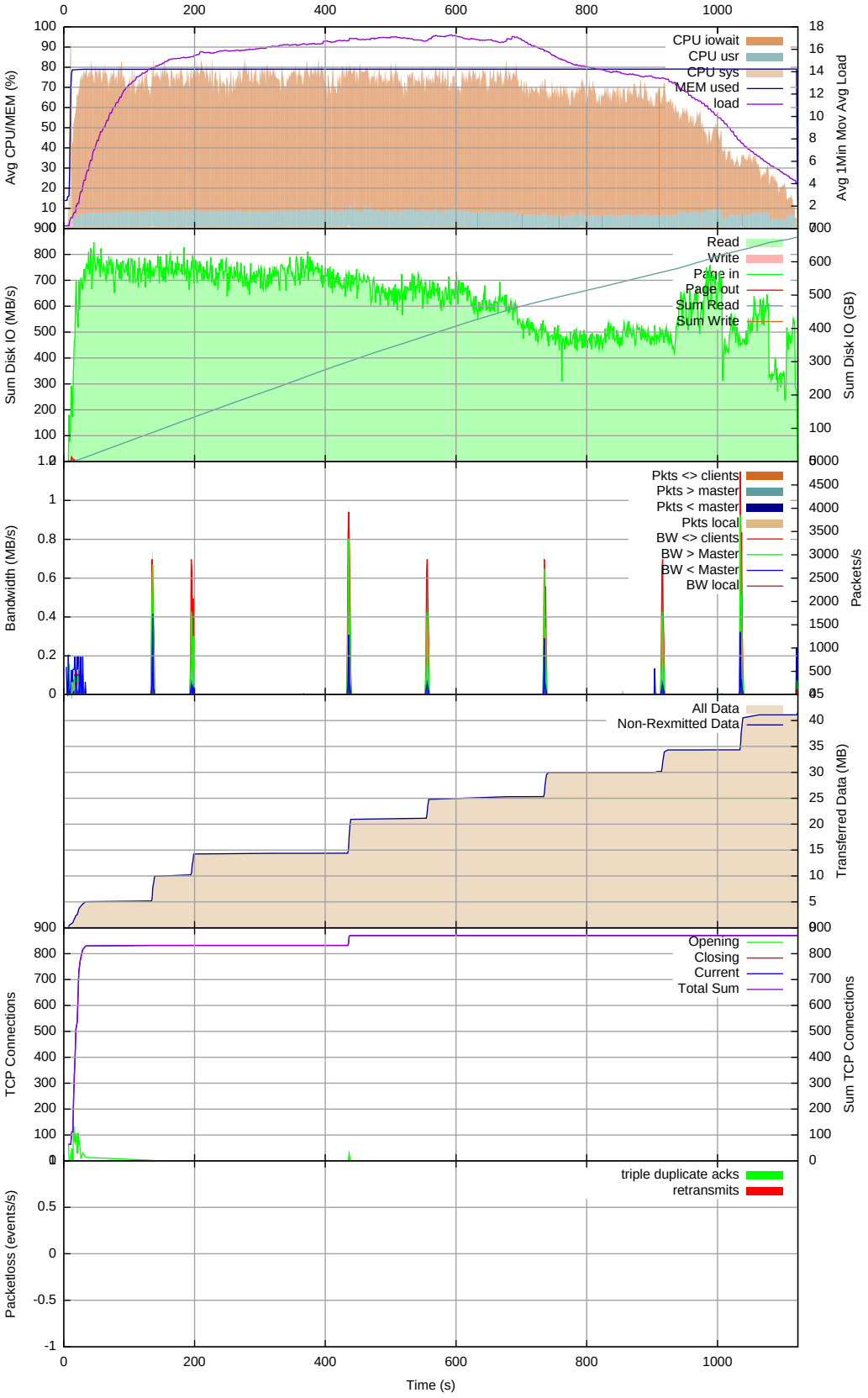

Abbildung 4.8.: DB2 collocated Join Report

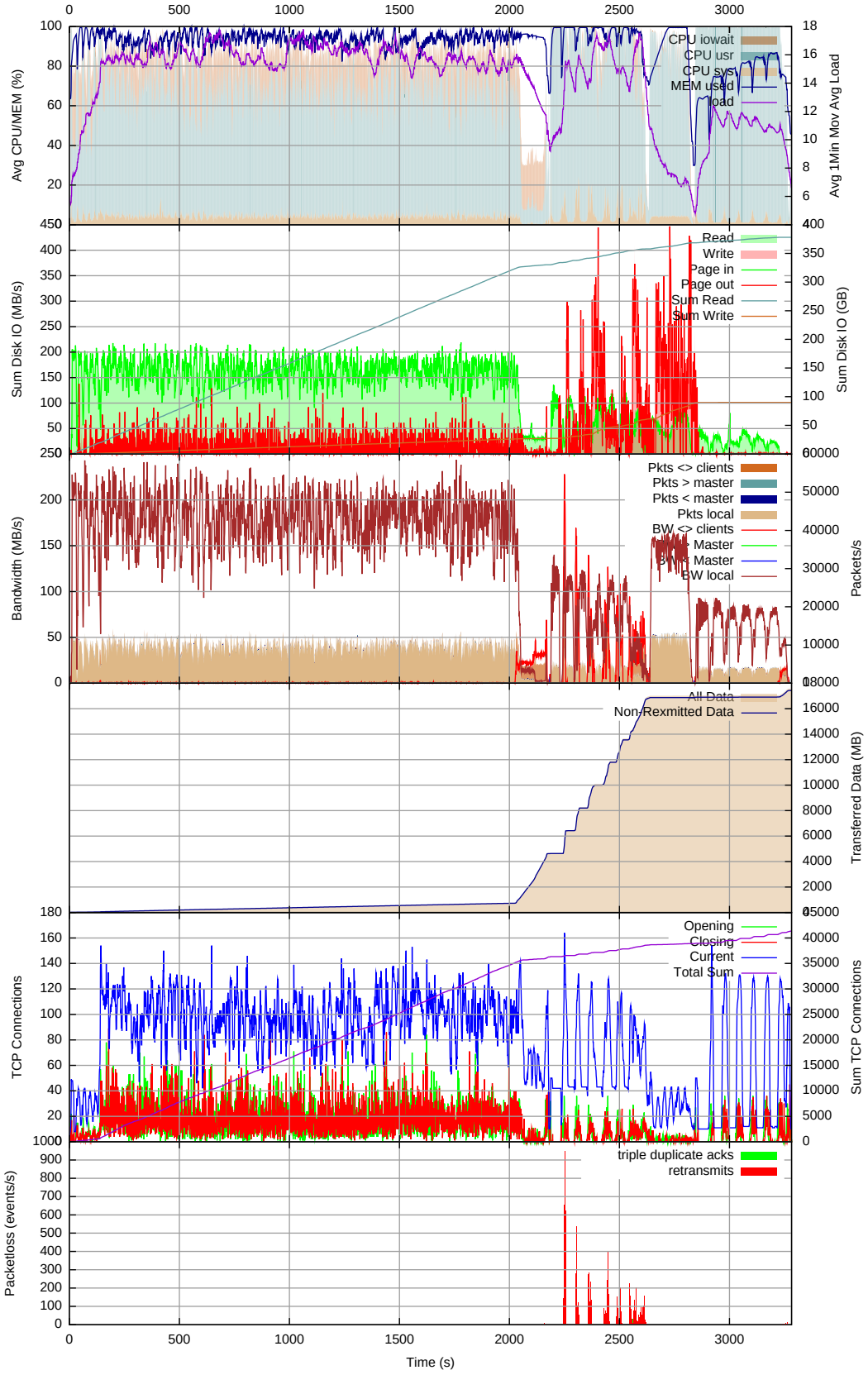

Abbildung 4.9.: Hadoop collocated Join Report

die Systeme werden jetzt erheblich durch die Mehrbelastung der Festplatten gebremst. Der Fast Communication Manager (FCM), der bei DB2 für die Kommunikation mit den anderen Partitionen zuständig ist, kann die anfallenden Daten, die in diesem Fall aufgrund der Gruppierung in Peaks auftreten, nicht schnell genug über eine TCP Verbindung versenden. Es müssen nach einiger Zeit, wie im Graphen TCP Connections zu erkennen, noch weitere TCP Verbindungen geöffnet werden. Insgesamt werden nur ca. 41 MB über das Netzwerk transportiert.

Bei Hadoop läuft der Join etwas anders ab. Der APP zeigt, dass für diese Anfrage drei MR Jobs benötigt werden, im ersten Job werden die Durchschnitte berechnet, im zweiten Job werden die Teilergebnisse mit der Latency Tabelle „gejoined“ und im dritten Job wird die Aggregation nach dem Feld Hour vorgenommen. Deutlich sind diese drei Jobs auch im MRT zu sehen. Die Reducer des ersten und dritten Job benötigen nur wenig Zeit, da nach den entsprechenden Mappern bereits Combiner zum Einsatz kamen, die die Datenmenge bereits reduziert haben. Der HDR zeigt eine Besonderheit von Hadoop. Im Mapper des ersten Jobs werden mit Hilfe einer „Split“ Operation die für den später erfolgenden Join benötigten Daten aus der Latency Tabelle zusammengefasst mit den Ergebnissen des Maps, als ein Teilergebnis gesichert. Dafür muss zwar etwas mehr geschrieben werden, allerdings kann das wiederholte Einlesen der Latency Tabelle beim Join vermieden werden. Der Sum Disc IO Graph zeigt, dass es Hadoop dadurch möglich ist, mit dem Lesen von ca. 370 GB und Schreiben von ca. 90 GB auszukommen. Das Schreiben der bis zum Ende des ersten Jobs angefallenen ca. 25 GB für den Join macht sich, im Vergleich zum erneuten Lesen von 350 GB wie bei der DB, bezogen auf die Laufzeit deutlich bemerkbar. Allerdings schreibt Hadoop zusätzlich weitere ca. 65 GB beim Join in Job 2, wofür nochmals Zeit benötigt wird. Die Ineffizienz des Joins in Job 2 zeigt sich auch im Transfered Data Graphen, hier sieht man, dass Hadoop insgesamt ca. 17 GB an Daten überträgt, wogegen es bei der Datenbank ca. 41 MB sind. Hadoop öffnet für diese Übertragung über 41.000 TCP Verbindungen. Hier erkennt man deutlich den Unterschied zwischen einem collocated Join, wie ihn die Datenbank ausführt, und einem directed Join bei Hadoop.

4.2.6. Replicated Join

Die Anfrage für den replicated Join aus Kapitel 3.4.8 ist etwas einfacher, als die im vorherigen Abschnitt besprochene Anfrage zum collocated Join.

APD:	Abb. D.6 auf Seite xiv
DNR:	Abb. 4.10 auf Seite 67
APP	Abb. D.14 auf Seite xxii
HDR:	Abb. 4.11 auf Seite 68
MRT	Abb. D.22 auf Seite xxvii

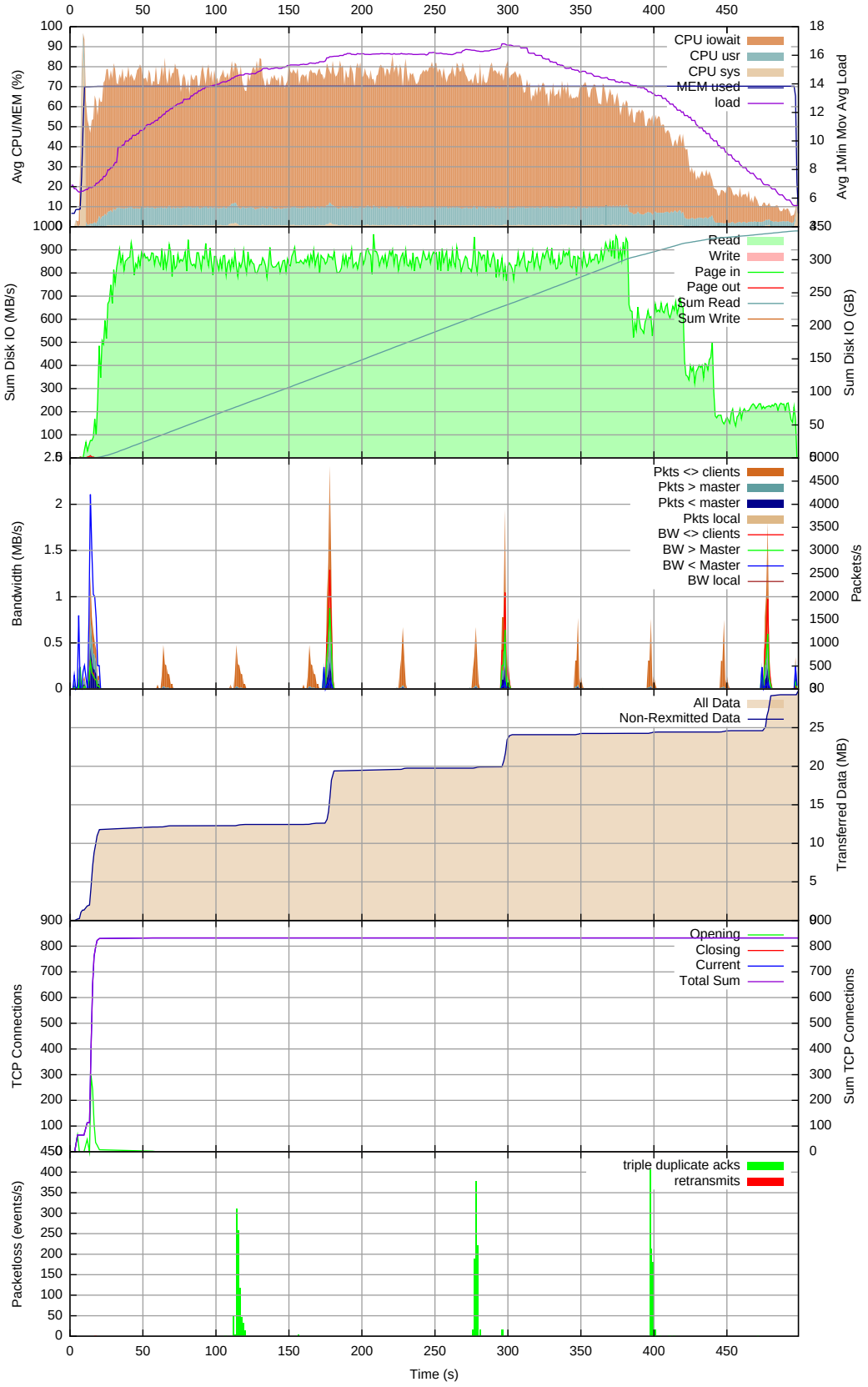

Abbildung 4.10.: DB2 replicated Join Report

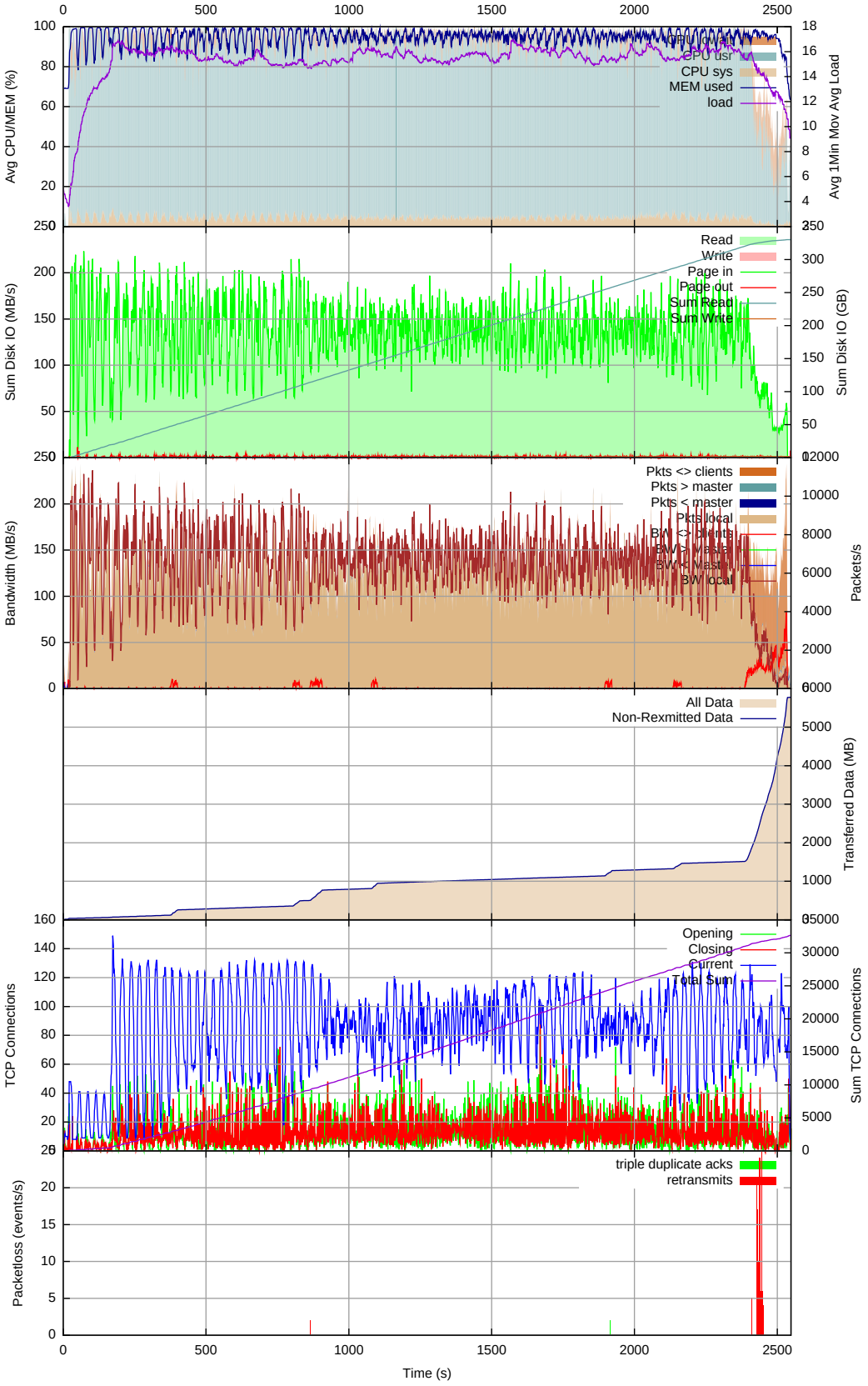

Abbildung 4.11.: Hadoop replicated Join Report

Im APD ist zu sehen, dass die Metadata Tabelle mittels einer „Broadcast Tablequeue“ vom Master auf alle Systeme repliziert wird, bevor es zum Join kommt. Da die Metadata Tabelle ein vielfaches kleiner als die Latency Tabelle ist, ist im DNR im Bandwith und Packets Graphen nur eine geringer Ausschlag innerhalb der ersten 20 Sekunden zu erkennen, in dem nach Transfered Data ca. 12 MB an Daten übertragen wurden. Für den Join wird ein Tablescan über Latency durchgeführt, anschließend lokal gruppiert und zum Schluss auf Master aggregiert.

Hadoop nutzt dem APP nach zwei Jobs. Der erste Job besteht nur aus einem Mapper über die Metadata Tabelle, um die Projektion für die benötigten Felder auszuführen. Der eigentliche Join wird im Mapper des zweiten MR Jobs ausgeführt. Aufgrund der geringen Größe der Metadata Tabelle, kann hier jeder Mapper die Tabelle komplett im Speicher halten, was dazu führt, dass der Join bereits im Mapper ausgeführt werden kann. Da die Gruppierung größtenteils bereits im Combiner erfolgt, erledigt der Reducer, wie im MRT zu sehen, die abschließende Aggregation innerhalb recht kurzer Zeit. Im Verhältnis zur Datenbank fällt bei Hadoop trotzdem noch ein vielfaches an Daten zum endgültigen Gruppieren an. Nach dem Transfered Data Graphen werden fast 6 GB an die Reducer übertragen.

4.2.7. Directed Join

Der directed Join aus Kapitel 3.4.9 verbindet Latency mit der Position Tabelle, die um mehr als den Faktor 1000 größer ist als die zuvor genutzte Metadata Tabelle. Hier wäre ein replicated Join nicht mehr sinnvoll, da unnötig viele Daten übertragen werden müssten.

APD:	Abb. D.7 auf Seite xv
DNR:	Abb. 4.12 auf Seite 70
APP	Abb. D.15 auf Seite xxiii
HDR:	Abb. 4.13 auf Seite 71
MRT	Abb. D.23 auf Seite xxviii

Der APD zeigt eine „Directed Tablequeue“, die in diesem Fall die Zeilen aus der Position Tabelle nach demselben Schema, nach dem auch die Latency Tabelle zwischen den Partitionen verteilt ist, hashed und so für jede Zeile eine eindeutige Zielpartition bestimmen kann. Dort kann im nächsten Operator mit Latency „gejoined“ und im Folgenden die restliche Aggregation durchgeführt werden. Auffällig ist, dass die Reduktion der Menge der Schlüssel für die Gruppierung in der Optimierung der Anfrage nicht erkannt wird, weshalb mit einer vielfach so großen Anzahl der Zeilen geplant wird, wie nötig. Im DNR sieht man im Sum Disk IO Graphen, dass die für den Join nötige Hashtabelle zu groß wird um sie komplett im Speicher zu halten. Die Datenbank wird gezwungen, die Tabelle auf die Festplatte auszulagern, wodurch auch die Leserate deutlich einbricht. Wie man im Avg CPU/MEM Graphen

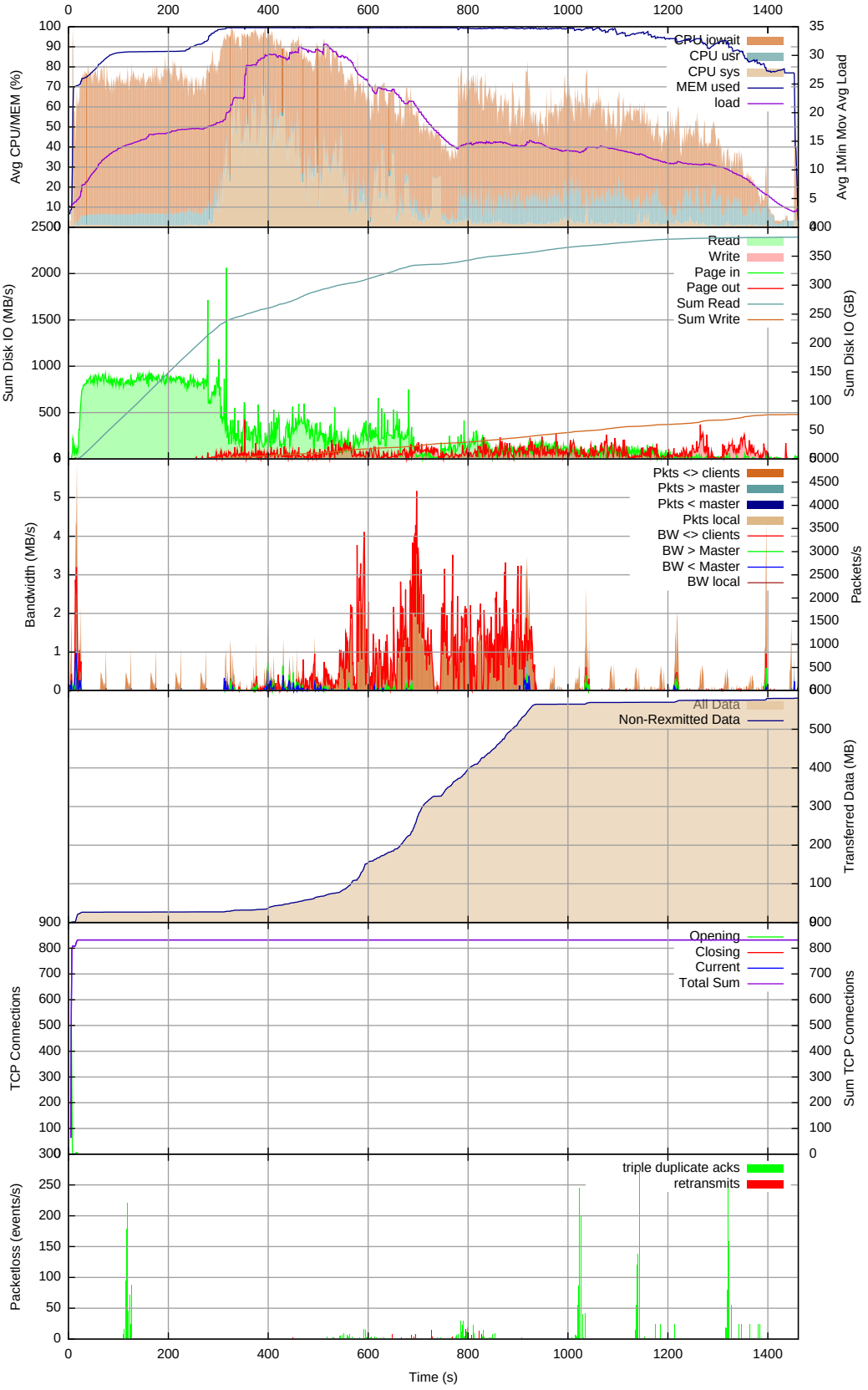

Abbildung 4.12.: DB2 directed Join Report

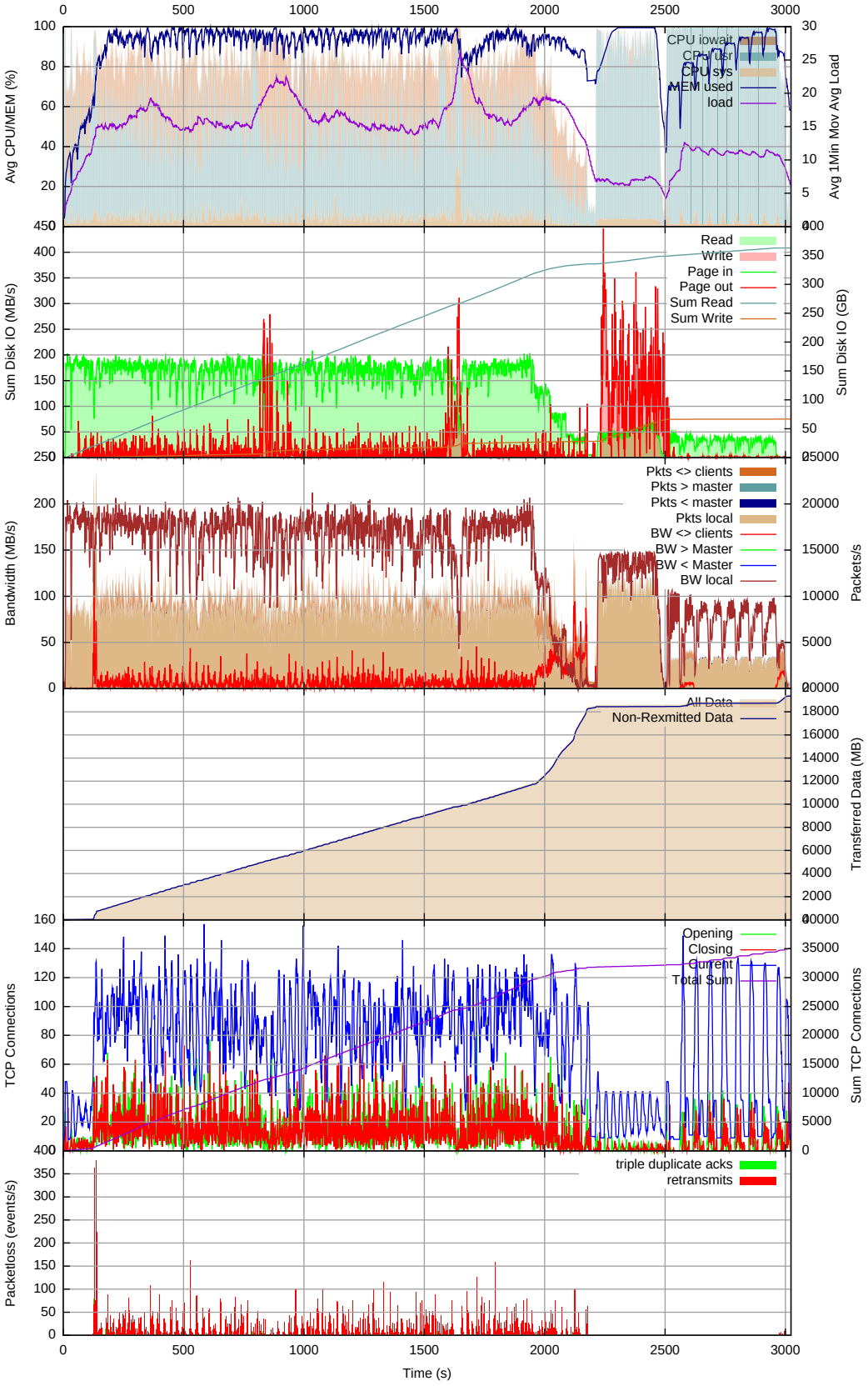

Abbildung 4.13.: Hadoop directed Join Report

erkennt, werden von ca. 300 Sekunden bis ca. 750 Sekunden kaum noch CPU Ressourcen für Userprozesse eingesetzt. Bei ca. 950 Sekunden endet, nach dem Transfered Data Graphen zu urteilen, der Hash Join. Bis dahin hat DB2 die Position Tabelle übertragen, die nach dem APD als „inner Table“ als zweite die Hashtabelle durchläuft, worauf folgend die Ergebnisse an den Sortieroperator weitergegeben werden. Nach dem Gruppieren bleiben nur noch wenige Zeilen zum Übertragen übrig.

Der APP zeigt, dass der Mapper des ersten Jobs aus den benötigten Feldern der Latency Tabelle sowie denen der Position Tabelle ein Tupel bildet, das im anschließenden Reducer „gejoined“ wird. Im Transfered Data Graphen des HDR sieht man, dass dazu über 18 GB über das Netzwerk versendet werden, wofür Hadoop in Summe über 36.000 TCP Verbindungen benötigt. Der zweite Job übernimmt, wie bei den bisherigen Tests auch, das Gruppieren am Schluss.

Zusätzlich zu den bereits gezeigten Graphen wurde beispielhaft für alle Anfragen beim directed Join zusätzlich die TCP Segmentgrößen der Kommunikation zwischen den Knoten visualisiert. Abb. 4.14 zeigt den Graphen für die Datenbank, Abb. 4.15 den für Hadoop. Horizontal ist auf dem Graphen die relative Zeit während der Ausführung der Anfrage abgetragen, während vertikal die TCP Segmentgröße der übertragenen Pakete dargestellt ist. Die Farbe gibt die Anzahl der übertragenen Pakete in der jeweiligen Größe und Sekunde an.

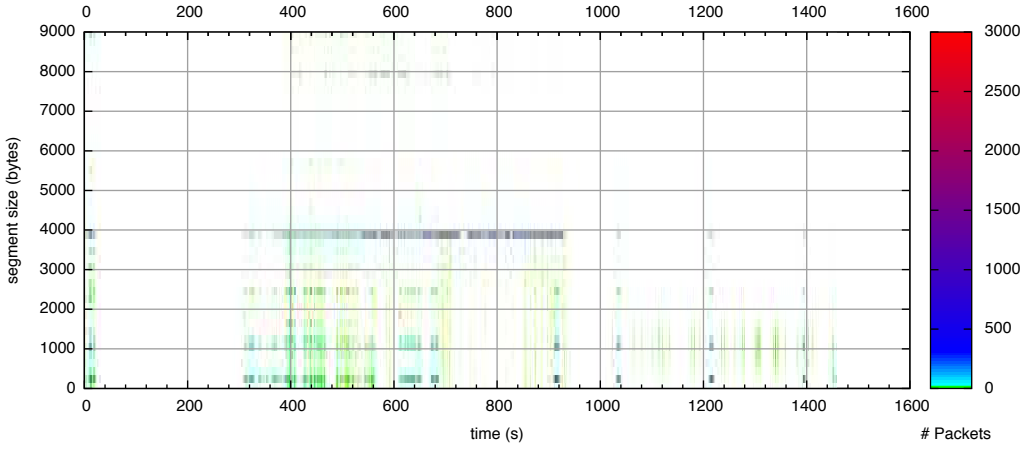

Abbildung 4.14.: Verteilung TCP Segmentgrößen DB2 directed Join

Wie im Versuchsaufbau beschrieben, werden im Netzwerk zwischen den Knoten Jumbo Frames mit bis zu 9000 Bytes Größe transportiert. Bemerkenswert ist, dass Hadoop die Jumbo Frames auch ausnutzt, während DB2 größtenteils nur ca. 4KB große Frames überträgt. Allerdings wurden die Tests zu einem Zeitpunkt erstellt, zu dem die Datenbank auf einem Tablespace mit 4K Seitengröße installiert war. Es ist davon auszugehen, dass DB2 auch größere Frames erstellt, sobald die Seitengröße erhöht wird, was aus zeitlichen Gründen nicht verifiziert werden konnte. Da die TCP-Segmente bei den Übertragungen nicht

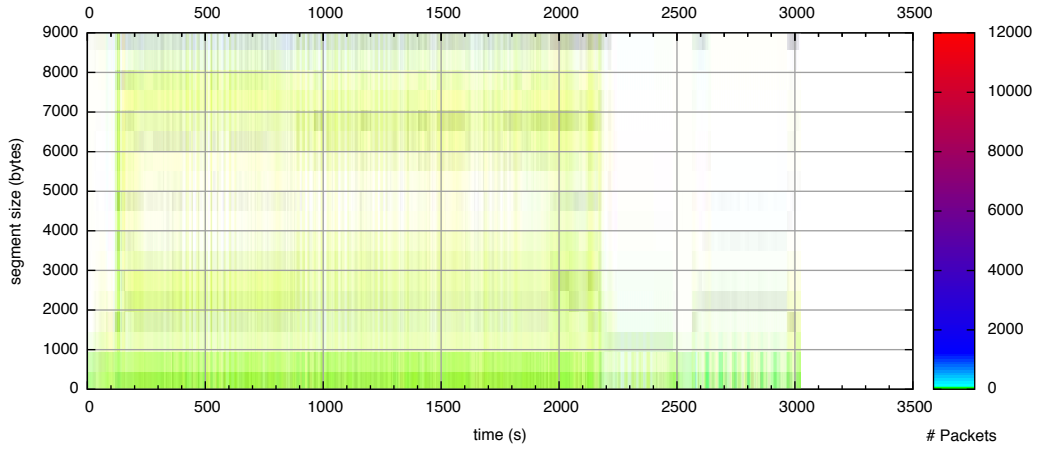

Abbildung 4.15.: Verteilung TCP Segmentgrößen Hadoop directed Join

„aufgefüllt“ werden, deaktiviert DB2 unter Linux offenbar den Nagle-Algorithmus, was nach Baragoin u. a. [Bar+03] von IBM aus Performancegründen für die Kommunikation der Knoten untereinander auch empfohlen wird.

4.2.8. Laden der Daten

Das Laden der Daten aus Kapitel 3.4.2 ist zwar normalerweise immer der erste Schritt, allerdings sehen diese Graphen anders aus als die Tests der Anfragen, weshalb dieser Abschnitt als letztes behandelt wird. Es wurde nur das Laden der größten Tabelle, der LATENCY Tabelle protokolliert. Da hierfür keine Anfrage auf die Datenbank stattfindet, bzw. keine Jobs auf dem MR Framework ausgeführt werden, existieren für diesen Test keine Ausführungspläne und kein MapReduce Taskscheduling. Der DNR stellt in diesem Test das Laden der Daten ohne Integritätsbedingungen, Indizes und Tabellenstatistiken dar, während der DIR das Laden mit Integritätsbedingungen, Indizes und Tabellenstatistiken zeigt. Der HDR bildet in diesem Test das Kopieren der Daten in das HDFS ab.

DNR: Abb. 4.16 auf Seite 74
DIR: Abb. 4.17 auf Seite 75
HDR: Abb. 4.18 auf Seite 76

Bei DB2 erfolgt der Ladevorgang, indem die koordinierende Partition die eingehenden Daten nach dem Verteilungsschema aus der Tabellendefinition vorsortiert und dann nur die für jede Partition notwendigen Zeilen entsprechend überträgt. Die koordinierende Partition befindet sich auf MASTER. Im TRANSFERED DATA Graphen im DNR fällt auf, dass von DB2 ca. die doppelte Menge an Daten übertragen werden, gegenüber der Menge, die in den Tabellen steht.

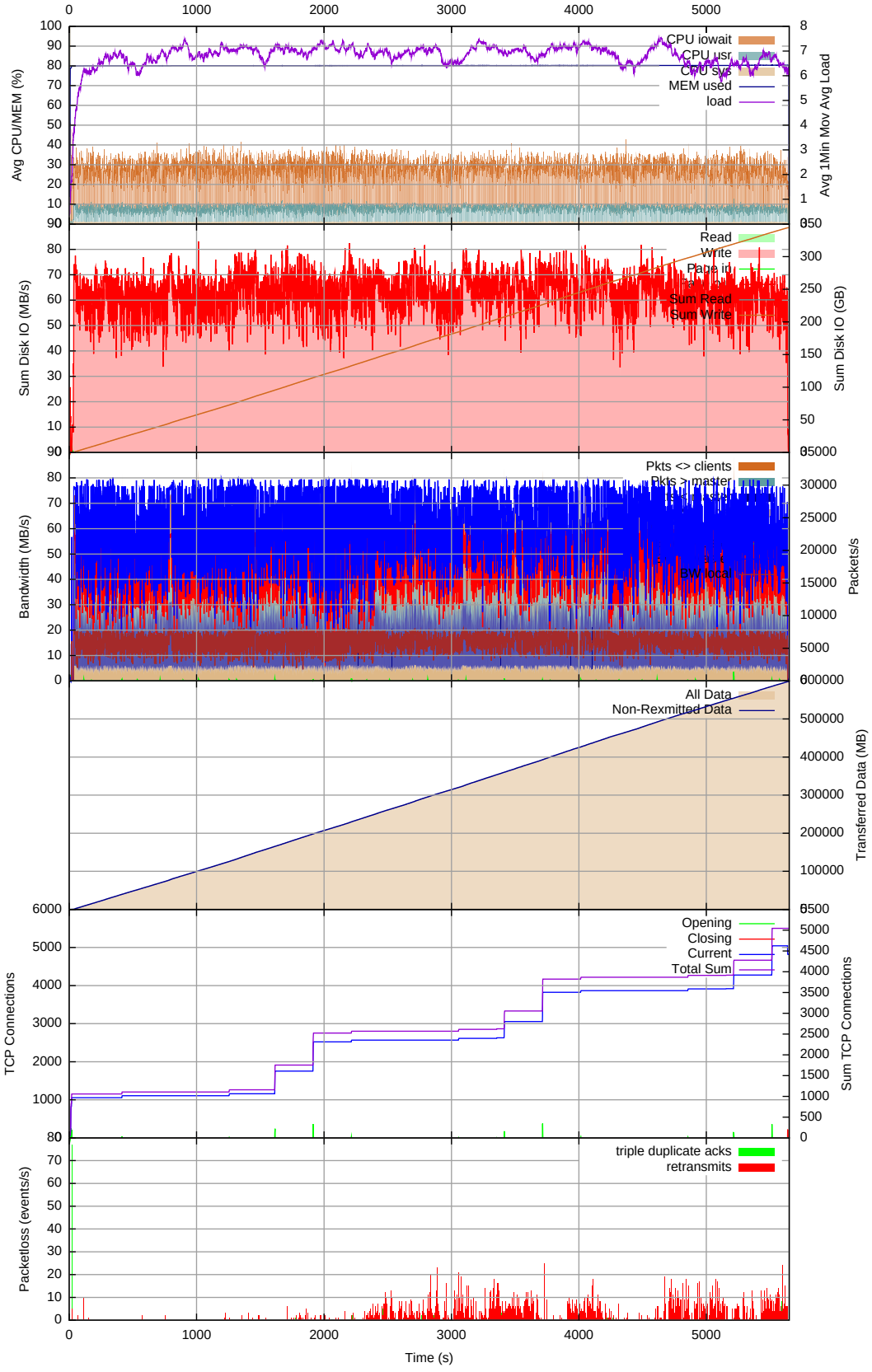

Abbildung 4.16.: DB2 Load Report ohne Index

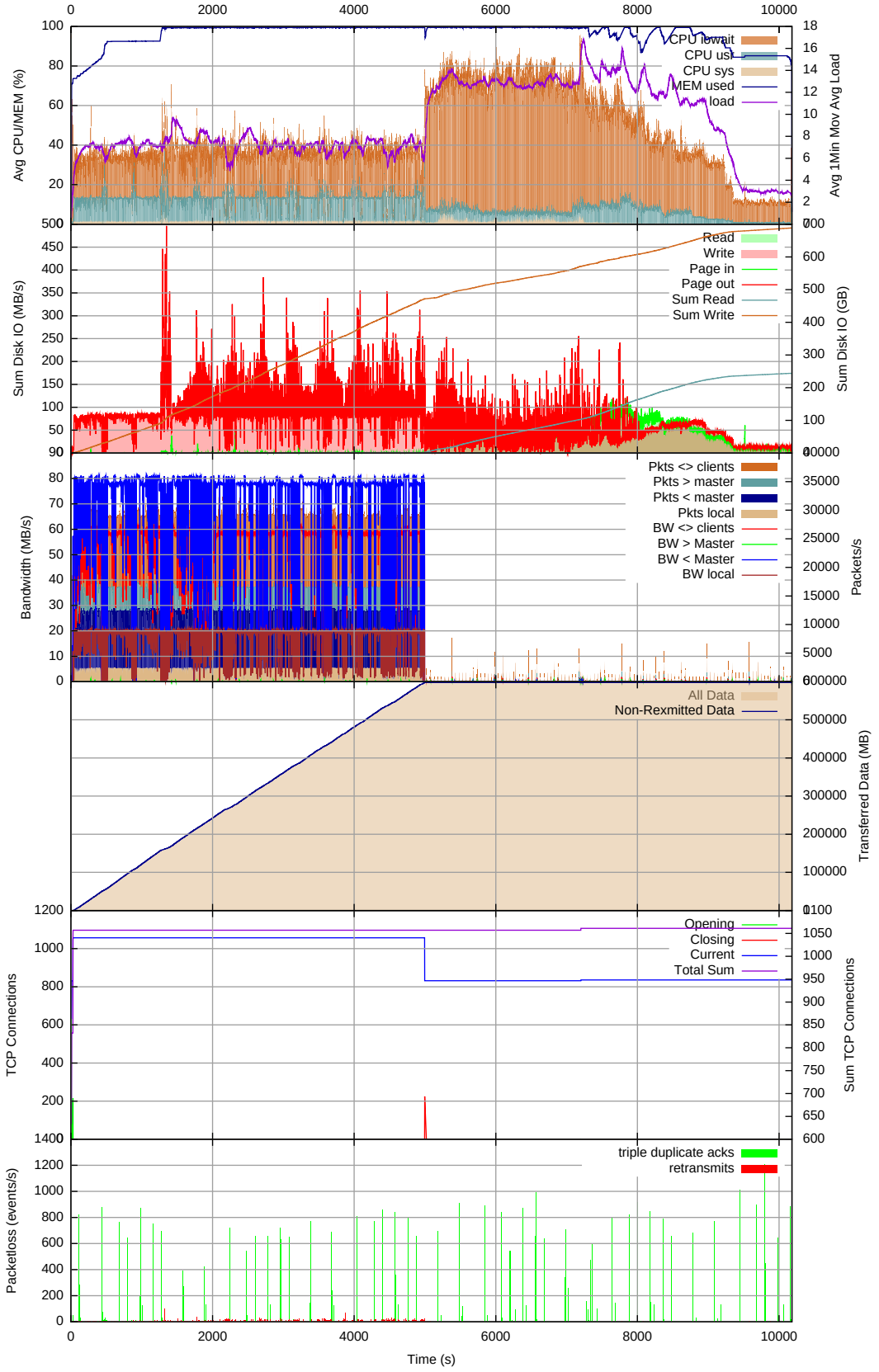

Abbildung 4.17.: DB2 Load Report

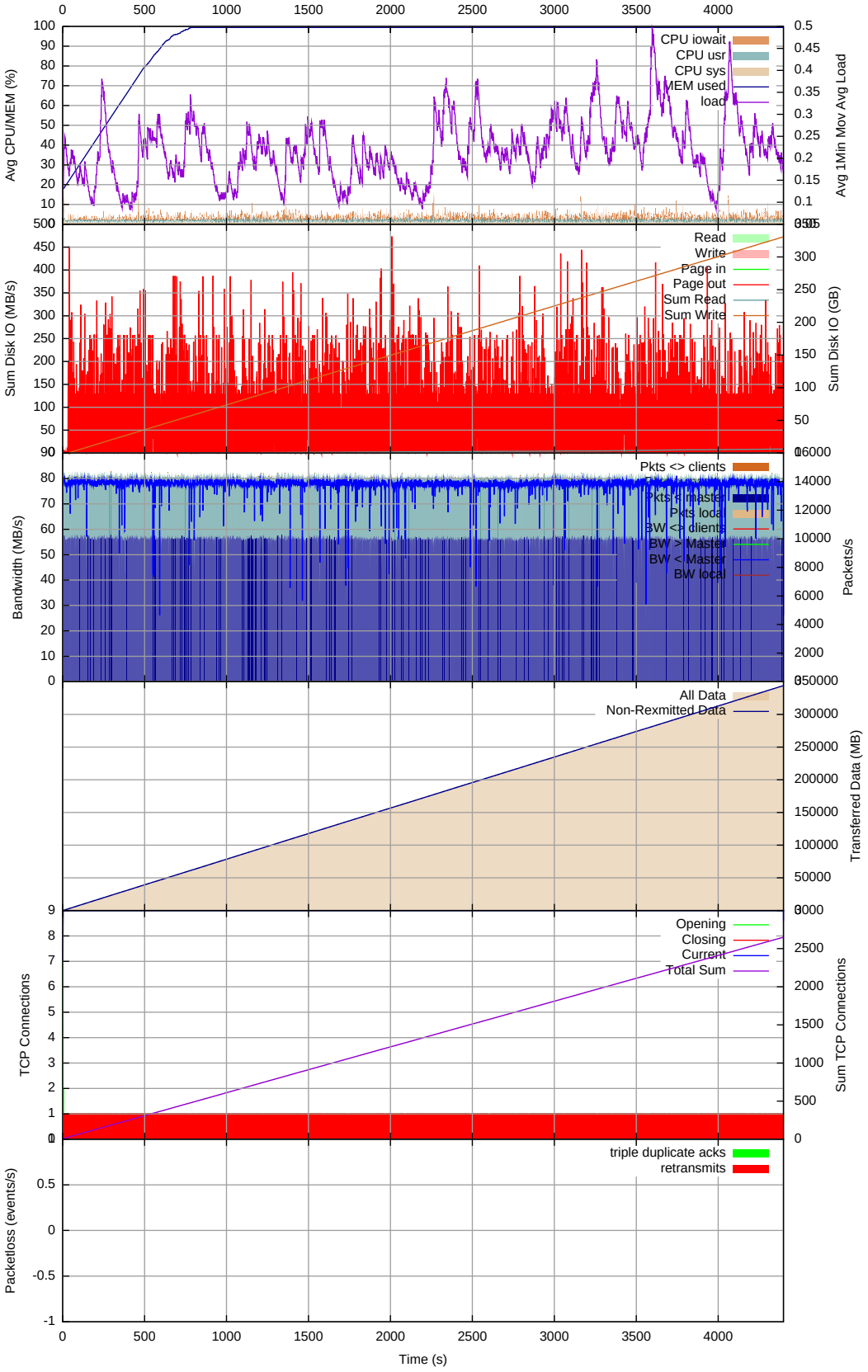

Abbildung 4.18.: Hadoop Load Report

Im DIR kann man den Mehraufwand sehen, der noch zusätzlich entsteht, wenn Integritätsbedingungen, Indizes und Tabellenstatistiken benötigt werden. Nachdem der Schreibvorgang auf allen Partitionen bei ca. 5000 Sek. beendet ist, folgt die Phase, in der die DB die Indizes anlegt, die Integritätsbedingungen überprüft und die Tabellenstatistiken erstellt. Es werden erneut nahezu alle Daten eingelesen und wie im Sum Disk IO Graphen zu sehen, die Indizes geschrieben. Es ist nicht notwendig, nochmals alle Daten komplett einzulesen, da die Daten, die sich zu Beginn dieser Phase noch in den Bufferpools befinden, nicht erneut gelesen werden müssen. Wenn man die Daten in Blöcken einladen würde, die kleiner als die Bufferpools sind, wäre es nicht erforderlich, die Daten erneut einzulesen.

Für Hadoop ergibt sich im HDR ein einfaches Bild, die Daten werden nicht interpretiert oder dekodiert sondern in gleichmäßigen Blöcken schnellstmöglich über alle Systeme verteilt. Dadurch gelingt es Hadoop, die Daten bedeutend schneller als die Datenbank zu speichern.

4.3. Prognosen zu den Tests mit Realdaten

Um die Aussagekräftigkeit der ermittelten Ergebnisse darstellen zu können werden mit Hilfe dieser Daten Prognosen für die noch zu testenden Anfragen auf den Realdaten erstellt. Nachdem dann im folgenden Unterkapitel die Ergebnisse der Anfragen erläutert werden, folgt zum Schluss eine Verifikation der hier erstellten Vorhersagen.

Die Laufzeiten der bisherigen Tests sind in Abb. 4.19 im Vergleich abgebildet.

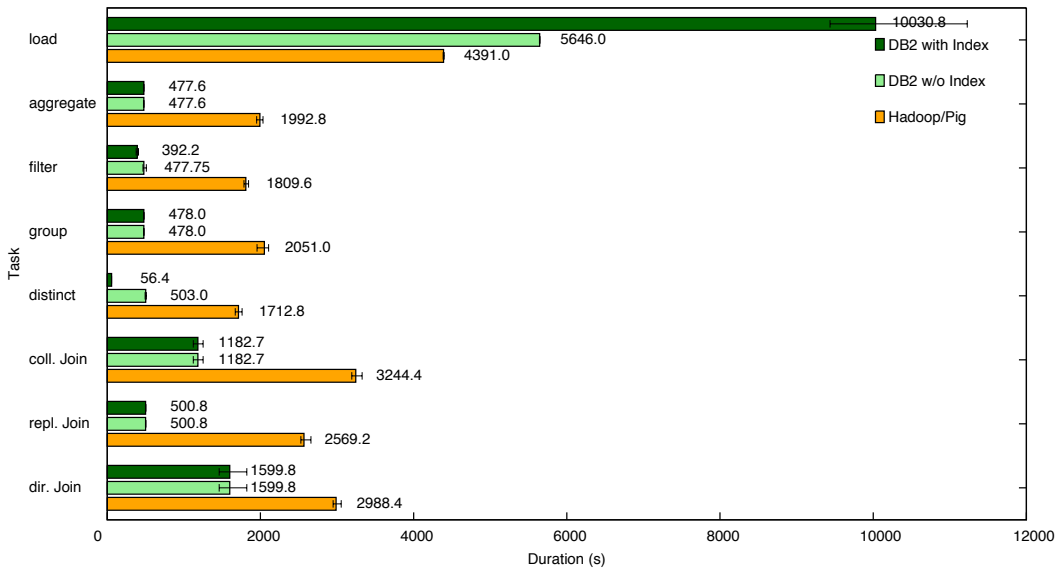

Abbildung 4.19.: Laufzeiten der synthetischen Tests

Ein Großteil des Laufzeitunterschieds der Anfragen wird nach den bisherigen Tests durch das langsamere Lesen der Daten durch Hadoop verursacht. Die Datenbank war an dieser Stelle immer um einen Faktor von mindestens 3 schneller. Auch das Übertragen der Daten mittels kurzlebiger TCP Verbindungen sorgt dafür, dass Hadoop nicht so effizient arbeitet, wie es möglich wäre. Allerdings hält sich die Menge der übertragenen Daten im Vergleich zu den verarbeiteten Daten dennoch im Rahmen.

Auch DB2 verhält sich nicht optimal, so führen zu viele parallele Plattenzugriffe, wie bereits in Kapitel 3.2.2 kurz erläutert wurde, zu Verzögerungen. Besonders beim Laden benötigt DB2 mehr Zeit als Hadoop. Das, wie man beim Vergleich der DNRs beim Laden der Daten sehen kann, ist allerdings gerade durch die Mechanismen bedingt, die die Datenbank bei späteren Anfragen beschleunigen. Die Zeiten, die die Datenbank zum eigentlichen Schreiben der geladenen Daten benötigt ist wie bei Hadoop hauptsächlich vom Plattendurchsatz abhängig.

Betrachtet man die Rahmenbedingungen für die Tests mit den echten Daten in Kapitel 3.5.5 an, fällt auf, dass die Ladetests nicht mit der gesamten großen Announces Tabelle durchgeführt werden können, da die Laufzeit dafür zu hoch wäre. Die Ladetests werden also nur für einzelne Timebins erstellt. Da allerdings jeder Ladevorgang gleich verläuft, lässt sich das Ergebnis eines einzelnen Vorgangs auf die gesamte Tabelle extrapolieren. Im Durchschnitt beträgt die Größe eines zu ladenden Timebins der Announces Tabelle ca. 1.1 GB.

Für Hadoop wird für eine Schätzung des Ladezeit aus den bisherigen Ladezeiten linear interpoliert. Die Datenbank benötigte ohne Indizes und Statistik ca. 30% länger, dieser Wert wird auch hier angesetzt. Für die Schätzung mit Index wird davon ausgegangen, dass die Daten nicht erneut gelesen werden müssen, da sie sich bereits im Speicher befinden. Auch ist der Index, wie in Kapitel 3.5.4 beschrieben, so angelegt, dass er beim Importieren weiterer Timebins nicht umsortiert werden muss, es werden lediglich Daten am Ende angehängt. Er beinhaltet nur 4 Integerfelder und die RowIDs pro Datensatz. Die zu schreibende Datenmenge ist also pro Timebin relativ klein. Es sind keine Integritätsbedingungen auf der Tabelle definiert und die Tabellenstatistik wird erst nach dem Laden aller Timebins erstellt. Es wird somit nur die Zeit benötigt, um den Indexbaum für die neu importierten Daten zu erstellen. Zu dieser Zeit kann mit den bisherigen Daten keine genaue Aussage getroffen werden, allerdings wird es geschätzt im Bereich weniger Sekunden liegen.

4.3.1. Abschätzung für BitTorrent Job 1

Bei der Abschätzung der Ausführungszeit von BT Job 1 aus Kapitel 3.5.6 hilf der vereinfachte Ausführungsplan in Abb. 3.9. Bei der Anfrage der Datenbank ohne Index muss ein Tablescan auf Announces ausgeführt werden. Im darauf folgenden Schritt werden die Daten sortiert und gruppiert. Anschliessend folgen die beiden Joins. Der Sortieroperator bremst

diese Kette von Operationen, da er erst nach dem Lesen aller Daten die Ergebnisse weiterreichen kann. Die Datenbank hat in den bisherigen Tests relativ konsistent mit ca. 900 MB/s gelesen. Allein zum Lesen der Daten wären somit ca. 2000 Sekunden nötig. Allerdings ist die nach der Sortierung entstehende Datenmenge mit 4 Integerfeldern pro Datensatz und 4.5 Mrd. Datensätzen nicht zu vernachlässigen und muss ein eingeplant werden. DB2 benötigt 4 Bytes für einen Integer, was bedeutet, dass bei einem Sort mindestens 72GB an Daten anfallen. Es wird nicht nur sortiert, sondern auch ein Distinct auf den Daten berechnet, d. h. mehrfache „Announces“ vom selben Peer im selben Swarm und Timebin fallen bei demselben Typ weg. Das sollte die Datenmenge zwar erheblich reduzieren, aber die Datenbank hat in der aktuellen Konfiguration nur ca. 19 GB Sortierspeicher, was bedeutet, dass die Daten zum Ende hin trotzdem noch ausgelagert werden müssen. Das reduziert, wie bereits beim „directed Join“ gezeigt, die Leseleistung auf bis zu 250 MB/s. Die restlichen Operatoren sind, bezogen auf die gesamte Ausführungszeit, vernachlässigbar. Die Schätzung der Ausführungszeit hängt darüber hinaus davon ab, wie viele gleiche Datensätze beim Distinct entfernt werden. Wenn überschlagen jeder zweite Datensatz doppelt ist, fallen im Distinct ca. 36 GB an. Nach der Berechnung von 19 GB, also ca. 2/3 der gruppierten Daten, muss ausgelagert werden. Demnach werden 2/3 der Daten mit ca. 900 MB/s gelesen und das restliche 1/3 mit ca. 250 MB/s. Dies ergibt somit ca. 1200 Sekunden für die ersten 2/3 und ca. 2300 Sekunden für das letzte Drittel und damit in Summe ca. 3500 Sekunden.

Bei der Ausführung mit Index ist eine Abschätzung der Laufzeit etwas einfacher, denn DB2 kann, da alle wichtigen Felder im Index stehen, einen „Index Only Lookup“ machen. Der Index müsste grob geschätzt ca. 100GB groß sein, was bedeutet, dass sich die Zeit, die zum Lesen der Daten benötigt wird, reduziert. Ein Indexscan hat zwar aufgrund der etwas komplexeren Struktur des Index nicht den Datendurchsatz eines Tablescans, jedoch wurden in dieser Arbeit zu diesem Durchsatz keine Daten gesammelt, weshalb hier ebenfalls eine Leserate von 900 MB/s angesetzt wird. In diesem Fall muss DB2, nach dem Sortieren des Index, Daten auslagern. Wenn man wie im vorigen Absatz annimmt, dass jeder zweite Datensatz wegfällt, wird auch hier nach ca. 2/3 der gelesenen Daten der Durchsatz reduziert werden. Damit werden 66GB mit 900 MB/s und 33GB mit 250 MB/s gelesen, was in Summe ca. 200 Sekunden macht. Da der Sortieroperator ca. 36 GB an temporären Daten erstellt, wird das Schreiben dieser Datenmenge zeitlich relevant. Bei einer Schreibbandbreite von ca. 100MB/s wie im „directed Join“ zu sehen, kommen nach dem Lesen der ersten 66GB, also nach ca. 75 Sekunden, weitere ca. 360 Sekunden zum Schreiben der temporären Daten dazu. Das Einlesen der Swarm und Timebin Tabellen in die entsprechenden Hashtabellen zum Joinen wurde bereits zu Beginn ausgeführt, jedoch aufgrund der geringen Größe vernachlässigt. Nun müssen die temporären ca. 36 GB wieder eingelesen werden und über den Gruppierungsoperator, der vernachlässigt werden kann, dem Joinoperatoren zugeführt werden. Dies wird aufgrund der Lesebandbreite nochmals ca. 40 Sekunden benötigen. Die

Laufzeit der Joins wird vernachlässigt. In Summe ergibt sich eine geschätzte Laufzeit von 475 Sekunden.

Für Hadoop ist in Abb. 3.10 der vereinfachte Ausführungsplan zu sehen. Aufgrund der geringen Tabellengröße verursachen die MapReduce Jobs 1 und 2 kaum zeitlichen Aufwand, sie können vernachlässigt werden. MR Job 3 muss im Mapper die komplette Announces Tabelle mit geschätzten 200 MB/s lesen, was ca. 7500 Sekunden dauert. Der Reducer benötigt zum Schreiben der ca. 36 GB nach dem Durchsatz beim „collocated Join“ von ca. 100 MB/s ca. 360 Sekunden. MR Job 4 benötigt zum erneuten Lesen dieser 36 GB kaum Zeit, da sich die Daten per Memorymapping bereits im RAM befinden. Der Reducer wird ebenfalls vernachlässigt. In Summe ergibt das ca. 8000 Sekunden für die Ausführung mit Hadoop.

4.3.2. Abschätzung für BitTorrent Job 2

Bei BT Job 2 aus Kapitel 3.5.7 wird wieder mit der Datenbank ohne Index angefangen und nach demselben Schema wie bei BT Job 1 vorgegangen. Zu Beginn würde die Datenbank unter Standardbedingungen die Peers Tabelle als „outer Table“ für den Join in die Hashtabelle laden. Da in diesem Fall für die weiteren Operationen der Anfrage nicht ausreichend temporärer Speicher zur Verfügung stand, musste der Optimizer entsprechend umkonfiguriert werden.Nun wird bei dieser Anfrage als erstes Announces verarbeitet und in die Hashtabelle geladen. Erst im nächsten Schritt wird Peers, nun als „inner Table“ dazu geladen. Diese Änderung sorgt zwar unter optimalen Bedingungen für eine verlängerte Ausführungszeit der Anfrage, hat aber in diesem Fall aufgrund der geringeren parallelen Lesebelastung der Festplatte jedoch positive Auswirkungen auf die Laufzeit.

Es wird nun also erst der Tablescan von Announces mit der Sortierung und Gruppierung ausgeführt. Der Sortieroperator benötigt auch hier mit mind. 56 GB mehr Platz als im Sortierspeicher zur Verfügung steht. Er muss die Felder Swarm, Peer und Timebin für jeden Datensatz zurück liefern. Hierdurch wird die Leserate also schon nach ca. einem Drittel einbrechen. Die Laufzeit bis zu dieser Operation beträgt also ca. 5200 Sek. Die Gruppierung wird ca. 60 Sekunden in Anspruch nehmen, da nur die bereits sortierten Ergebnisse eingeladen werden müssen. Auch für die Hashtabelle des Joins sollte der RAM zwar im Prinzip ausreichen, allerdings läuft deren Erstellung parallel zum Gruppieren, was bedeutet, dass voraussichtlich Teile der Tabelle ausgelagert werden müssen, was den Durchsatz wiederum senkt. Ab hier wird es schwer, Prognosen zu treffen, da unklar ist, wie schnell die Daten in diesem Fall von der Platte gelesen, bzw. auf sie geschrieben werden können und wie viele Daten ausgelagert werden müssen. Deshalb wird an dieser Stelle von mehr als 5200 Sekunden ausgegangen.

Bei der Ausführung mit Index ist das Lesen von Announces für die Datenbank nicht erforderlich, es stehen alle notwendigen Daten im Index. Da sich die Datenmenge jedoch erst nach der Gruppierung verringert, muss bereits während des Sortierens, nach dem Lesen von ungefähr der Hälfte des Index, ausgelagert werden. Es werden also ca. 50GB mit 900MB/s gelesen und 50GB mit 250MB/s. Das benötigt in Summe ca. 250 Sekunden. Das Gruppieren überlappt sich mit dem Sortiervorgang, weshalb hierfür keine weitere Zeit eingeplant wird. Wenn davon ausgegangen wird, dass jeder Peer im Durchschnitt in zwei Timebins auftaucht, entstehen für die Hashtabelle ca. 50GB an Daten. Anschließend erfolgen das Einlesen der Peers Tabelle in die Hashtabelle und ein weiterer Sortiervorgang. Da der Sortierspeicher bereits durch die Hashtabelle gefüllt ist, müssen die Ergebnisse des Joinvorgangs ebenfalls ausgelagert werden. Die Größe des Ergebnisses des Joinvorgangs ist stark von der Länge der Client-Zeichenkette abhängig. Bei der Annahme von 10 Bytes pro Zeichenkette ergibt sich eine Größe von ca. 50GB. Deshalb wird die Dauer des Joinvorgangs durch die Schreibgeschwindigkeit begrenzt und benötigt bei einer Bandbreite von ca. 100MB/s, wie sie im „directed Join“ Beispiel zu sehen ist, ca. 500 Sekunden. Bis zu diesem Zeitpunkt sind ca. 750 Sekunden vergangen. Für die Laufzeit der folgenden Sortierung und Gruppierung kann keine Schätzung abgegeben werden, da sie sehr stark von der Anzahl der unterschiedlichen Clientprogramme abhängt. Es wird also ein Dauer von über 750 Sekunden angesetzt.

Durch das Batchprocessing bei Hadoop ist dort die Berechnung einfacher. Im vereinfachten Ausführungsplan in Abb. 3.12 ist zu sehen, dass insgesamt 3 MR Jobs benötigt werden. Der erste Job liest die 1.5 TB großeAnnounces Tabelle im Mapper und benötigt dafür mit den gemessenen 200 MB/s ca. 7500 Sekunden. Der Reducer benötigt ungefähr 560 Sekunden zum Schreiben der ca. 56 GB an Teilergebnissen. Die Mapper des zweiten Jobs lesen diese 56 GB und die 25 GB große Peers Tabelle. Da sich die von den Mappern geschriebenen 56 GB noch im Cache befinden, addieren sich nur ca. 130 Sekunden für Peers. Die Ergebnisse des Joins werden unter 25 GB liegen, das Schreiben wird also maximal 250 Sekunden dauern. Der letzte Job kann vernachlässigt werden, da sich bis dahin alle Zwischenergebnisse im RAM befinden. Die Laufzeit beträgt also geschätzt 8500 Sekunden.

Tabelle 4.2 zeigt die geschätzten Ausführungszeiten.

Job	Dauer DB2 ohne Index	Dauer DB2 mit Index	Dauer Hadoop
Laden eines Timebin	ca. 20 s	ca. 25 s	ca. 15 s
Job 1	ca. 3500 s	ca. 475 s	ca. 8000 s
Job 2	> 5200 s	> 750 s	ca. 8500 s

Tabelle 4.2.: Schätzungen der Ausführungszeiten

4.4. Tests mit den BitTorrent Daten

Diese Tests wurde auf der Datenbank sowohl mit als auch ohne Index ausgeführt. Wie im vorhergehenden Abschnitt wird mit DIR auf den DB2 Report mit Index verwiesen, mit DNR auf den DB2 Report ohne Index und mit HDR auf den Hadoop Report. APD bezeichnet den Ausführungsplan von DB2, APP den Ausführungsplan von Pig und MRT das MapReduce Taskscheduling.

4.4.1. Laden der Daten

Da ein einzelner Timebin innerhalb von wenigen Sekunden geladen wird, sind in den folgenden Graphen einige Artefakte zu sehen, die nichts mit dem eigentlichen Ladeprozess zu tun haben und bisher in der Datenmenge untergegangen sind. Der APD, APP und der MRT existieren beim Laden nicht, der DNR wurde nicht aufgenommen.

DIR: Abb. 4.20 auf Seite 83
HDR: Abb. 4.21 auf Seite 84

Im DIR ist bereits zu Beginn bei ca. 9-15 Sekunden im Avg CPU/MEM Graphen zu erkennen, dass die CPU viel Zeit für Systemcalls benötigt (CPU sys), gleichzeitig erhöht sich der Speicherverbrauch drastisch. Diese Zeitspanne sollte im Grunde genommen nicht mit eingerechnet werden, sie wird von der Datenbank zum Starten benötigt. Am abfallenden Speicherverbrauch bei ca. 55 Sekunden wird das Herunterfahren der Datenbank sichtbar. Nach dem Start der Datenbank vergehen noch ca. 15-20 Sekunden bis, wie im Transfered Data Graphen zu sehen, Master damit beginnt die Daten an Node1 - Node4 zu übertragen. In dieser Zeit werden die Daten nach dem Verteilungsschema in der Tabellenbeschreibung auf Master vorsortiert. Die eigentliche Übertragung und das Schreiben der Daten nimmt dann nur noch ca. 15 Sekunden in Anspruch. Bei ca. 53-54 Sek. scheint dem Sum Disc IO Graphen nach der Index geschrieben zu werden. Der Bandwidth und Packets Graph zeigt, dass mit einer Bandbreite von knapp über 100 MB/s das Gigabit Netzwerk von Master fast komplett ausgenutzt wird. Da Node1 - Node4 zeitgleich ebenfalls mit ca. 80 MB/s untereinander kommunizieren, kommt es in diesem Zeitraum vermehrt zu Übertragungsfehlern, wie der Packetloss Graph zeigt.

Hadoop geht beim Laden effizienter vor. Da Node1 - Node4 hier nicht miteinander kommunizieren müssen, kann Master mit der ihm gegebenen maximalen Bandbreite von ca. 118 MB/s übertragen. Der Sum TCP Connections Graph zeigt, dass hierfür insgesamt 22 TCP Verbindungen benötigt werden. Die Daten auf Node1 - Node4 werden aufgrund ihrer geringen Größe innerhalb der gemessenen Zeit nicht auf die Festplatte geschrieben. Durch das Memorymapping der Datein kann dies durch das Betriebsystem zu einem späteren Zeitpunkt erfolgen.

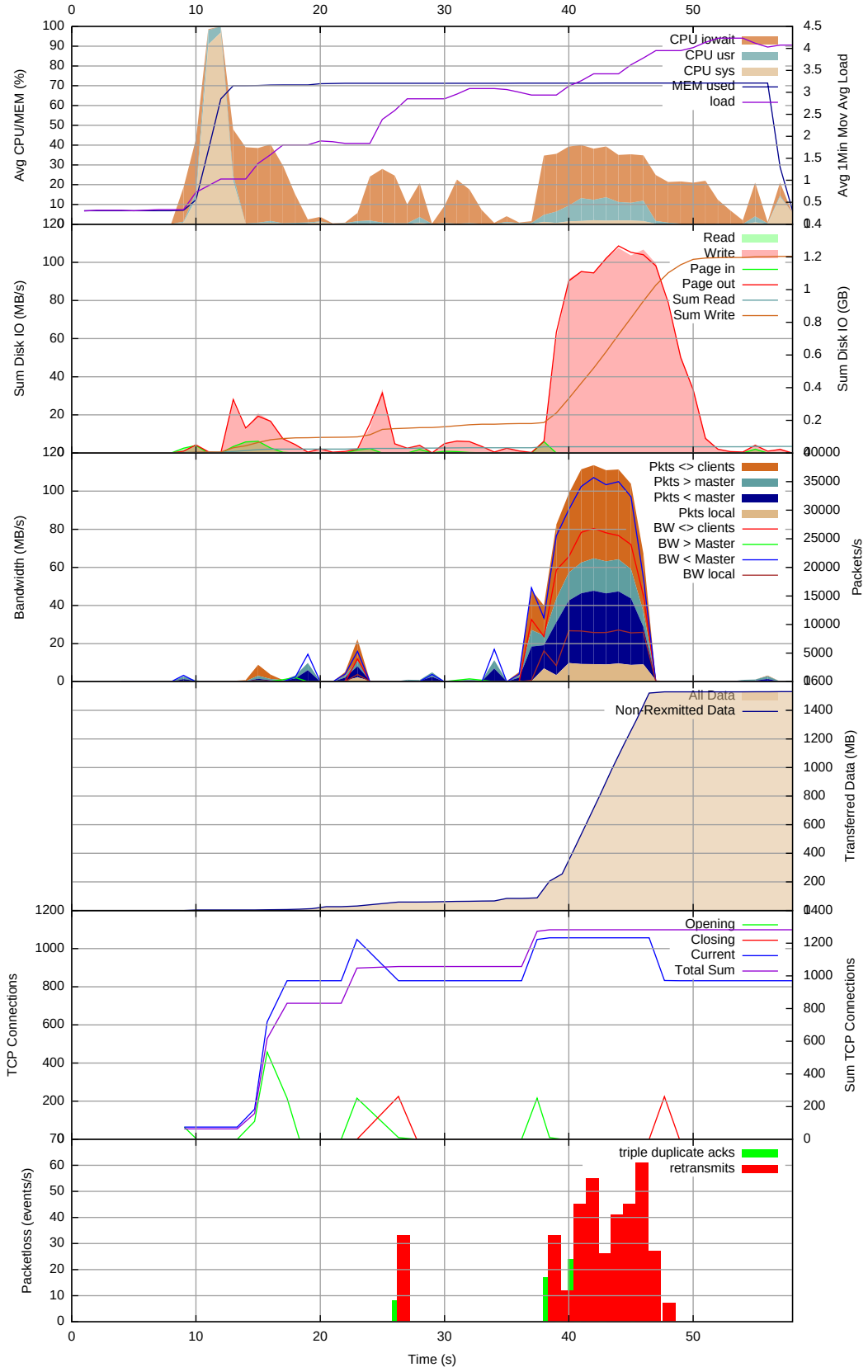

Abbildung 4.20.: DB2 BitTorrent Load mit Index Report

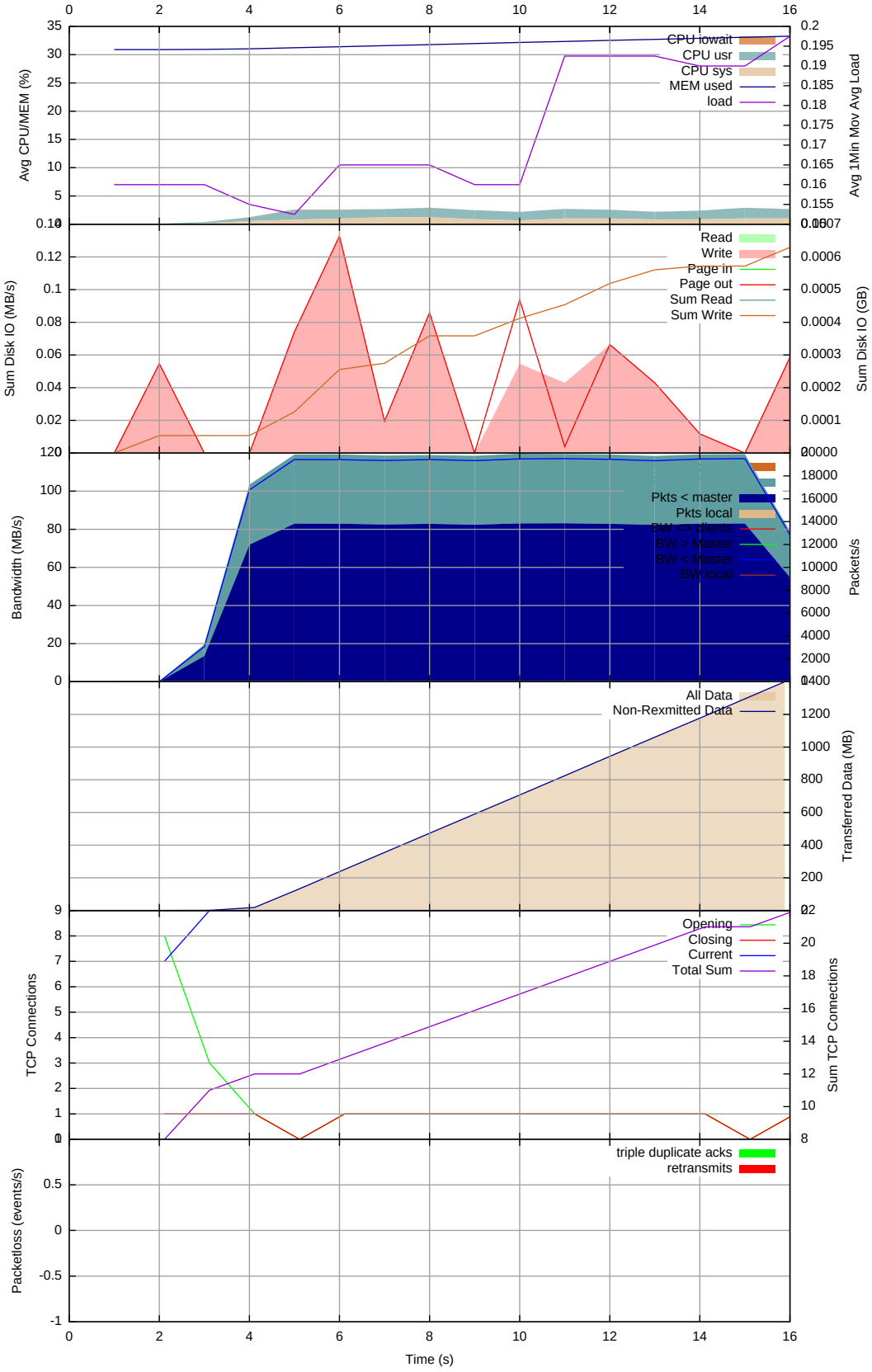

Abbildung 4.21.: Hadoop BitTorrent Load Report

4.4.2. BitTorrent Job 1

Die Messergebnisse für BT Job 1 aus Kapitel 3.5.6 sind auf folgenden Abbildungen zu finden:

APD ohne Index:	Abb. D.26 auf Seite xxxiii
APD mit Index:	Abb. D.24 auf Seite xxx
DNR:	Abb. 4.22 auf Seite 86
DIR:	Abb. 4.23 auf Seite 87
APP	Abb. D.28 auf Seite xxxvi
HDR:	Abb. 4.24 auf Seite 88
MRT	Abb. D.30 auf Seite xxxviii

Nach dem Graphen im DNR entspricht die Schätzung in Kapitel 4.3.1 dem Ablauf dieser Anfrage, weshalb die Interpretation an dieser Stelle nicht wiederholt wird. Allerdings ist die nach dem „Distinct“ erzeugte Datenmenge geringer als angenommen, weshalb sich die Laufzeit der Anfrage in der Realität reduziert.

Dagegen weichen die Ergebnisse der Anfrage mit Index stark von den geschätzten Werten ab. Es zeigt sich im DIR zwar das erwartete Verhalten, aber im Sum Disc IO Graphen ist zu beobachten, dass der Index, im Gegensatz zu den geschätzten 900 MB/s, nur mit ca. 150 MB/s gelesen wird, wodurch sich die Lesedauer verlängert. Die nach dem Sortiervorgang erstellten temporären Daten werden in dem Zeitraum von ca. 650 Sek. bis 820 Sek ebenfalls mit einer recht geringen Bandbreite von nur ca. 200 MB/s gelesen. Auch dieser Wert liegt deutlich unter dem der Schätzung.

Hadoop geht, nach dem HDR zu urteilen, mit den Zwischenergebnissen der Mapper auf den Timebins und Swarms Tabellen weniger effizient um. Der Transfered Data Graph zeigt in den ersten 100 Sekunden, dem MRT nach die Dauer der ersten beiden MR Jobs, die Übertragung von ca. 12 GB an Daten. Ebenfalls bemerkenswert ist, dass die Lesebandbreite im Sum Disc IO Graphen nun ca. 250 MB/s anstatt der bisher gemessenen ca. 200 MB/s erreicht. Dies bewirkt, dass sich die Ausführungszeit von den geschätzten 8000 Sekunden auf ca. 7500 Sekunden reduziert. Hadoop öffnet zum Übertragen der Daten während der Anfrage insgesamt etwas über 151.000 TCP Verbindungen.

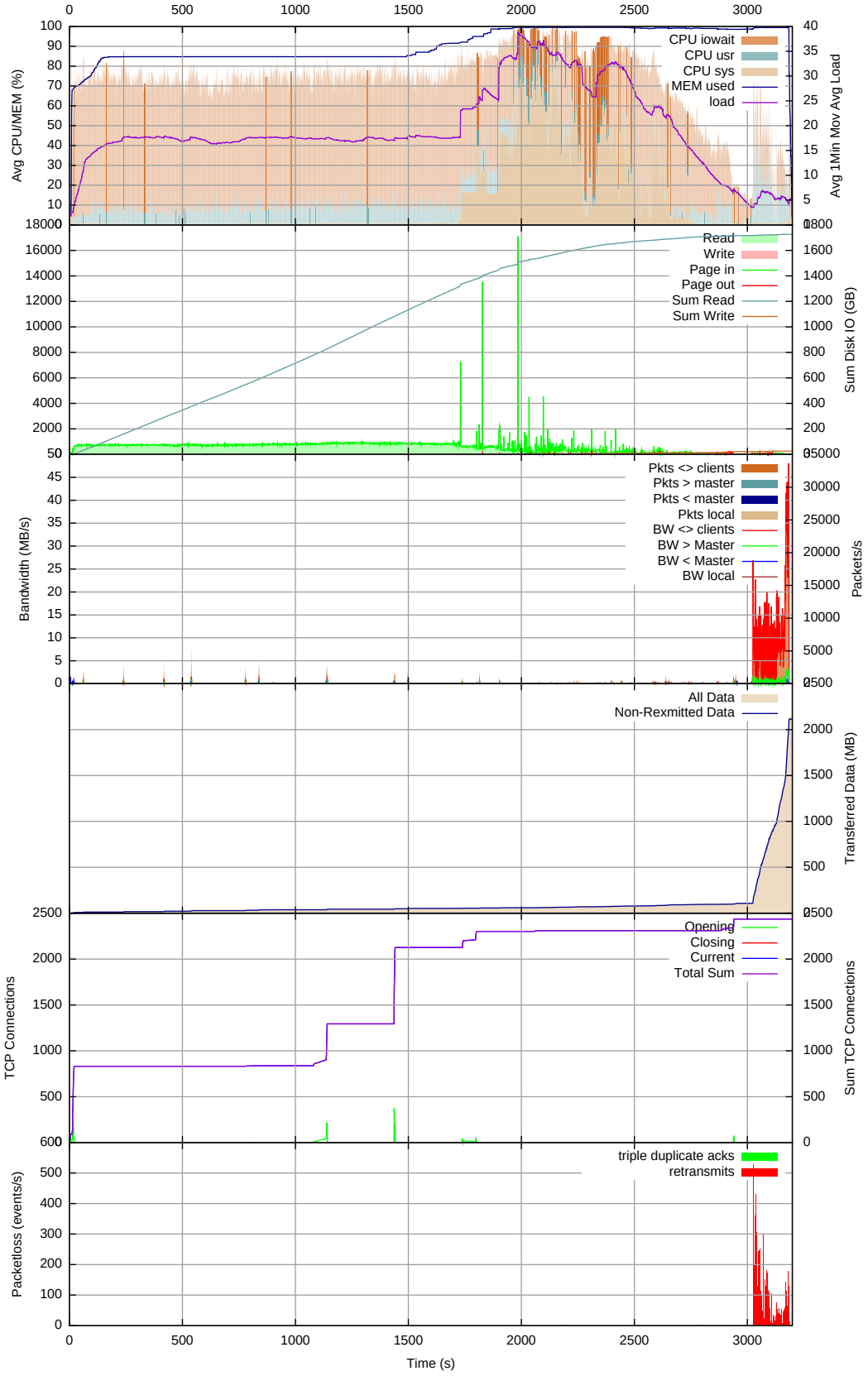

Abbildung 4.22.: DB2 BitTorrent Job1 ohne Index Report

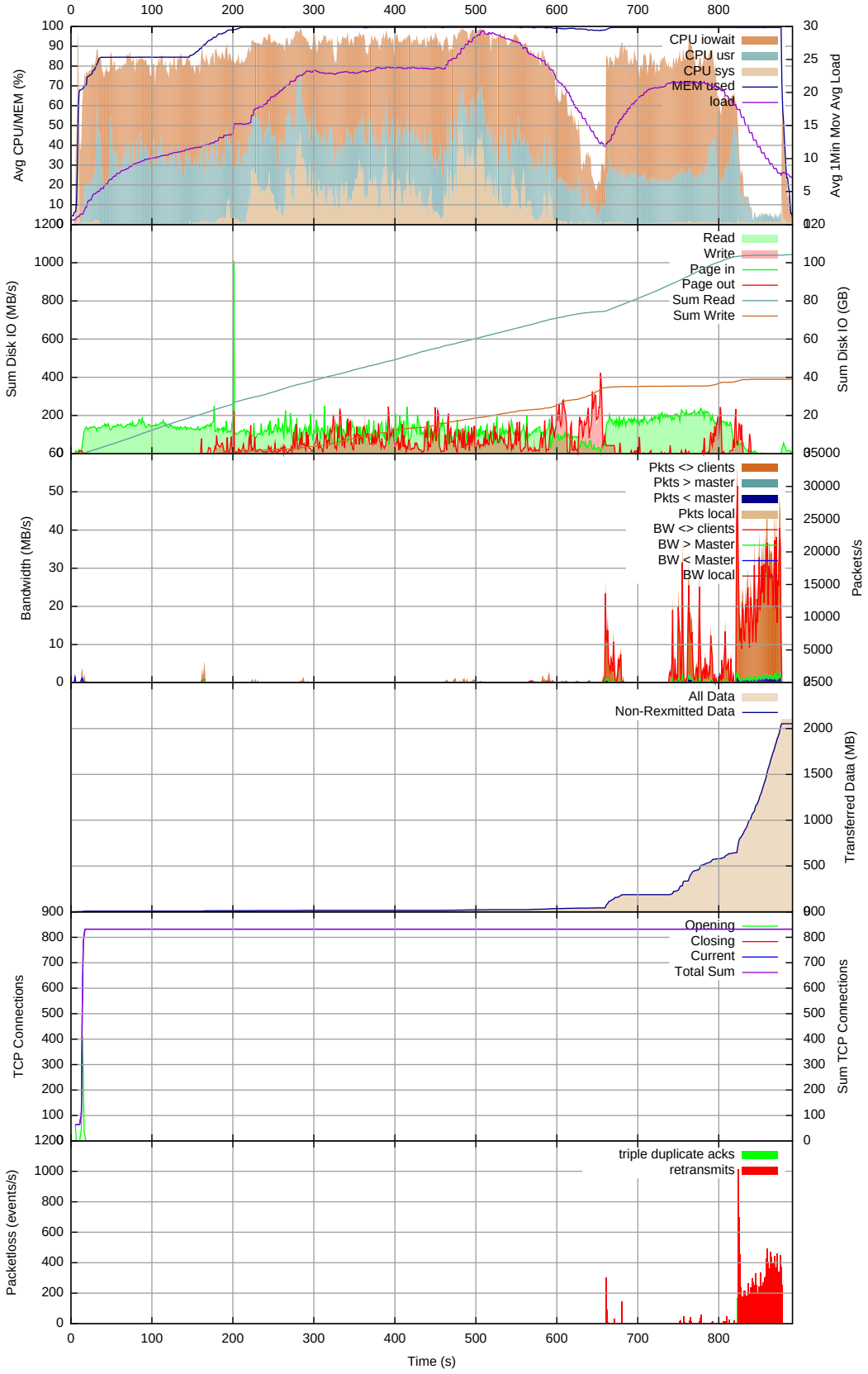

Abbildung 4.23.: DB2 BitTorrent Job1 mit Index Report

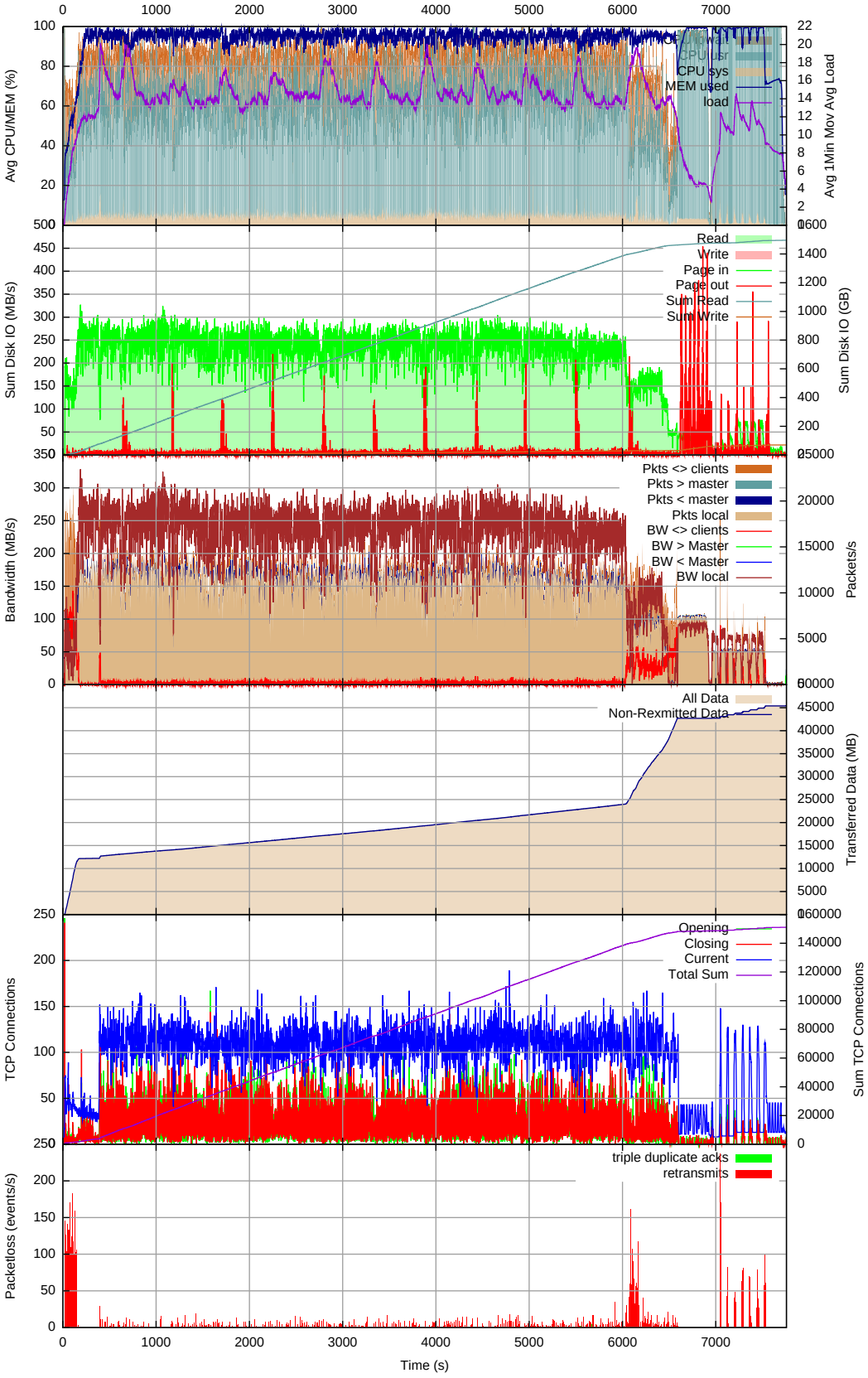

Abbildung 4.24.: Hadoop BitTorrent Job1 Report

4.4.3. BitTorrent Job 2

Die Messergebnisse für BT Job 2 aus Kapitel 3.5.7 sind auf folgenden Abbildungen zu finden:

APD ohne Index:	Abb. D.27 auf Seite xxxiv
APD mit Index:	Abb. D.25 auf Seite xxxi
DNR:	Abb. 4.25 auf Seite 90
DIR:	Abb. 4.26 auf Seite 91
APP	Abb. D.29 auf Seite xxxvii
HDR:	Abb. 4.27 auf Seite 92
MRT	Abb. D.31 auf Seite xxxviii

Der DNR von Job 2 wird weitesgehend durch die Schätzung in Kapitel 4.3.2 beschrieben. Allerdings musste erst nach ca. der Hälfte der gelesenen Daten der Announces Tabelle ausgelagert werden, wodurch sich die Laufzeit der Anfrage verkürzte. In der Lesebandbreite im Sum Disc IO Graphen sind recht hohe Peaks zu sehen, die durch Cachingeffekte entstehen.

Der DIR zeigt eine höhere Laufzeit als erwartet. Die Index- und temporären Daten werden, wie bei Job 1, nicht mit den angenommenen 900MB/s, sondern nur mit ca. 150MB/s gelesen, und die Ergebnisse des Joinvorgangs mit nur ca. 50MB/s geschrieben. Zuzüglich der Dauer der anschließenden Gruppierung, die nicht abgeschätzt werden konnte, ergibt sich eine mehr als doppelt so lange Laufzeit der Anfrage wie geschätzt.

Da Hadoop im Vergleich zu DB2 in dieser Konfiguration eher CPU-lastig ist, lässt sich die Zeit mittels der Bandbreiten zum Lesen und Schreiben der Daten nicht ganz so gut vorhersagen. Die Schätzung mit ca. 8500 Sekunden liegt recht nah am Ergebnis von ca. 9100 Sekunden, die Differenz sind Aufwände, die durch benötigte Rechenzeit zu erklären sind. Hadoop überträgt bei diesem Job ca. 60GB an Daten zwischen den Knoten und öffnet und schließt dafür ca. 155.000 TCP Verbindungen.

In Abb. 4.28 ist eine Verteilung der CPU Ressourcen bei der Ausführung des BitTorrent Job 2 zu sehen. Der Graph steht symbolisch für alle Anfragen, da er bei allen durchgeführten Tests ein vergleichbares Bild liefert. Er stellt dar, wie viel Zeit die CPU im Verhältnis für die Abarbeitung von Interrupts (irq), Systemcalls (sys), Userprozesse (usr) sowie das Warten auf I/O (iowait) benötigt. Die Idletime wurde in diesem Diagram nicht mit dargestellt.

Deutlich ist zu erkennen, dass für die Datenbank auf dem gegebenen Cluster die Plattenzugriffe den limitierenden Faktor darstellen, während es bei Hadoop eher die CPU ist. Hadoop benötigt im Verhältnis zur Datenbank für dieselbe Aufgabe bedeutend mehr CPU-Ressourcen.

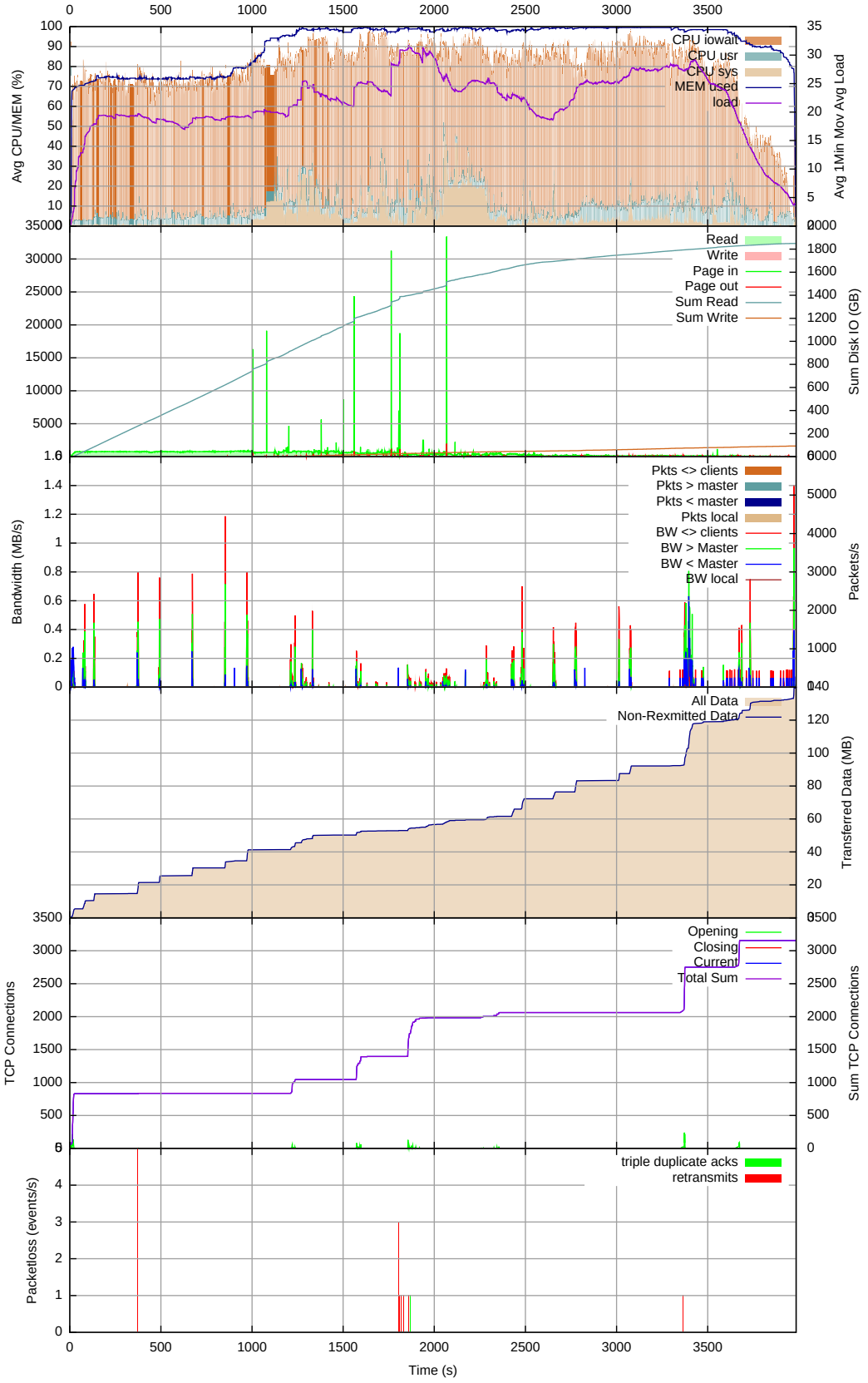

Abbildung 4.25.: DB2 BitTorrent Job2 ohne Index Report

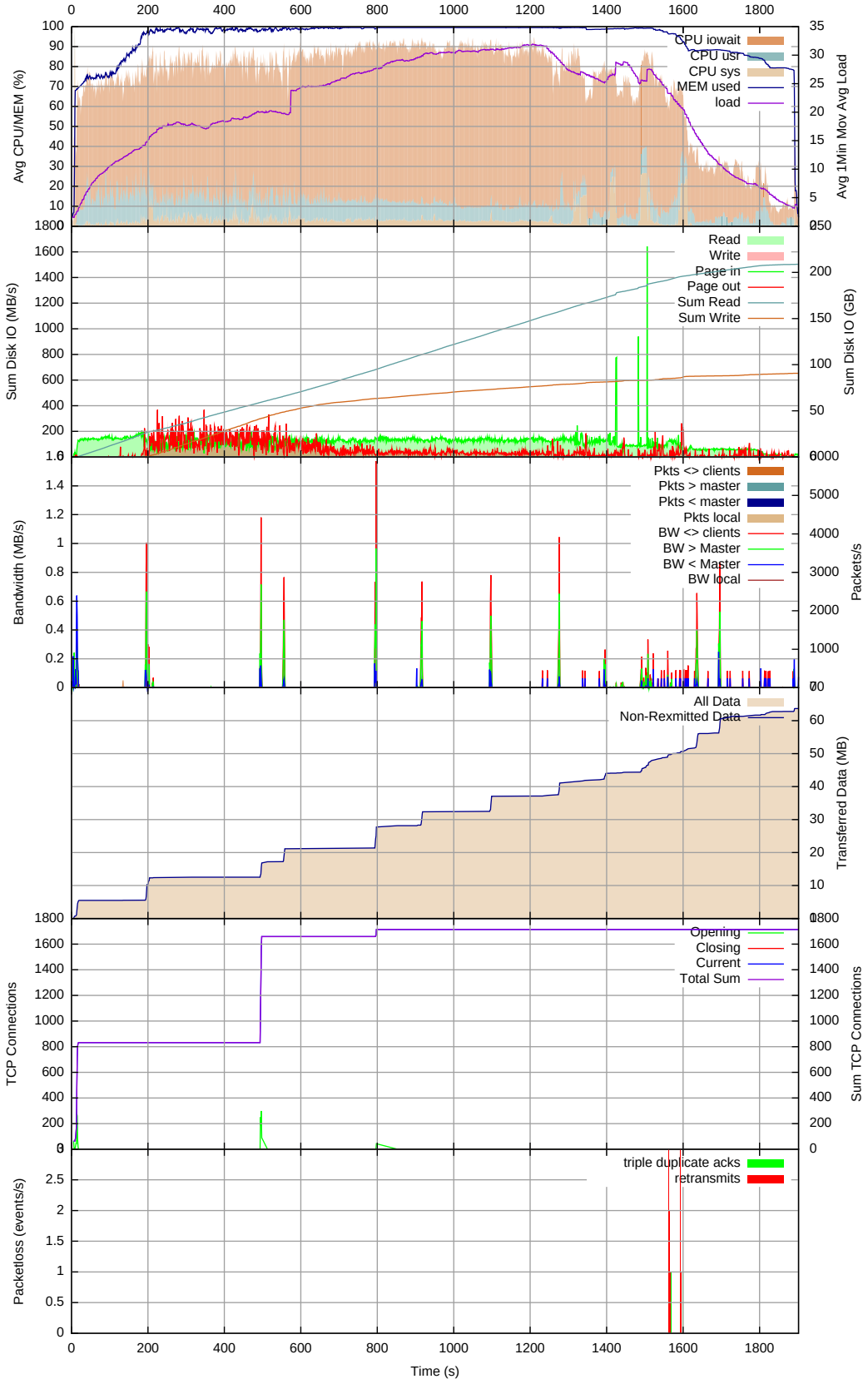

Abbildung 4.26.: DB2 BitTorrent Job2 mit Index Report

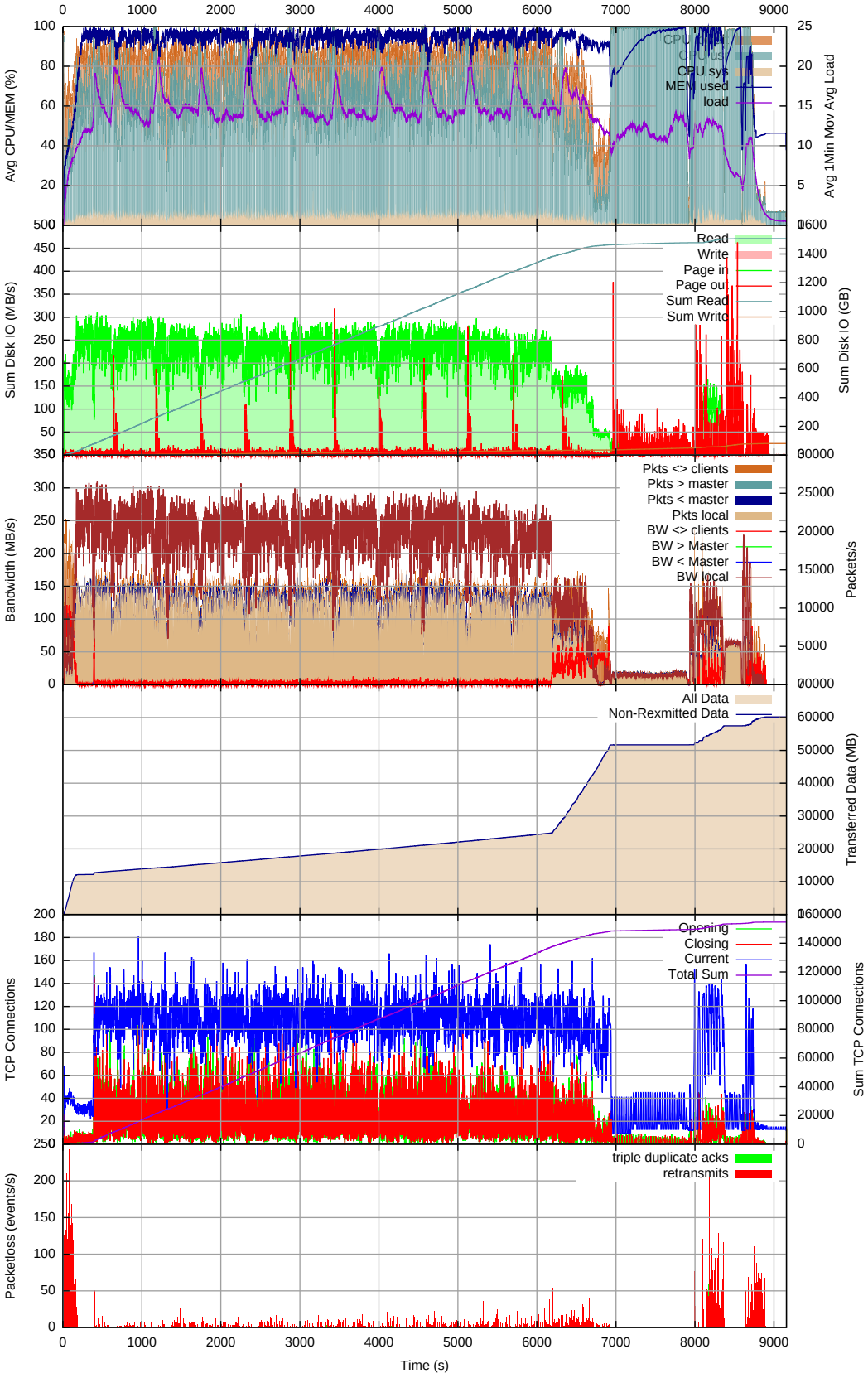

Abbildung 4.27.: Hadoop BitTorrent Job2 Report

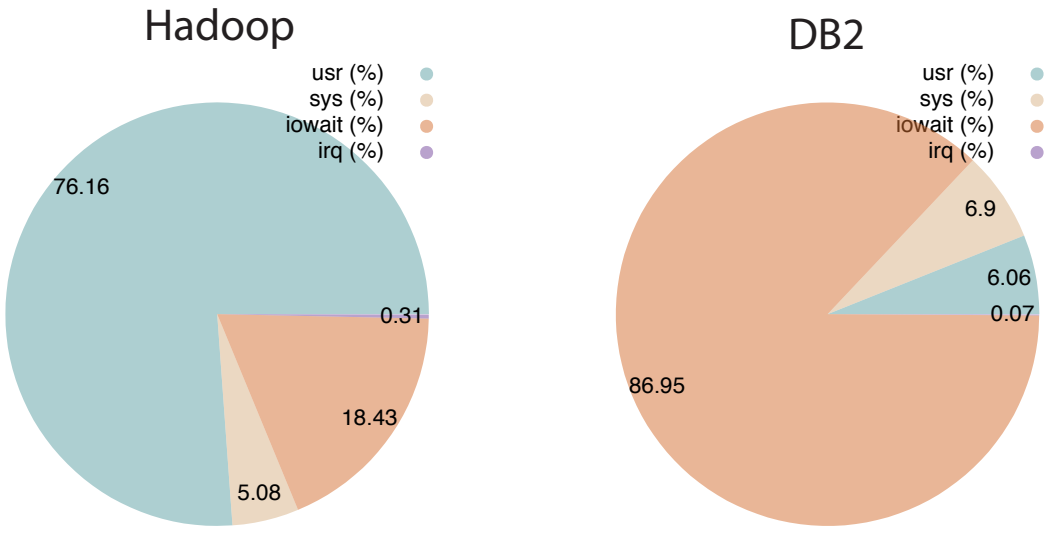

Abbildung 4.28.: Verteilung der CPU Resourcen BitTorrent Job 2

4.5. Verifikation der Prognosen

In Abb. 4.29 ist ein Überblick über die geschätzten und gemessenen Laufzeiten der Tests mit den BitTorrent Daten zu sehen. Bei den gemessenen Werten sind sowohl der Mittelwert als auch Minimal- und Maximalwerte dargestellt. Zu beachten ist, dass aus Skalierungsgründen bei den Ladezeiten die benötigten Zeiten zum Laden eines Timebins mit dem Faktor 100 multipliziert dargestellt sind. Dadurch sind die Minimal- und Maximalwerte recht weit vom Mittelwert entfernt, was nicht der Fall wäre, wenn die benötigte Zeit zum Laden von 100 Timebins dargestellt wäre.

Während der Schätzfehler der Anfragen bei Hadoop im Bereich von ca. 10% liegt, sind die Fehler der Schätzung bei der Datenbank deutlich höher. Dies ist zum einen darauf zurückzuführen, dass die Datenbank viele Operationen parallelisiert, wodurch sich schwer prognostizieren lässt, wann die Datenbank Daten in den temporären Speicher auslagert und wie sich dabei die Lese- und Schreibbandbreiten verhalten und zum anderen daran, dass die Bandbreiten für den Index-Only-Zugriff aufgrund fehlender Daten falsch eingeschätzt wurden.

Bei Hadoop liegt die Schätzung der Ladezeit recht nah am gemessenen Resultat, aber die Ladezeiten der Datenbank liegen deutlich über den geschätzten Werten. Die benötigte Zeit für das Vorsortieren der Daten auf Master wurde beim Schätzen vernachlässigt. Der Ladeagent benötigt, nach den Messungen der Ladezeiten der BitTorrent Daten zu urteilen, ähnlich lange zum Einladen und Sortieren der Daten, wie Node1 - Node4 im nächsten Schritt zum Schreiben. Da die jeweils pro Timebin geschriebene Datenmenge relativ gering ist und die Puffer zum Vorsortieren recht groß sind, kann der Ladeagent das Sortieren und Schreiben an Node1 - Node4 in diesem Test nicht parallelisieren. Auch das Starten der Datenbank fällt

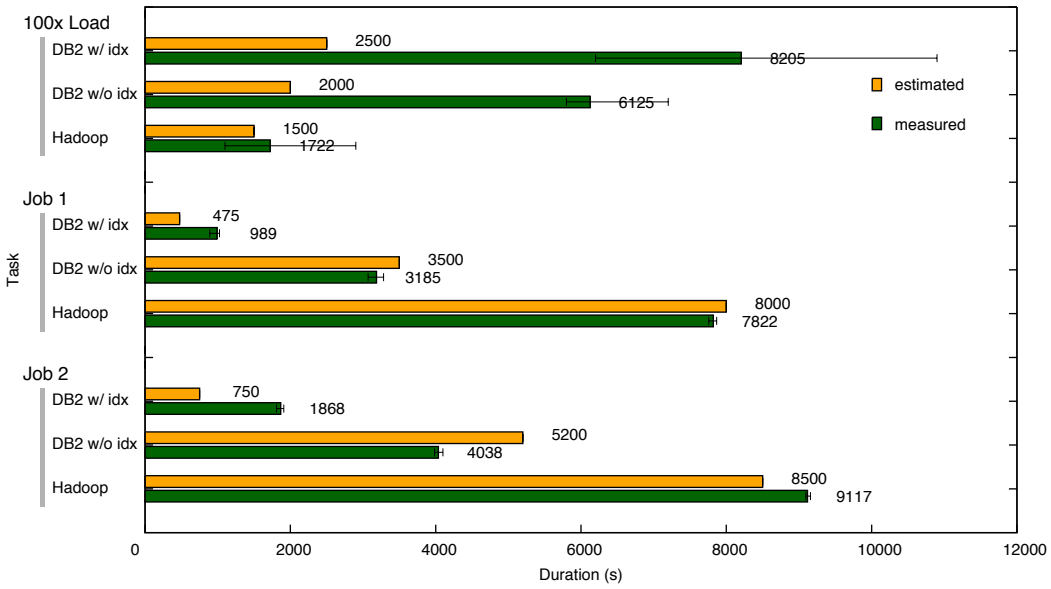

Abbildung 4.29.: Laufzeiten der BitTorrent Tests

bei so geringen Zeiträumen messbar ins Gewicht. Subtrahiert man die Zeit von ca. 17 Sekunden, die zum Starten und Stoppen der Datenbank benötigt wird von den gemessenen Werten, beträgt die Zeit des Ladevorgangs ohne Index auf der Datenbank im Durchschnitt nur noch ca. 49 Sekunden pro Timebin. Plant man nun bei der Schätzung den großen Sortierpuffer auf Master ein und nimmt an, dass das Schreiben der Daten auf Node1 - Node4 erst nach dem Lesen und Vorsortieren erfolgt, dass ungefähr dieselbe Zeit benötigt, ergibt die neue Schätzung rund 40 Sekunden. Damit liegt der Fehler zwar immer noch bei ca. 20%-25% was aber weit besser als die vorhergehenden ca. 300% ist.

Interessant wäre ein Vergleich der Ausführungszeiten der Jobs mit den Originalanfragen von Krenc [Kre12] gewesen. Die Anfragen wurden dort allerdings semiautomatisch durchgeführt und können deshalb nicht vergleichen werden.

5. Zusammenfassung

Ziel der vorliegenden Arbeit war es, die beiden folgenden Fragen zu untersuchen:

1. Eignet sich besser eine verteilte relationale Datenbank oder ein MapReduce Framework angesichts der spezifischen Eigenschaften der Systeme zur Auswertung von Netzwerkstatistiken?
2. Wie sieht der Netzwerkverkehr zwischen den Knoten in einem entsprechenden Cluster aus, und was muss man deshalb gegebenenfalls beim Skalieren beachten?

Zu diesem Zweck wurde ein Versuch durchgeführt, in dem die Anwendbarkeit beider Techniken in kleinem Maßstab getestet wurde. Als Datenbank wurde DB2 in einer Version für verteilte Systeme eingesetzt, als MapReduce Framework wurde Hadoop genutzt.

Während dieser Tests wurden Statistiken über die Auslastung der verschiedenen Subsysteme angelegt, um im Nachhinein die Ressourcennutzung vergleichen zu können. Im ersten Teil des Versuches wurden diese Statistiken für Basisfunktionen der Datenverarbeitung sowohl auf einer verteilten Datenbank als auch in einem MapReduce Framework erstellt. Mittels dieser Daten wurde das Verhalten der „echten“ Anfragen prognostiziert, um im anschließenden Vergleich mit den gemessenen Werten ein Verständnis über die Aussagekraft dieser Statistiken zu erlangen. Es zeigte im Verlaufe der Tests, dass sich die Datenbank und MapReduce unterschiedlich verhielten.

Aufgrund der Parallelisierung der Operationen in der Datenbank war die Prognose der Ausführungszeiten einer Anfrage eine sehr komplexe Aufgabe. Da die Datenbank in den Tests stark I/O gebunden war, konnten hierfür, sofern alle notwendigen Daten für eine Prognose vorhanden waren, relativ genaue Vorhersagen erzielt werden. Hadoop zeigte sich in den Tests eher CPU-lastig, was die Prognose erschwerte. Allerdings ließen sich durch das einfache Batchverhalten von Hadoop ebenfalls recht gute Vorhersagen treffen. Die Qualität der Vorhersagen spricht für ein stabiles Verhalten der Systeme, wodurch die Aussagekraft der gemessenen Ergebnisse unterstützt wird.

Die Frage, welche der beiden Systeme sich besser zur Auswertung von Netzwerkstatistiken eignet, lässt sich allerdings nicht eindeutig beantworten. Sowohl für die Datenbank als auch MapReduce lassen sich Vor- und Nachteile aufzeigen.

Hinsichtlich der Ausführungszeit der Anfragen zeigte sich die Datenbank der MapReduce Lösung überlegen. Die Datenbank bietet höhere Lese- und Schreibbandbreiten sowie eine

Parallelisierung der einzelnen Operatoren innerhalb der Anfrage. Die Zwischenergebnisse werden, um Zeit zu sparen, durch Pipelining automatisch zum jeweils nächsten Operator weiter geleitet, und durch Kenntnis der Datenverteilung und Statistiken wird der Anfrageplan auf bestmögliche Ressourcenausnutzung optimiert. Bei der Datenbank ist zudem jede Partition schon zu Beginn einer Anfrage darüber informiert, zu welchem Zeitpunkt welche Operation ausgeführt werden muss und Indizes und vorberechnete Teilergebnisse helfen, die nötigen I/O-Operationen zu reduzieren.

Andererseits konnte dargestellt werden, dass Hadoop im Vergleich zur Datenbank nur sehr geringe Ladezeiten benötigt, die sich besonders bei großen Datenmengen bemerkbar machen. Wenn nur wenige Anfragen auf einer Datensammlung ausgeführt werden müssen, ist häufig Hadoop die bessere Wahl, da die Datenbank teilweise ein Vielfaches der Zeit einer Anfrage zum Einspielen benötigt. Falls jedoch zu Beginn einer Auswertung noch unklar ist, welche Daten ausgewertet werden oder in den Daten erst nach Zusammenhängen gesucht werden muss, bietet sich aufgrund der geringen Ausführungszeiten der Anfragen eher die Datenbank an.

Wenn die Anfragen nicht in Pig sondern in Java entwickelt worden wären, könnte ein Experte typischer Weise bis zu 10% der Laufzeit einsparen. Eine Prämisse dieser Arbeit war jedoch, dass die Nutzer mit diesem System nur ihre Daten auswerten. Es ist deshalb nicht zu erwarten, dass sie Experten auf dem Gebiet der MapReduce Programme in Java sind. Eine durch diese Nutzer erstellte Umsetzung der Anfragen in Java wäre voraussichtlich sogar langsamer als die Version in Pig. Zusätzlich ist der Programmieraufwand durch komplexeren Code in Java um einen Faktor von ca. 10-20 höher als in Pig.

Weitere Faktoren, die die Entscheidung für eine verteilte Datenbank oder ein MapReduce Framework beeinflussen können, sind z. B. Lizenzkosten oder Administrationsaufwand. Während Hadoop im Gegensatz zu DB2 als OpenSource kostenlos zur Verfügung steht und einfach zu installieren und zu warten ist, fallen für DB2, in der eingesetzten „Enterprise Server Edition“, die für das verteilte Arbeiten notwendig ist, hohe Lizenzgebühren an. Zudem sind, aufgrund der Komplexität der Software, sowohl zum Aufsetzen als auch zum Betrieb weitreichende Vorkenntnisse nötig.

Der wichtigste Faktor, wenn es um BigData Auswertungen geht, ist die Größe der Daten. Denn der eigentliche Grund für die Entwicklung von Hadoop war nicht die Performance der Anfragen, sondern eine Möglichkeit zu schaffen, Anfragen möglichst einfach auch über tausende von Rechnern zu verteilen. Bei Clustern in dieser Größenordnung ergibt sich ein völlig neues Problem, dass bei kleinen Clustern nicht ins Gewicht fällt: die Verfügbarkeit der Hardware.

Die Datenbank ist im Gegensatz zu MapReduce darauf angewiesen, dass zur Ausführung einer Anfrage alle Knoten zur Verfügung stehen. Wenn man z. B. einen Cluster mit 1000 Knoten annimmt und von qualitativ sehr guter Hardware mit einer Verfügbarkeit von 99,9%

ausgeht, bedeutet das für jeden Knoten nur eine Ausfallzeit von max. 44min im Monat. Die Gesamtverfügbarkeit in einem System, bei dem jedes Teil verfügbar sein muss, entspricht dem Produkt aller Einzelverfügbarkeiten. Bei 1000 Knoten entspricht das einer Gesamtverfügbarkeit von ca. 36,8% und damit einer Ausfallzeit von ca. 19 Tagen im Monat.

Hadoop bietet genau hierfür eine Lösung, indem jeder Datenblock automatisch mehrfach auf verschiedene Knoten repliziert werden kann. Weiterhin kann jeder Map- oder Reducetask seine Daten von jedem beliebigen anderen Knoten laden. Der Ausfall eines Knotens würde zwar bewirken, dass im selben Moment auf diesem Knoten ausgeführte Map- oder Reducetasks nochmals auf einem anderen Knoten ausgeführt werden müssten, die Anfrage an sich würde jedoch nicht unterbrochen.

Welches der beiden Systeme besser geeignet ist, kommt also auf die genauen Anforderungen an. Für den Einsatz im FG INET könnte Hadoop interessant sein, denn auch wenn die Anfragen etwas mehr Zeit benötigen, lässt sich hier mit wenig administrativem Aufwand ein System etablieren, dass von den Anwendern ohne viel Vorwissen genutzt werden kann.

Hinsichtlich der Auffälligkeiten im Netzwerkverkehr zwischen den Knoten konnte gezeigt werden, dass dieser bei DB2 besonders stark minimiert wurde. Während bei DB2 versucht wurde, nicht nur die Menge der zu transferierenden Daten, sondern auch die Anzahl der genutzten TCP-Verbindungen so gering wie möglich zu halten, indem bereits bestehende Verbindungen wiederverwendet werden, geht Hadoop sehr großzügig mit den Verbindungen um. Hier wird für nahezu jeden zu übertragenden Datenblock eine neue Verbindung geöffnet. Da Hadoop keine Informationen über die Verteilung der Daten besitzt, muss gerade bei Join-Operatoren ein Vielfaches der Datenmenge, die bei DB2 notwendig wäre, über das Netzwerk übertragen werden. Hier böte sich auch bei Hadoop an, den Netzwerkverkehr und die Latenzzeiten zu reduzieren, indem bereits bestehende Verbindungen wiederverwendet werden.

Ein weiteres auffälliges Merkmal in der Kommunikation der Knoten bei Hadoop ist der Zugriff der MapReduce Jobs auf das HDFS. Um aus Abstraktions- und Redundanzgründen die Unabhängigkeit der Lage der Daten im Verhältnis zum ausführenden Prozess zu gewährleisten, ist der Zugriff auf Daten im HDFS über Netzwerksockets geregelt. Aufgrund der Arbeitsweise von Hadoop werden die Daten soweit möglich auf demselben Knoten verarbeitet, auf dem sie jeweils gespeichert sind. Der lokale Zugriff auf diese Daten über das Loopbackdevice, im Vergleich zum direkten Zugriff auf die Festplatte wie bei der Datenbank, reduzierte die Zugriffsbandbreite auf die Festplatte erheblich. Hier liegt das Potential zu einer enormen Geschwindigkeitssteigerung von Hadoop im Bereich der Anfragen, indem in den Fällen, in denen die Daten lokal liegen, auch direkt auf die Daten zugegriffen wird. Eine weitere Einschränkung könnte die I/O Implementierung in Java sein, die in dieser Arbeit nicht getestet wurde.

Der Testcluster in dieser Arbeit war, mit fünf Maschinen im Verhältnis zu der möglichen Größe von Clustern mit den eingesetzten Systemen von teilweise mehreren tausend Maschinen, relativ klein. So ist es mit den gemessenen Statistiken nicht möglich, über die Probleme, die bei einer Skalierung der beiden Systeme auftreten können, eindeutige Vorhersagen zu machen. Einzig auffällig ist das relativ hohe Übertragungsvolumen bei Hadoop, bzw. auch die recht hohe Anzahl von TCP-Verbindungen. Man sollte sich hier bei größeren Clustern an die Empfehlungen der Entwickler halten und das Feature der „Rack Awareness“ nutzen. So kann dafür gesorgt werden, dass ein Großteil des Netzwerkverkehrs lokal innerhalb des jeweiligen Racks bleibt und damit der Verkehr zwischen den Racks reduziert wird. Zur weitergehenden Erforschung des Themas empfehle ich daher in Zukunft weitere Überprüfungen im größeren Rahmen und auch im Vergleich mit unterschiedlichen Clustergrößen durchzuführen.

Abschließend kann festgestellt werden, dass ein MapReduce Framework aufgrund seiner Architektur eine Anfrage auf strukturierten Daten niemals schneller beantworten kann als eine korrekt konfigurierte Datenbank auf der selben Hardware, dafür aber kürzere Ladezeiten und einfachere Verwaltung bietet. Andererseits sind der Skalierung und Verfügbarkeit einer verteilten Datenbank Grenzen gesetzt, was sie ab einer gewissen Größenordnung ausschließt.

A. Konfigurationen

A.1. DB2

Abweichende Konfiguration zur Standard Installation.

```
#!/bin/bash
#
db2 update dbm cfg using INTRA_PARALLEL YES
```

Listing A.1: config_dbm.sh

```
#!/bin/bash

db2 "create db bench automatic storage yes on /mnt/db2"

db2 "connect to bench"

db2 "create database partition group ALL on all dbpartitionnums"
db2 "create database partition group MASTER on dbpartitionnum(0)"
db2 "create database partition group NODESONLY on dbpartitionnums(1,2,3,4,5,6,7,8,
     9,10,11,12,13,14,15,16, 17,18,19,20,21,22,23,24, 25,26,27,28,29,30,31,32)"
db2 "alter bufferpool IBMDEFAULTBP size 30000" # ~100MB
db2 "create bufferpool bp32K size 35000 pagesize 32K" # ~ 1.1 GB
db2 "update db cfg for bench using SORTHEAP 100000" # ~ 390MB
db2 "update db cfg for bench using SHEAPTHRES_SHR 100000"
db2 "update db cfg for bench using dft_extent_sz 256"

db2 "create temporary tablespace TMP4K pagesize 4K extentsize 256K transferrate
    0.56 bufferpool ibmdefaultbp"
db2 "create temporary tablespace TMP32K pagesize 32K extentsize 256K transferrate
    0.56 bufferpool bp32K"
db2 "drop tablespace TEMPSPACE1"

db2 "create regular tablespace MASTERTS in database partition group MASTER
    pagesize 4K managed by automatic storage initialsize 100M autoresize yes
    increasesize 100M maxsize 1G EXTENTSIZE 256K transferrate 0.56 bufferpool
    ibmdefaultbp"

db2 "create regular tablespace DATATS in database partition group NODESONLY
    pagesize 32K managed by automatic storage initialsize 20G autoresize yes
    increasesize 1G maxsize 60G EXTENTSIZE 256K TRANSFERRATE 0.56 BUFFERPOOL bp32K
    "

db2 "connect reset"
```

Listing A.2: create_db.sh

A.2. Hadoop

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>dfs.name.dir</name>
    <value>/mnt/dfs/name</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>/mnt/dfs/data</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.block.size</name>
      <value>134217728</value>
    </property>
</configuration>
```

Listing A.3: hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>mapred.job.tracker</name>
      <value>hdfs://master:54311</value>
  </property>
  <property>
    <name>mapred.local.dir</name>
      <value>/mnt/dfs/mapred</value>
  </property>
  <property>
    <name>mapred.system.dir</name>
      <value>/mapred/system</value>
  </property>
  <property>
    <name>mapreduce.jobtracker.staging.root.dir</name>
      <value>/user</value>
  </property>
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
      <value>8</value>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
```

```
29      <value>8</value>
30    </property>
31    <property>
32      <name>mapred.child.java.opts</name>
33      <value>-Xmx1024m -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps</value
            >
34    </property>
35    <property>
36      <name>io.sort.mb</name>
37      <value>256</value>
38    </property>
39    <property>
40      <name>fs.inmemory.size.mb</name>
41      <value>200</value>
42    </property>
43    <property>
44      <name>io.sort.factor</name>
45      <value>100</value>
46    </property>
47    <property>
48      <name>io.file.buffer.size</name>
49      <value>131072</value>
50    </property>
51    <property>
52      <name>mapred.compress.map.output</name>
53      <value>true</value>
54     </property>
55    <property>
56      <name>mapred.map.output.compression.type</name>
57      <value>BLOCK</value>
58     </property>
59    <property>
60      <name>mapred.map.output.compression.codec</name>
61      <value>org.apache.hadoop.io.compress.SnappyCodec</value>
62     </property>
63    <property>
64      <name>mapred.output.compress</name>
65      <value>true</value>
66     </property>
67  </configuration>
```

Listing A.4: mapred-site.xml

B. SYSSTAT Parameter

Modul	Parameter	Description
CPU	%usr	Percentage of CPU utilization that occurred while executing at the user level (application). Note that this field does NOT include time spent running virtual processors.
	%nice	Percentage of CPU utilization that occurred while executing at the user level with nice priority.
	%sys	Percentage of CPU utilization that occurred while executing at the system level (kernel). Note that this field does NOT include time spent servicing hardware and software interrupts.
	%iowait	Percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.
	%steal	Percentage of time spent in involuntary wait by the virtual CPU or CPUs while the hypervisor was servicing another virtual processor.
	%irq	Percentage of time spent by the CPU or CPUs to service hardware interrupts.
	%soft	Percentage of time spent by the CPU or CPUs to service software interrupts.
	%guest	Percentage of time spent by the CPU or CPUs to run a virtual processor.
	%idle	Percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

Tabelle B.1.: SYSSTAT Parameter Modul CPU

Modul	Parameter	Description
PAGING	pgpgin	Total number of kilobytes the system paged in from disk per second.
	pgpgout	Total number of kilobytes the system paged out to disk per second.
	fault	Number of page faults (major + minor) made by the system per second. This is not a count of page faults that generate I/O, because some page faults can be resolved without I/O.
	majflt	Number of major faults the system has made per second, those which have required loading a memory page from disk.
	pgfree	Number of pages placed on the free list by the system per second.
	pgscank	Number of pages scanned by the kswapd daemon per second.
	pgscand	Number of pages scanned directly per second.
	pgsteal	Number of pages the system has reclaimed from cache (pagecache and swapcache) per second to satisfy its memory demands.
	%vmeff	Calculated as pgsteal / pgscan, this is a metric of the efficiency of page reclaim. If it is near 100% then almost every page coming off the tail of the inactive list is being reaped. If it gets too low (e.g. less than 30%) then the virtual memory is having some difficulty. This field is displayed as zero if no pages have been scanned during the interval of time.

Tabelle B.2.: SYSSTAT Parameter Modul PAGING

Modul	Parameter	Description
IO	tps	Total number of transfers per second that were issued to physical devices. A transfer is an I/O request to a physical device. Multiple logical requests can be combined into a single I/O request to the device. A transfer is of indeterminate size.
	rtps	Total number of read requests per second issued to physical devices.
	wtps	Total number of write requests per second issued to physical devices.
	bread	Total amount of data read from the devices in blocks per second. Blocks are equivalent to sectors and therefore have a size of 512 bytes.
	bwrtn	Total amount of data written to devices in blocks per second.

Tabelle B.3.: SYSSTAT Parameter Modul IO

Modul	Parameter	Description
MEM	kbmemfree	Amount of free memory available in kilobytes.
	kbmemused	Amount of used memory in kilobytes. This does not take into account memory used by the kernel itself.
	%memused	Percentage of used memory.
	kbbuffers	Amount of memory used as buffers by the kernel in kilobytes.
	kbcached	Amount of memory used to cache data by the kernel in kilobytes.
	kbcommit	Amount of memory in kilobytes needed for current workload. This is an estimate of how much RAM/swap is needed to guarantee that there never is out of memory.
	%commit	Percentage of memory needed for current workload in relation to the total amount of memory (RAM+swap). This number may be greater than 100% because the kernel usually overcommits memory.

Tabelle B.4.: SYSSTAT Parameter Modul MEM

Modul	Parameter	Description
QUEUE	runq_sz	Run queue length (number of tasks waiting for run time).
	plist_st	Number of tasks in the process list.
	ldavg_1	System load average for the last minute. The load average is calculated as the average number of runnable or running tasks (R state), and the number of tasks in uninterruptible sleep (D state) over the specified interval.
	ldavg_5	System load average for the past 5 minutes.
	ldavg_15	System load average for the past 15 minutes.

Tabelle B.5.: SYSSTAT Parameter Modul QUEUE

Modul	Parameter	Description
These Parameters are available for every network interface.		
IFACE	rxpck	Total number of packets received per second.
	txpck	Total number of packets transmitted per second.
	rxkB	Total number of kilobytes received per second.
	txkB	Total number of kilobytes transmitted per second.
	rxcmp	Number of compressed packets received per second (for cslip etc.).
	txcmp	Number of compressed packets transmitted per second.
	rxmcst	Number of multicast packets received per second.

Tabelle B.6.: SYSSTAT Parameter Modul IFACE

C. Tabellendefinitionen

```
CREATE TABLE latency (
  unit INTEGER NOT NULL,
  hour SMALLINT NOT NULL,
  minute SMALLINT NOT NULL,
  target VARCHAR(40) NOT NULL,
  rtt DOUBLE NOT NULL,
  payload char(192),
  PRIMARY KEY ( hour, minute, unit, target )
)
IN DATATS
DISTRIBUTE BY HASH (unit,hour);

CREATE TABLE position(
  unit INTEGER NOT NULL,
  hour SMALLINT NOT NULL,
  minute SMALLINT NOT NULL,
  lat DOUBLE NOT NULL,
  lon DOUBLE NOT NULL
)
IN DATATS
DISTRIBUTE BY HASH(lat,lon);

CREATE TABLE metadata (
  unit INTEGER NOT NULL PRIMARY KEY,
  isp VARCHAR(40)
)
IN MASTERTS;

CREATE INDEX "DB2INST1"."LAT_TARGET" ON "DB2INST1"."LATENCY"
  ("TARGET" ASC)
  ALLOW REVERSE SCANS COLLECT SAMPLED DETAILED STATISTICS;

CREATE INDEX "DB2INST1"."LAT_HOUR_UNIT" ON "DB2INST1"."LATENCY"
  ("HOUR" ASC, "UNIT" ASC)
  ALLOW REVERSE SCANS COLLECT SAMPLED DETAILED STATISTICS;

CREATE INDEX "DB2INST1"."LAT_UNIT_HOUR" ON "DB2INST1"."LATENCY"
  ("UNIT" ASC, "HOUR" ASC)
  ALLOW REVERSE SCANS COLLECT SAMPLED DETAILED STATISTICS;
```

Listing C.1: synt_tables.sql

```
create table timebins (
  id integer,
  name varchar(12),
  ts timestamp
)
IN MASTERTS;

create table swarms (
  id integer,
  hash char(40)
)
IN MASTERTS;

create table peers (
  id integer,
  swarm integer,
  ip varchar(40),
  port integer,
  client varchar(40),
  hash char(40)
)
IN DATATS
DISTRIBUTE BY HASH (id,swarm);

create table announces (
  peer integer,
  swarm integer,
  timebin integer,
  ts integer,
  type smallint,
  chunks integer,
  bitvector varchar(32000)
)
IN DATATS
DISTRIBUTE BY HASH (peer,swarm);

CREATE INDEX "DB2INST1"."ANN_TB_S_P_T" ON "DB2INST1"."ANNOUNCES"
  ("TIMEBIN" ASC, "SWARM" ASC, "PEER" ASC, "TYPE" ASC)
    ALLOW REVERSE SCANS COLLECT SAMPLED DETAILED STATISTICS;
```

Listing C.2: bt_tables.sql

D. Ausführungspläne

D.1. Synthetische Tests

D.1.1. DB2 Ausführungspläne

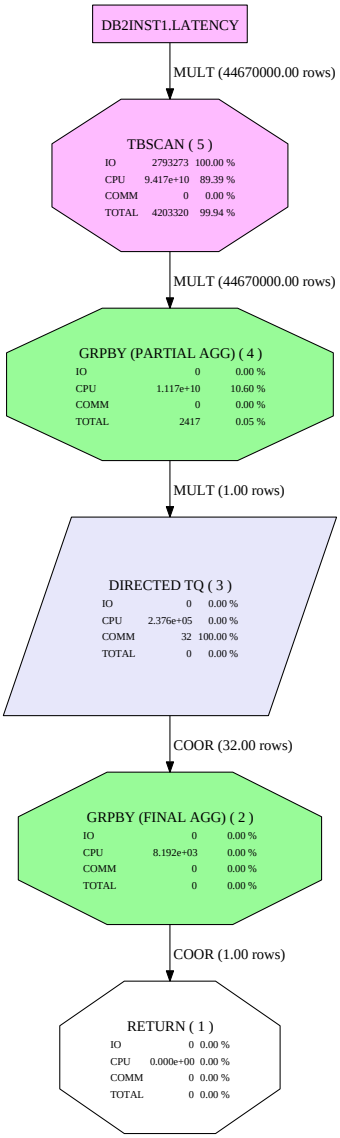

Abbildung D.1.: DB2 Aggregation

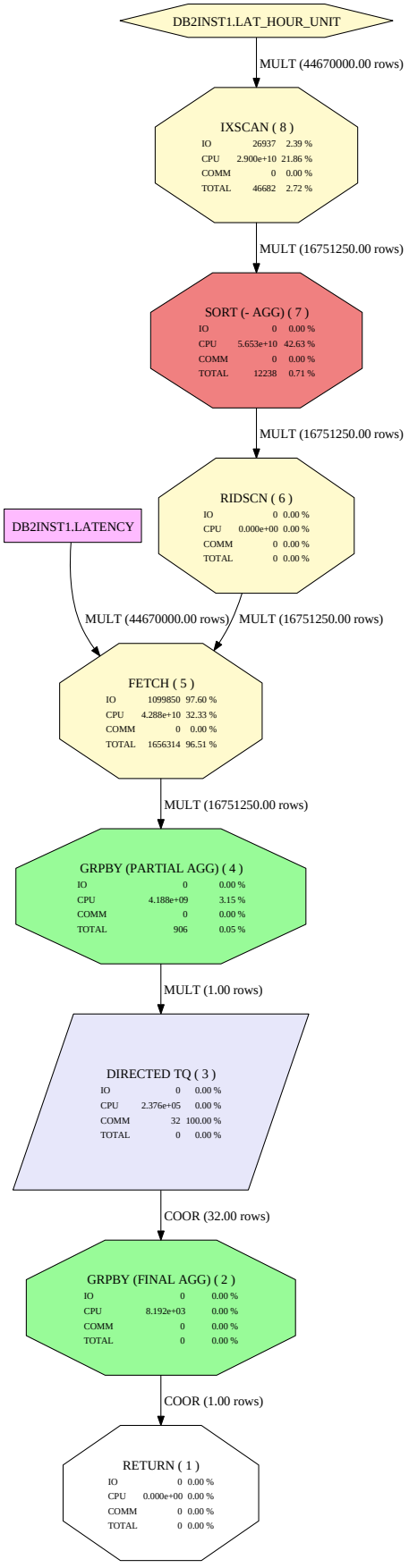

Abbildung D.2.: DB2 Filter

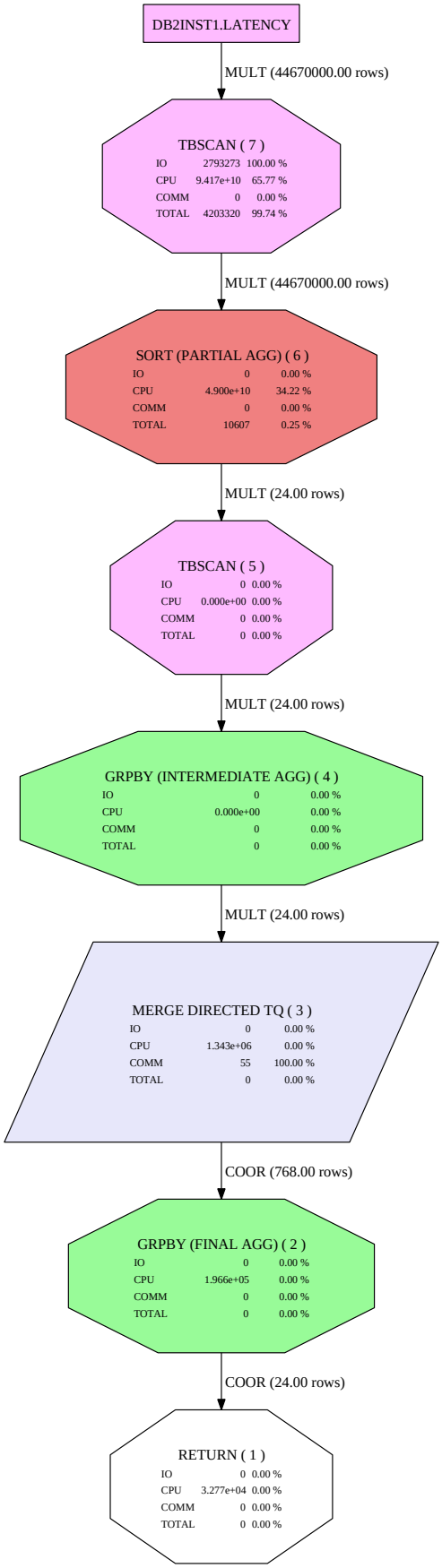

Abbildung D.3.: DB2 Group

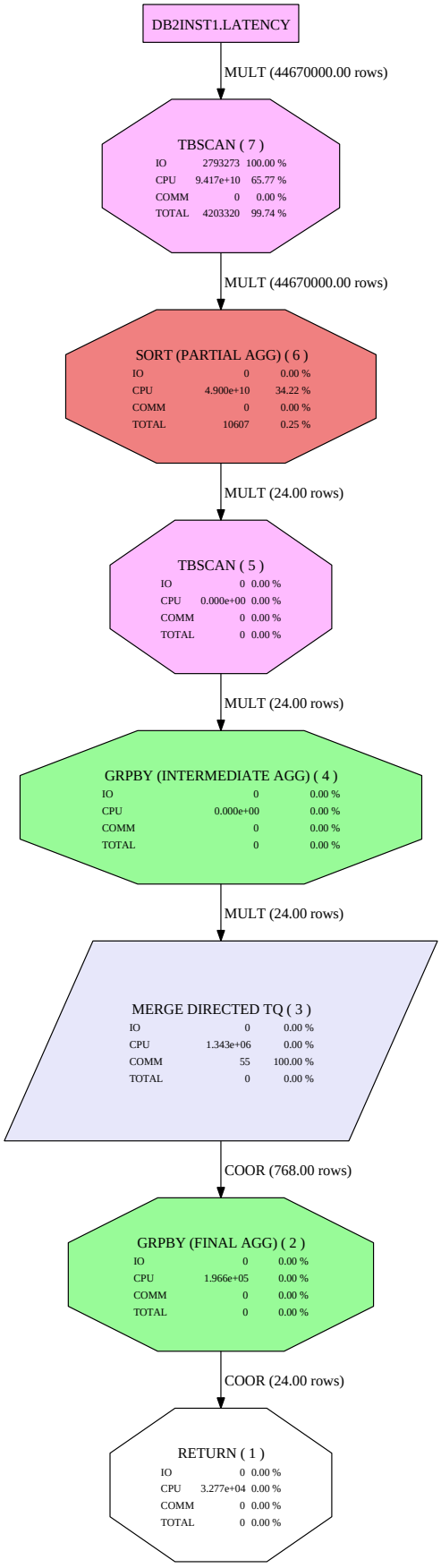

Abbildung D.4.: DB2 Distinct

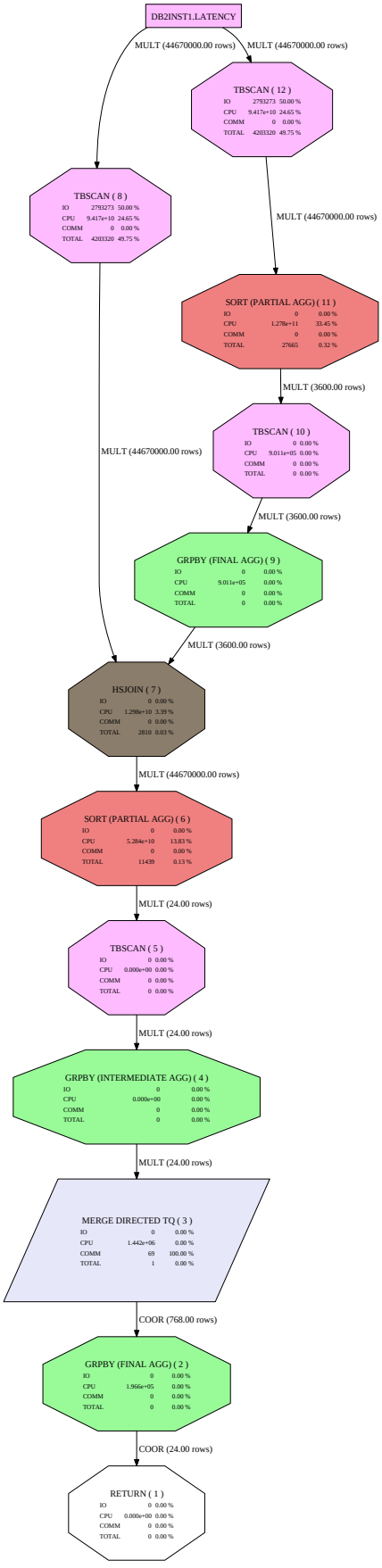

Abbildung D.5.: DB2 collocated Join

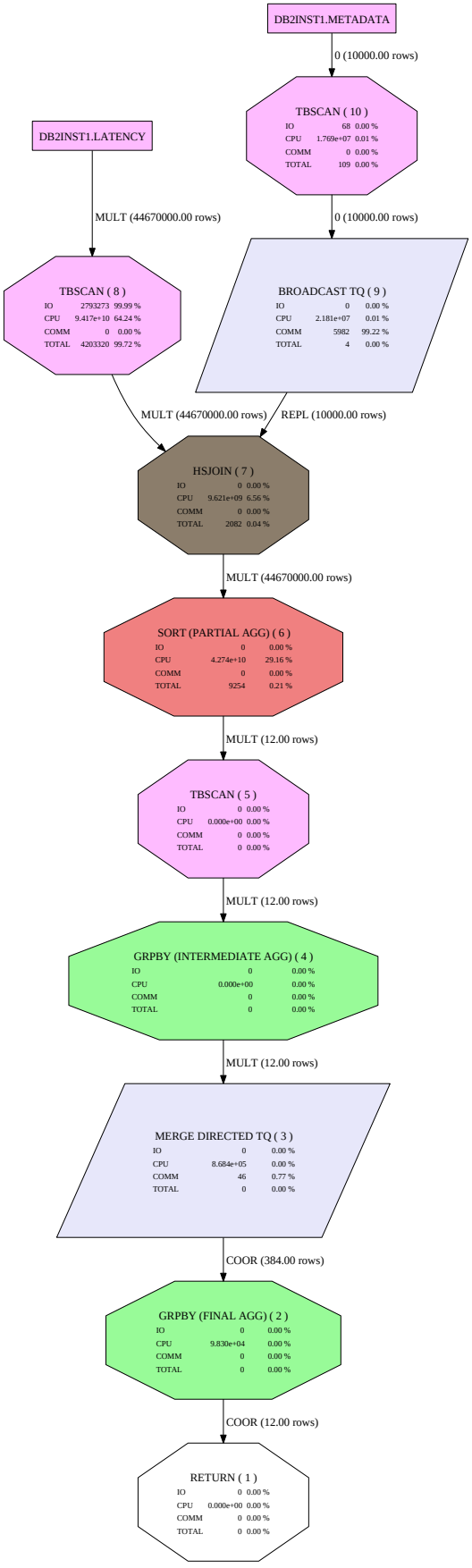

Abbildung D.6.: DB2 replicated Join

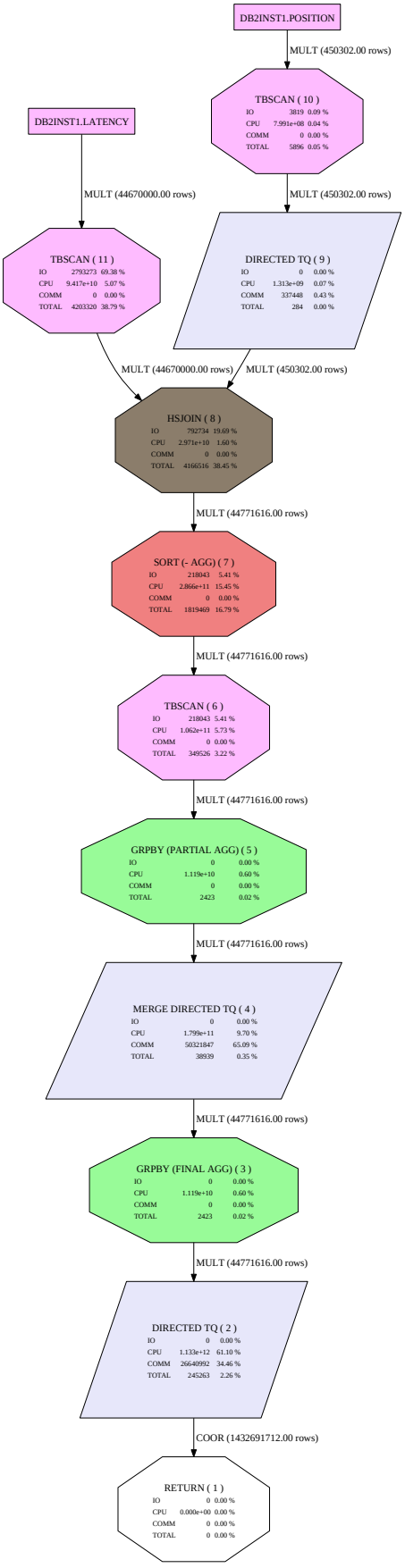

Abbildung D.7.: DB2 directed Join

D.1.2. Hadoop Ausführungspläne

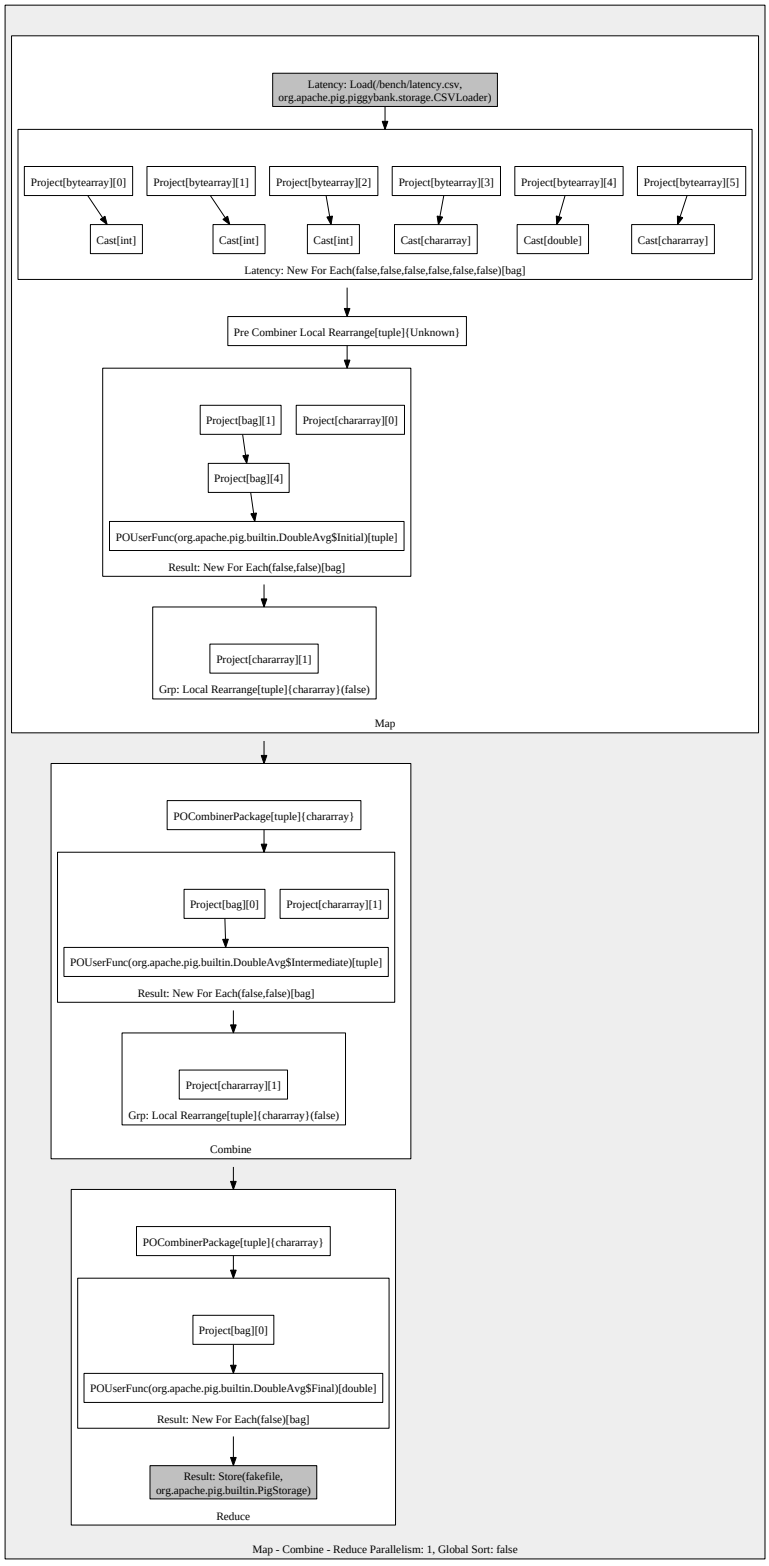

Abbildung D.8.: Hadoop Aggregation

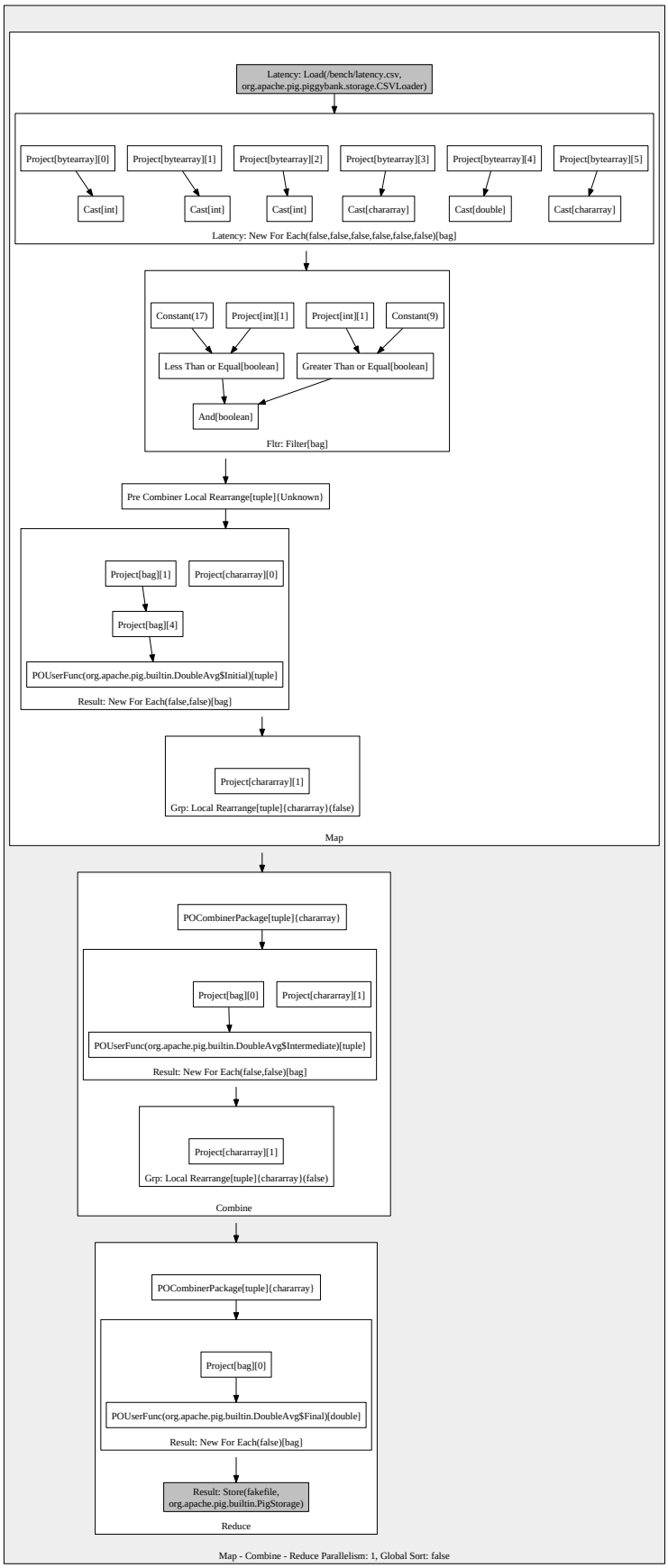

Abbildung D.9.: Hadoop Filter

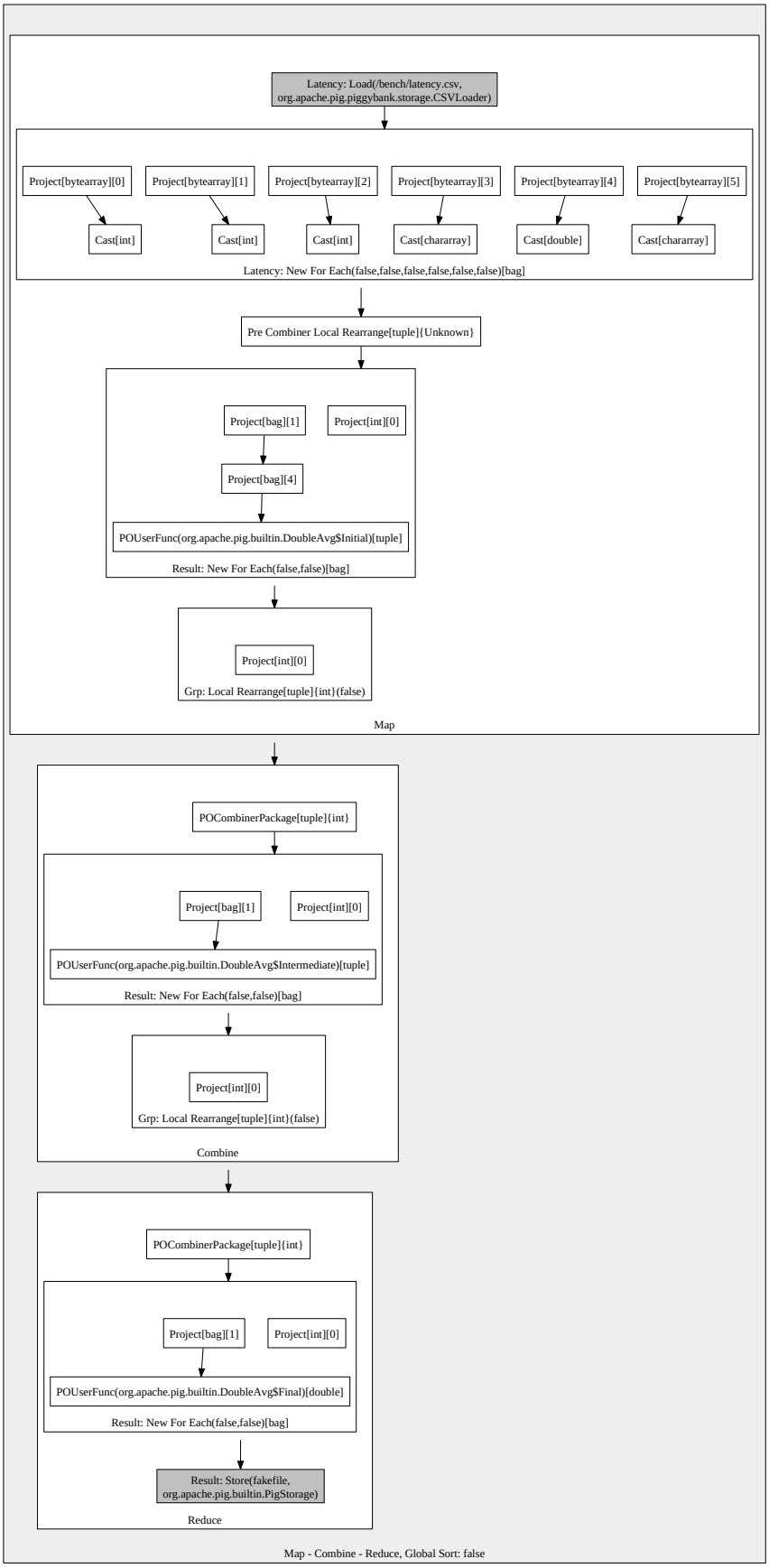

Abbildung D.10.: Hadoop Group

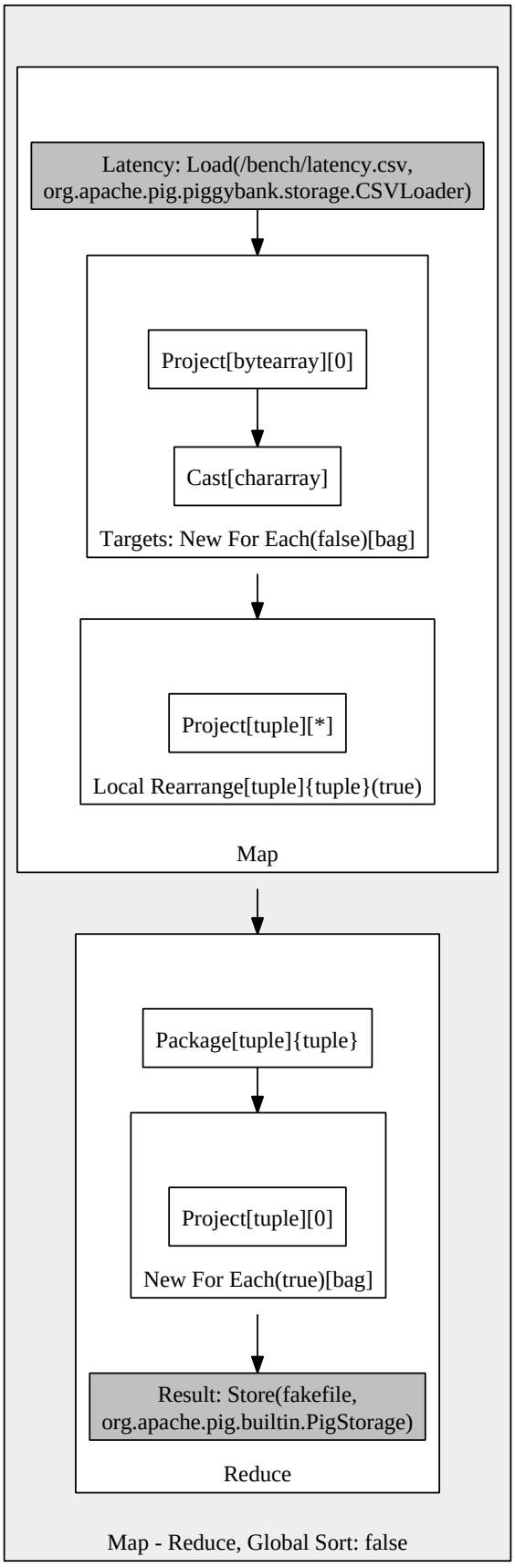

Abbildung D.11.: Hadoop Distinct

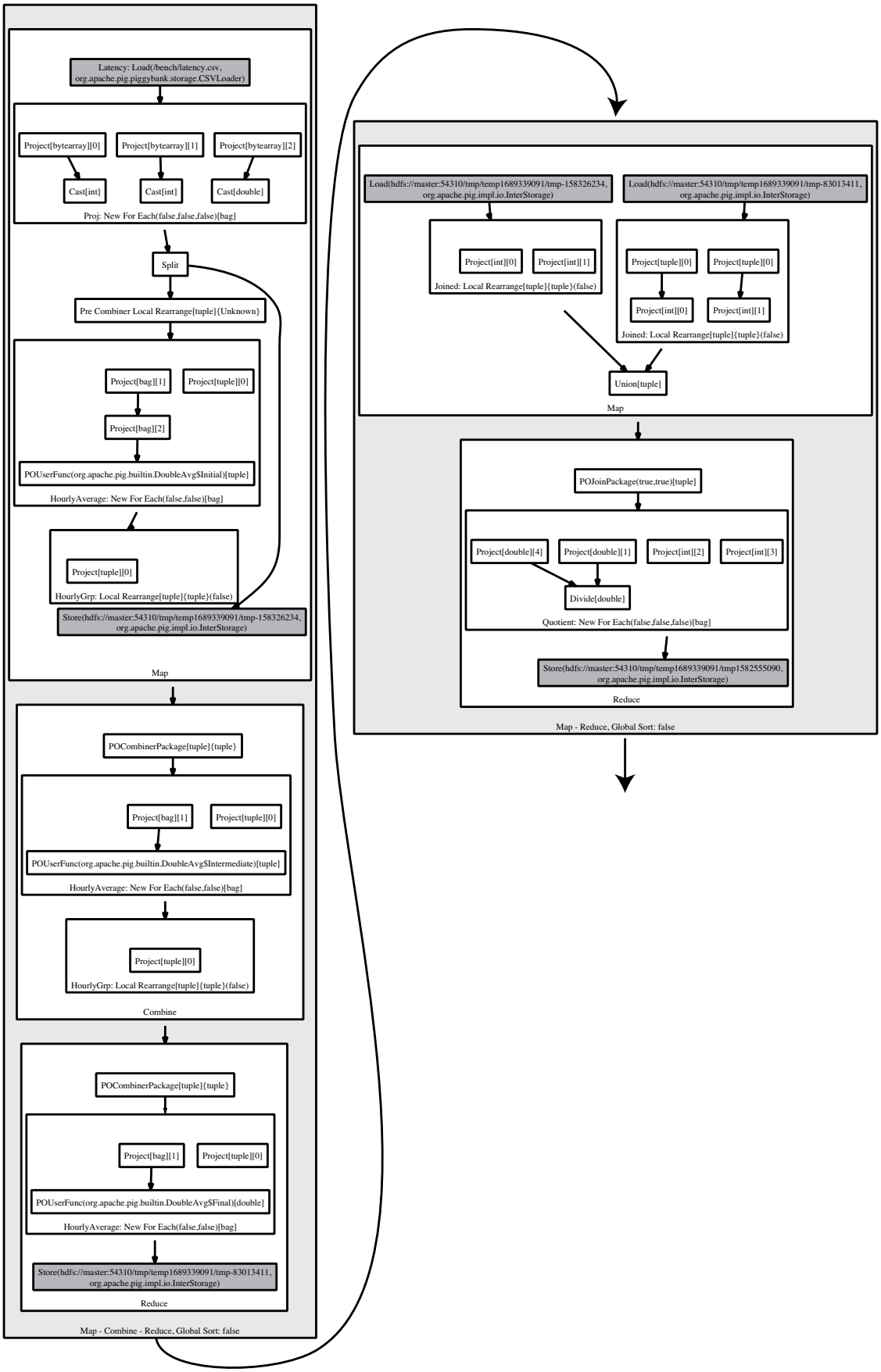

Abbildung D.12.: Hadoop collocated Join Teil1

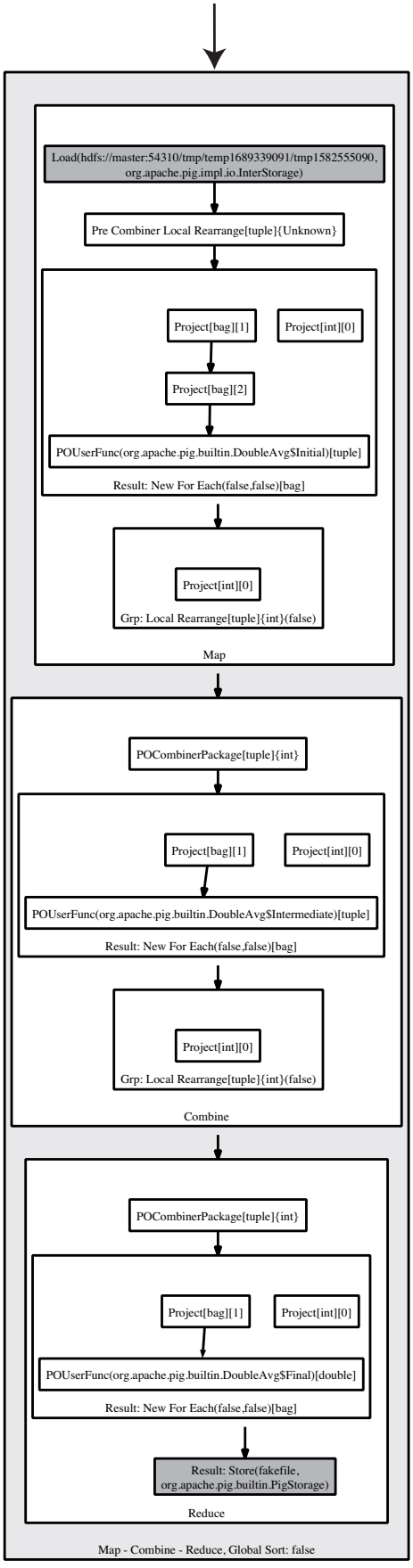

Abbildung D.13.: Hadoop collocated Join Teil2

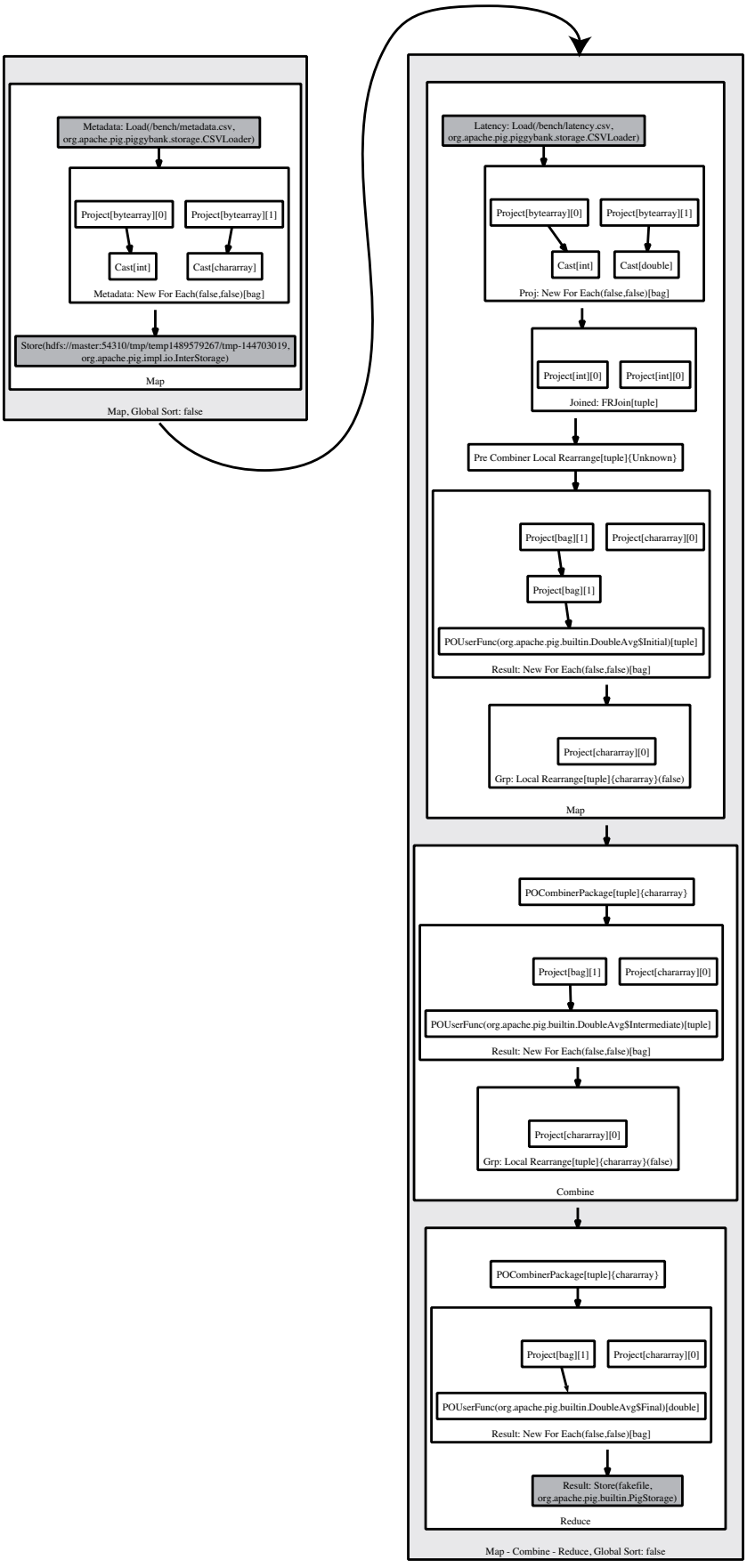

Abbildung D.14.: Hadoop replicated Join

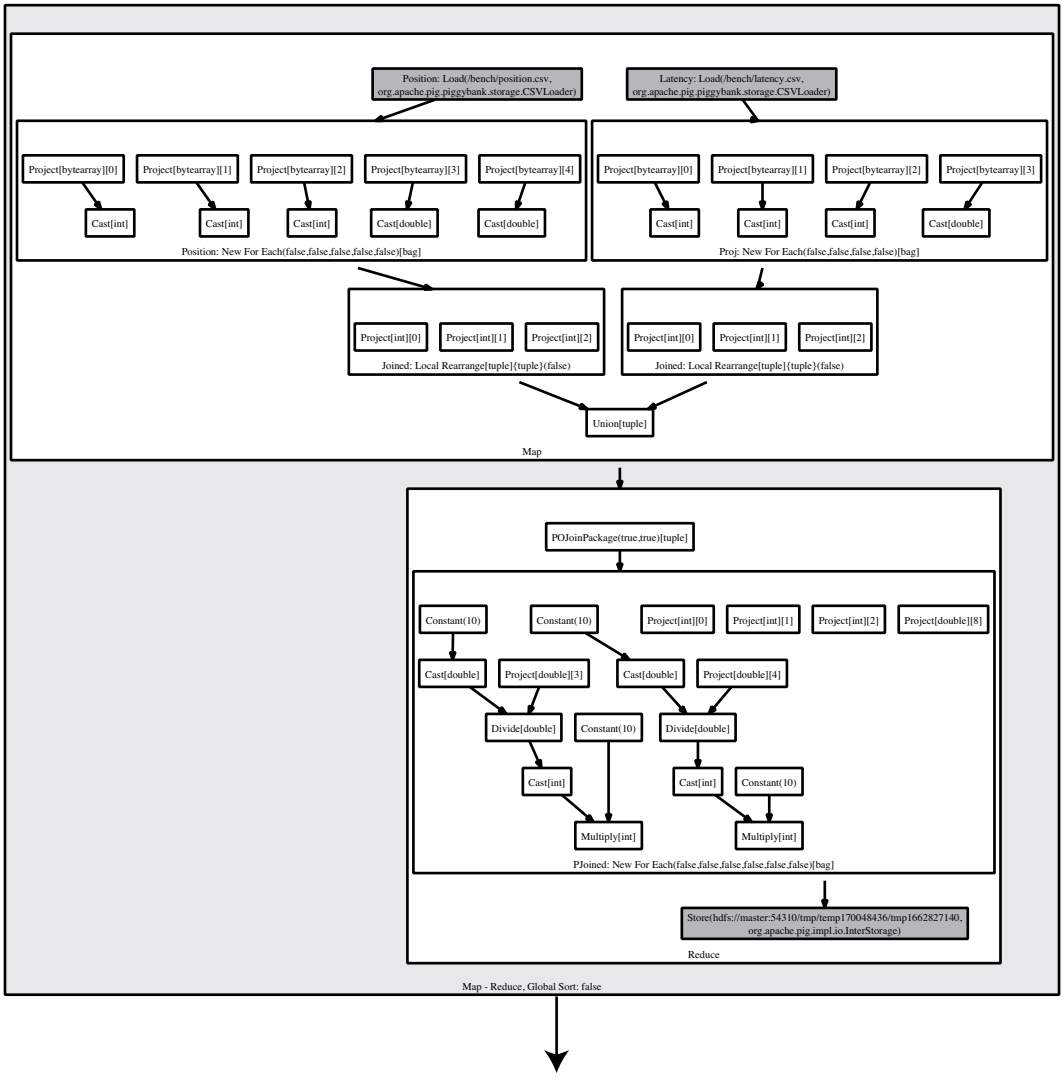

Abbildung D.15.: Hadoop directed Join Teil 1

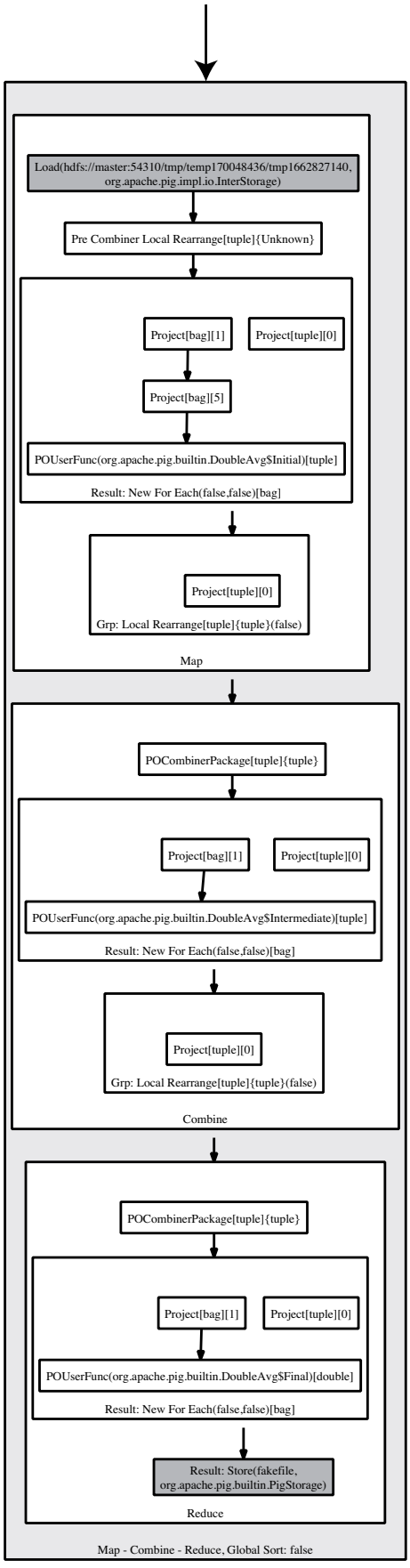

Abbildung D.16.: Hadoop directed Join Teil 2

D.1.3. MapReduce Taskscheduling

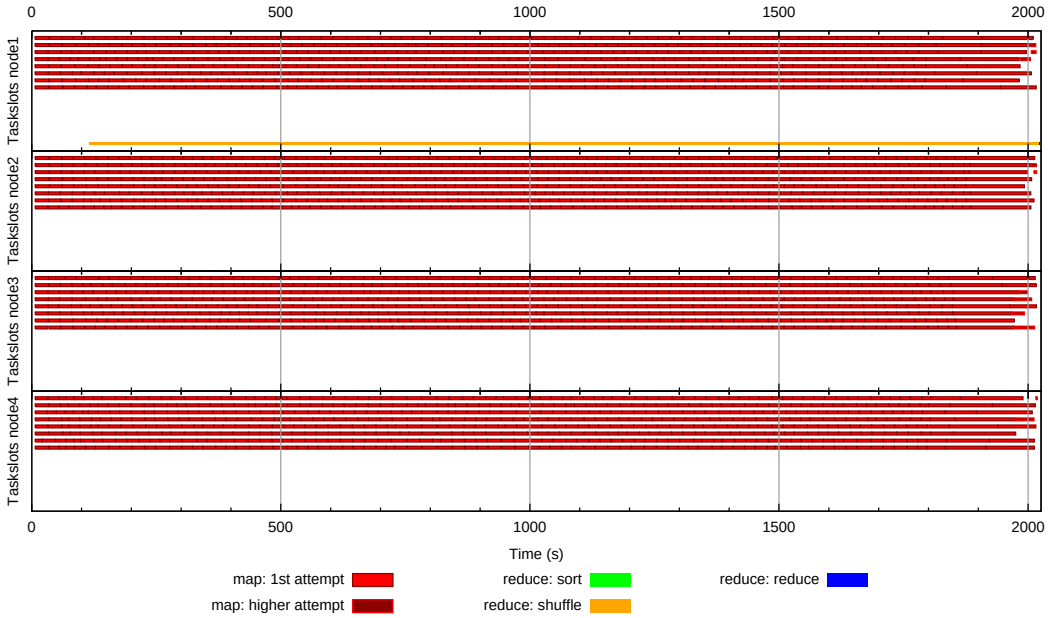

Abbildung D.17.: MR Taskscheduling Aggregation

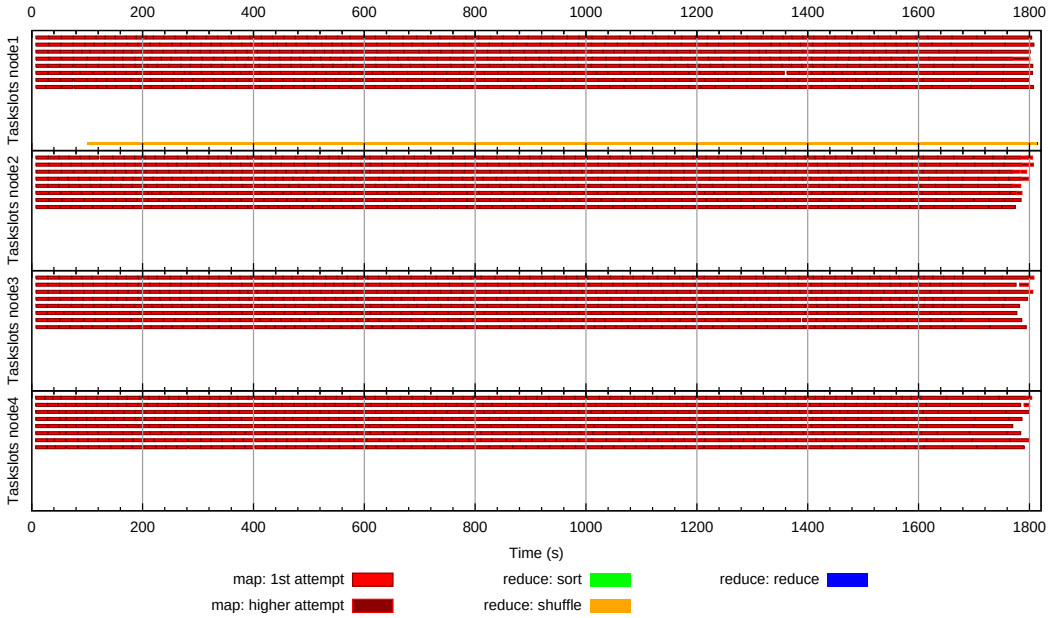

Abbildung D.18.: MR Taskscheduling Filter

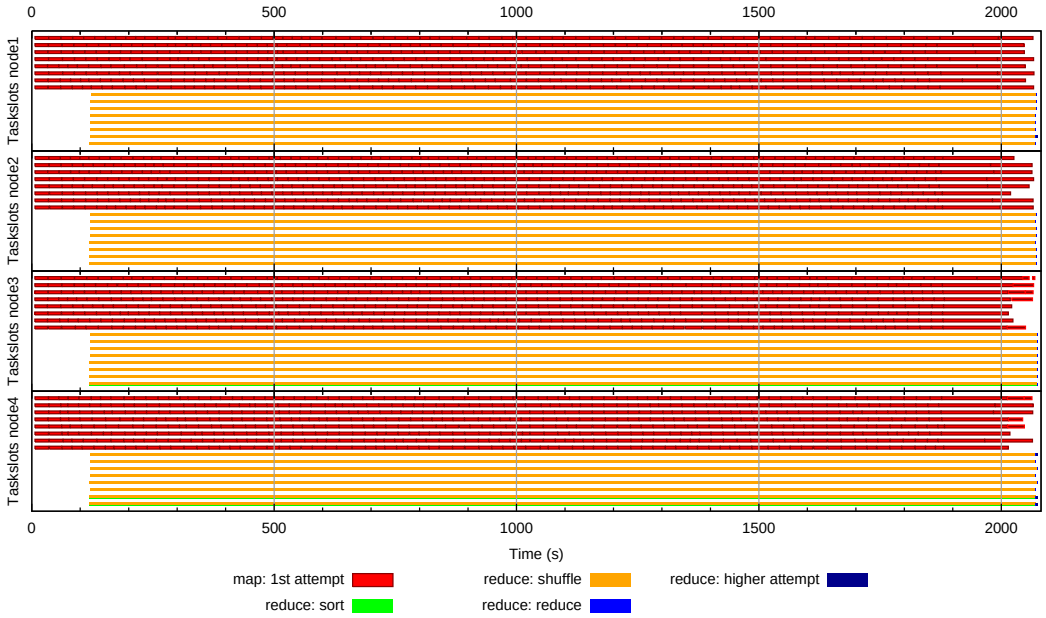

Abbildung D.19.: MR Taskscheduling Group

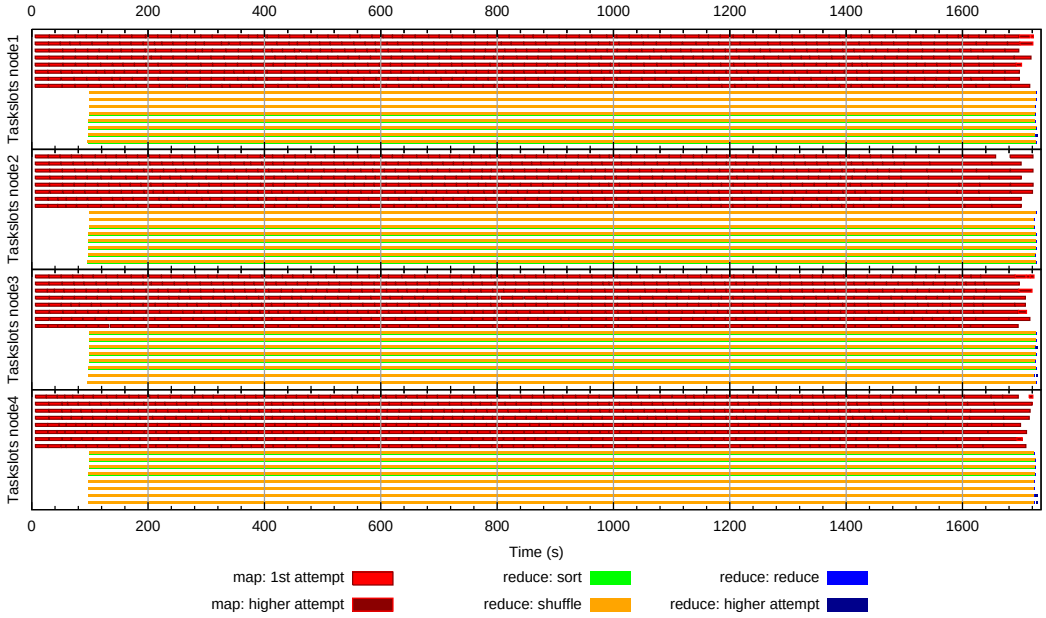

Abbildung D.20.: MR Taskscheduling Distinct

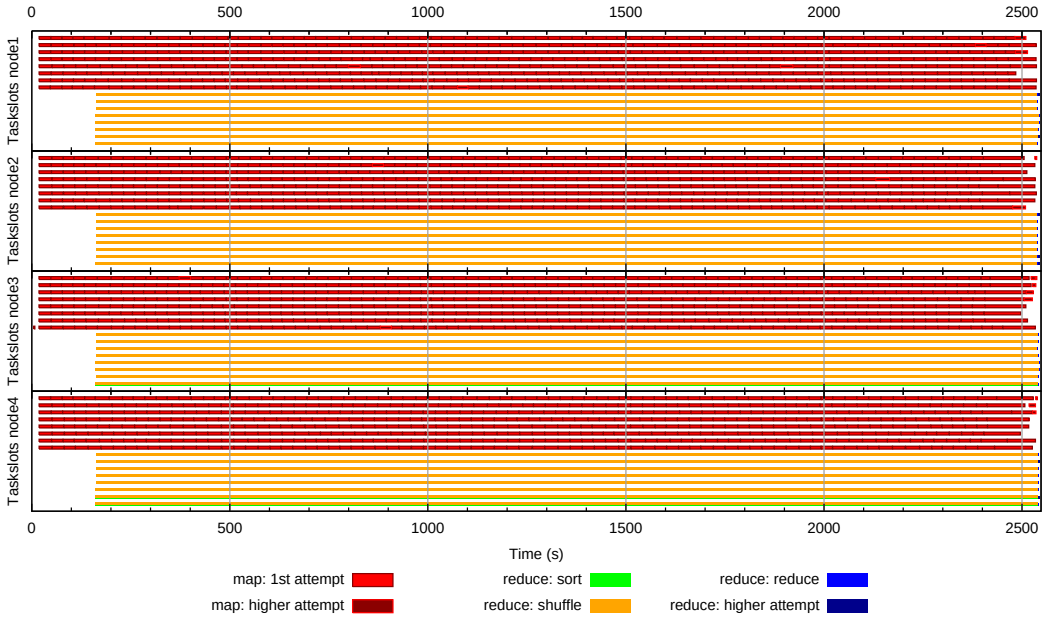

Abbildung D.21.: MR Taskscheduling collocated Join

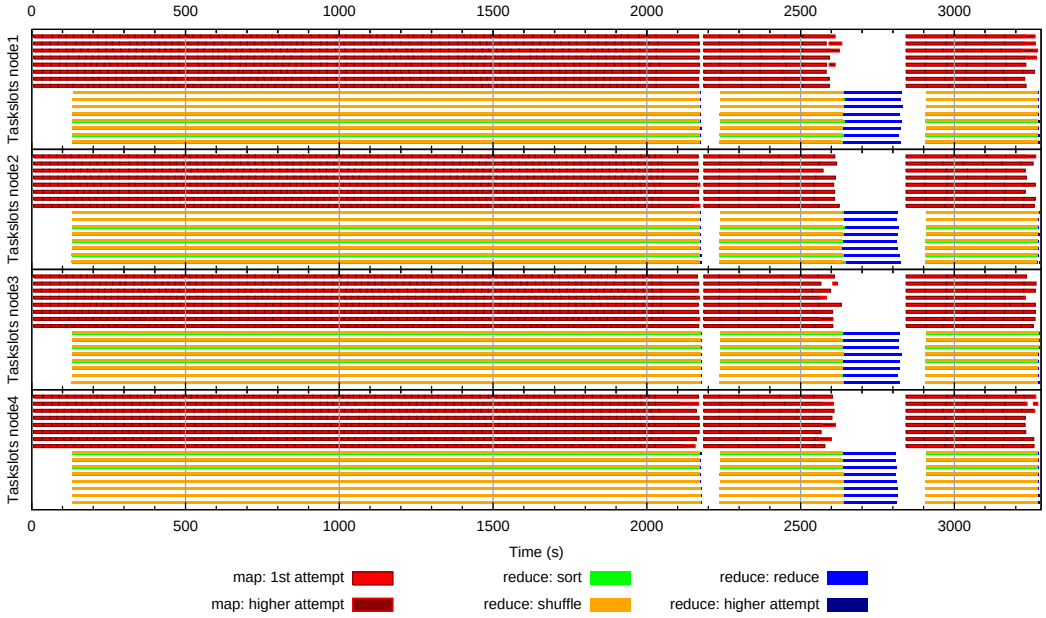

Abbildung D.22.: MR Taskscheduling replicated Join

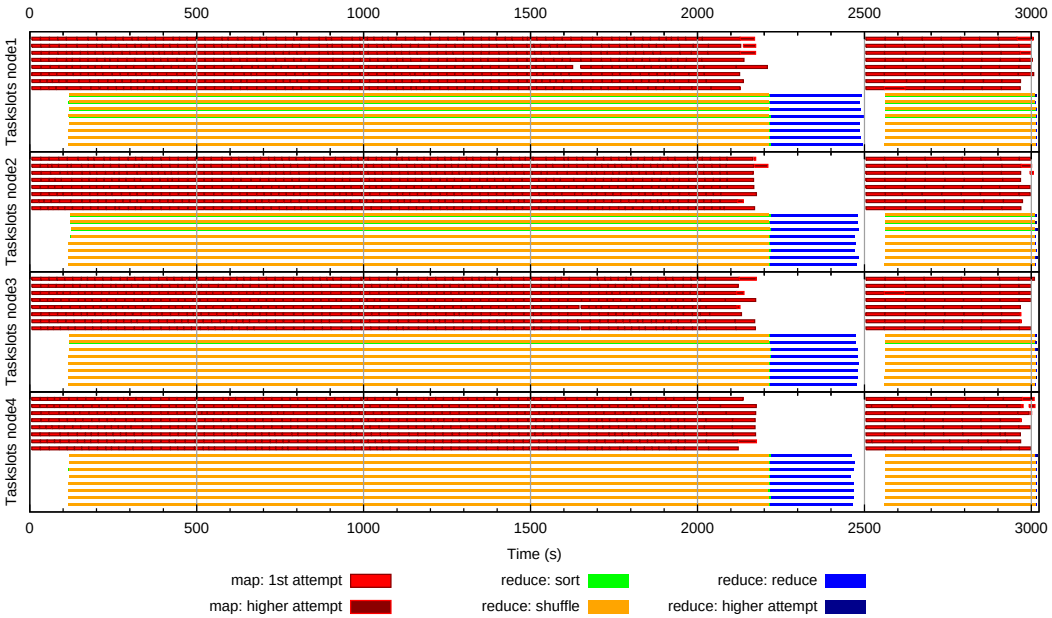

Abbildung D.23.: MR Taskscheduling directed Join

D.2. BitTorrent Tests

D.2.1. DB2 Ausführungspläne mit Index

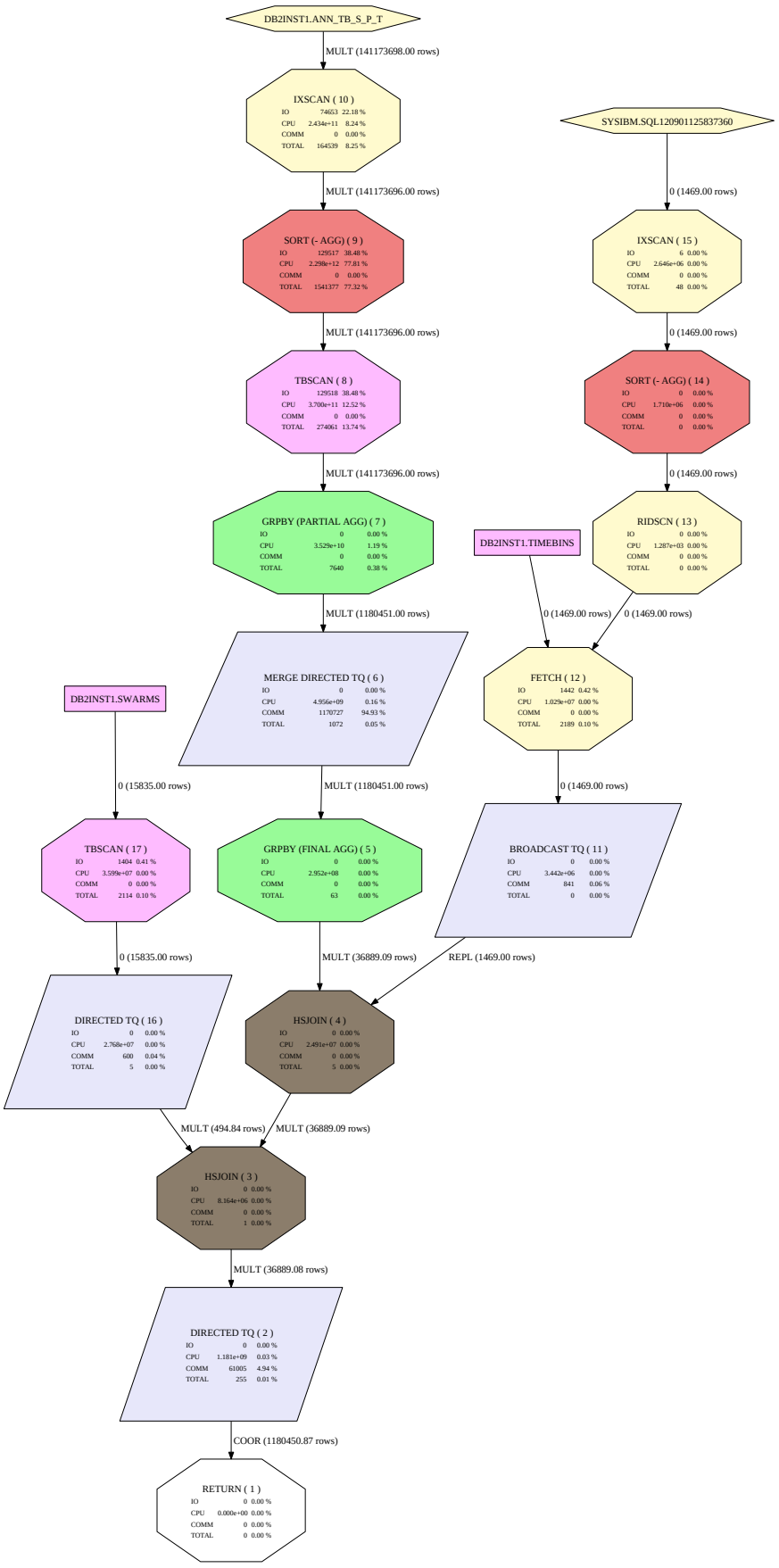

Abbildung D.24.: DB2 BitTorrent Job1

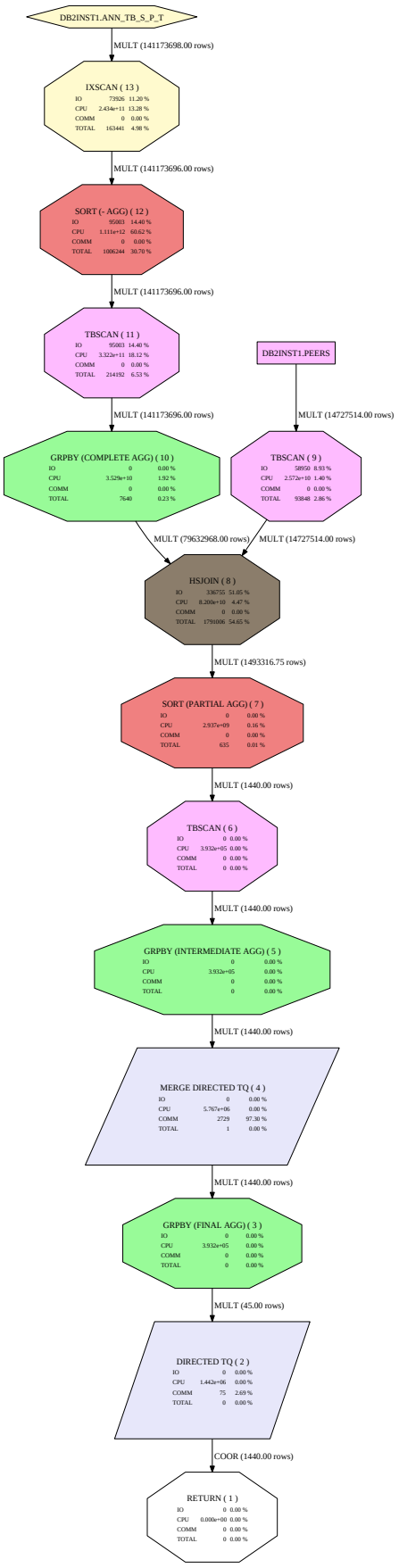

Abbildung D.25.: DB2 BitTorrent Job2

D.2.2. DB2 Ausführungspläne ohne Index

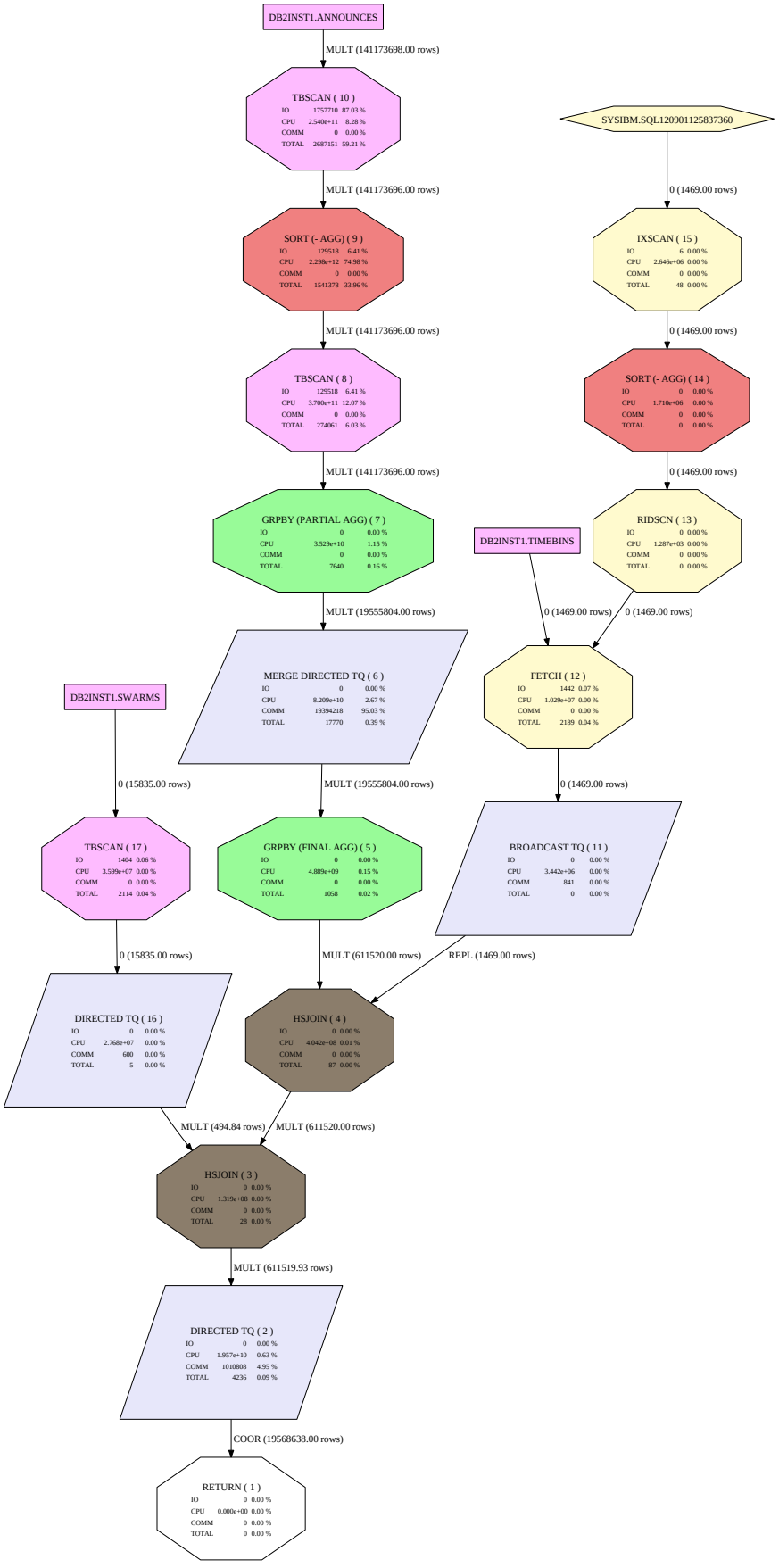

Abbildung D.26.: DB2 BitTorrent Job1 ohne Index

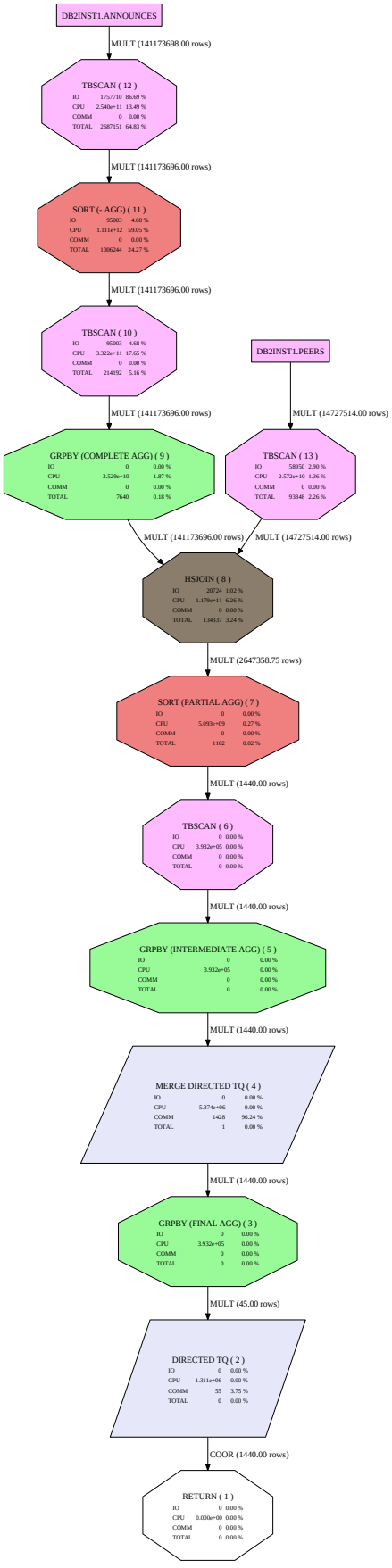

Abbildung D.27.: DB2 BitTorrent Job2 ohne Index

D.2.3. Hadoop Ausführungpläne

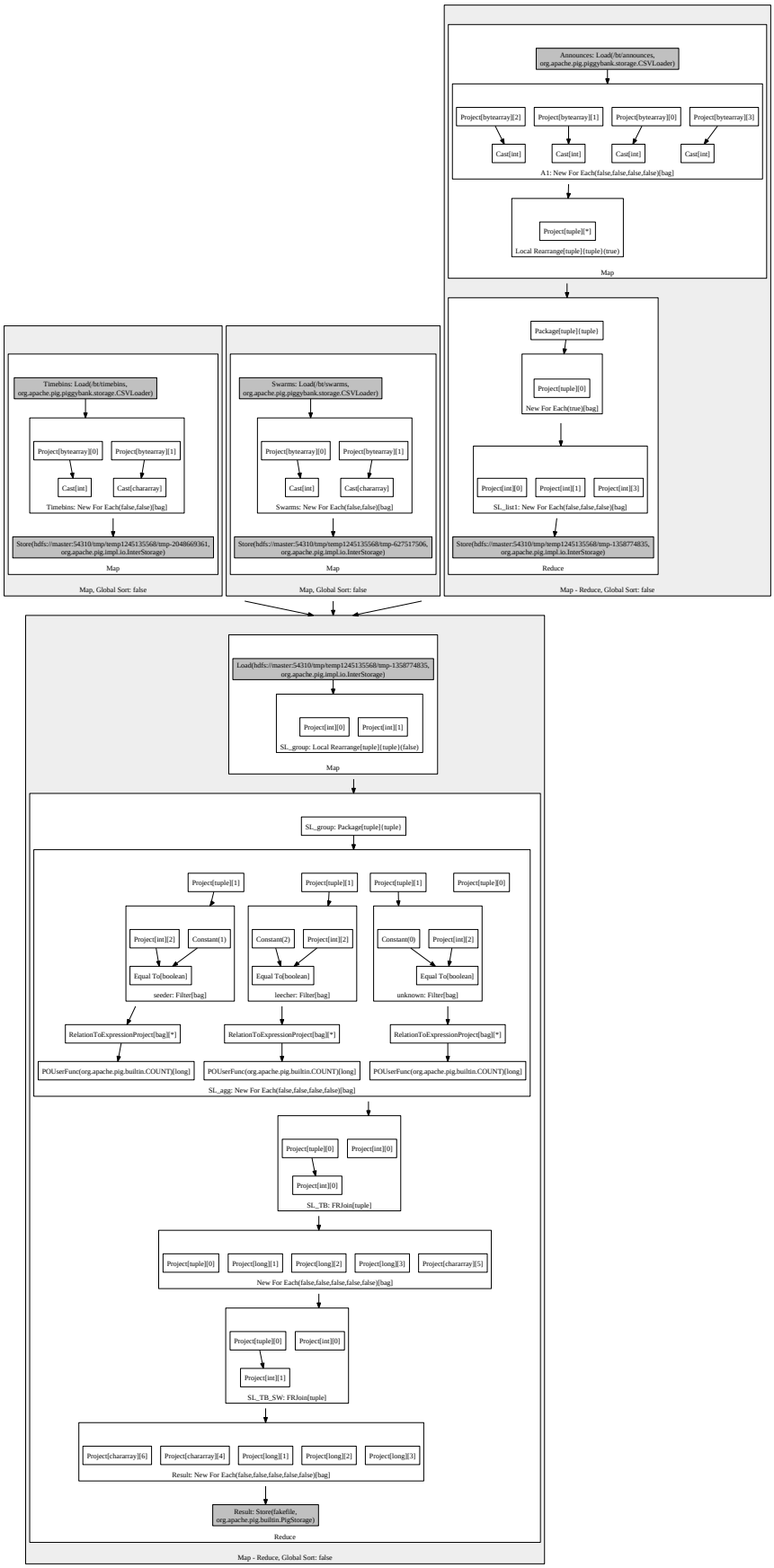

Abbildung D.28.: Hadoop BitTorrent Job1

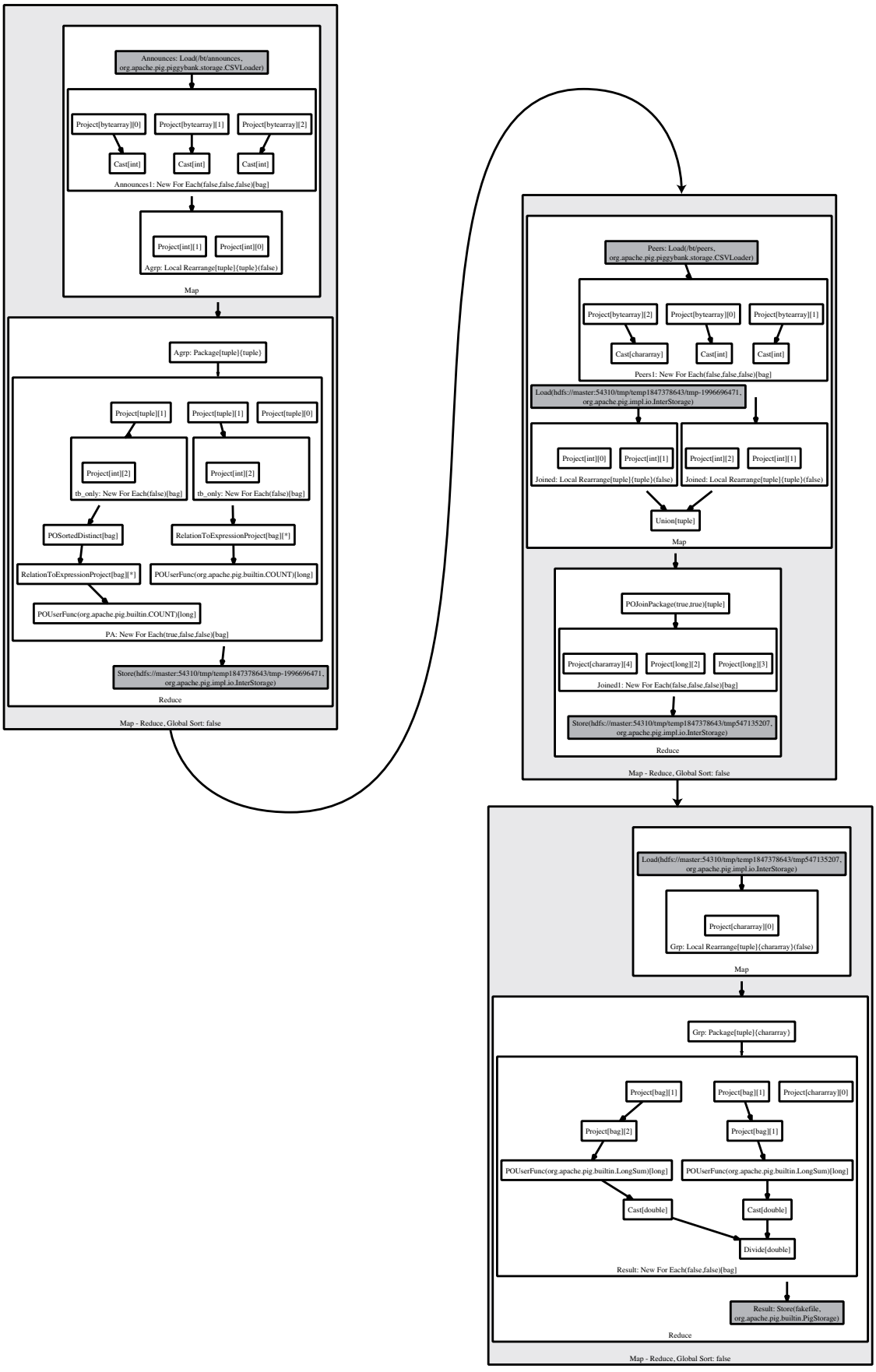

Abbildung D.29.: Hadoop BitTorrent Job2

D.2.4. MapReduce Taskscheduling

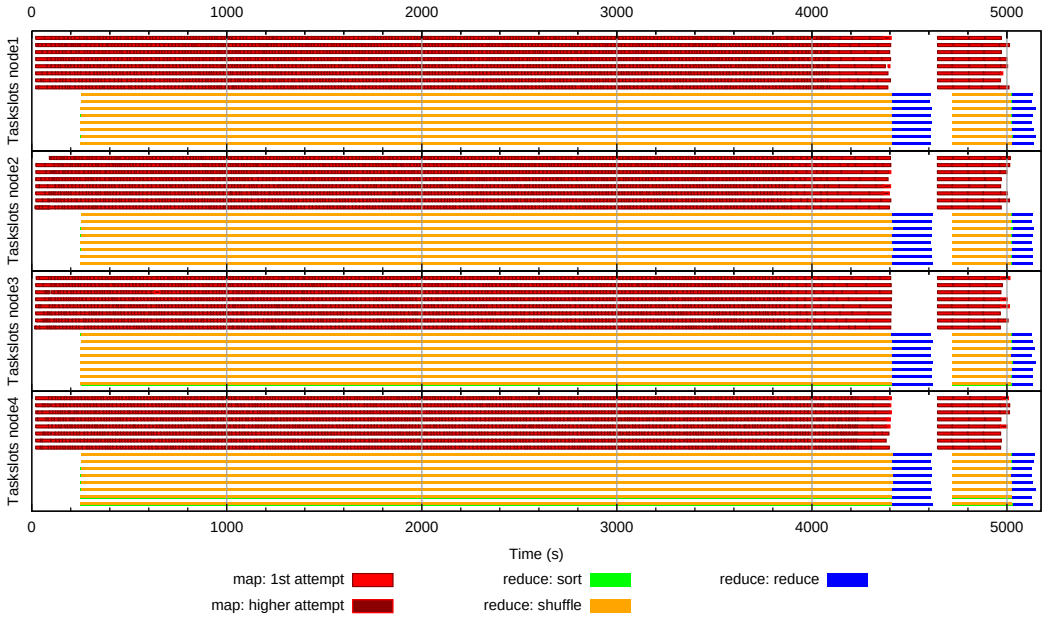

Abbildung D.30.: MR Taskscheduling BitTorrent Job1

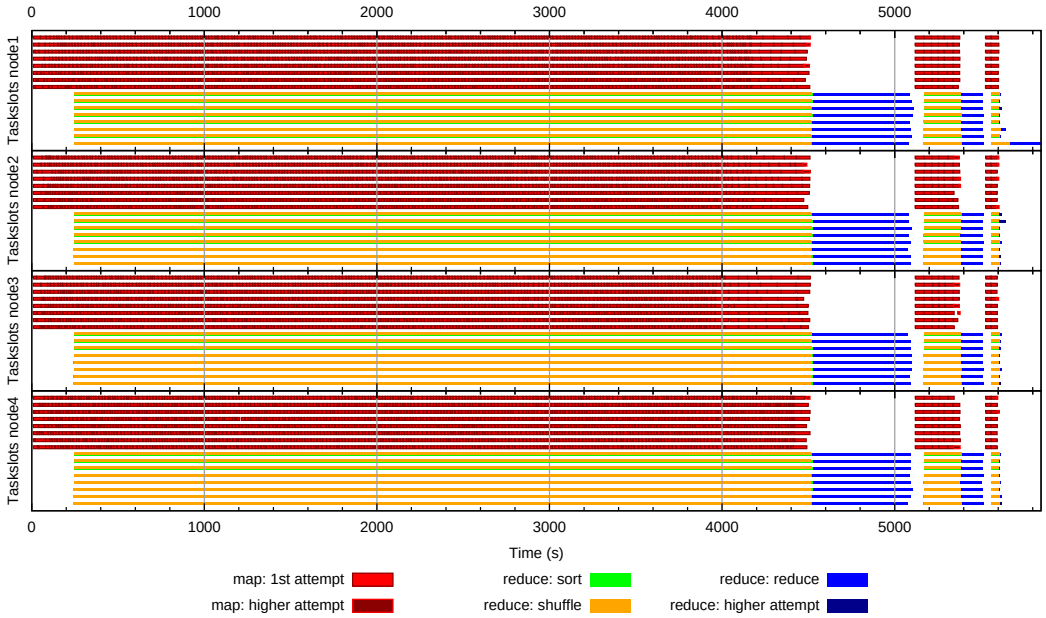

Abbildung D.31.: MR Taskscheduling BitTorrent Job2

E. Weitere Reports

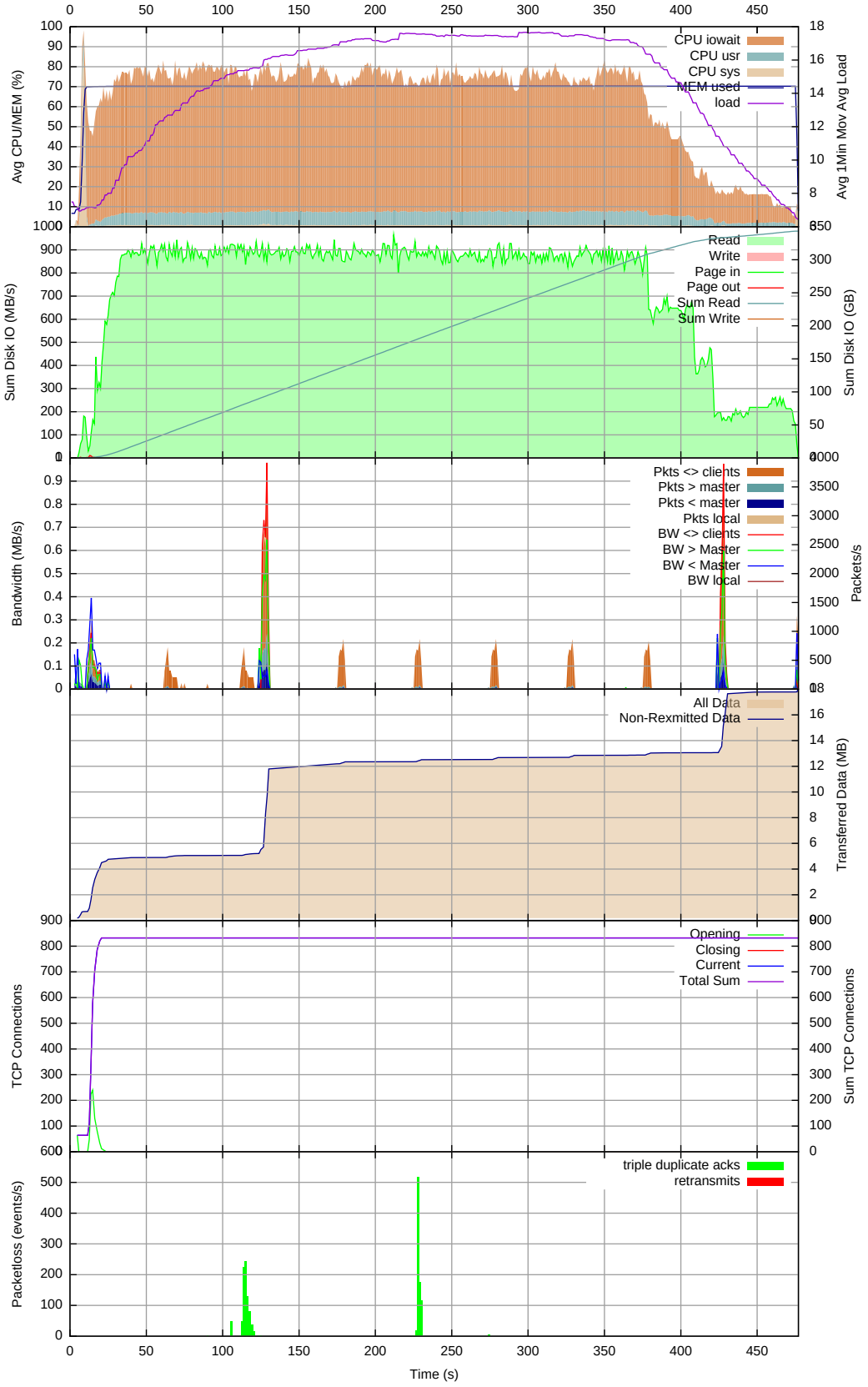

Abbildung E.1.: DB2 Group Report

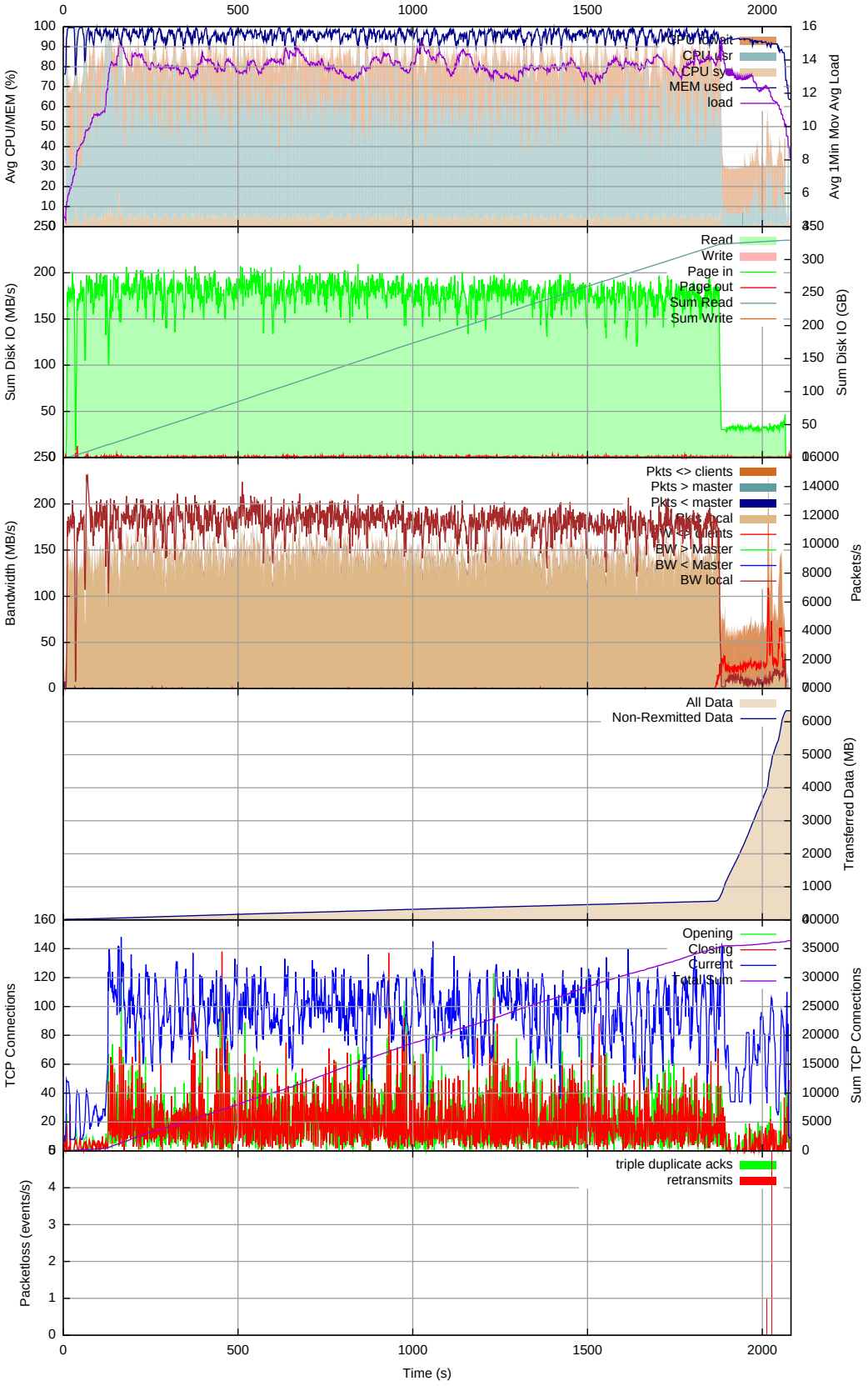

Abbildung E.2.: Hadoop Group Report

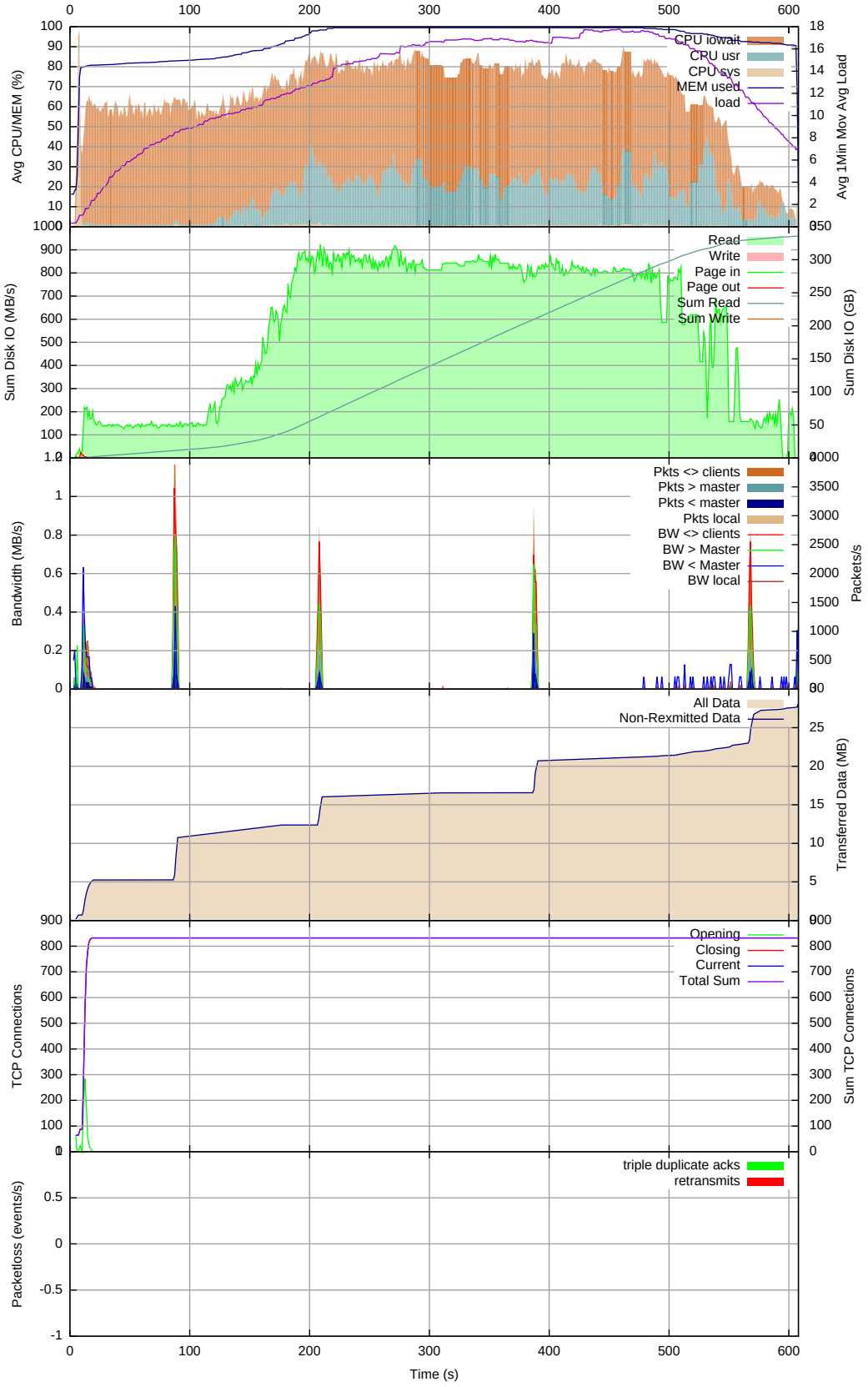

Abbildung E.3.: DB2 Distinct ohne Index Report

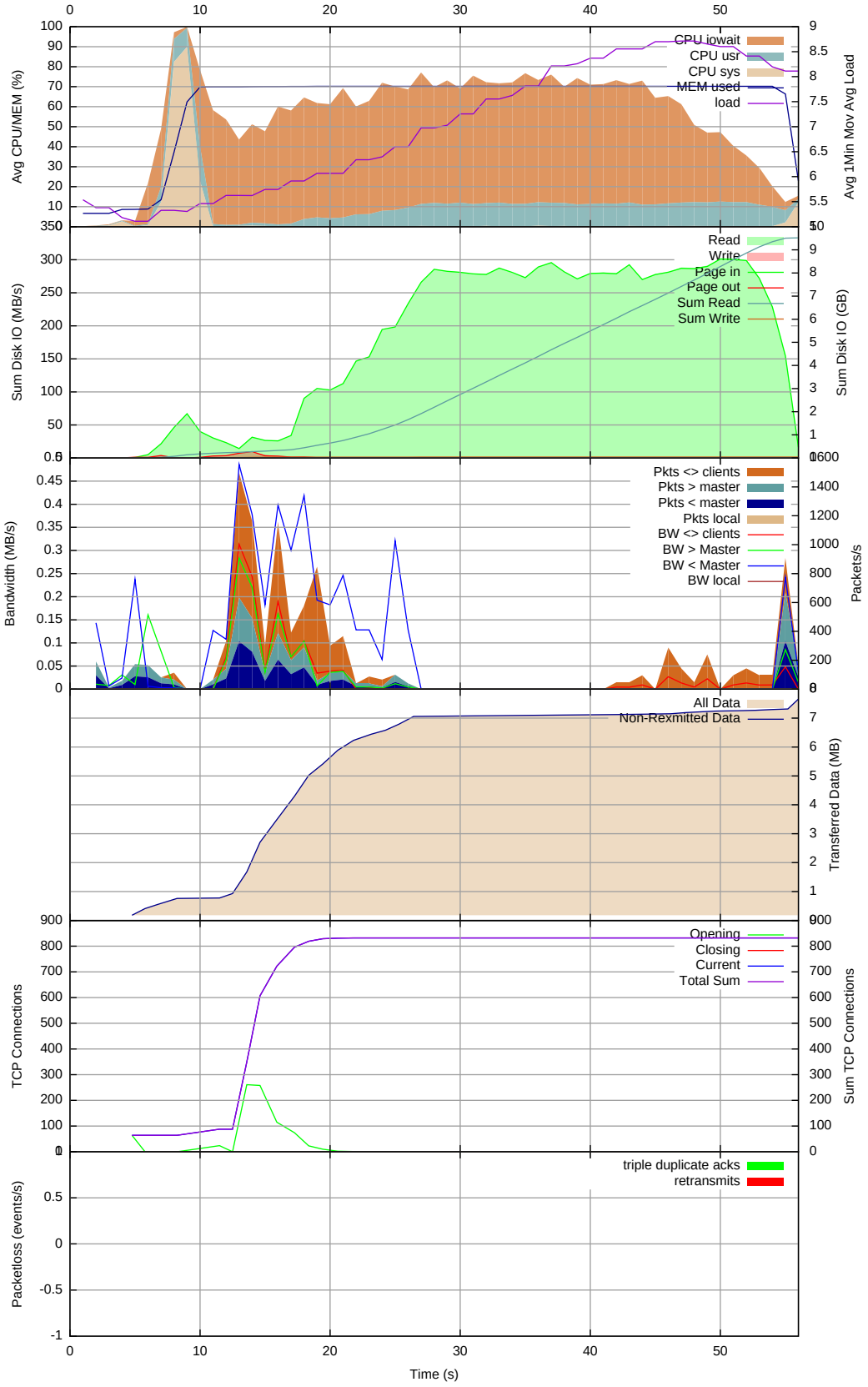

Abbildung E.4.: DB2 Distinct Report

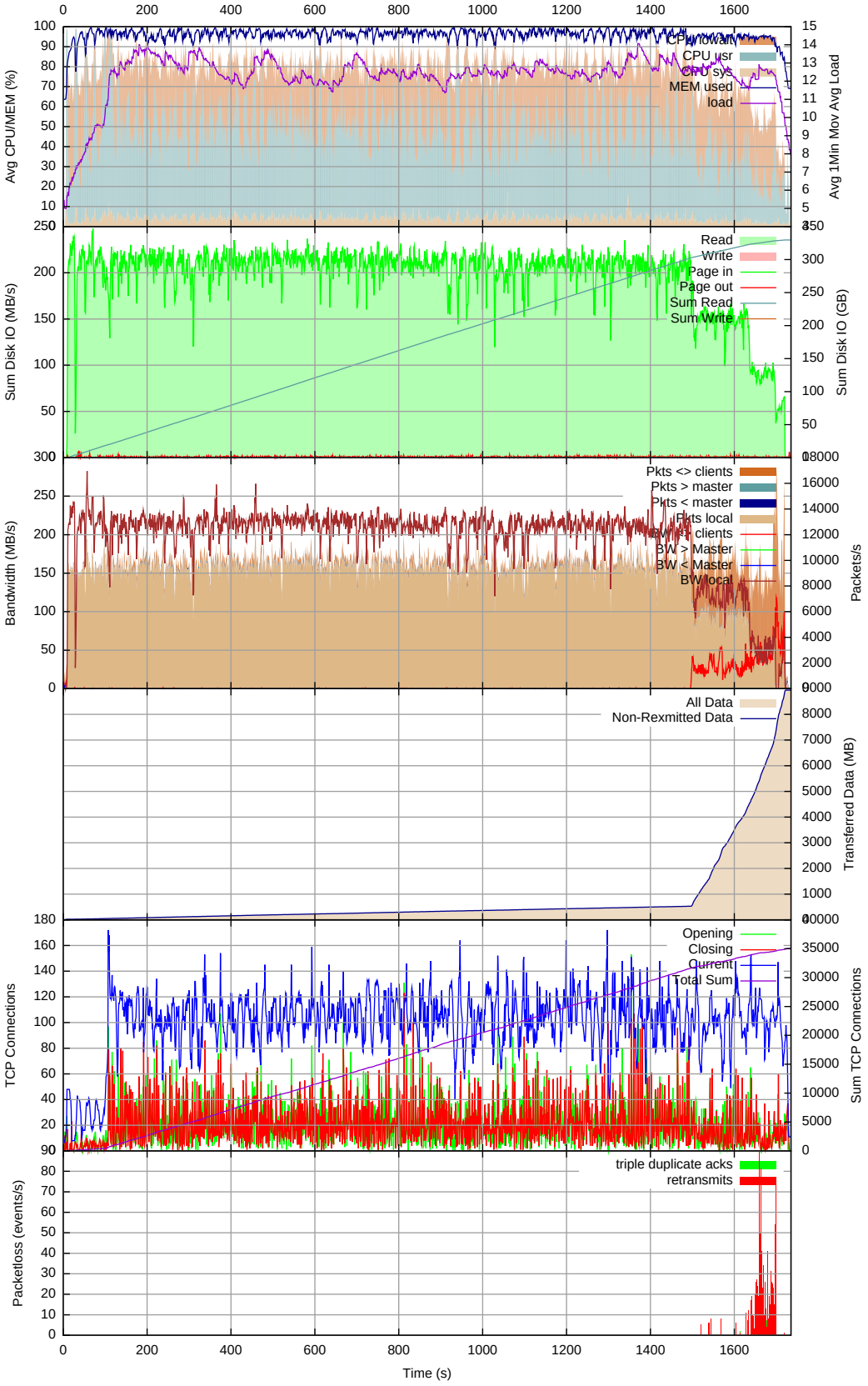

Abbildung E.5.: Hadoop Distinct Report

Literaturverzeichnis

[Bar+03] Corinne Baragoin, Alain Fournil, Andrea Hirata Miqui, George Latimer, Susan Schuster und Calisto Zuzarte. „Up and Running with DB2 UDB ESE: Partitioning for Performance in an e-Business Intelligence World“. In: *IBM Redbooks* (2003). URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg246917.pdf.

[Bär+11] Arian Bär, Antonio Barbuzzi, Pietro Michiardi und Fabio Ricciato. „Two Parallel Approaches to Network Data Analysis“. In: *ACM SIGOPS LADIS 2011*. 2011.

[Dat87] C. J. Date. „What Is a Distributed Database System?“ In: *Relational Database Writings* (1987).

[DG04] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: OSDI '04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, USA, Dez. 2004. URL: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/de//archive/mapreduce-osdi04.pdf.

[God12] Sebastien Godard. *The SYSSTAT Utilities*. 2012. URL: http://sebastien.godard.pagesperso-orange.fr.

[Kre12] Thomas Krenc. „Measurement and Characterization of Content Distribution in BitTorrent“. Master Thesis. Berlin, Germany: Technische Universität Berlin, März 2012.

[Ols+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar und Andrew Tomkins. „Pig Latin: A Not-So-Foreign Language for Data Processing“. In: *ACM SIGMOD 08*. 2008.

[Ost12] Shawn Ostermann. *tcptrace*. 2012. URL: http://http://www.tcptrace.org.

[Pav+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden und Michael Stonebraker. „A comparison of approaches to large-scale data analysis“. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD '09. New York, NY, USA: ACM, 2009, S. 165–178. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559865. URL: http://doi.acm.org/10.1145/1559845.1559865.

[Sto+10] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo und Alexander Rasin. „MapReduce and parallel DBMSs: friends or foes?“ In: *Commun. ACM* 53.1 (Jan. 2010), S. 64–71. ISSN: 0001-0782. DOI: 10.1145/1629175.1629197. URL: http://doi.acm.org/10.1145/1629175.1629197.

[Tra12] Transaction Processing Performance Council. *TPC-H*. 2012. URL: http://www.tpc.org/tpch/default.asp.

[Wik12] Wikipedia. *MapReduce*. 2012. URL: http://de.wikipedia.org/wiki/MapReduce.

[Wol12] Prof. Dr. M.-R. Wolff. *XML und Internet-Datenbanken*. 2012. URL: http://winfor.uni-wuppertal.de/fileadmin/wolff/Downloads/Hauptstudium/WIModul1/DBMS/WS05_06/XML_und_Internetdatenbanken.pdf.

Abbildungsverzeichnis