2012•2013
FACULTEIT WETENSCHAPPEN
master in de informatica: databases

Masterproef
Conjunctive Regular Path Queries in MapReduce

Promotor :
Prof. dr. Frank NEVEN

Bas Ketsman
*Masterproef voorgedragen tot het bekomen van de graad van master in de informatica ,
afstudeerrichting databases*

2012•2013
FACULTEIT WETENSCHAPPEN
master in de informatica: databases

Masterproef
Conjunctive Regular Path Queries in MapReduce

Promotor :
Prof. dr. Frank NEVEN

Bas Ketsman
Masterproef voorgedragen tot het bekomen van de graad van master in de informatica , afstudeerrichting databases

Abstract

In the current big data era, in which we are overloaded with huge amounts of data, there is a large demand for alternatives to traditional querying systems. In our context, big data refers to the petabyte scale data analysis to which industry and academia are exposed today; and, where hundreds or thousands of machines, running in parallel, are required to finish computations in a reasonable amount of time. However, the current data landscape also has a complex and non-traditional structure, which typically fits well into the graph model. In semi-structured data, the traditional relational query languages fall short. Hence, we consider the conjunctive regular path queries (CRPQs); a simple, but reasonably expressive language for querying graph data. This thesis is about the evaluation of CRPQs in MapReduce, a framework that provides a programming abstraction that enables the design of algorithms that can be executed automatically on a cluster of machines in a fault-tolerant way.

Acknowledgements

After a year of working on this thesis, I would like to express my special thanks to a lot of people. Foremost among them is of course Prof. Dr. Frank Neven, without whom this thesis would not have been possible. Thanks for the help to find this interesting topic, for the guidance to keep track on the subject, and for the bunch of papers that I would have missed out myself. I also thank Dr. Joris Gillis; for his helpful remarks, which enabled further refinement of this text.

Aside to content, a lot of time was spent on writing this thesis. Especially since I have never written more than two pages in English before. Therefore, I owe much to Laure Jacobs, whom I have harassed with questions on linguistic matters. Nevertheless, any remaining errors are mine alone.

Last but not least, I owe much to my family and friends; my family, for the opportunities they have given me, and their support and encouragement throughout my study; my friends, for the moments of distraction.

Dutch Summary
Nederlandse samenvatting

Deze thesis behandelt de evaluatie van conjunctieve reguliere pad queries (CRPQs) in MapReduce. Het doel van de thesis is drieledig. Ten eerste wordt er een omschrijving gegeven van de complexiteitsresultaten van CRPQs in de sequentiële context. Vervolgens wordt er een bespreking gegeven van MapReduce, enkele varianten, en de complexiteitsmaten en modelen die er voor MapReduce bestaan. In deze context worden er ook enkele algoritmen besproken voor gerelateerde problemen die reeds in de MapReduce literatuur voorkomen, zoals het berekenen van joins en het oplossen van het undirected s-t-connectivity probleem. Tenslotte worden de eerste twee delen gecombineerd om te komen tot algoritmen voor het evalueren van RPQs en CRPQs in MapReduce.

Big data

In het database onderzoek en de IT management literatuur is *big data* het modewoord van het moment. Big data is een subjectief, tijd- en contextafhankelijk begrip, maar met een gemeenschappelijke kern: het gaat over data die ons door zijn grootte verplicht om verder te kijken dan de beproefde technologie [37]. Want, hoewel hardware snel evolueert, zowel qua capaciteit als verwerkingsvermogen, groeit de hoeveelheid data waarmee we te maken krijgen nog veel sneller. Om hiermee om te kunnen gaan moet er dus verder worden gekeken dan de traditionele computer en de sequentiële algoritmen. Een populaire methode om met deze problematiek om te gaan is het distribueren van gegevens over een cluster van computers om hier vervolgens in parallel berekeningen op uit te voeren. Omdat het in het algemeen goedkoper is om 8 commodity computers te kopen dan één machine met 8 keer de capaciteit van een enkele computer [37], worden dit soort clusters ook in de praktijk veelvuldig gebruikt. Dit is ook een erg schaalbare strategie, omdat er makkelijk computers kunnen worden toegevoegd wanneer de nood aan capaciteit opnieuw toeneemt.

In onze context verwijst big data naar de verwerking van data waarvan de grootte in petabytes wordt uitgedrukt. Het zijn deze groottes waar de industrie en academische wereld vandaag mee te maken krijgen. Het gaat dan ook niet over clusters van 8 computers, maar over clusters van honderden of zelfs duizenden computers die met elkaar verbonden zijn in een netwerk [19]. Jammer genoeg klinkt het allemaal eenvoudiger dan het is. Parallellisatie van gegevens en distributie van data introduceren een heleboel moeilijkheden die er niet zijn in een sequentiële context. Zo is er nood aan communicatie, coördinatie en synchronisatie tussen de betrokken processen. Hier komt nog bij dat de kans dat er een fout optreed tijdens een berekening op een cluster van duizenden computers ongeveer 100% bedraagt door de grote hoeveelheid hardware componenten die erbij betrokken zijn. Daarom zijn fouttolerantie en fout afhandeling eveneens van onmisbaar belang. Kortom, het schrijven van een algoritme voor een dergelijke omgeving is allesbehalve evident.

MapReduce

In 2004 werd MapReduce gepubliceerd. MapReduce [19] is een programmeerabstractie met bijhorende implementatie, ontwikkeld door Google, waarbinnen algoritmen automatisch op een fouttolerante manier kunnen worden uitgevoerd op een dergelijke configuratie. MapReduce werd al intensief gebruikt bij Google, maar is sindsdien wijd in gebruik geraakt in de industrie en academische wereld via zijn opensource implementatie Hadoop [1]. Ruwweg wordt een MapReduce stap opgebouwd uit twee functies die door de gebruiker kunnen worden ingevuld: een map functie en een reduce functie. De map functie verdeelt de input in fragmenten, de reduce functie wordt in parallel op deze fragmenten uitgevoerd. In veel gevallen reduceert de reduce functie de gegevens waarop ze wordt uitgevoerd, maar dit is niet noodzakelijk. De elementaire basisblokken in een MapReduce programma zijn paren van de vorm $\langle key; value\rangle$. Hier representeert *value* de inhoudelijke data, *key* fungeert als een soort logisch adres. Zowel de input, output als intermediaire resultaten worden gezien als verzamelingen van dit soort paren. Meer formeel wordt een map functie gedefiniëerd als een functie die een individueel key-value paar afbeeldt op een verzameling van key-value paren. De reduce functie wordt gedefiniëerd als een functie die een verzameling van key-value paren, allen met een zelfde key, afbeeldt op een verzameling van key-value paren, waarbij alle paren nog steeds diezelfde key hebben.

Een MapReduce programma is een sequentie van MapReduce stappen waarin achtereenvolgens de volgende fasen plaatsvinden:

- MAP FASE: de map functie wordt uitgevoerd op ieder afzonderlijk paar. Dit gebeurt in parallel. Het resultaat is een verzameling van intermediaire key-value paren;
- SHUFFLE FASE: op basis van de *key* velden worden de paren gesorteerd en verdeeld in groepen waarin ieder paar dezelfde key heeft; en
- REDUCE FASE: de reduce functie wordt uitgevoerd op iedere groep van paren, ook dit gebeurt in parallel.

Merk op dat de shuffle fase een intensieve berekening inhoudt. Verder kan de shuffle fase pas plaatsvinden nadat ieder resultaat van de map functie is berekend. Daarom spreekt men in deze context over een *blocking property.*

Een typisch voorbeeld van een programma in MapReduce is Wordcount. Beschouw een verzameling van woorden gecodeerd als key-value paren van de vorm $\langle k; (woord)\rangle$. We kunnen nu gemakkelijk de woorden tellen door de map functie een paar $\langle k; (woord)\rangle$ te laten afbeelden op een paar van de vorm $\langle woord; 1\rangle$. De reducer streamt een groep van woorden en telt de achtervoegsels bij elkaar op. De output van de reducer is dus een tupel van de vorm $\langle woord; sum(input\ achtervoegsels)\rangle$ en vormt de oplossing voor het probleem. Er bestaan verschillende varianten voor dit algoritme. In plaats van woorden te groeperen op basis van het woord zelf, kan er bijvoorbeeld ook gebruik worden gemaakt van een hash functie, of zelfs een random partitionering. Mogelijk moet er dan wel een extra MapReduce stap worden geïntroduceerd. Jammer genoeg is niet ieder probleem zo gemakkelijk in MapReduce op te lossen als het Wordcount probleem. Meestal zorgt de noodzakelijke communicatie en coördinatie ervoor dat er in de praktijk heel wat extra overhead bij komt kijken.

Net als in de sequentiële context zijn we ook binnen MapReduce geïnteresseerd in de complexiteit van algoritmen. Het uitdrukken van de complexiteit en de bruikbaarheid van MapReduce algoritmen is echter niet triviaal. In een sequentiële context wordt typisch de hoeveelheid tijd en ruimte dat een algoritme nodig heeft gemeten, maar in een gedistribueerde context zoals MapReduce komen hier verschillende extra parameters bij kijken. Ten eerst is er het aantal stappen: hoe meer MapReduce stappen er moeten worden uitgevoerd, hoe vaker de shuffle fase moet plaatsvinden. Het spreekt voor zich dat het aantal MapReduce stappen dus best zo laag mogelijk wordt gehouden. Een andere belangrijke maat is de hoeveelheid communicatie die plaatsvindt. Ook hier spreekt het voor zich dat de hoeveelheid communicatie best zo laag mogelijk wordt gehouden. Ook het geheugen en de uitvoertijd zijn natuurlijk van belang: eenerzijds dat van individuele map functie en reduce functie instanties, anderzijds dat van het gehele programma.

Complexiteitsmaten vormen een belangrijk onderdeel in deze thesis. Hoewel er verschillende modellen en maten worden besproken, speelt met name de key- en sequentiële complexiteit [29] een belangrijke rol. De *key complexiteit* bestaat uit drie maten:

- de maximum grootte van de input en output van een mapper en reducer;
- de maximum uitvoertijd van een mapper of reducer; en
- het maximum geheugen dat door een mapper of reducer wordt gebruikt.

Ten tweede is er de *sequentiële complexiteit*, die een maat vormt voor de complexiteit van alle mappers en reducers in één MapReduce stap. De sequentiële complexiteit heeft twee delen:

- de grootte van alle inputs en outputs van mappers en reducers; en
- de totale uitvoertijd voor alle mappers en reducers.

Een ander model dat een belangrijk rol speelt in deze thesis is $\mathcal{MRC}$ [39] (MapReduce Class). Dit model probeert op design niveau grenzen te formuleren die de praktische uitvoerbaarheid van een algoritme moeten garanderen. Omdat zelfs een lineair aantal machines, of een lineaire hoeveelheid geheugen ten opzichte van de input grootte in onze context onrealistisch groot zijn, wordt er in dit model gewerkt met sublineare bovengrenzen. Formeel wordt het model als volgt gedefinieerd:

Definitie: Fixeer een $\varepsilon > 0$. Een MapReduce algoritme is in $\mathcal{MRC}^i$ als:

- het algoritme een correcte antwoord oplevert met kans minstens 3/4;
- iedere mapper μ een (gerandomiseerde) functie is, die kan worden geïmplementeerd op een RAM met woorden van grootte $\mathcal{O}(\log n)$, die $\mathcal{O}(n^{1-\varepsilon})$ ruimte gebruikt en slechts een polynomiale hoeveelheid tijd nodig heeft in n;
- iedere reducer ρ een (gerandomiseerde) functie is, die kan worden geïmplementeerd op een RAM met woorden van lengte $(\log n)$, waarvan de hoeveelheid ruimte begrenst is door $\mathcal{O}(n^{1-\varepsilon})$ en de tijd polynomiaal is in n;
- de totale hoeveelheid ruimte die wordt ingenomen door alle intermediare key-value paren (output van mappers) in een enkele MapReduce stap begrensd is door $\mathcal{O}(n^{2-2\varepsilon})$; en
- het aantal stappen begrensd is door $\mathcal{O}(\log^i n)$.

Noem $\mathcal{MRC} = \bigcup_{i \geq 0} \mathcal{MRC}^i$ en noem $\mathcal{DMRC}$ de deterministische variant. Het idee achter dit model is dat algoritmen die aan deze grenzen voldoen gegarandeerd kunnen worden uitgevoerd in MapReduce met slechts een sublinear aantal machines met ieder een sublineare hoeveelheid geheugen, ten opzichte van de input grootte.

Naast een omschrijving van MapReduce zelf worden er in deze thesis ook omschrijvingen gegeven van enkele meer recente MapReduce-achtige abstracties die in onze context interessant lijken. Het gaat over Pregel [47], GraphLab [46], PowerGraph [30], en een methode voor het gedistribueerd evalueren van Datalog programma's [5]. Deze abstracties worden beknopt besproken en dienen vooral als illustratie van de golf van abstracties die MapReduce in gang heeft gezet, alsook ter illustratie van de beperkingen in de MapReduce abstractie zelf, die deze modellen reeds trachten te verhelpen.

RPQs en CRPQs

Naast de grote hoeveelheid data, doet er zich in het huidige datalandschap een tweede probleem voor: data nemen vaak een complexe structuur aan. Typische voorbeelden hiervan zijn sociale media, zoals Facebook en Twitter, die bestaan uit complexe netwerken van zelf-omschrijvende structuren [23]. Er zijn ook tal van voorbeelden te vinden in andere domeinen, zoals de financiële wereld, de bioinformatica en de chemie. In het kort komt het erop neer dat de vorm van deze data niet toelaat dat ze op triviale wijze in het traditionele relationele model past. Daarom is het in de praktijk nu al vaak gebruikelijk om relationele databases aan te vullen met niet-relationele systemen, ook wel NoSQL databases genoemd. In onze context zijn we vooral geïnteresseerd in semi-gestructureerde data die op natuurlijke wijze binnen het graafmodel kan worden gemodelleerd.

Het kunnen ondervragen van data is essentieel om inzicht te kunnen krijgen in de inhoud of structuur ervan. SQL is de ondervragingstaal bij uitstek binnen het relationele model, maar schiet tekort in expressiviteit voor de ondervraging van graafdata.

Met name pad expressies ontbreken. De vraag welke querytalen het meest geschikt zijn voor het ondervragen van semi-gestructureerde data is echter niet nieuw [60]. De ondervragingstaal die populair is in deze context is een combinatie van de conjunctieve queries (CQs), het in de praktijk meest gebruikte deel van SQL, en reguliere pad queries (RPQs). De Conjuntieve reguliere pad queries (CRPQs) zijn voor semi-gestructureerde data ongeveer hetzelfde als wat CQs zijn voor het relationele model. Met andere woorden, een taal met voldoende uitdrukkingskracht om alledaagse vragen te kunnen formuleren, maar tegelijkertijd ook een taal met heel wat beslisbare eigenschappen. Dit verklaart de grootte interesse voor CRPQs binnen de wetenschappelijke gemeenschap [11, 17, 41, 61]. De expressiviteit van CRPQs wordt bovendien (gedeeltelijk) omvat in query talen die in de praktijk worden gebruikt voor de ondervraging van graafdata, zoals SPARQL, voor de ondervraging van semantisch web data, en XPath/XQuery voor de ondervraging van XML.

Syntactisch wordt een CRPQ als volgt gedefiniëerd.

Definitie: Een *conjunctieve reguliere pad querie* (CRPQ) over een eindig alfabet Σ is een expressie van de vorm:

$$ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, r_i, y_i),$$

waar $m > 0$, iedere x_i en y_i is een knoop-variabele, iedere r_i is een reguliere expressie over Σ, en iedere z_i in $\bar{z}$ is een x_j of y_j.

Voorbeeld: Gegeven een graaf waarin een boog van knoop u naar knoop v met label *ouder* aangeeft dat v een ouder is van u. We kunnen de relatie vinden van personen met gemeenschappelijke voorouders via de volgende CRPQ :

$$ans(u, w) \leftarrow (u, ouder^*, z) \wedge (w, ouder^*, z).$$

In deze thesis worden de syntax, semantiek en complexiteit van conjunctieve queries in de sequentiële context besproken. In het bijzonder wordt er aandacht besteed aan de verschillende manieren om complexiteit uit te drukken: data complexiteit, expressie complexiteit, en gecombineerde complexiteit. Het verschil zit in de elementen op basis waarvan de complexiteit wordt uitgedrukt, dit wil zeggen, enkel de grootte van de input, enkel de grootte van de query, of de combinatie van beiden. In het bijzonder wordt de NP-compleetheid (gecombineerde-complexiteit) en AC^0 bovengrens (data complexiteit) van conjunctieve queries besproken. Vervolgens behandelen we een gemakkelijker deel van de conjunctieve queries, namelijk de acyclische conjunctieve queries. Hiervan behandelen we de PTIME bovengrens. Voor CRPQs doen we hetzelfde: we behandelen de NP-compleetheid binnen de gecombineerde-complexiteit en de NL-compleetheid binnen de data-complexiteit. Vervolgens wordt er ingegaan op de acyclische conjunctieve reguliere pad queries die NL-compleet zijn wat betreft gecombineerde-complexiteit.

CQs en USTCON in MapReduce

CRPQs worden op het moment van schrijven niet in de literatuur rond MapReduce bestudeerd. Daarom wordt er ingegaan op algoritmen voor enkele gerelateerde problemen die wel reeds in de literatuur aan bod komen.

In de eerste plaats zijn dit CQs en joins. In de MapReduce context encoderen we een relatie instantie steeds als een verzameling MapReduce paren, waarbij ieder tuple uit een relatie overeenkomt met een paar, waarvan de *value* zowel het tuple als de relatienaam omvat. We bespreken hoofdzakelijk twee algoritmen voor het uitvoeren van joins in MapReduce. De meest natuurlijke manier om joins in de context van MapReduce uit te voeren is door, via de map functie, alle paren te groeperen op basis van de waarde van hun join-attribuut. Op deze manier wordt de input opgesplitst in strikte fragmenten waartussen er geen join moet worden berekend. De reduce functie voert vervolgens de join uit op paren binnen zo een fragment. Het probleem met deze methode is dat, wanneer de frequenties van waarden voor de join-attributen sterk afwijken van elkaar, ook de groottes van de fragmenten sterk zullen verschillen. Mogelijk zijn bepaalde fragmenten zelfs groter dan een enkele machine kan verwerken, wat maakt dat deze methode enkel praktisch bruikbaar is als er een begrenzing op de frequenties bestaat. Het alternatief is een benadering die de reducers centraal plaatst. Hierbij treed er een overhead op in de communicatie doordat er overlap ontstaat tussen de fragmenten die worden beschouwd, maar we bezitten wel over garanties over de hoeveelheid paren die door een individuele reducer zullen moeten worden verwerkt.

Verder werden er twee eenvoudige graaf problemen bestudeerd. In het bijzonder is dit het *undirected s-t-connectivity* (USTCON) probleem, dat gegeven een ongerichte graaf vraagt of twee knopen s en t in de graaf met elkaar verbonden zijn. Ook het gerelateerde *finding connected components* probleem, waarbij er gezocht wordt naar de verzamelingen van knopen in een graaf die met elkaar geconnecteerd zijn, wordt behandeld.

RPQs en CRPQs in MapReduce

Het laatste onderdeel bestaat uit een studie van de evaluatie van RPQs en CRPQs in MapReduce zelf, door de inzichten die werden bekomen uit de domeinstudie rond CRPQs in de sequentiële context, MapReduce, de complexeitsmaten, en de algoritmen voor gerelateerde problemen te combineren. In het bijzonder worden er algoritmen voorgesteld en geanalyseerd die gebaseerd zijn op de partiële evaluatie van padqueries door gebruik te maken van automaten. Er wordt eveneens ingegaan op acyclische CRPQs in deze context. In het algemeen kan geconcludeerd worden dat het uitvoeren van joins in MapReduce niet evident is. De kost van het berekenen van multi-way joins neemt dan ook sterk de bovenhand in onze analyse.

Conclusie en vooruitzicht

De klassieke complexiteitsgrezenden voor de evaluatie van CQs, RPQs, en CRPQs uit de traditionele sequentiële context worden besproken. Zowel voor het algemene probleem als voor de acylische tegenhangers. Het MapReduce model en de verschillende complexiteitsmaten en modellen worden besproken. Er worden eveneens enkele alternatieven besproken die bepaalde problemen in de MapReduce abstractie aanpakken, in het bijzonder het verwerken van grafen. In aanloop naar de evaluatie van RPQs en CRPQs werden er enkele algoritmen bestudeerd die reeds in de literatuur in MapReduce werden onderzocht. Tenslotte werden de inzichten uit de voorgaande delen gebundeld om te komen tot algoritmen voor de evaluatie van RPQs en CRPQs.

Bepaalde delen van de literatuur werden niet behandeld. Zo worden er enkel complexiteitsresultaten gegeven die nuttig lijken in onze context. We gaan er steeds vanuit dat queries worden geëvalueerd zoals ze zijn. Er worden geen technieken voor het herschrijven van queries behandeld, die in de praktijk natuurlijk wel een belangrijke rol spelen. In de literatuur wordt er reeds aandacht besteed aan de relaties tussen MapReduce en andere theoretische modellen, zoals PRAM, BSP en streaming modellen. Omdat we niet verwachtten dat hier voor onze context bruikbare resultaten uit te halen waren, werd dit deel van de MapReduce literatuur eveneens niet behandeld. Verder wordt er een selectie aan MapReduce varianten besproken, maar er bestaan er natuurlijk veel meer. Tenslotte, door een gebrek aan ondergrenzen hebben we geen informatie over de nauwheid van de complexiteitsgrenzen voor de evaluatie algoritmen die werden geformuleerd. Het uitvoeren van experimenten had mogelijk een beter inzicht in de praktische bruikbaarheid kunnen opleveren. Er werd echter niet beschikt over een dataset die tot een relevante analyse kon leiden. In het algemeen lijkt er wel nog heel wat verbetering mogelijk te zijn, al dan niet binnen MapReduce.

Hoewel MapReduce veelvuldig wordt gebruikt in de praktijk, heeft het model ook heel wat tekortkomingen. Hieronder vallen ondermeer de hoge kost voor het berekenen van joins, en de moeilijkheden in het omgaan met graafdata; beiden onmisbaar in de context van CRPQs.

MapReduce heeft het onderwerp van gedistribueerde en parallelle berekeningen nieuw leven ingeblazen en een golf van abstracties in gang gezet. Verschillende van deze nieuwe abstracties trachten reeds tekortkomingen van MapReduce aan te pakken. Daarom zien we MapReduce eerder als een beginpunt dan een eindpunt. We verwachten dat er nog heel wat modellen zullen volgen.

Contents

List of Figures

List of Algorithms

Nomenclature

$\Gamma(.)$	set of neighbour nodes
$\Delta(G)$	upper-bound on degree of graph G
δ	transition relation
η	path in a graph
θ	accepting run
ϑ	valuation function
$\lambda(.)$	labeling function
μ	map function
μ	mean
π	ordering
ρ	reduce function
Σ	alphabet
A	NFA
a	edge label ($\in \Sigma$)
d	the diameter of a graph
$deg(.)$	degree (incoming degree $deg_i(.)$, or outgoing degree $deg_o(.)$)
E	set of edges in a graph
$E_{v\to *}$	set of outgoing edges of v
$E_{*\to v}$	set of incoming edges of v
F	set of accepting states in automaton
G	graph

$\mathbf{I}, \mathbf{J}$	db instances
$L(.)$	language
Q	set of states in automaton
q	query
$\mathbf{R}$	db schema
R, S, T	db relations
R_r	replication rate
s	state
s_0	initial state
$\bar{s}, \bar{t}$	tuples of constants
u, v, w	nodes
$\bar{u}, \bar{v}, \bar{w}$	tuples of nodes
V	set of nodes in a graph
X, Y, Z	attributes
x, y, z	variables
$\bar{x}, \bar{y}, \bar{z}$	tuples of variables (and/or constants)

1 Introduction

Big data; in current database research and IT management literature it is all about *big data.* But what is big data? It is a subjective, time, and context dependant buzzword. However, there also is a common crux:

> "Big data is data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time [37]."

Although hardware capacity evolves quickly, the amounts of data we are faced with tend to grow even faster. Therefore, this definition perfectly catches the challenges of processing big data.

Nowadays, it is generally cheaper to buy 8 off-the-shelf commodity machines than one with 8 times the capacity of a single machine [37]. Hence, it is a common strategy to use large clusters of cheap commodity machines connected in a network in order to evade the limitations of a single machine. In our context, big data refers to the petabyte scale data analysis to which industry and academia are exposed today; and, where hundreds or even thousands of machines, running in parallel, are required to finish computations in a reasonable amount of time [19]. Unfortunately, parallelization and distribution of data also brings a lot of drawbacks: the need for coordination, communication, and synchronization are only a few of the causes that make a simple algorithm extremely hard to implement. Furthermore, in a cluster of this size, where the chance of possible errors is near 100 percent, fault tolerance and error handling are of inevitable importance. In 2004, Google introduced MapReduce [19], a framework that provides a programming abstraction that enables the design of algorithms that can be executed automatically on a cluster of machines in a fault-tolerant way. MapReduce has been used intensively at Google, and has since widely been adopted in industry and academia throughout its open-source implementation, Hadoop [1].

Although size matters, much data in the current data landscape also has a complex and non-traditional structure. Social media that consist of complex, interlinked networks of self-describing structures [23], like Facebook and Twitter, are typical examples. Other challenging examples can be found in, for example, finance, bioinformatics, and chemistry. It is already common to complement traditional databases with NoSQL databases (i.e., non-relational systems) in order to handle the variety of elements in the current data landscape; especially semi-structured data, which naturally fits into the graph model. Semi-structured data is not new, nor is the question how to query it [60]. SQL, and in particular the conjunctive part, is the main query language of interest in relational databases. In semi-structured data, however, these languages fall short due to a lack of navigational expressiveness. Therefore, ubiquitous, large graph data has led to a renewed interest in path queries and graph pattern matching languages [61]. In this context, regular path queries (RPQs) and conjunctive regular path queries (CRPQs) are considered. CRPQs are reasonably expressive, but still have a lot of decidable properties, which explains the great interest of the scientific community [11, 17, 41, 61]. Furthermore, CRPQs are (partially) subsumed by many graph and tree languages that are used in practice; like SPARQL [34], the default language for querying semantic web data; and XPath/XQuery, which is the default language for querying XML.

Driven by size and shape of the data to which we are exposed, the challenges of cluster computing, and the recent wave of abstractions to handle them, the goal of this thesis is threefold:

1. understanding the established research of CQs and CRPQs in the (traditional) sequential context;

2. performing a domain study of MapReduce, its properties, and complexity measures; and

3. designing and analyzing algorithms for the evaluation of CRPQs in MapReduce by combining the knowledge gained from the previous parts.

Thesis Outline

In Chapter 2, the background concepts that are heavily used throughout this thesis, are introduced. Part I contains a domain study of conjunctive queries (CQs), regular path queries (RPQs) and conjunctive regular path queries (CRPQs) in a sequential context. Chapter 3 covers the syntax and semantics of CQs together with the classical complexity results. In Chapter 4 an analogous study is performed on CRPQs. Part II is twofold: it consists of the domain study of MapReduce and MapReduce-like environments; and introduces algorithms to evaluate CRPQs on large graphs with MapReduce. First, in Chapter 5, a theoretical study of MapReduce is performed, including a description of its complexity measures. Since RPQs and CRPQs are not yet studied in the context of MapReduce, some related problems are studied in the MR abstraction. In particular, in Chapter 6, the evaluation of conjunctive queries and joins is studied in MapReduce. In Chapter 7, a similar analysis is performed for the problem of undirected s-t-connectivity and the related problem of finding connected components. MapReduce has initiated a wave of parallel abstractions. Some of these MapReduce extensions or alternatives are discussed in Chapter 8. Finally, in Chapter 9 and Chapter 10, algorithms for the evaluation of RPQs and CRPQs are introduced. In Chapter 11 the conclusions and outlook are given.

2 Preliminaries

The aim of this chapter is to give a brief overview of the theoretical results that are used throughout this thesis. First, a short description of the relational and graph model are provided. Second, the complexity of queries in a sequential setting is discussed, including the parallel complexity class AC^0. Next, an application of Chernoff bounds is described, which we will use several times. Finally, some basic notations of regular expressions are covered.

2.1 Relational Databases

When referring to a database, we usually mean a database in the relational model which was introduced by Codd. Formally, a relational database is an instance of a database schema. A *database schema* $\mathbf{R}$ specifies the structure of a database and is defined as set of relation schemas $\{R_1, R_2, \ldots, R_2\}$. In turn, a *relation schema* R_i is a set of attributes $\{X_1, X_2, \ldots, X_n\}$, where R itself is called the relation name. Furthermore, a countable and infinite set of constants **dom** is defined as the underlying domain. Since we only make use of the equality relation, the exact shape of the structure associated to **dom** is unimportant in our context.

A *database instance* $\mathbf{I}$ over a database schema $\mathbf{R}$ is an image that maps every relation schema $R \in \mathbf{R}$ on a finite set of tuples over R, written $\mathbf{I}(R)$. A tuple $\bar{t}$ over R is an image from the set of attributes in R on **dom**. Moreover, when talking about constants we mean elements in **dom**.

In the context of queries defined over a database schema $\mathbf{R}$ we sometimes refer to instances that are *compatible* to a query q. By this we mean instances from the domain associated to $\mathbf{R}$. In addition, when referring to a tuple compatible to a query q, we also assume that this tuple has the same arity as the output relation of q.

2.2 Semi-Structured Data and Graphs

Data in the relational model is highly structured. Munch of the data in the current data landscape, however, does not conform to this formal structure. We use the term *semi-*

structured data to denote any data in between raw and, highly structured, relational data. A natural way to model semi-structured data is by means of a graph, or graph database [10].

A graph is a mathematical structure defined as a pair (V, E), where V is a finite set of nodes and $E \subseteq V \times V$ is a set of edges between these nodes. A graph can be directed or undirected. Most of the time we refer to directed graphs, i.e, graphs where edges are ordered pairs of nodes. When referring to an undirected graph, we assume that for every edge in E, the other direction is available in E as well.

Sometimes, we consider graphs where the edges or nodes are labeled. A Σ-labeled graph is a graph where Σ represents an underlying finite domain of labels. Formally, labels are associated to nodes or edges by means of a labeling function. Hence, we define a labeling function on nodes $\lambda_v : V \to \Sigma$ as a total function on the nodes in G that maps every node on its associated label. Analogous we define a labeling function on edges $\lambda_e : E \to \Sigma$ as a total function on the edges in G that maps every edge on its associated label. When it is clear from the context λ_e and λ_v are shorthanded by λ to denote the appropriate function.

A path η in G from v_1 to v_k is a sequence of edges between v_1 and v_k, or equivalent, a sequence of nodes $v_1v_2v_3 \dots v_{k-1}v_k$ such that $(v_i, v_{i+1}) \in E$ for each $i \in [1, k-1]$. For notational simplicity we sometimes use an alternative definition for edges that combines the end-nodes and the labeling function on edges, i.e., $E \subseteq V \times \Sigma \times V$. We extend λ_e to paths $\eta = e_1e_2 \dots e_k$ in the natural way, that is, $\lambda_e(\eta) = \lambda_e(e_1e_2 \dots e_k) = \lambda_e(e_1)\lambda_e(e_2) \dots \lambda_e(e_k)$, where $e_i \in E$ for every $i \in [1, k]$.

Let $G = (V, E)$ be an undirected graph. We use the notation $\Gamma(v)$, where $v \in V$ to denote the set of neighbours of v. The degree of v, written $deg(v)$, is the number of adjacent edges of v. In a directed graph, we typically make a distinction between the incoming edges, i.e., $deg_i(v)$, and the outgoing edges, i.e., $deg_o(v)$. We use $\Delta(G)$, or simply Δ when G is clear from the context, to denote the maximum degree of a node in G.

2.3 Query Complexity

There are three different ways to evaluate the complexity of queries [59]: data complexity, expression complexity[1] and combined complexity. The difference between these types is in the way the complexity is expressed; that is, in the size of the instance, the size of the query, or the size of both the query and the instance. As usual, the complexity is measured in terms of a decision problem instead of the evaluation itself. For query languages the decision problem is called the recognition problem. More formally, for relational query language Q the recognition problem of Q is defined as the problem to decide whether $\bar{t} \in q(\mathbf{I})$, where $\mathbf{I}$ is a relation instance, q a query in Q, and $\bar{t}$ a tuple.

We define the *data complexity* of a relational query language Q as the complexity of its recognition problem where q is fixed, i.e., the complexity of Q-EVAL(q).

[1]Expression complexity is sometimes called query complexity, but its value depends more on the expression than on the actual query [59].

PROBLEM:	Q-EVAL(q)
INPUT:	An instance $\mathbf{I}$ and tuple $\bar{t}$ that are compatible with q.
OUTPUT:	$\bar{t} \in q(\mathbf{I})$?

The *expression complexity* of a relational query language Q is defined as the complexity of its recognition problem where $\mathbf{I}$ is fixed. That is, the complexity of Q-EVAL$(\mathbf{I})$.

PROBLEM:	Q-EVAL$(\mathbf{I})$
INPUT:	A query $q \in Q$ and a tuple $\bar{t}$ that is compatible with q.
OUTPUT:	$\bar{t} \in q(\mathbf{I})$?

Eventually, the *combined complexity* of a relational query language Q is the complexity of its recognition problem, where no input parameters are fixed. We define the corresponding decision problem as Q-EVAL.

PROBLEM:	Q-EVAL
INPUT:	An instance $\mathbf{I}$, a query $q \in Q$ and a tuple $\bar{t}$ compatible with q.
OUTPUT:	$\bar{t} \in q(\mathbf{I})$?

The data complexity, expression complexity, and combined complexity for graph query languages is defined in an analogous way. For convenience, the definition of combined complexity for queries over graph databases is given below.

PROBLEM:	Q-EVAL
INPUT:	Let Σ be a finite alphabet, G a graph database over Σ, $q \in Q$ a query, and $\bar{v}$ a tuple of nodes in G compatible with q.
OUTPUT:	$\bar{v} \in q(G)$?

2.4 AC^0

A *boolean circuit* C is a directed acyclic graph with n input nodes (wit no incoming edges) and 1 output node (with no outgoing edges). Every non-input node is called a gate and is labeled with a logical connective OR, AND, or NOT. The size of C is defined as its number of nodes.

The evaluation of a boolean circuit C over input $x \in \{0,1\}^n$ is defined inductively as follows. The ith input node evaluates to the ith input value. Every non-input node evaluates to the value corresponding to the evaluation of the associated connective on the values on its incoming wires.

It is important to note that Boolean circuits are non-uniform models of computation. Nevertheless, it is possible to place a uniformity condition on the circuits. A circuit family $\{C_n\}$ is *logspace-uniform* if there is an implicitly logspace computable function mapping 1^n to the description of circuit C_n [9].

Definition 2.1: For every i, Q-EVAL(q) is in AC^i, if it can be decided by a logspace-uniform family of circuits $\{C_n\}$ where every C_n has a polynomial size in n and depth $\mathcal{O}(log^i n)$. An unbounded amount of input wires is allowed for OR and AND gates.

In particular we consider AC^0, in which the depth of the circuits is constant. AC^0 is contained in the complexity class L ; i.e., the class of logspace algorithms.

2.5 Chernoff Bounds

Let $X_1, X_2, \dots, X_m$ be independent Bernoulli trails, i.e., a set of experiments where the output can be either 1 or 0. Let p_i be the chance that $X_i = 1$ and $1 - p_i$ be the chance that $X_i = 0$. Then let $X = \sum_{i=1}^{m} X_i$ and E$[X]$ be the expectation of X. Moreover:

$$\mathrm{E}[X] = \sum_{i=1}^{m} p_i.$$

We also write μ to denote the expectation.

In the context of randomized algorithms it is a frequent necessity to show that properties are true with high probability. Chernoff bounds provide a tool that is perfectly suitable for this task. In particular, in our context we use Chernoff bounds to prove that $X > (1 + \delta)\mu$, for a given δ .

There is no such thing as a Chernoff bound, rather, Chernoff bounds are applications of Markov's inequality (i.e., $\Pr[X \geq a] \leq \frac{\mathrm{E}[X]}{a}$, where a is a non-negative integer) to e^{tX} for some wel chosen t. The complete theory of Chernoff bounds would lead us to far. Therefore, we refer to [49] and [48] for a more in detail overview. Nevertheless, it is a classic result that for any $\delta > 0$

$$\Pr[X \geq (1 + \delta)\mu] < \left(\frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}} \right)^{\mu}. \tag{2.1}$$

Moreover, when $\delta \geq 2e - 1$, Equation (2.2) holds as well:

$$\left(\frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}} \right)^{\mu} \leq \left(\frac{e^{(1+\delta)}}{(1 + \delta)^{(1+\delta)}} \right)^{\mu} \leq \left(\frac{1}{2} \right)^{\mu(1+\delta)}, \tag{2.2}$$

which leads to a simplified version of Equation (2.1) when $\delta \geq 2e - 1$:

$$\Pr[X > (1 + \delta)\mu] < 2^{-\mu(1+\delta)}.$$

2.6 Regular Expressions and NFAs

Let Σ be an alphabet. When referring to regular expressions we mean the simple regular expressions that are typically used in literature. ε and $a \in \Sigma$ are regular expressions. Furthermore, regular expressions are closed under concatenation, set union and the Kleene star operation, i.e., $r_1 r_2$, $r_1|r_2$, and r_1^* are regular expressions. We denote the language containing every string over Σ that is defined by a regular expression r with $L(r)$.

It is a classical result that the languages defined by a regular expressions are equivalent to the languages that are expressible by Nondeterministic Finite Automata (NFAs). We define an NFA A as a 5-tuple $(\Sigma, Q, \delta, s_0, F)$, where Σ is the alphabet, Q a set of states, $\delta : Q \times \Sigma \times Q$ a transition relation, $s_0 \in Q$ a start state, and $F \subseteq Q$ the accepting states. We usually express the size of an NFA in terms of its states, i.e., $|Q|$. More particular, in our context it is important that:

- A regular expressions r can be transformed into a (ε-free) NFA in $\mathcal{O}(|r| \log |r|)$ time [35], where $|r|$ denotes the length of r.
- Let A_1 and A_2 be NFAs. We are able to generate an NFA A that recognizes the intersection of languages recognized by A_1 and A_2, $L(A) = L(A_1) \bigcup L(A_2)$, in polynomial time such that A is polynomial in the size of A_1 and A_2.

A *run* of A is a sequence of states that agrees to the transition relation of A. An *accepting run* of A, write θ, is a run that starts in s_0 and ends in a state from F.

Part I

Sequential Context

3 Conjunctive Queries

A conjunctive query is a very simple query that, despite its limited expressiveness, allows to formulate every day queries. Moreover, the language of conjunctive queries forms the basis of several mayor query languages, making it the subject of considerable research. In this chapter we study the complexity of the conjunctive queries and an important restriction, the acyclic conjunctive queries, in a traditional, sequential setting.

3.1 Syntax and Semantics

Conjunctive queries appear in several forms. One such form, closely related to the syntax used for conjunctive regular path queries in the next chapter, is given below. An extended overview can be found in [2]. For historical reasons we will only consider the relational case for now.

Definition 3.1: Let $\mathbf{R}$ be a database schema. A *conjunctive query* (CQ) over $\mathbf{R}$ is an expression of the form

$$ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m} R_i(\bar{x}_i),$$

where $m \geq 0$, $R_1, \ldots, R_m$ relation names in $\mathbf{R}$ and $\bar{z}, \bar{x}_1, \ldots, \bar{x}_m$ tuples of variables and constants that have appropriate arities. Every variable in $\bar{z}$ has to appear at least once in $\bar{x}_1, \ldots, \bar{x}_m$.

Example: Let $Parent(X_1, X_2)$ and $Male(Y)$ be database relations. It is easy to formulate a conjunctive query that asks for the grandfather relation:

$$ans(x, z) \leftarrow Parent(x, y) \wedge Parent(y, z) \wedge Male(z).$$

We call $ans(\bar{z})$ the *head* of the query and $\bigwedge_{1 \leq i \leq m} R_i(\bar{x}_i)$ the *body* of the query. A conjunctive query is a *Boolean conjunctive query* (BCQ) if its head has arity 0.

A valuation ϑ over a set V of variables is a total function from V to the set **dom** of constants. This function can be extended with the identity of **dom** and further extended to tuples in the natural fashion (i.e., $\vartheta(\bar{x}) = (\vartheta(x_1), \ldots, \vartheta(x_k))$ where $\bar{x} = (x_1, \ldots, x_k)$).

Definition 3.2: Let q be a conjunctive query over $\mathbf{R}$ of the form $ans(\bar{z}) \leftarrow R_1(\bar{x}_1) \wedge \cdots \wedge R_m(\bar{x}_m)$ and $\mathbf{I}$ an instance over $\mathbf{R}$. The *image* of q over $\mathbf{I}$ is defined as follows:

$$\begin{aligned} q(\mathbf{I}) = \{\vartheta(\bar{z}) \mid\ & \vartheta \text{ a valuation over the set of variables in } q \\ & \text{en } \vartheta(\bar{x}_i) \in \mathbf{I}(R_i), \text{for all } i \in [1, m]\}. \end{aligned}$$

The *active domain* of an instance $\mathbf{I}$ or a query q is the finite subset of **dom** that contains every element of **dom** occurring in $\mathbf{I}$ or q.

One of the reasons why the language of CQs is so popular in literature is that its containment and equivalence problem are decidable (NP-complete [17] to be precise), in contrary to more expressive languages such as SQL. Containment and equivalence are especially useful in the context of query optimization.

3.2 Data Complexity

It is not hard to see that a conjunctive query over a database instance can be evaluated in a logarithmic amount of space in the size of the input instance [2]. We only have to iterate the possible values of body atoms by using pointers to the corresponding elements in the input instance. Consequently, CQ-EVAL(q) $\in$ LOGSPACE. It turns out, however, that we can further restrict this complexity result to AC^0.

Theorem 3.1: CQ-EVAL$(q) \in \text{AC}^0$ [2].

Proof: Let $\mathbf{R} = \{R_1, \ldots R_k\}$ be a relational schema, q a fixed query over $\mathbf{R}$, $\mathbf{I}$ a database instance over $\mathbf{R}$, and $\bar{t}$ a tuple compatible with q.

Assume $\mathbf{J}$ to be the maximal instance over $\mathbf{R}$ with active domain the union of the active domain of $\mathbf{I}$ and q. Let n be the size of $\mathbf{J}$. Construct an input gate for each pair $\langle R, s\rangle$, where $R(\bar{x})$ is an atom in the body of q and $\bar{s}$ is a tuple consisting of constants from the active domain of $\mathbf{I}$ and q. We assume $\bar{s}$ to have the same arity as $\bar{x}$. Also construct the output gate associated to $\langle ans, \bar{t}\rangle$.

Next, we build the Boolean circuit network from output to input. Connect the output gate to a newly added OR-gate. This OR gate will connect the distinct configurations of existentially quantified variables in the body of q. Connect to this OR-gate a set of newly added distinct AND-gates, where each AND-gate is associated to a single configuration of existentially quantified variables compatible to $\bar{t}$. Every AND-gate is connected to the associated input gates. An example construction is illustrated in Figure 3.1.

The depth of the resulting circuit is fixed. The circuit size (i.e., the number of gates) is polynomial in the size of the input. It is not hard to see that this circuit can be constructed by a TM on input 1^n in logspace. □

Theorem 3.1 is not only applicable to conjunctive queries; in fact, the evaluation problem for the entire domain calculus is in AC^0 [2].

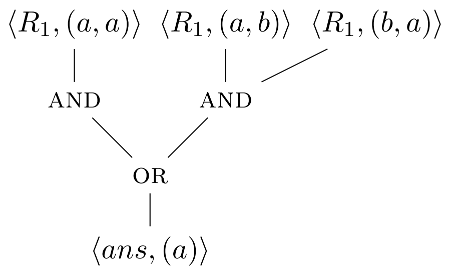


Figure 3.1: Example circuit for query $ans(x) \leftarrow R_1(x,y) \wedge R_1(y,x)$ over database schema $\mathbf{R} = \{R_1\}$, domain {a,b} and tuple $t = (a)$.

3.3 Combined Complexity

In [17] it is proven that every Boolean conjunctive query is logspace complete in NP with respect to combined complexity, but it is easy to generalize this proof to the language of (not necessary boolean) conjunctive queries.

Theorem 3.2: CQ-EVAL is logspace complete in NP.

Proof: Let q be an arbitrary CQ, $\mathbf{I}$ a compatible instance, and $\bar{t}$ a tuple compatible to q. A witness for the recognition problem $\bar{t} \in q(\mathbf{I})$ is a valuation ϑ such that $\vartheta(\bar{z}) = \bar{t}$ and $\vartheta(\bar{x}_i) \in \mathbf{I}(R_i)$ for each $i \in [1, m]$. The verification of both conditions clearly is possible in time polynomial in the size of the input. A valuation is a function that maps precisely the variables in q on the constants in **dom**, and thus has a fixed size. Now, it is not hard to see that there exists a non-deterministic TM that guesses ϑ and checks if ϑ is a witness for $\bar{t} \in q(\mathbf{I})$ in a polynomial amount of time.

Completeness can be derived via a reduction from k-clique, say $\langle k, G\rangle$, which asks fo the existence of a complete subgraph of size k in graph G. First, assume that G has at least k edges. This can be verified in a logarithmic amount of space. Second, construct the instance $\mathbf{I}$ over a binary relation R and construct a Boolean query q that represents a k-clique pattern as below:

$$\begin{aligned}
ans() \leftarrow R(x_1, x_2) \wedge R(x_1, x_3) \wedge \ldots\ & \wedge R(x_1, x_k) \\
\wedge R(x_2, x_3) \wedge \ldots\ & \wedge R(x_2, x_k) \\
& \ \vdots \\
& \wedge R(x_{k-1}, x_k).
\end{aligned}$$

The construction of instance $\mathbf{I}$ requires only a logarithmic amount of space. To achieve this, iterate through the edges in G by means of a pointer, and construct two tuples (one for each direction) in $\mathbf{I}$ for each edge. The construction of q also requires only a logarithmic amount of space by means of two pointers of size $\mathcal{O}(\log k)$. Since we assumed k to be greater than the number of edges in G $\mathcal{O}(\log k)$ implies $\mathcal{O}(\log n)$, where n is the size of the input. If $|G| < k$ we return a trivial untrue combination, for example $() \leftarrow R(a)$ and instance $R(\mathbf{I}) = \{\}$. $\square$

Alternatively, NP-completeness follows immediately from the NP-completeness of the containment problem for conjunctive queries combined with the homomorphism theorem [2] which states that the containment problem $q \subseteq q'$ for CQs is equivalent to the problem $\bar{z} \in q'(\mathbf{I})$ where $\mathbf{I}$ is the canonical instance of q and $ans(\bar{z})$ the head-atom of q. In [44] the completeness is derived via a reduction to 3-COLOR.

3.4 Expression Complexity

Theorem 3.3: CQ-EVAL($\mathbf{I}$) is logspace complete in NP.

The proof is completely analogues to the one presented for the combined complexity of CQs in the previous section. Since $\mathbf{I}$ is fixed, $\vartheta(\bar{x}) \in \mathbf{I}(R)$ is verifiable in a constant amount of time.

3.5 Acyclic Conjunctive Queries

In the previous sections, proofs are given for the intractability of the evaluation problem for conjunctive queries, even when restricted to Boolean conjunctive queries. In this section we consider an important subclass of CQs where the complexity of the evaluation problem is tractable.

A cause of the NP-completeness of the evaluation problem for Boolean CQs can be found in the possible exponential size of intermediate results while computing such an evaluation. However, some conjunctive queries have an order in which the conjunctions can be computed with only polynomial sized intermediate outputs in the size of the input and output. Consequently, for Boolean conjunctive queries this implies polynomial sized intermediate outputs in the size of the input. Informally, a conjunctive query is called acyclic if such an ordering exists.

There exist several notions that are equivalent to the notion of acyclic queries. The most important ones for our case are situated in the context of acyclic database schemas. The term acyclicity is derived from equivalent notions on hypergraphs. We refer to [2, 12] for a more complete overview of properties.

Property 3.1: *A database schema* $\boldsymbol{R}$ *is acyclic iff* $\boldsymbol{R}$ *has a join forest.*

A join forest of a database schema $\mathbf{R}$ is a forest consisting of $\mathbf{R}$ as its nodes, where every edge is labeled by the shared attributes of its adjacent nodes and between every pair of distinct nodes there exists a path containing only edges that have a label including the shared attributes. Intuitively a join forest implies the existence of an ordering such that $\pi_X(\bowtie_{1 \leq i \leq m} R_i)$ can be computed with only polynomial sized intermediate instances relative to the size of the input and output. A join tree is a connected join forest.

Definition 3.3: Let $\mathbf{I}$ be an instance over $\mathbf{R} = \{R_1, \ldots, R_m\}$. $\mathbf{I}$ is *pairwise consistent* iff for each $i, j \in [1, m] : \pi_X(R_i \bowtie R_j) = R_i$, where X are the attributes in R_i. $\mathbf{I}$ is *globally consistent* iff for each $i \in [1, m] : \pi_X(R_1 \bowtie \ldots \bowtie R_m) = R_i$ where X are the attributes of R_i.

Property 3.2: *A database schema* $\boldsymbol{R}$ *is acyclic iff every pairwise consistent instance over* $\boldsymbol{R}$ *is globally consistent.*

From instance $\mathbf{I}$ it is possible to compute a pairwise consistent instance $\mathbf{I'}$ by means of a semijoin program. A semijoin between relation R and S, write $R \ltimes S$, is defined as $\pi_X(R \bowtie S)$, where X is the set of attributes of R. Hence, a semijoin program is a program that consists of rules of the form $R_i := R_i \ltimes R_j$. In a worst-case scenario, the semijoin of every relation has to be taken with every other relation. Hence, this computation can be performed in polynomial time. Property 3.2 implies that the resulting $\mathbf{I'}$ in the case of an acyclic database schema is globally consistent. A semijoin program for $\mathbf{R}$ is called a *full reducer* if the result of the execution of the program on every instance $\mathbf{I}$ over $\mathbf{R}$ results in an equivalent globally consistent instance $\mathbf{I'}$.

Property 3.3: *A database schema* $\boldsymbol{R}$ *is acyclic iff* $\boldsymbol{R}$ *has a full reducer.*

Now we turn to the main theorem of this section by combining Property 3.1 and Property 3.3.

Theorem 3.4 (Yannakakis [62, 2]): Let $\mathbf{R}$ be an acyclic database schema. For every instance $\mathbf{I}$ over $\mathbf{R}$, the expression $\pi_X(\bowtie_{1 \leq i \leq m} R_i)$ can be computed in time polynomial in the size of the input and the output.

Proof: Let $\mathbf{I'}$ be the result of the execution of a full reducer over $\mathbf{I}$ and T a join forest of $\mathbf{R}$. Without loss of generality we assume that T is a join tree. If not, the construction below can be applied on the individual trees. Throughout this proof, we use R not only as the relation symbol, but also to refer to the corresponding set of attributes.

Choose a root for T and call it R_1. For every subtree T_k from T with $R_k \neq R_1$ as its root, let $X_k = X \cap (\cup\{R \mid R \in T_k\})$ en $Z_k = R_k \cap R_p$ where R_p denotes the parent node of R_k in T. Let $\mathbf{J}(R_k) = \mathbf{I'}(R_k)$ for each $k \in [1, m]$.

Systematically remove each node R_k in T from leaf to root by first removing node R_k and replacing $\mathbf{J}(R_p)$ with $\mathbf{J}(R_p) \bowtie \pi_{X_k Z_k}\mathbf{J}(R_k)$.

We prove that when R_k is removed, $\mathbf{J}(R_k) = \pi_{X_k R_k}(\bowtie_{R_l \in T_k} \mathbf{I}(R_l))$. If R_k is a leaf in the original T, this immediately follows from the definition of $\mathbf{J}(R_k)$. Now assume for R_k that every child node R_i satisfies this property $\mathbf{J}(R_i) = \pi_{X_i R_i}(\bowtie_{R_l \in T_i} \mathbf{I'}(R_l))$ at the moment before removal. When R_k is removed $\mathbf{J}(R_k) = \mathbf{I'}(R_k) \bowtie (\bowtie_{R_i \text{ child of } R_k} \pi_{X_k Z_k}\mathbf{J}(R_i))$. Moreover, from the induction hypothesis we get Equation (3.1) and Equation (3.2).

$$\mathbf{J}(R_k) = \mathbf{I'}(R_k) \bowtie (\bowtie_{R_i \text{ child of } R_k} \pi_{X_k Z_k}(\pi_{X_i R_i}(\bowtie_{R_l \in T_i} \mathbf{I'}(R_l)))) \tag{3.1}$$

$$\mathbf{J}(R_k) = \mathbf{I'}(R_k) \bowtie \pi_{X_k Z_k}(\bowtie_{R_l \in T_k \setminus \{R_k\}} \mathbf{I'}(R_l)) = \pi_{X_k R_k}(\bowtie_{R_l \in T_k} \mathbf{I'}(R_l)) \tag{3.2}$$

Consequently, when R_1 is deleted we get $\mathbf{J}(R_1) = \pi_X(\bowtie_{R_l \in T} \mathbf{I'}(R_l)) = \pi_X(\bowtie_{R_l \in T} \mathbf{I}(R_l))$.

Note that the size of every intermediate instance is bounded by $|\mathbf{I'}|.|\pi_X(\bowtie_{1 \leq i \leq m} \mathbf{I}(R_i))|$ and thus is polynomial in the size of the input and output. □

3.6 Acyclic Complexity

Theorem 3.5: ACQ-EVAL is in PTIME.

Proof: Let q be an arbitrary ACQ over **R**, **I** an arbitrary instance over **R** and t an arbitrary tuple compatible with q. We first transform q into a Boolean query q' by instantiating each variable in the body of q that appears in $\bar{z}$ with the corresponding nodes in t.

To use Theorem 3.4 we transform this Boolean problem. Let **R'** be a database schema where each relation $\bar{x}_i$ in the body of q' has its own relation schema $R_i(\bar{z})$ where each constant is replaced by a new variable. We also construct the instance **I'** over **R'** with respect to the original **I** over **R**, and remove every tuple in **I'**(R_i) that is not compatible with its associated relation in the body of q.

Now consider the evaluation problem of $q'' := \pi_{\emptyset}(\Join_{1\leq i\leq m} (R_i))$ over I' with fixed database schema **R'**. This problem is equivalent to the original problem, and **R'** is an acyclic database schema. Since q'' is a Boolean query, Theorem 3.4 implies that its evaluation is possible in time polynomial in the size of the input. Consequently ACQ-EVAL is possible in time polynomial in the size of the input. □

In particular, boolean acyclic conjunctive queries are complete in LOGCFL [32]; a complexity class in between NL and AC^1.

4 Conjunctive Regular Path Queries

In the previous chapter we explored the complexity of conjunctive queries in the context of relation databases. In this chapter we take a look at the complexity of an extension of the conjunctive queries in the context of graph databases. First, we define conjunctive queries in the context of graph databases and introduce the language of regular path queries. Next, we combine both to form the language of conjunctive regular path queries. The main subject of this chapter is a study of the complexity of conjunctive regular path queries in a sequential setting.

4.1 Conjunctive Queries on Graph Databases

The language of conjunctive queries itself can be seen as a very simple query language on graph databases, where E is the only relational schema. A formal, slightly modified version of Definition 3.1 applied to graph databases is given below.

Definition 4.1: A *conjunctive query* (CQ) over a finite alphabet Σ is an expression of the form

$$ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, a_i, y_i),$$

where $m \geq 0$, x_i, y_i are node variables, and $a_i \in \Sigma$.

The evaluation of a CQ q over a directed, Σ-labeled graph database $G = (V, E)$, written $q(G)$, is defined analogously to the relational case. We define a valuation ϑ over a set W of node variables as a total function from W to the set V of nodes in G. This function can be extended to tuples in the natural way.

$$q(G) := \{\vartheta(z) \mid \vartheta \text{ a valuation such that for every } i \in [1, m] : \vartheta(x_i, a_i, y_i) \in E\}.$$

Essentially, Definition 4.1 implies a restriction of Definition 3.1 to queries over a single ternary relation. Note however that the complexity results from Chapter 3 remain applicable for this restricted definition of conjunctive queries, including the completeness in Theorem 3.2.

4.2 Regular Path Queries

An important shortcoming in the expressiveness of CQs over graph databases is the lack of navigational capabilities. A basic query language that contains just this functionality is the language of regular path queries.

Definition 4.2: A *regular path query* (RPQ) over a finite alphabet Σ is an expression of the form

$$ans(x, y) \leftarrow (x, r, y),$$

where x, y are node variables and r is a regular expression over Σ.

Example: Let G be a Σ-labeled graph and Σ an alphabet, where $parent \in \Sigma$. Assume that nodes represent persons and a *parent*-labeled edge from node u to node v indicates that v is a parent of u. The ancestor relation can be formulated with a regular path query:

$$ans(u, v) \leftarrow (u, parent^*, v).$$

Intuitively, the evaluation of a regular path query q over a graph databases G is defined as the set of node pairs (u, v) such that u and v are connected by a path in G which agrees to the regular expression r. More formally, let $L(r)$ be the language containing each string defined by regular expression r. The evaluation of q over Σ-labeled graph G, written $q(G)$, is defined as follows:

$$q(G) := \{(u, v) \in V^2 \mid \begin{array}{l}\text{there exists a path } \eta \text{ from } u \text{ to } v\\ \text{such that } \lambda(\eta) \in L(r)\}.\end{array}$$

Although RPQs are considered simple, their evaluation is still hard, especially on large and unrestricted graphs [40], in contrary to, e.g., path queries on XML trees. Since the expressions themselves have a huge influence on the evaluation complexity, query rewriting [16] is an important aspect of the evaluation of RPQs in practice. In this document, however, we assume that queries are evaluated as is.

4.3 Conjunctive Regular Path Queries

The combination of conjunctive queries and regular path queries leads to a simple but powerful query language on graph databases, comparable to what conjunctive queries are in the relational model. The same as for CQs, both the containment and equivalence problem for RPQs and CRPQs are decidable. In particular, the containment problem for CRPQs is in EXPSPACE [26].

Definition 4.3: A *conjunctive regular path query* (CRPQ) over a finite alphabet Σ is an expression of the form

$$ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, r_i, y_i),$$

where $m > 0$, every x_i and y_i are node variables, every r_i is a regular expression over Σ, and every z_i in $\bar{z}$ is an x_j or y_j.

We call $ans(\bar{z})$ the *head* of the query and $\bigwedge_{1 \leq i \leq m}(x_i, r_i, y_i)$ the *body* of the query. A conjunctive regular path query is a *Boolean conjunctive regular path query* (BCRPQ) if its head has arity 0.

Example: Recall the example of Section 4.2. We can use a CRPQ to express a query that asks for the relation of persons that have a common ancestor:

$$ans(u, v) \leftarrow (u, parent^*, w) \wedge (v, parent^*, w).$$

The evaluation of conjunctive regular path queries is defined in a similar manner as the evaluation of CQs and RPQs in the previous sections:

$$\begin{aligned} q(G) := \{\vartheta(\bar{z}) \mid \vartheta \text{ a valuation such that for every } i \in [1, m] \text{ there exists a path } \eta \\ \text{from } \vartheta(x_i) \text{ to } \vartheta(y_i) \text{ such that } \lambda(\eta) \in L(r_i)\}. \end{aligned}$$

4.4 Data Complexity

Proposition 4.1: *Let q be a CRPQ.* CRPQ-EVAL(q) *is* NL*-complete.*

Proof: A witness for $\bar{v}$ and G with respect to $q := ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m}(x_i, r_i, y_i)$ is a tuple $(u_1, \ldots, u_m, u'_1, \ldots, u'_m, \eta_1, \ldots, \eta_m)$ where for every $i \in [1, m]$: u_i and u'_i are nodes in G, η_i is a path in G from u_i to u'_i, and every pair (u_i, u'_i) is compatible to q. Let z be the jth element of $\bar{z}$ and $z = x_i$ or $z = y_i$, than u_i or u'_i is the same as the jth element of $\bar{v}$, respectively. For each $i \in [1, m] : \lambda(\eta_i)$ is defined by regular expression r_i.

Construct an NFA $\mathcal{A}_i$ for every regular expression r_i in q. This construction only depends on q and can thus be done in constant time.

Now, consider a non-deterministic TM that, on input $\bar{v}$ and G, guesses nodes for $u_1, u_2, \ldots, u_m, u'_1, u'_2, \ldots, u'_m$. The number of nodes depends only on q, and therefore is fixed. Since we only keep pointers to nodes, this is possible with a logarithmic amount of space. For every η_i assume a pointer to u_i and systematically choose nodes that are connected to this node in G with respect to $\mathcal{A}_i$. Reuse the selected space for each consequent node. If we reach node u'_i start guessing the next path. Otherwise, reject the input. When we find a path for every path variable, accept the input. Clearly, this non-deterministic TM requires only a logarithmic amount of space in the size of the input. Therefore we have proven that CRPQ-EVAL(q) is in NL.

The completeness is a result of the completeness of PATH, say $\langle G, s, t\rangle$, which asks for the existence of a path from s tot t in directed graph G. Let Σ be the alphabet of G. Now it is possible to detect if there exists a path from s to t in G with the CRPQ $ans() \leftarrow (s, \Sigma^*, t)$. □

4.5 Combined Complexity

Proposition 4.2: CRPQ-EVAL *is* NP*-complete.*

Proof: Let q be a conjunctive regular path query over the finite alphabet Σ, $G = (E, V)$ a directed Σ-labeled graph, and $\bar{z}$ a tuple with nodes in G such that $|\bar{v}| = |\bar{z}|$.

Define a witness for $\bar{v}$ with respect to q and $q(G)$ as in the proof of Proposition 4.1. We can construct an NFA $\mathcal{A}_i$ based on the regular expression a_i, for every $i \in [1, m]$ that accepts the language defined by a_i, in polynomial time. We call Q_i the set of states in $\mathcal{A}_i$.

When we assume that for every $\bar{v} \in q(G)$ there exists a witness such that the length of every path of that witness is bounded by $M = |V| \times \max\{|Q_i| \mid i \in [1, m]\} \times |\Sigma| + 3$ it is possible to construct a non-deterministic TM that guesses every possible witness for $\bar{v}$ and checks if $\bar{v} \in q(G)$ by validating these witnesses. More precise, this is possible in a polynomial amount of time.

To prove the assumption, we only need to prove that if a witness exists for $\bar{v}$, there also exists a witness for $\bar{v}$ where every path has length smaller or equal to M. Assume a witness where at least one path has length greater than M, that is

$$(u_1, \ldots, u_m, u'_1, \ldots, u'_m, \eta_1, \ldots, \eta_m),$$

such that $\eta_i = v_1 a_1 v_2 a_2 \ldots v_l a_l v_{l+1}$ where $i \in [1, m]$ and $l + 1 > M$. Let θ be the accepting run of $\mathcal{A}_i$ for η_i where $\theta(0)$ is the initial state. Then there exists an $i, j \in [1, m]$ such that $v_i = v_j, a_i = a_j$ and $\theta(i - 1) = \theta(j - 1)$. Clearly, when we replace η_i by $v_1 a_1 v_2 a_2 \ldots v_{i-1} a_{i-1} v_j a_j \ldots v_l a_l v_{l-1}$ the remaining is still a witness for $\bar{v}$. Repeating this step yields an algorithm that results in a witness for $\bar{v}$ where every path has length smaller than M, which proves the assumption.

The completeness follows directly from the completeness of the conjunctive queries, which was proven in the previous chapter. □

The proof above is based on the proof in [11] for the combined complexity of an extended form of CRPQs that allow path variables in the output.

4.6 Expression Complexity

Proposition 4.3: *Let G be a directed Σ-labeled graph database.* CRPQ-EVAL(G) *is* NP*-complete.*

The proof of Proposition 4.3 is analogous to the proof of Proposition 4.2.

4.7 The Acyclic Case

Analogous to the acyclic case for conjunctive queries as described in Chapter 3, we define an *acyclic conjunctive regular path query* (ACRPQ) as a CRPQ where the conjunctive part is acyclic. That is, if the graph containing precisely the edges (x_i, y_i) for $i \in [1, m]$ is acyclic [10].

Proposition 4.4: *Let q be an acyclic conjunctive regular path query.* ACRPQ-EVAL(q) *is* NL-*complete.*

A close look at the proof of Proposition 4.1 reveals that the data complexity of CRPQs does not depend on the conjunctive part of the query, but is straightforwardly related to the RPQ part. Therefore, it is easy to see that acyclicity has no effect on this result.

Proposition 4.5: ACRPQ-EVAL *is in* PTIME.

Proof: Let $\bar{v}$ be a tuple, G a directed Σ-labeled graph database and q a fixed query over Σ. First, we transform q into a boolean query q' by replacing the variables of $\bar{z}$ in the body of q by the associated nodes in v and replacing the head of the query by $ans()$.

For each regular path query (x_i, r_i, y_i), $i \in [1, m]$ construct the associated NFA A_i and construct an NFA that recognizes the intersection $L(\mathcal{A}_i) \cap L(G)$. Note that G can be seen as an automaton. When both x_i and y_i are variables, assume that every state in G is an initial and final state. Otherwise, we only consider initial and final states compatible with x_i and y_i. The construction is possible in polynomial time.

Now, construct for every regular path query (x_i, r_i, y_i) an instance over the binary relation R_i containing the tuples (v_i, v_i') where v_i and v_i' are connected by a path whose label is in the language $L(r_i)$. The number of relations depends on q only. The construction of relations R_i is possible in polynomial time.

Finally, we only need to check if there exists an assignment to the node variables in the body of q' that is compatible to every regular path query of q'. We do this by evaluating the conjunctive query $q'' := ans() \leftarrow \bigwedge_{1 \leq i \leq m} R_i(x_i, y_i)$ over the constructed instance, where x_i and y_i are node variables as in q'. q is acyclic by definition, and so is q''. Consequently, the evaluation is possible in polynomial time. □

This result remains valid even when we generalize ACRPQs to allow path variables in the output. The proof above is a based on the proof for the extended case in [11].

Part II

Parallel Context

5 MapReduce

MapReduce is a programming model and corresponding computational infrastructure introduced by Google [19] to simplify the implementation of distributed algorithms on large-scale clusters of commodity machines. Informally a MapReduce job consists of the parallel execution of two user defined functions: a map function that maps elements into buckets and a reduce function that processes the elements inside these buckets. Although not required, ideally a reducer reduces the number of elements.

MapReduce has become a kind of *de facto* standard for robust distributed computations at Google [20], but it has also attracted a lot of attention by other companies through its popular open source implementation Hadoop [1].

We start this chapter with an in depth description of the MapReduce programming model and a brief discussion of its computational infrastructure. Next, we discuss several models and complexity measures for comparing MapReduce algorithms among each other and for examining their feasibility in practice.

5.1 Programming Model

The description below is based on insights from several existing descriptions; i.e., [19, 39, 3, 29]. We call $\mathbf{U}_{\mathrm{K}}, \mathbf{U}_{\mathrm{V}} \subseteq \Sigma^*$ the key and value domains. The basic building blocks in a MapReduce computation are binary strings of the form $\langle key; value\rangle \in \mathbf{U}_{\mathrm{K}} \times \mathbf{U}_{\mathrm{V}}$. Input, output, as well as intermediary outputs are multisets of key-value pairs. We will frequently look at a multiset of key-value pairs with a common key as a special type of "aggregated" key-value pair of the form $\langle k; \{v_1, v_2, \ldots, v_k\}\rangle \in \mathbf{U}_{\mathrm{K}} \times \mathcal{P}(\mathbf{U}_{\mathrm{V}})$. Nevertheless, conceptually we have that

$$\langle k; \{v_1, v_2, \ldots, v_k\}\rangle \equiv \{\langle k, v_1\rangle, \langle k, v_2\rangle, \ldots \langle k, v_k\rangle\}.$$

We call a pair of this form that contains every value associated to k a *reduce record* and k its *reduce key*.

A *MapReduce step* is defined as a tuple (μ, ρ) of two user defined functions *map* and *reduce*. Formally, a *map* function $\mu : \mathbf{U}_{\mathrm{K}} \times \mathbf{U}_{\mathrm{V}} \to \mathcal{P}(\mathbf{U}_{\mathrm{K}} \times \mathbf{U}_{\mathrm{V}})$ is a function that maps a key-value pair on a finite multiset of key-value pairs. A *reduce* function ρ is a function that maps a *reduce record* $\langle k; \{v_1, v_2, \ldots, v_n\}\rangle$ on a multiset of key-value pairs $\langle k, value\rangle$ where k is the reduce key:

$$\rho : \mathbf{U}_{\text{K}} \times \mathcal{P}(\mathbf{U}_{\text{V}}) \to \mathcal{P}(\mathbf{U}_{\text{K}} \times \mathbf{U}_{\text{V}}) : \langle k; \{v1, \ldots, v_l\}\rangle \mapsto \{\langle k; w_1\rangle, \ldots, \langle k; w_m\rangle\}.$$

Both the map and reduce function are stateless; the output only depends on the input arguments. The empty set is allowed as output.

From a programmer's viewpoint the execution of a MapReduce job has two phases: a map phase and a reduce phase. From an infrastructural viewpoint there are three phases: a map phase, a shuffle phase, and a reduce phase. Although the shuffle phase is completely hidden from the programmer it is an essential and expensive phase in the execution of a MapReduce step.

A *MapReduce program* as defined in [19] is a sequential program in which the programmer is able to formulate MapReduce jobs that are executed in parallel automatically by the underlying system. Nevertheless we formalize it, analogously to [39] as a sequence of MapReduce jobs $\langle \mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_r, \rho_r\rangle$. Let U_0 be a multiset of key-value pairs that serves as the input to the MapReduce program. The execution of step $1 \leq i \leq r$ goes as follows:

- MAP PHASE: Run map function μ_i on each $\langle k; v\rangle \in U_{i-1}$. Let

  $$U_i' = \bigcup_{\langle k,v\rangle \in U_{i-1}} \mu_i(\langle k; v\rangle)$$

  be the intermediate key-value pairs. During the map phase several processes execute μ_i in parallel.

- SHUFFLE PHASE: The underlying implementation groups the intermediate outputs per key, write $V_{k,i} = \{v \mid \langle k; v\rangle \in U_i'\}$.

- REDUCE PHASE: Run each reduce function ρ_i on reduce record $\langle k; V_{k,i}\rangle$. During the reduce phase several processes execute ρ_i in parallel. Let $U_i = \bigcup_k \rho_i(\langle k; V_{k,r}\rangle)$. Since the input for an individual reducer can be much larger than the size of its memory, reducers have access to their input via an iterator.

Finally the execution finishes when the last reducer ρ_r finishes. We call U_r the output of the MapReduce program.

An important property of MapReduce is its blocking property; the shuffle phase (and reduce phase) can only start when the map function has finished on every key-value pair.

In the literature, the instances of the map and reduce function are commonly called mappers and reducers, though the exact definition differs. Some (e.g., [3, 43, 29]) define a reducer as a process that runs the reduce function sequentially on several reduce records, that is, several reduce records are processed by the same reducer. In this definition there is, however, a correlation between the number of reducers and the number of available machines. In other words, the number of reducers depends on the implementation and the available infrastructure. Therefore it is not a very satisfying definition at the model level. We prefer the competing definition from [6] and use the term reducer to denote a particular function instance. In this definition the number of reducers equals the number of reduce keys and therefore represents a theoretical bound on parallelization rather than a physical and implementation dependent bound based on the number of available machines. We use the term *reduce worker* to refer to the former definition.

Since MapReduce is designed to handle big data it is common, even for the input of a single reducer, to handle data that is much larger than the available amount of memory on a single machine. Hence, the input of a reduce instance in the original model [19] as well as in Hadoop [1] needs to be accessed by an iterator. This is definitely an important limitation. Ideally, a balanced distribution of data may cause the input of individual reduce instances to be small enough to enable the load of the entire input stream into memory and to allow random access to it.

5.2 Computational Infrastructure

The computational infrastructure of MapReduce as described in [19, 20] is somewhat less important for our purposes, but it is helpful to have a least a basic notion of a what *may* happen underneath to understand the choices in the subsequent sections.

MapReduce essentially provides an easy to use and fault-tolerant programming interface to perform parallel computations on files of a special form, that is, multisets of key-value pairs. It is build on top of a distributed file system such as Google its GFS [27] or Hadoop its HDFS [1] and therefore is able to store files in a reliable environment by use of, e.g., replication. The input and output of a MapReduce job consists of sets of files which are stored into this file system.

The MapReduce infrastructure itself [19] consists of a single master node and several worker nodes. The input files are fragmented into pieces which are distributed over the worker nodes. Each fragment is processed by a map worker that parses the fragment and executes the user defined map function on every single pair in the fragment. The mapper output is written to local disk and the output is divided in r regions, where every region represents a reduce worker. The position of every region is communicated to the master node, which on his turn communicates the position of these regions to the corresponding reduce workers. Note that these can only start when all mappers are finished. A reduce worker reads all the intermediate data from its corresponding regions and sorts them on the intermediate keys (not the values), that is, reduce records are extracted. Next, the reduce worker executes the user defined reduce function on every reduce record that it received. The reducer outputs are stored on files in the distributed file system.

Apart from the map and reduce function itself, the implementation in both [19] and Hadoop [1] allow several other customizations. For example, it is possible to customize the fragmentation function. Hadoop also allows customization of the group function, which is used to determine what key-value pairs in a particular reduce worker are processed together by a reduce function.

5.3 Relation of MapReduce with Other Models

Driven by practical need, MapReduce has become an important player in parallel computing, but it has little theoretical foundation yet. The need for sound theoretical foundations gives rise to a growing interest in relation with the established computational models like Parallel Random Access Machines (PRAM) and Bulk Synchronous Parallel (BSP) model. Although these are beyond the scope of this thesis, it is interesting to mention that, for example, the relation between PRAMS and MapReduce is studied in [39]. In particular, the simulation of a class of PRAMS in MapReduce is discussed. The relation between MapReduce and BSP [58] is investigated in [31, 52]. In contrary to PRAMS, which are generally known to be overly simplistic models of computations, the BSP model is more inline with modern cluster configurations. Further, in [25], MapReduce is linked to streaming algorithms, which can be seen as a counter method for processing large amounts of data.

5.4 Parallel DBMS

Some critique exists in that MapReduce disregards a lot of lessons learned from parallel DBMS research [21]. A parallel DBMS takes advantage of parallelization and distribution of data, but retains the traditional relational setting. In general, most agree that MapReduce should not be seen as a replacing or competing technology to parallel DBMSs, but rather as a complementary technology, for example, when it comes to processing read once data or semi-structured data [55].

5.5 Stragglers and Data Skew

Parallel and distributed systems as MapReduce give raise to problems that don't exist in a sequential context. Stragglers are one such problem. A *straggler* is an individual machine that slows down the entire computation. They are especially important in MapReduce since computations have to wait for each other at regular times, i.e., in every iteration before the reduce phase can start. Although stragglers are commonly referred to as a result of hardware and software failures, which are not the responsibility of the algorithm designer, algorithms themselves can cause stragglers to occur as well. For example, if data partitioning relies on the structure of the input data then some reducers may receive almost the entire dataset where others receive only a few pairs. This is called skew. The phenomenon of 99% of the reducers that are finished and 1% of the reducers to wait for is also referred to as "the curse of the last reducer" [57].

In [42] an intensive study of skew in MapReduce is performed with the aim to add automated skew mitigation to MapReduce. Moreover, they provide an overview of different types of algorithm related skew. A brief overview of the most important types is listed below.

- *Record skew*: some pairs may be much larger than others and consequently are much harder to process.

- *Heterogeneous functions*: mappers or reducers may depend heavily on the type of input pairs for their running time.
- *Partition skew*: skew caused by mappers who map their output on reducers in an unbalanced way.

The latter type of skew is also studied in [41]. They try to prevent partition skew by introducing a MapReduce like model that imposes very strict bounds on the partitioning of pairs among reducers. More particular, the model requires that the reduce records are perfectly balanced.

Notwithstanding that skew is an important and non-negligible aspect of MapReduce algorithms, we have no intention to focus on types of skew and approaches to handle skew in general. However, we will regularly refer to skew and try to avoid the occurrence of skew in our algorithms.

5.6 A Canonical Example

A popular problem to illustrate the working of MapReduce is Word Count [19, 3, 31, 6]; that is, given a collection of documents, calculate the number of occurrences for each word in these documents. Assume that this collection of documents is encoded as multiset of key-value pairs, where every word, say w, is represented by a key-value pair $\langle k, w\rangle$, where k is an arbitrary key. This problem can be translated into a very simple and effective MapReduce algorithm: map each pair $\langle key, word\rangle$ onto a pair $\langle word, 1\rangle$ and let the reduce function sum the values of pairs that it receives as input, and output the pair $\langle word, sum\rangle$.

Note that the proposed algorithm may suffer from data skew. Nevertheless, we can also hash words uniformly at random and distribute words in a more balanced fashion. Every reducer then outputs a pair for every word it receives and its partial frequency. A second MapReduce step is required to compose the final output. In this step, every word has at most as many pairs as there are reducers in the first step.

Unfortunately, the implementation of MapReduce algorithms for most problems is much less convenient. Basically, MapReduce only supports the effective execution of problems which are *embarrassingly parallel* [45, 6], i.e., problems that are decomposable into a large number of independent subroutines. In general, when a not embarrassingly parallel problem is computed with MapReduce, more parallelization incures a higher communication cost [6].

5.7 The Communication Cost Model

Similar to the sequential context, we are interested in complexity measures for algorithms in MapReduce. Therefore, in this and consecutive sections we discuss several complexity measures and cost models on MapReduce algorithms.

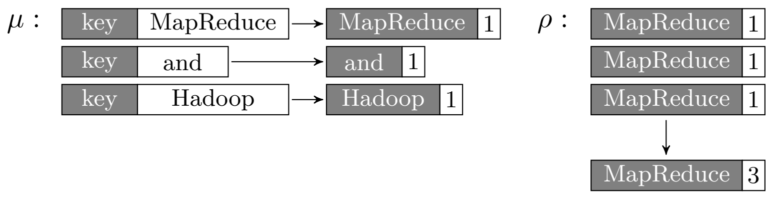


Figure 5.1: Word Count in MapReduce.

Afrati and Ullman [7] use the communication cost, also called data volume cost [5], to compare the complexity of algorithms in MapReduce and related environments. The idea is that when tasks are distributed appropriately, the cost of reading data from disk and communicating it via a network dominates the cost of the actual computation. The communication cost therefore is taken as an approximation to the complexity of algorithms in this setting. Moreover, since the amount of data that is exchanged is usually very large, the time taken to communicate will depend more on the size of the data than on, for example, network implementation details. These insights lead to the following definitions:

- The *communication cost* of a process is the size of the input of this process.
- The *total communication cost* of an algorithm is the sum of the communication costs of all processes. It can be seen as a surrogate of *renting time* [5], for example on a public cloud.

Consider a MapReduce computation as a directed acyclic graph of processes, where every edge indicates that the target process relies on the output of the source processes for its input.

- The *elapsed communication cost* is defined as the maximum sum of communication costs of processes in a path in this acyclic graph.

Afrati and Ullman ignore the output cost as it is taken into account as input of the next operation, or that it is the output of the whole calculation and thus is likely to be small enough to neglect.

Unfortunately, the less parallelization, the less communication and thus the better the algorithm will score in the communication cost model. Moreover, since the shuffle phase causes significant overhead, it is common to constrain the number of MapReduce steps. Therefore, many algorithms score best with only a single reducer. Hence, appropriate upper-bounds on the input size for individual reducers are necessary to make this model meaningful. Furthermore, a restriction on the size of the reduce records may enable computations to be done in main memory [5].

Further note that the communication cost and elapsed communication cost are defined in terms of processes (i.e., map and reduce workers) and therefore depend on the implementation of MapReduce. Clearly, this is not applicable to the total communication cost.

5.8 Key and Sequential Complexity

Goel and Munagala [29] give two complexity measures to analyse the performance differences between MapReduce algorithms. First, there is the *key complexity*, which is a measurement for individual mappers and reducers. This measurement consists of three parts:

- The maximum size of a key-value pair input to or output by a mapper or reducer;
- The maximum running time for a mapper or reducer;
- The maximum memory used by a single mapper or reducer to process a key-value pair.

Second, there is the *sequential complexity*, which is a measurement over all mappers and reducers. It consists of two parts:

- The size of all inputs and outputs by mappers and reducers;
- The total running time for all mappers and reducers.

Notice that the total amount of memory is not measured. Goel and Munagala argument that the total memory depends on the number of reduce workers operating at a given time and thus is a property of the MapReduce deployment rather than of the algorithm. Therefore, it is omitted as a sequential complexity measure.

The algorithms on which the sequential complexity is illustrated in [29] are one-step algorithms. Many algorithms in this thesis, however, consist of multiple MapReduce steps. Hence, the sequential complexity can be viewed as a bound on the complexity of single MapReduce steps, or as a bound on the complexity of the entire program. In our context, we use the sequential complexity to denote the complexity of single MapReduce steps as this gives us a better understanding of the feasibility of an algorithm.

5.9 The $\mathcal{MRC}$ Model

In Section 5.6 it was mentioned that, except for embarrassingly parallel algorithms, it is not clear which algorithms in MapReduce are practically feasible. The MapReduce Class, $\mathcal{MRC}$ [39], is an attempt to capture this notion into a formal model.

Clearly, from a practical point of view it is not feasible to assume an infinite supply of machines or memory. Moreover, since MapReduce is especially used to process big data, even a linear number of machines or a linear amount of memory in the size of the input generally is infeasible to assume. The $\mathcal{MRC}$ model is based on the assumption that we have only a sublinear number of machines and a sublinear amount of memory per machine. To be precise, $\mathcal{MRC}$ provides a set of bounds on the MapReduce programming model that guarantee that every algorithm can be executed with a sublinear number of machines each with a sublinear amount of space.

Definition 5.1: Fix an $\varepsilon > 0$. A MapReduce algorithm is in $\mathcal{MRC}^i$ if:

- It outputs the correct answer with probability at least 3/4.
- Each μ is a randomized map function implemented on a RAM with words of length $\mathcal{O}(\log n)$, that uses $\mathcal{O}(n^{1-\varepsilon})$ space and time polynomial in n.
- Each ρ is a randomized reduce function implemented on a RAM with $(\log n)$ length words, that uses $\mathcal{O}(n^{1-\varepsilon})$ space and time polynomial in n.
- The total amount of space used by key-value pairs outputted by map functions in a single map phase is $\mathcal{O}(n^{2-2\varepsilon})$.
- The number of rounds is $\mathcal{O}(\log^i n)$.

Let $\mathcal{MRC} = \bigcup_{i\geq 0} \mathcal{MRC}^i$ and call $\mathcal{DMRC}$ its deterministic variant. The model allows random access to the reducer input.

The number of reduce keys is not limited, rather the bounds in Definition 5.1 guarantee that there exists an implementation of the shuffle phase such that every record is associated to a machine and no machine receives more than a sublinear amount of input pairs [39]. This guarantee is based on a heuristic called Graham's algorithm for the minimum makespan problem [33]. The minimum makespan problem is the problem that asks, given a number of machines and a set of tasks with associated processing times, to assign these tasks to machines such that the processing time for machines is balanced. Of course, in our case we replace time by memory. Assume a sequence of tasks in a given order. Consecutively map a task on the machine that has the least memory in use. Hence, before the last task is mapped every machine has at least as much memory in use as the machine to which the last task is mapped. Consequently, within the asymptotic analysis it can be proven that Graham's Greedy algorithm maps no more than the balanced load plus the maximum load of a single task on a single machine:

$$\frac{\text{total memory}}{\text{number of machines}} + \max(\text{reduce record}) \leq \frac{\mathcal{O}(n^{2-2\varepsilon})}{\Theta(n^{1-\varepsilon})} + \mathcal{O}(n^{1-\varepsilon}).$$

Therefore, we have that the total amount of memory received by a single machine is bounded by $\mathcal{O}(n^{1-\varepsilon})$.

Unfortunately, if ε is small, the actual number of required machines can be close to linear. Since n can be very large, $n^{1-\varepsilon}$ may still be unreasonably high. Furthermore, most algorithms implicitly restrict the value of ε towards zero.

The polynomial bound on the processing time for individual mappers and reducers may seem rather high, but similar as in the communication cost model, in $\mathcal{MRC}$ the focus is on communication rather than on the time taken by the actual computations. Another issue is that it is a hard task to design algorithms for MapReduce, and it often requires intensive use of advanced probability theory (e.g. uniform hash functions and Chernoff bounds). In Section 5.12 a technique is described to simplify the design of algorithms in $\mathcal{MRC}$.

In terms of the communication cost model [7], as was defined in Section 5.7, we have that in every $\mathcal{MRC}$ algorithm the communication cost is bounded by $\mathcal{O}(n^{1-\varepsilon})$ and the total communication of a each MapReduce step is bounded by $\mathcal{O}(n^{2-2\varepsilon})$. Note that in $\mathcal{MRC}$ the total communication cost is bounded explicitly, while the communication cost is bounded implicitly by the guarantees about the shuffle phase. Analogous to the communication cost the elapsed communication cost of an algorithm in $\mathcal{MRC}$ is bounded by $\mathcal{O}(r.n^{1-\varepsilon})$, where r is the number of rounds.

In terms of key complexity, the size of key-value pairs in an algorithm that is in $\mathcal{MRC}$ is trivially bounded by $\mathcal{O}(n^{1-\varepsilon})$, the time is polynomial, and the memory is bounded by $\mathcal{O}(n^{1-\varepsilon})$. The Sequential complexity of each MapReduce step is likewise bounded by $\mathcal{O}(n^{2-2\varepsilon})$ for size and takes a polynomial amount of time. Although not taken into account by the sequential complexity, in the $\mathcal{MRC}$ setting the total memory is also bounded by $\mathcal{O}(n^{2-2\varepsilon})$.

5.10 Generalizations of $\mathcal{MRC}$

It may be argued that the $n^{1-\varepsilon}$ bound on the size of reduce records and the $n^{2-2\varepsilon}$ bound on the total size are rather arbitrary chosen and should not be imposed at model level. Therefore [31] and [53] propose a kind of generalized model where concrete bounds are replaced by parameters. In [31] a parameter is used to constrain the memory of individual mappers and reducers. In [53] both the size of individual mappers and reducers, and the total size are parameterized. In this section we describe the model of [53].

Informally, an MR-algorithm is a MapReduce algorithm where the available amount of space in every round is specified by the parameters m and M rather than $n^{1-\varepsilon}$ and $n^{2-2\varepsilon}$.

Definition 5.2: Let P be a MapReduce algorithm and n the size of the input. Call $m_{k,i}$ the size of the input and amount of workspace (excluding the space taken by the output) used by reducer k (on a RAM) in round i of P. P is called an *MR-algorithm*, write $MR(m, M)$, if for each k, i we have that $m_{k,i} \in \mathcal{O}(m)$ and in every round i we have that $\sum_{k \in \mathbf{U}_{\mathrm{K}}} m_{k,i} \in \mathcal{O}(M)$. The final output has to fit in $\mathcal{O}(M)$. Each reducer is required to run in polynomial time in n.

As usual, the size of the output of individual reducers is not explicitly restricted. Also note that this model takes into account that the final output does not necessarily has to be small, the only requirement is that it fits in the global amount of available space.

Further, [53] makes a distinction between the output of a reducer that is intended as the input for the next round and the output which is part of the final output of the algorithm and may be excluded from the computation prematurely. Formally, a reducer k in round i transforms its input $V_{k,i}$ into a multiset of key-value pairs $W_{k,i+1}$ and a multiset of key-value pairs O_i. $W_{k,i+1}$ serves as input for the next round, O_i is part of the final output. The final output itself is now defined as $\bigcup_{i \geq 1} O_i$.

Clearly this model is closely related to the $\mathcal{MRC}$ model in [39]. Moreover, if $m \in \mathcal{O}(n^{1-\varepsilon})$ and $M \in \mathcal{O}(n^{2-2\varepsilon})$ for a fixed $\varepsilon > 1$ then $MR(m, M)$ fits in the $\mathcal{MRC}$ bounds (with exception of the polylogaritmic number of rounds).

5.11 Replication Rate and Reduce-Record Size

Many problems and algorithms do not fit so well in the MapReduce model as, for example, the canonical Word-Count problem that we described in Section 5.6. Generally, when the amount of parallelization increases, the communication cost increases as well. Therefore, in this section, we take a look at the replication rate, that is, the overhead in communication cost compared with the input size. Formally, the replication rate is defined as below [6]:

$$R_r = \sum_{i=1}^{r} \frac{m_i}{|I|},$$

where r is the number of reducers and for every $i \in [1, r]$, m_i is the size of the reduce record associated to reducer i.

Example: Recall the Word-Count algorithm from section Section 5.6, where every *word* is mapped on a tuple $\langle word; 1\rangle$ and every reducer counts the values of the pairs that it receives. Since every input pair is mapped on exactly one pair, it follows that $\sum_{i=1}^{r} m_i = |I|$. Hence, $R_r = 1$ for this algorithm.

The replication rate of the join algorithm in Section 6.2 is an example where $R_r > 1$.

Let m be an upper bound on the size of the reduce records. In [6] a recipe is provided to find lower bounds on the replication rate for a problem, where the amount of memory for reducers is bounded by m. Since problems can have multiple mapper implementations, or *mapping schemas*, we have to take all of these into account. However, only schemas that satisfy the following conditions should be considered:

- the input size of every reducer is bounded by m; and
- for every output pair there is a reducer that receives as input the required pairs to be able to generate that particular output pair.

Now, the recipe goes as follows:

- find an upper bound on the number of outputs that a reducer with input size m can generate, write $g(m)$;
- count the total number of inputs and outputs, $|I|$ and $|O|$ respectively.

When there are r reducers, the reduce-record size is bounded by m, and, consequently, the output size for reducers is bounded by $g(m)$, it follows that:

$$\sum_{i=1}^{r} g(m_i) \geq |O|. \tag{5.1}$$

Next, we have to manipulate Equation (5.1) to find a lower bound on the replication rate. However, it turns out that, when $g(m_i)/m_i$ is monotonically increasing in m_i, we can use the fact that $m_i \leq m$ to find one such lower bound [6]:

$$\sum_{i=1}^{r} m_i \frac{g(m)}{m} \geq \sum_{i=1}^{r} m_i \frac{g(m_i)}{m_i} \geq \sum_{i=1}^{r} g(m_i) \geq |O|. \tag{5.2}$$

In particular, Equation (5.2) leads to the following result:

$$R_r = \sum_{i=1}^{r} \frac{m_i}{|I|} \geq \frac{m.|O|}{g(m).|I|}.$$

This technique is used in Section 6.5 to provide a lower bound on the replication rate for joins.

5.12 $\mathcal{MRC}$ Parallelizable Functions

In order to prove that an algorithm is in $\mathcal{MRC}$ we can think of it as a composition of subroutines. In general, however, even for these subroutines the $\mathcal{MRC}$ bounds are not trivial to meet. Many times we need to go into a lot of superfluous detail while designing MapReduce algorithms to make sure they fit in the $\mathcal{MRC}$ bounds. Especially to avoid overloading of individual reducers.

In this respect [39] introduces the so called $\mathcal{MRC}$-parallelizable functions. Intuitively, an $\mathcal{MRC}$-parallelizable function is a function that can be executed in two MapReduce steps, such that the function can be solved within the $\mathcal{MRC}$-bounds. In other words, when some conditions are met, we may assume that there exists a MapReduce algorithm that implements the desired function such that it fits in the $\mathcal{MRC}$ bounds.

Definition 5.3: A function f on a set S is *$\mathcal{MRC}$-parallelizable* if there are functions g and h such that:

- for any partition $\{T_1, T_2, \ldots, T_k\}$ of S, where $\bigcup_{1\leq i\leq k} T_i = S$ and $T_i \cap T_j = \emptyset$ for each $i \neq j$ in $[1, k]$, we have $f(S) = h(g(T_1), g(T_2), \ldots, g(T_k))$;
- functions g and h can be expressed in $\mathcal{O}(\log n)$ bits; and
- functions g and h can be computed in polynomial time in the size of S and every output of g can be expressed in $\mathcal{O}(\log n)$ bits.

This definition gives rise to an interesting lemma which generalizes the essence of this definition to families of $\mathcal{MRC}$-parallelizable functions on subsets of the same universe.

Lemma 5.1 (Functions lemma [39]): *Consider a universe $\mathcal{U}$ of size n and a collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ of subsets of $\mathcal{U}$, where $S_i \subset \mathcal{U}$, $\Sigma_{i=1}^{k} |S_i| \leq n^{2-2\varepsilon}$ and $k \leq n^{2-3\varepsilon}$. Let $\mathcal{F} = \{f_1, f_2, \dots, f_k\}$ be a collection of $\mathcal{MRC}$-parallelizable functions. Then the output $f_1(S_1), \dots, f_2(S_2), \dots, f_k(S_k)$ can be computed using $\mathcal{O}(n^{1-\varepsilon})$ reducers each with $\mathcal{O}(n^{1-\varepsilon})$ space.*

We first give the MapReduce algorithm for such a subroutine and then prove that it fits in the $\mathcal{MRC}$ bounds. Assume that we have $R = n^{1-\varepsilon}$ reducers grouped in blocks of size $B = n^{\varepsilon}$. This implies that we have $t = n^{1-2\varepsilon}$ blocks of reducers.

As input we assume for each $i \in [1, k]$ and each $u \in S_i$ a pair of the form $\langle i; u \rangle$. We also consider the functions g_i and h_i as part of the input. Further we assume $hash_1, hash_2 : [1, k] \to [1, t]$ to be uniform hash functions.

In the first step, map each $\langle i; u \rangle$ on $\langle r; (u; i) \rangle$, where r is a reducer, uniformly chosen from the block $hash_1(i)$. We map each function g_i and h_i on every reducer in the block $hash_1(i)$. Each reducer runs g_i on the fragments $T_j \subseteq S_i$ that it receives and outputs the record $\langle r; (g_i(T_j), i, h_i) \rangle$.

In the next step, map every reducer output onto a pair $\langle hash_2(i); (g_i(T_j), h_i) \rangle$. The reducers in this step receive every partial result $g_i(T_i)$ for certain $i \in [1, k]$ and the corresponding functions h_i. Consequently, every reducer computes the final output,

$$\langle r; h_i(g_i(T_1), g_i(T_2), \dots, g_i(T_B)),$$

for each i that it received.

Since it is a classical result that $\log^k n = o(n^{\varepsilon})$ for every $\varepsilon > 0$, we have that, for sublinear bounds of the form $n^{1-\varepsilon}$, the soft-O and big oh notation can be interchanged.

Lemma 5.2: *For each $\varepsilon > 0$ we have that there exists a $\varepsilon' > 0$ such that $\tilde{\mathcal{O}}(n^{1-\varepsilon})$ is in $\mathcal{O}(n^{1-\varepsilon'})$.*

Proof: Let ε be an arbitrary number such that $\varepsilon > 0$. By definition we have that $\tilde{\mathcal{O}}(n^{1-\varepsilon})$ is a shorthand for $\mathcal{O}(n^{1-\varepsilon} \log^k n)$ for some k.

Let ε be an arbitrary number such that $\varepsilon > \varepsilon' > 0$ (note that there always exists such an ε'), and let $\delta = \varepsilon - \varepsilon'$. Clearly, $\delta > 0$. As we stated earlier, in asymptotic analysis $n^{1-\varepsilon} \log^k n$ is bounded by $n^{1-\varepsilon} n^{\delta}$. Therefore we have that $\tilde{\mathcal{O}}(n^{1-\varepsilon})$ is in $\mathcal{O}(n^{1-\varepsilon'})$. □

The correctness of this MapReduce algorithm is easy to verify from the definition of $\mathcal{MRC}$-parallelizable functions. Next, we show that it fits in the $\mathcal{MRC}$ bounds.

Clearly, every mapper ander reducer takes only a polynomial amount of time. The total number of mapper outputs in the first step is bounded by $\mathcal{O}(n^{2-2\varepsilon} + \log n . n^{\varepsilon})$ and in the second step is bounded by $\mathcal{O}(\log n . n^{\varepsilon} . n^{2-3\varepsilon})$. Next, we prove that no reducer gets overloaded in the first and second step. Although we left out a formal proof for the space bounds, analogous arguments can be used for space.

Lemma 5.3: *Each reducer in the first step has $\tilde{\mathcal{O}}(n^{1-\varepsilon})$ elements mapped on it with high probability.*

Proof: Since fragments are mapped uniformly at random on particular reducers in a block, it suffices to prove that every block receives at most $\tilde{\mathcal{O}}(n)$ elements with high probability.

First, group subsets based on their size. Let $G_j := \{S_i \in \mathcal{S} \mid 2^{j-1} < |S_i| \leq 2^j\}$. Since $|S_i|$ is bounded by n, we have $\log n$ groups. Define the volume of group G_j as $V_j := |G_j|.2^j$, which is the maximum number of pairs that can occur in group G_j.

Even if we map every group that has a volume which is smaller or equal than $n \log n$ on the same block of reducers, we do not violate any input restriction. This statement is true since we have at most $\log n$ groups and thus at most $\mathcal{O}(n \log^2 n)$ pairs.

In the remainder of the proof we focus on groups where $V_j > n \log n$. From the definition of G_j and V_j we derive that G_j has between $n \log n/2^j$ and $2n \log n/2^j$ elements. Fix a block of reducers. Let X_i be a variable that has value 1 if $S_i \in G_j$ maps on the fixed block of reducers and 0 otherwise. Since there are $n^{1-2\varepsilon}$ blocks, the chance of $S_i \in G_j$ to map on that particular block is $n^{2\varepsilon-1}$. Let X be the number of subsets in G_j that map on the fixed block of reducers, that is $X = \Sigma_{\{i \mid S_i \in G_j\}} X_i$. These variables are independent of eachother. The expectation μ of X is $2n^{2\varepsilon} \log n/2^j$. When we apply the Chernoff bound from Section 2.5 where $\delta = n^{1-2\varepsilon}$, we find that:

$$2^{-(1+\delta)\mu} = n^{-2^{1-j}(n^{2\varepsilon}+n)} \tag{5.3}$$

$$\leq n^{-2(n^{2\varepsilon-1}+1)} \tag{5.4}$$

$$\leq \frac{1}{n^2}. \tag{5.5}$$

Equation (5.4) follows from the fact that j is at most $\log n$. Equation (5.5) is a result of $n^{2\varepsilon-1}$ being greater than zero.

The chance that the fixed reducer will get too much content from any group G is bounded by $\mathcal{O}(\log n/n^2)$. The chance that any reducer gets too much content from any group is bounded by $\mathcal{O}(n^{2\varepsilon} \log n/n^3)$. Since $\varepsilon < 1$ we have $\mathcal{O}(\log n/n)$. □

Lemma 5.4: *With high probability each reducer in step two receives at most $n^{1-\varepsilon}$ values of g_i.*

Proof: $hash_2$ is universal and maps every set on a fixed reducer of a block. The sum of fragmented outputs $\Sigma_j |g_i(F_j)|$ for certain i is bounded by $n^{\varepsilon}. \log n$ by the number of reducers per block and the definition of $\mathcal{MRC}$-parallelizable functions.

By expectation the number of sets mapped on a particular block is k/t. If $k < t$ then every set can be mapped on its own block. If $k \geq t$ we have $k/t \leq n^{2-3\varepsilon}/n^{1-2\varepsilon} = n^{1-\varepsilon}$. Therefore, by expectation, the size of the input for each block is bounded by $\tilde{\mathcal{O}}(n^{1-\varepsilon})$.

We now prove that the size of the input for every reducer (i.e. every block) is at most $\tilde{\mathcal{O}}(n^{1-\varepsilon})$ with high probability.

$$\Pr[X > (1 + \log n)\frac{k}{t}] < 2^{-(1+\log n)\frac{k}{t}} \leq \frac{1}{n}.$$

Since there are $n^{1-2\varepsilon}$ blocks, the probability that any block gets overloaded is bounded by $\mathcal{O}(1/n^{2\varepsilon})$. □

Example: Let $s = a_1 a_2 \dots a_n$ be a string of characters from Σ. The k^{th} *frequency moment* of s is defined as $\sum_{a\in\Sigma} |S_a|^k$, where $|S_a|$ represents the number of occurrences of a in s. Let $f_1(S) = |S|^k$ and $f_2(S) = (\sum_{x\in S} x)$. Hence,

$$\sum_{a\in\Sigma} |S_a|^k = f_2(\{f_1(S_a) | a \in \Sigma\}).$$

Let $\{T_1, T_2, \dots, T_k\}$ be an arbitrary partitioning of S_a. Let $g(T) = |T|$ and

$$h(x_1, x_2, \dots, x_k) = |\sum_{i=1}|^k.$$

Hence, f_1 is an $\mathcal{MRC}$-parallelizable function. Similar, for any partitioning $\{T_1, T_2, \dots, T_m\}$ of $\{f_{a_1}, f_{a_2}, \dots, f_{a_n}\}$ we have $g(T) = \sum_{x\in T} x$ and

$$h(x_1, x_2, \dots, x_m) = \sum_{i=1}^{m} x_i.$$

Hence, f_2 is an $\mathcal{MRC}$-parallelizable function. Therefore, from the functions lemma follows that the k^{th} frequency moment can be computed in $\mathcal{MRC}$.

6 Conjunctive Queries in MapReduce

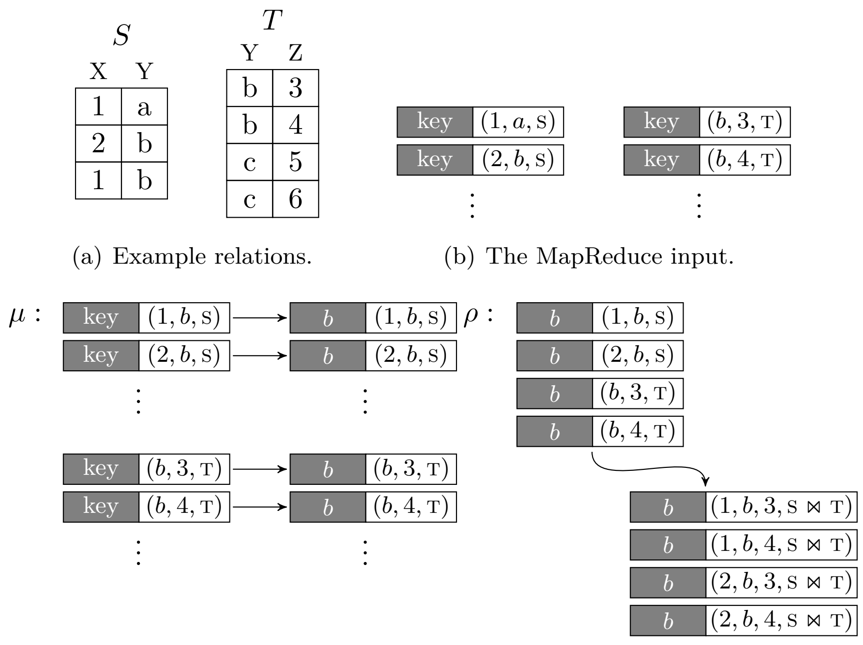
In Chapter 3, the complexity of conjunctive queries is described in a sequential setting. In this chapter we discuss approaches for the evaluation of conjunctive queries in MapReduce. Essentially, a conjunctive query consists of join operations and a projection. It is, however, a common critique to MapReduce that it does not deal well with the popular join operation [51, 13, 43, 50], though a fundamental operator in conjunctive queries. Therefore, we first discuss two strategies to compute full two-way joins in MapReduce and eventually discuss briefly the generalization to multi-way joins.

6.1 Data-Centered Join

Before going into detail on particular join algorithms, note that MapReduce only allows a single multiset of key-value pairs as its input, and that the join operator by definition expects two input relations as its input. Fortunately, it is easy to simulate the effect of having multiple input arguments by adding a origin reference to the value of each key-value pair. Therefore, a join between relation R and S can be simulated on the MapReduce framework by taking as input the union of all the tuples from R and S, where each tuple has an additional value that refers to the relation R or S respectively.

One of the most natural algorithms to perform a full join $S(X, Y) \bowtie T(Y, Z)$ in MapReduce is partitioning the tuples based on the join-keys and joining the resulting fragments in parallel [7, 13, 43]. We call this algorithm *data-centered* since its fragmentation is entirely based on the input instances, in contrary to the *reducer-centered* approach in Section 6.2. In the literature, this algorithm is also referred to as the *repartition join* [13].

Let S and T be relation schemas. As input to the MapReduce program assume for each tuple from S and T a key-value pair where the value field contains both the tuple and a reference to its origin table (i.e., S or T). Next, map every key-value pair onto a pair with key the value of the join attributes of S and T. This algorithm is illustrated in Figure 6.1. Note that, since MapReduce only performs a sort on the keys, the reduce



(a) Example relations. (b) The MapReduce input.

(c) The map and reduce function.

Figure 6.1: Data-centered join in MapReduce.

function has to buffer both the partition of S and the partition of T to be able to perform the join. Moreover, when the data is skewed, the input size of an individual reducer can be very large and it may be impossible to keep both fragments in memory.

Practical Improvements

Several practical improvements to the previously described approach exist based on implementation features of MapReduce and Hadoop [13, 43]. One such improvement is described in [13]. In essence it is the repartition algorithm as described before, but instead of performing a sort based only on the join key, a sort is performed on both the join key and the relation name. Therefore, the reducer will have to keep only one of the fragments in memory and will be able to stream the other. Although MapReduce does not perform a sort on values explicitly, it is possible to simulate this effect by exploiting some customizable functions.

We use the same input encoding as before. The map function maps every key-value pair onto a pair where the key is based on both the join key and the relation name. Consequently, the shuffle phase will perform a sort based on both arguments. By defining a customized partitioning function which is only based on the join key, we are able to overcome the fragmentation of tuples with the same join key, which would naturally occur. We need another customized function to group the reducer inputs based only on the join key, to ensure that both tuples of R and tuples of S that have the same join key will be processed together as input to the reduce function.

This algorithm depends on the ability to customize the fragmentation and grouping function. From a theoretical perspective, the worst-case complexity is the same as the general approach above.

Complexity

Let n be the sum of $|S|$ and $|T|$, without loss of generality assume that $|S| \leq |T|$. An important observation is that the shuffle overhead in this algorithm is very low since this approach requires only a single MapReduce step and the replication rate is 1. On the other hand, it relies entirely on the structure of the input data for the distribution of its intermediate pairs among reducers. Hence, it is a very inflexible approach.

In general, the size measures of key and sequential complexity take into account both the size of the input and output. Since it is typical for a join operator to have an output that is larger than its input, in what follows we make a distinction between input size and output size.

Proposition 6.1: *The data-centered join can be computed in a single step.*

- ***Key complexity:*** *the input size and amount of space are bounded by $\mathcal{O}(|S| + |T|)$; the output size is bounded by $\mathcal{O}(|S \bowtie T|)$; and the running time is bounded by $\mathcal{O}(|S||T|)$.*
- ***Sequential complexity:*** *the same bounds apply.*

Proof: Clearly, since every input pair is mapped on a single reducer, the total communication cost (or sequential input size) is bounded by $\mathcal{O}(|S| + |T|)$. The sequential output size, however, may be much larger and is bounded by $\mathcal{O}(|S \bowtie T|)$. Since we need to compare every input tuple from S to every input tuple from T, the sequential time to compute this algorithm is bounded by $\mathcal{O}(|S||T|)$.

As noted before, the distribution of input tuples to particular reducers depends entirely on the input data. Therefore, in the worst-case scenario, every tuple has the same value for its join attribute, and consequently, a single reducer has to do all the work. Therefore, the key input size, output size and time equal the sequential input size, output size and time, that is, they are bounded to $\mathcal{O}(|S| + |T|)$, $\mathcal{O}(|S \bowtie T|)$, and $\mathcal{O}(|S||T|)$ respectively. Since the entire input of a reducer has to be buffered, there is no need to differentiate between streamed or batched input. Hence, the key memory of this algorithm is bounded by $\mathcal{O}(|S| + |T|)$. □

The improved data-centered algorithm may be more feasible in practice, but yields no improvements from a theoretical perspective. Although we need to buffer nothing but the smallest relation, we have no guarantees about the relative size of S and T. Therefore, the key complexity of the improved repartition algorithm is still $\mathcal{O}(|S| + |T|)$ for the input size, $\mathcal{O}(|S \bowtie T|)$ for output size, and $\mathcal{O}(|S|.|T|)$ for time. For streamed inputs, the key memory is decreased to $\mathcal{O}(|S|)$, but may still be as high as $n/2$. An equivalent analysis can be performed for the sequential complexity, i.e., $\mathcal{O}(|S|.|T|)$ for time, $\mathcal{O}(|S|)$ for input size, and $\mathcal{O}(|S \bowtie T|)$ for output size.

6.2 Reducer-Centered Join

A disadvantage in the previously described algorithm is that data is partitioned based on the structure of the data itself. A more interesting approach would be to partition the input in a way that is independent of the input's structure.

If one of the relations is much smaller than the other relation, say $|S| \ll |T|$, it easy to achieve an independent fragmentation by performing a broadcast join and mapping the smallest relation to every reducer. Although the total communication cost increases, we have the opportunity to split the larger relation in a more balanced way in order to overcome reducer overloading.

A fundamentally different approach to perform joins and crossproducts in MapReduce than the data-centered approach, for the case when S is not much smaller than T, is proposed in [50]. They introduce the notion of *max-reducer-input* and *max-reducer-output* to denote the maximum input size and output size for reducers, which is used to evaluate the performance of join algorithms. The goal of the proposed algorithm is to balance the output size of reducers and optimize the input size based on the size of the output.

The main concept is to consider the *join matrix*, where each row represents a tuple from one relation and every column represents a tuple from the other relation. Consequently, the matrix represents the cross product $S \times T$, and every cell represents a tuple in this cross product. In terms of a join we can see a cell as a potential entry in the join $S \bowtie T$. Essentially, we try to map regions on this matrix such that every region has approximately the same size and the associated number of rows and columns is near optimal. An example join matrix and reducer mapping is illustrated in Figure 6.2(a).

The algorithm to perform the join will consist of R reducers, where every reducer is associated to a particular region. Although we are not really interested in cross products, this method provides some interesting upper bounds. Nonetheless these bounds do not guarantee a balanced number of output tuples for joins in general. In practice, this algorithm may still suffer from data skew up to a certain amount (i.e., restricted by the upper bounds). Furthermore, it is not obvious how to replace $|S|.|T|$ with $|S \bowtie T|$ in the following propositions. Note that, in this algorithm, the optimal choice for R is the number of available machines. Therefore, the number of reduce keys and the number of reduce workers coincides by assumption.

Proposition 6.2 ([50]): *Let S and T be relations and $R > 0$. If $|S| < |T|/R$ then the join matrix can be covered such that every region has $\mathcal{O}(|S|.|T|/R)$ cells and the sum of rows and columns is bounded by $\mathcal{O}(|S| + |T|/R)$.*

Proof: If $|S| < |T|/R$ then divide the matrix into R regions by fragmenting the tuples of T into R equal sized partitions. Clearly, the surface of every region has size $|S|.|T|/R$ and the number of columns and rows for each region is $|S| + |T|/R$. □

An example of this mapping is illustrated in Figure 6.2(b). Next, a proof is given for upper bounds for the complement result.

Proposition 6.3 ([50]): *Let S and T be relations and $R > 0$. If $|T|/R \leq |S| \leq |T|$ then its join matrix can be covered such that every region has $\mathcal{O}(|S|.|T|/R)$ cells and the sum of rows and columns is bounded by $\mathcal{O}(\sqrt{|S|.|T|/R})$.*

Proof: Let c_S and c_T be as below:

$$c_S = \left\lfloor \frac{|S|}{\sqrt{\frac{|S||T|}{R}}} \right\rfloor$$

$$c_T = \left\lfloor \frac{|T|}{\sqrt{\frac{|S||T|}{R}}} \right\rfloor$$

Here, c_S and c_T represent the number of optimal squares that fit in the size of S and T respectively. When we map regions of height and width $\sqrt{|S||T|/R}$ onto the join matrix starting at the upper left, however, it is likely that at the bottom and upper right of the matrix there are some cells uncovered. This situation is illustrated in Figure 6.2(c). Clearly, the number of uncovered rows and columns is less than $\sqrt{|S||T|/R}$, and therefore we are able to cover these cells by increasing the height and width of the existing R regions by a factor that is not higher than $1 + 1/\min\{c_S, c_T\}$. Since $c_S, c_T \geq 1$, as demonstrated by Equations (6.1) and (6.2), this factor is at most 2. Consequently, when $|T|/R \leq |S| \leq |T|$, the number of columns and rows for each region is bounded by $4\sqrt{|S||T|/R}$ and the number of cells in each region is bounded by $4|S||T|/R$.

$$c_S = \left\lfloor \frac{|S|}{\sqrt{\frac{|S||T|}{R}}} \right\rfloor \geq \left\lfloor \frac{|S|}{\sqrt{\frac{|T|^2}{R}}} \right\rfloor = \left\lfloor \frac{|S|}{\frac{|T|}{\sqrt{R}}} \right\rfloor \geq \left\lfloor \frac{|S|}{\frac{|T|}{R}} \right\rfloor \geq 1 \tag{6.1}$$

$$c_T = \left\lfloor \frac{|T|}{\sqrt{\frac{|S||T|}{R}}} \right\rfloor \geq \left\lfloor \frac{|T|}{\sqrt{|S|^2}} \right\rfloor = \left\lfloor \frac{|T|}{|S|} \right\rfloor \geq 1 \tag{6.2}$$

□

Remark that a reducer has to receive at least $\sqrt{c}$ tuples to be able to generate c output tuples [50]. Therefore, the result above is rather tight.

The above insights give rise to a MapReduce algorithm where every reducer receives key-value pairs based on the precomputed regions of the join-matrix. Unfortunately, it would require a lot of effort to associate every key-value pair to a particular column or row in advance. Therefore, [50] proposes a randomized alternative in which the regions are computed in advance based on the size of S and T, but the key-value pairs are associated to a particular row or column uniformly at random by the mappers.

Since tuples are mapped uniformly at random on particular reducers, it can be verified that significant variations on the number of tuples mapped on a particular reducer are very unlikely. This can be done by an application of the Chernoff Bound as in [50].

Propositions 6.2 and 6.3 provide upper bounds on the input and output size for individual reducers. Hence, we have the following complexity results:

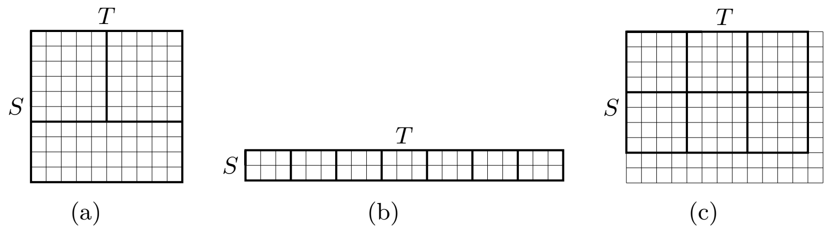


Figure 6.2: Illustration of region mappings on join matrices.

Proposition 6.4: *Let S, T be relations such that $|S| \geq |T|/R$. The reducer-centered approach computes the join $|S \bowtie T|$ in a single MapReduce step.*

- ***key complexity:*** *The input size is bounded by $\mathcal{O}(\sqrt{|S||T|/R})$. The output size, running time, and space are bounded by $\mathcal{O}(|S||T|/R)$.*
- ***sequential complexity:*** *the total input size is bounded by $\mathcal{O}(\sqrt{R|S||T|})$. The output size is bounded by $\mathcal{O}(|S \bowtie T|)$. The running time is bounded by $\mathcal{O}(|S||T|)$.*

Replication Rate

First, we analyze the complexity of the reducer-centered join for the case $|S| < |T|/R$. As in the data-centered join, only a single MapReduce step is required, but now, the replication rate is bounded by 2.

Proposition 6.5: *For the reducer-centered approach, when $|S| < |T|/R$, we have that $R_r < 2$.*

Proof: From $|S| < |T|/|R|$ and Proposition 6.2, we have that:

$$R_r = \frac{R(|S| + \frac{|T|}{R})}{|S| + |T|} < \frac{2|T|}{|T|} \leq 2. \tag{6.3}$$

□

Concerning key complexity, we have that the input size, as well as reducer memory, is bounded by $\mathcal{O}(|S|+|T|/R)$. The output size and running time is bounded by $\mathcal{O}(|S||T|/R)$. The sequential complexity is $\mathcal{O}(R|S|+|T|)$ for input size, $\mathcal{O}(|S \bowtie T|)$ for output size, and $\mathcal{O}(|S||T|)$ for time.

For the case $|S| \geq |T|/R$, a single MapReduce step suffices, but the replication rate is less convenient.

Proposition 6.6: *When $|S| \geq |T|/R$, the replication rate of the reducer-centered approach is bounded by $4\sqrt{R}$.*

Proof: If $|T|/|R| \leq |S|$ then it follows from Proposition 6.3 that the replication rate equals the leftmost expression of Equation (6.4):

$$\frac{4R\sqrt{\frac{|T||S|}{R}}}{|S|+|T|} \leq \frac{R\sqrt{\frac{|n|^2}{R}}}{n} \leq 4\sqrt{R} \tag{6.4}$$

□

Replication rate is often expressed as a function of the available amount of memory per reducer [6]. Therefore, we rewrite this expression. Let m be the amount of space that is available for individual reducers. For the algorithm to work correctly, m and R should agree to Equation (6.5), or equivalent, to Equation (6.6).

$$m \geq 4\sqrt{\frac{|S||T|}{R}} \tag{6.5}$$

$$R \geq \frac{16|S||T|}{m^2} \tag{6.6}$$

Since we don't need more than $16|S||T|/m^2$ space, it is safe to assume that R equals this expression, and hence, to replace the R in Equation (6.4) by the right most expression in Equation (6.6). Therefore we have the following corollary.

Corollary 6.1: *When $|S| \geq |T|/R$, the replication rate of the reducer-centered approach is bounded by $16\sqrt{|S||T|}/m$, where m denotes the available size for individual reducers.*

6.3 A Hybrid Approach

The data-centered approach is a very efficient approach if the frequencies of join attributes are not skewed and are small enough to let every reduce record fit in a single machine. The reducer-centered approach yields an overhead in communication cost, but is guaranteed to utilize parallelization independent from the input dataset; therefore, it is able to handle even those datasets that are highly skewed or contain join attributes with a high frequency. In this section, we take a look at a hybride approach for performing two-way joins that combines both approaches.

Assume that every tuple contains frequency information (i.e., instead of tuples $\langle k, (a, b)\rangle$ we have tuples $\langle k, (a : f, b)\rangle$, where a is a join key and f is its frequency). Based on this information, the map function can decide if the data-centered approach or the reducer-centered approach is preferred. In fact, we split the computation in two separated parts: the low-frequency tuples are joined in a data-centered approach, and the high-frequency tuples are joined in the reducer-centered approach. The main improvement is that by discarding the low-frequency tuples from the join-matrix, the overhead of the reducer-centered approach is reduced.

However, an important aspect of this new approach is that frequency information has to be available. On the one hand, the problem of counting frequencies is equivalent to the word count problem, which is an embarrassingly parallel problem. But on the other hand, a merge between the frequencies and the input set should take place; in other words, a two-way join between the frequency relation and the input relations. Therefore, even when the frequency relation is small (i.e., the number of distinct join values is small) computing frequencies for purpose alone leads to a huge overhead in running time.

6.4 Joins in $\mathcal{MRC}$

The data-centered approach does not fit in the $\mathcal{MRC}$ bounds. Nevertheless, it may be reasonable to assume that the cardinality of join keys is restricted in practice.

First note that it is a property of the join operation that its output may be much larger than its input. Therefore the analysis below states on two assumptions:

- there is enough space to store the entire output, which may be as high as $\mathcal{O}(n^2)$; and
- reducers are able to stream there output pairs.

Note that the first assumption is not a necessity to fit in $\mathcal{MRC}$ since there are no explicit bounds on the size of reducer outputs and the approaches described earlier require only a single MapReduce step.

Property 6.1: *Let S and T be relations such that the number of tuples that match an arbitrary value for the join attributes is bounded by $\mathcal{O}(n^{1-\varepsilon})$. Then, $S \bowtie T$ is computable in $\mathcal{MRC}^0$ with the data-centered join algorithm.*

Proof: Clearly, every reducer receives at most $\mathcal{O}(n^{1-\varepsilon})$ input pairs and the total number of mapper outputs is bounded by $\mathcal{O}(n)$. Therefore, it takes at most $\mathcal{O}(n^{2-2\varepsilon})$ time to compute the output for a single reducer. Although the output of a reducer may be high, there is no need to keep track of the produced tuples, and hence, streaming the output tuples makes this algorithm fit in $\mathcal{MRC}^0$. $\square$

The reducer-centered join algorithm looks more promising. Indeed, we are able to compute a simple join with the reducer-centered approach in $\mathcal{MRC}^0$ without restrictions on the input.

Proposition 6.7: *Let S and T be relations. $S \bowtie T$ is computable in $\mathcal{MRC}^0$ with the reducer-centered join algorithm.*

Proof: Let $R = n^{2\varepsilon}$, where $1/3 \geq \varepsilon > 0$. Consequently $R \in \mathcal{O}(n^{1-\varepsilon})$. We prove that the reducer-centered join algorithm fits in the $\mathcal{MRC}^0$ bounds by breaking the problem into two distinct cases.

When $|S| < |T|/R$ every reducer receives at most $\mathcal{O}(|S| + |T|/R)$ input pairs. Consequently, the input of every reducer is bounded by $\mathcal{O}(n/R)$, which is $\mathcal{O}(n^{1-2\varepsilon})$. The time to compute the output is at most $\mathcal{O}(n^2/R^2)$. The total size of mapper outputs is bounded by $\mathcal{O}((|S| + |T|/R).R)$ and thus bounded by $\mathcal{O}(n)$. Since we assume that it is possible to stream the output of a reducer, the amount of space equals the size of the input and hence, for $S < |T|/R$, this proves the proposition.

When $|T|/R \leq |S| \leq |T|$ we have that the size of reducer inputs is bounded by $\mathcal{O}(\sqrt{|S||T|/R})$ and consequently by $\mathcal{O}(n^{1-\varepsilon})$. The total number of mapper pairs is bounded by $\mathcal{O}(\sqrt{|S||T|/R}.R)$ and since $\varepsilon \leq 1/3$ we have that $\mathcal{O}(n^{1+\varepsilon})$ is in $\mathcal{O}(n^{2-2\varepsilon})$. The time for a reducer to compute its output is bounded by $\mathcal{O}(n^2/R)$. The amount of required space again is equal to the size of the input. □

6.5 Lower Bound on Replication Rate

Given a set of size m containing tuples from S and T, the output of a join of these tuples is at most $m^2/4$. That is, the worst-case situation is that we have $m/2$ tuples of S and equally much tuples from T, where every tuple has the same value for its join attribute. In terms of the recipe from Section 5.11 we have $g(m) = m^2/4$.

The input size is bounded by $|S| + |T|$ and the output size is bounded by $|S \bowtie T|$. Therefore, we have that:

$$\sum_i g(m_i) \geq |S \bowtie T|,$$

and by use of the trick in Section 5.11 that:

$$R_r = \frac{\sum_i m_i}{|S| + |T|} \geq \frac{4|S \bowtie T|}{m(|S| + |T|)}.$$

Clearly, the trivial bound $R_r \geq 1$ should be considered as well.

6.6 The Projection Operator

In the previous sections we described two approaches for computing full two-way joins, i.e., queries of the form $ans(x, y, z) \leftarrow S(x, y) \wedge (y, z)$. It is easy to add a projection to these types of queries by computing the projection for each reducer output before emitting the output tuples. Moreover, the addition of a projection has no influence on the previously described complexity results.

6.7 Multi-way Joins

An intuitive generalization of two-way joins is the iterative implementation of multi-way joins, e.g., $R \bowtie S \bowtie T$ by first computing $R \bowtie S$ and then computing the join of the result of $R \bowtie S$ and T. Unfortunately, this approach yields a very high complexity. In the first place, it takes m MapReduce steps, where m is the number of relations to join. Further, as we have seen in Chapter 3, the intermediate results of a multi-way join can become very large when compared to the actual output. The following proposition gives rough upper bounds on the complexity of this approach.

Proposition 6.8: *Let $T_1, T_2, \ldots, T_m$ be relations. The iterative implementation of two-way joins to compute $T_1 \bowtie T_2 \bowtie \ldots \bowtie T_m$ can be done in $m-1$ steps.*

- ***Key complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{|T_1||T_2| \ldots |T_m|/R})$; the output size, running time and space are bounded by $\mathcal{O}(|T_1||T_2| \ldots |T_m|/R)$.*
- ***Sequential complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{R|T_1||T_2| \ldots |T_m|})$; the output size and running time are bounded by $|T_1||T_2| \ldots |T_m|$.*

The output size of the last step, of course, equals $|T_1 \bowtie T_2 \bowtie \ldots \bowtie T_m|$, but the (sequential) inputs and outputs in the intermediate steps may be much larger than that. However, for simplicity we assumed that no (intermediate) relation is significantly smaller (as in $|S| < |T|/R$) or larger than the relation to join with.

In [7] Afrati and Ullman extend the fragmentation join algorithm to a ternary join algorithm in one step. Take $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$, and assume tuples of S as reduce key. Hence, every tuple of S is mapped on a particular reduce bucket. Tuples $(a,b) \in R$ are mapped on several buckets, i.e., every bucket that is associated to a tuple $(b,*)$. On the other hand, tuples of $(b,c) \in T$ are mapped on every bucket that is associated to a tuple $(*,c)$. Clearly, in practice the number of reducers is much less than the number of possible tuples in S, and therefore, hash functions should be used to map tuples on particular reducers. Moreover, the number of hash buckets for each attribute can be customized to further tighten the communication cost. In [7] the optimization of this approach is discussed in detail.

In the context of log processing [13] we are typically interested in the join between one very large relation and several small relations. Take for example $L(A,B,C) \bowtie S(A,D) \bowtie R(B,E)$, where L is large and S and R are small. A variation on the broadcast join can be utilized for this type of joins, by broadcasting the smaller tables to every reducer and distributing the larger table in several fragments.

Inspired from [50], in [63] a generalization of the reducer-centered join for multi-way joins is considered based on the cross-product conceptualization of multiple relations, i.e., a multidimensional matrix.

7 Undirected s-t Connectivity

The evaluation of RPQs is only briefly described in the literature. Hence, to gain a better understanding of graph problems in MapReduce, we take a look at a related problem. Undirected s-t-connectivity (STCON) is a very simple graph problem that has already been studied in the MapReduce context [18, 39, 54]. In this chapter, we give an overview of the results and algorithms for the STCON problem to allow a better understanding of graph problems in general, in MapReduce, and to get a step closer to the evaluation of RPQs and CRPQs.

7.1 Problem Definition

Definition 7.1: Let $G = (V, E)$ be an undirected graph and $s, t \in V$. *s-t-connectivity* $\langle G, s, t\rangle$ is the decision problem that asks if there is a path between s and t.

The s-t-connectivity problem is very related to the problem of finding connected components. In fact, when the connected components are known, STCON is reduced to the question if s and t are in the same connected component.

Definition 7.2: Let $G = (V, E)$ be an undirected graph. *Finding connected components* is the problem that asks for a partitioning $\{T_1, T_2, \dots, T_k\}$ of V such that $(T_i, \{(u, v) \in E \mid u, v \in T_i\})$ is a connected graph for $i \in [1, k]$, and there is no $u \in T_i, v \in T_j$ such that $(u, v) \in E$, where $i \neq j$.

To decide if $\langle G, s, t\rangle \in STCON$, it suffices to find the connected components of G and to check if both s and t are in the same partition. This approach is utilized in [18] and [39]. Unfortunately, both algorithms require a linear number of rounds in the diameter of G. More recently, [54] studied the problem of finding connected components in MapReduce in a logarithmic number of rounds.

7.2 Finding Connected Components

In this section we describe the algorithm for the connected components problem that is described in [18]. A similar algorithm is proposed in [54], called the *hash-min algorithm*, which in its turn is partially credited to [38]. In this algorithm, every node is expected to have a label that represents a zone. Initially, every node has a distinct label indicating its own distinct zone. In every round, node labels are compared to each other based on edge relations and updated accordingly. At the end of the algorithm, every node in a connected component has the same label, which indicates that hey are in the same zone.

The MapReduce model used in [18] allows reducers to generate key-value pairs that have a key which differs from the reduce key, wherefore map functions are somewhat superfluous. However, the original MapReduce definition in [19] does not allow this practice. Therefore, we present a slightly modified version of this algorithm below. For clarity, the algorithm is illustrated in Figure 7.1.

Consider both the set of edges and the set of nodes as input elements. Each edge consists of two nodes and each node in V has an associated label (or zone). Furthermore, assume the existence of an ordering π on node labels. Initially all nodes have distinct labels, e.g., their identity. Formally, the input of the MapReduce algorithm is a set of key-value pairs of the form[1]:

$$\langle k_1, e_1\rangle, \dots \langle k_m, e_m\rangle, \langle k_{m+1}, (v_1, \textsc{v}_1)\rangle, \dots, \langle k_{m+l}, (v_l, \textsc{v}_l)\rangle.$$

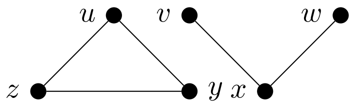
In the first step (Figure 7.1(c)), map every edge $\langle k; e\rangle$ on two key-value pairs: $\langle v; e\rangle$ and $\langle u; e\rangle$; one for each end node of e. Map every node pair $\langle k; (v, \textsc{v})\rangle$ on the pair $\langle v; \textsc{v}\rangle$. Consequently, the edges and nodes are binned per node, and the reducer associated to node v receives every edge adjacent to v and its associated label $\textsc{v}$. The reduce function generates for each edge e that it receives an output pair of the form $\langle v; (e, \textsc{v})\rangle$, where v is the node associated to the reducer, i.e., an end node of e.

In the second step (Figure 7.1(d)), map every reducer output of the previous step $\langle v; (e, \textsc{v})\rangle$ onto a pair $\langle e, \textsc{v}\rangle$ with the edge itself as the key. Consequently, every reduce record contains two labeled edges. The reduce function checks which label is the lowest according to π and outputs a pair $\langle e, (\textsc{z}, \textsc{z}^*)\rangle$ where $\textsc{z} \geq \textsc{z}^*$. Intuitively, $\textsc{z}^*$ is proposed as a replacement for the old zone represented by $\textsc{z}$.

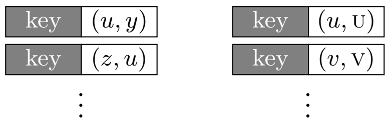
In the third step (Figure 7.1(e)) map every pair $\langle e; (\textsc{z}, \textsc{z}^*)\rangle$ onto a pair of the form $\langle \textsc{z}; \textsc{z}^*\rangle$ and map every pair $\langle v; \textsc{z}\rangle$ onto a pair $\langle \textsc{z}; v\rangle$. Intuitively, buckets receive (several) update proposals for their associated zone and the nodes that are associated to this (original) zone. The reduce function computes which of the proposed labels is the lowest in π and outputs for each node v a pair $\langle \textsc{z}; (v, \textsc{z}^*)\rangle$, where $\textsc{z}^*$ is the replacement for $\textsc{z}$.

We have to repeat these steps until convergence occurs. Call this algorithm $\textsc{stcon}_1$.

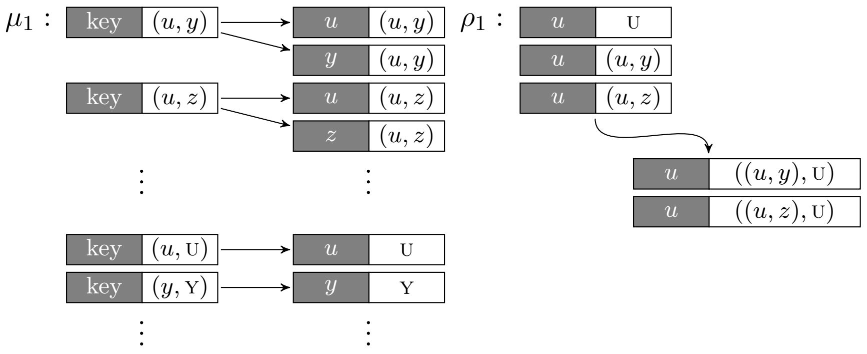
[1]For clarity, labels are written in small caps to distinguish between node-ids and node-labels.



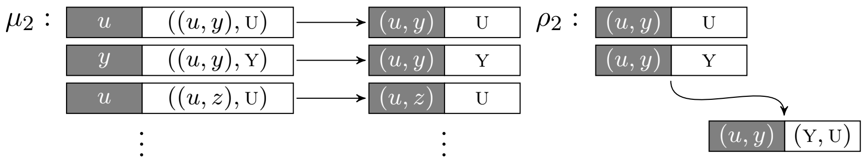
(a) The example graph.



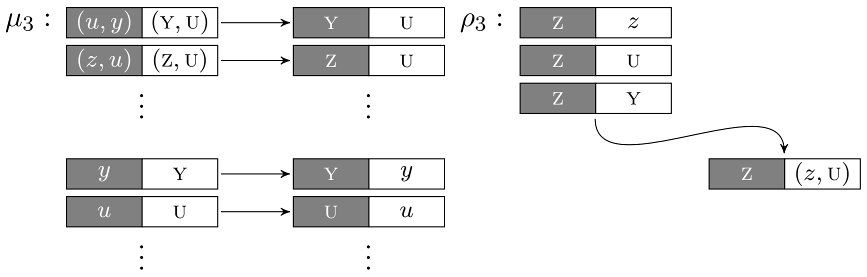
(b) The MapReduce input. We assume π to be the alphabetical ordering.



(c) The map and reduce function of the first step.



(d) The map and reduce function of the second step.



(e) The map and reduce function of the third step.

Figure 7.1: A visualization of the STCON$_1$ algorithm.

Proposition 7.1: *Let $G = (V, E)$ be an undirected graph, let d be the diameter of G, and Δ an upper bound on the degree of G.* STCON$_1$(G) *runs in $\mathcal{O}(d)$ steps.*

- ***Key complexity:*** *the input and output as well as the running time are bounded by $\mathcal{O}(|V|)$; the size is bounded by $\mathcal{O}(\Delta)$.*
- ***Sequential complexity:*** *the input and output size are bounded by $\mathcal{O}(|E| + |V|)$; the running time is bounded by $\mathcal{O}(|E| + |V|)$.*

Proof: During the algorithm every node has exactly one zone associated to it. Therefore, the number of zones is fixed throughout the algorithm and bounded by $\mathcal{O}(|V|)$.

In the first step, every bin contains a node and every adjacent edge. Therefore, the key input and output are bounded by $\mathcal{O}(\Delta)$. In the second step, the input and output is constant. Note that zone comparisons, after the first step, are not restricted to comparisons between labels of adjacent edges. Therefore, in the third step, the input and output are bounded by $\mathcal{O}(|V|)$. Clearly, the running times of the mappers and reducers are directly related to the input sizes. Hence, the running time for mappers and reducers is bounded by $\mathcal{O}(|V|)$. Since buffering of the input is only necessary for the first map phase (due to the unknown ordering), the size is bounded by $\mathcal{O}(\Delta)$ when assuming stream inputs.

The total input and output size of the first reduce phase is $2|E| + |V|$, of the second step is $2|E|$ and in the third step is $|E| + |V|$. Hence, the total input and output size is bounded by $\mathcal{O}(|E| + |V|)$. Since the running time is directly correlated with the size of the input, and every edge appears twice, the total running time in the first step is bounded by $\mathcal{O}(|E| + |V|)$. The running time in the third step depends on the number of zone comparisons, which is no more than E. Hence, the total running time of a single round is bounded by $\mathcal{O}(|E| + |V|)$.

Depending on the structure of the graph and the label ordering π, this algorithm can take up to as many iterations as there are nodes in the largest connected component. Therefore, the worst-case number of rounds is $\mathcal{O}(|V|)$. $\square$

There is no straightforward method to check if the termination condition has actually been reached. To deal with this, we can either keep track of the number of iterations and stop after d or $|V|$ steps, where d is the diameter of the input graph, or let every reducer of the third step output if a zone has changed or not. These values can be binned and reduced to a single value 'stop' or 'continue' that can be used by the coordinating application.

Note that the zone relation occurs as input in the first step and the third step of every iteration. Although it is modified in the third step, it is not in the first. In the second it not used at all. By strictly following the formal definition, as given in Section 5.1, the entire zone set has to be passed from the first step, through the second step, to the third step. In practice, however, there is no need to involve these data in the second step.

7.3 Finding Connected Components in $\mathcal{MRC}$

Clearly, the algorithm in Section 7.2 is not in $\mathcal{MRC}$. There are two main issues:

- If there are nodes with a high degree, the input size of individual reducers in step one is not guaranteed to be sublinear in the size of the input.
- The same problem occurs for the input sizes of the reducers in step three: if particular zones receive a lot of updates, the number of updates is not guaranteed to be sublinear.

In the first step of the algorithm we used the end nodes to map edges into particular buckets. Unfortunately, if a node has many edges, the size of an individual bucket can be as high as $\mathcal{O}(n)$, where $n = |V|$. An alternative to this approach is to associate blocks of buckets to each end node and map every end node uniformly at random on a particular bucket inside the associated block. Assuming blocks of size n^{ε}, we have that every reduce record has an expected size of at most $\mathcal{O}(n^{1-\varepsilon})$. Since the reduce function only marks edges with zones, the only requirement to let this new approach work is that the zone labels itself are sent to every bucket in the block of associated nodes. Hereby, increasing the communication cost to $\mathcal{O}(n^{1+\varepsilon})$.

In the last MapReduce step we used the zones as keys in the mapper outputs. Again, if many zone proposals for a particular node exist, the reduce records can be very large. Therefore, we introduce blocks of n^{ε} buckets for each node and map every zone proposal on the block associated to its target node and uniformly at random on a particular bucket. To merge the zone proposals and guarantee that every node has a unique zone proposal, we add a MapReduce step in which every remaining zone proposal is mapped on a particular bucket based on its node and the reduce function merges the input proposals to a single, smallest proposal. The size of the reduce records in the third step now is bounded by $\mathcal{O}(n^{1-\varepsilon})$ by assumption. In the newly added step the total number of mapper outputs is still bounded by the number of nodes, but the size of particular reduce records is guaranteed to fit in $\mathcal{O}(n^{\varepsilon})$.

Instead of mapping the nodes on buckets based on their zone label in step three, we skip this action to step five, where every node is mapped uniformly at random on a bucket in the associated block (every block consists of n^{ε} buckets) and we map the zone proposals, which are bounded to n^{ε} per zone, on every bucket in the particular zone. Therefore, we have a communication cost of $\mathcal{O}(n^{1+\varepsilon})$.

Proposition 7.2: *The adapted algorithm for finding connected components requires at most $\mathcal{O}(n^{1-\varepsilon})$ space per reducer, and the total size is bounded by $\mathcal{O}(n^{1-\varepsilon})$, where $\varepsilon > 0$ and fixed.*

Note that the first step of this algorithm actually is a data-centered join between $R(U, V)$ and $S(V, L)$, where R is the directed variant of E (i.e., every edge from E is represented twice in R; once for each direction) and S is the node-zone relation. Consequently, this step suffers from the same problems as the data-centered join in Section 6.1.

Therefore, instead of a mapping edges randomly on a set of reducers, the reducer-centered join (Section 6.4) could be used instead.

In [39] an algorithm for the undirected s-t-connectivity problem is described for $\mathcal{MRC}$ that runs in a logarithmic number of rounds. Let $G = (V, E)$ be an undirected graph and μ an arbitrary order on the nodes. Define $\Gamma(v)$ as the set of neighbour nodes of v and its extension $\Gamma(S)$, where $S \subseteq V$ is a set of nodes, that represents the neighbours nodes of nodes in S that are not in S itself. Further, $\Gamma'(v)$ is the set $\Gamma(\{u \mid \lambda(u) = v\})$, i.e., the set of nodes that are neighbours of nodes with label v.

- Initially, every node $v \in V$ is active and $\lambda(v) = v$.
- For $i := 1 \ldots \mathcal{O}(\log |V|)$ do
  - Call each node a leader with probability 1/2.
  - For every active non-leader node w find the smallest node $w^* \in \Gamma'(w)$.
  - If $\Gamma'(w)$ is not empty, mark w passive and relabel each node with label w to label w^*.
- Output true if s and t have the same label, false otherwise.

In [39] a proof is given that the algorithm reaches its converged state after a logarithmic number of rounds, with high probability. The proof is mainly based on the fact that nodes get marked passive with a constant factor. Furthermore, at any point if two nodes have the same label then they are in the same component.

Unfortunately, it seems that this algorithm actually has a rather high error rate and, although it has similarities with [18], the implementation in MapReduce is not very clear.

Proposition 7.3: *The algorithm described has an error rate that is above* $1/4$ *for graphs containing a chain of c nodes, where c is at least* 4*, and converges to* 1 *when c increases.*

Proof: First, remark that the label of a zone can only change if the node that is identified with that label is active. Moreover, a node with label v, where node v is passive, will have v as its label in the converged state.

In the first round every node is active and has a unique label (e.g. its identity). We assume that every node has at least one neighbor. Each node is called 'leader' with probability 1/2 and consequently each node is called 'follower' with probability 1/2. If a node is selected as a follower, we have a chance of 1/2 that its neighbor with smallest label is also selected as a 'follower'. Therefore, in the first round every node has a chance of 1/4 to change its label (and get marked passive) such that node v is also marked passive.

Now take a graph G that contains a chain with c nodes. We call one of the end nodes s and the other end node t. By construction we have that every node in the chain has at least one neighbour and every node in the component has to get the same label to be able to conclude that s and t are connected. Consequently, if at least two nodes run into the situation as described before, the algorithm will inevitable erroneously decide that s and t are unconnected.

The chance that no node runs into this situation in the first round is $(3/4)^c$. The chance that only one node runs into it is $c.1/4\,(3/4)^{c-1}$. Every other scenario inevitably leads to an erroneous result. Therefore, the chance that the result is erroneous is at least as great as one minus the sum of the changes for the first two scenarios:

$$1 - \left(\frac{3}{4}\right)^c - c.\frac{1}{4}\left(\frac{3}{4}\right)^{c-1}. \qquad \square$$

In particular, given the formula in the proof of Proposition 7.3, when $c = 4$ we have a chance bounded below by 0.26171875; when $c = 7$, we already have 0.55505371; when c is, say 100, this chance approaches 1 very close.

7.4 Finding Connected Components in Logarithmic Rounds

For the previous algorithm we unfortunately have no bound on the number of rounds which is tighter than $\mathcal{O}(d)$, where d is the diameter of the input graph. In [54] several approaches to identify the connected components in a graph are discussed, including two approaches that require only a logarithmic number of rounds with high probability. In this section we give the *hash-greater-to-min algorithm*, which according to [54] has the best theoretical bounds.

Associate a set D_v to every node v. Initially $D_v = \{v\}$. Iteratively repeat the following three steps:

- For every node v, send the smallest node in D_v to every neighbour node and send D_v to v.
- Repeat the previous step. If a new minimum is known in this or the previous step, send it to the current minimum [2].
- For every node v compute the set $D_{\geq v} = \{u \mid u \in D_v \text{ and } u \geq v\}$. Send $D_{\geq v}$ to the smallest node, say $v_{\min}$, in D_v and send $v_{\min}$ to every $u \in D_{\geq v}$.

Conceptually, the hash-greater-to-min algorithm constructs a topology on top of G based on the sets D_v. Let G_{D_v} be a directed graph where V are the nodes, and the set of edges is defined by $\{(v, u) \mid u \in D_v\}$. Initially, every node in G_{D_v} is an unconnected component with a directed edge to itself. In the first and second step, however, nodes are spread among neighbours in G. In the third step, many nodes are aware of a node that is smaller than itself. We call the minimal nodes in these sets the local minima of G_{D_v}. In the third round, every node v sends $D_{\geq v}$ to its local minimum and sends the local minimum to itself. When the sets D_v converge, we have a situation where every connected component has a global minimum m that has an edge to every other node in the connected component, and every node has an edge to its minimum. The concept

[2] The communication of the minimum is not mentioned in [54], but seems necessary to be able to conclude the proof of Proposition 7.6

behind the hash-greater-to-min algorithm is that halfway through the algorithm there are several local minima and clusters around these minima. Assume a cluster with minima m. In the first step of the algorithm m is spread from the outer nodes of the cluster to neighbours in other clusters. Let v be such a neighbour. In the second step - if m is the smallest node that v received - m is communicated to the current local minimum m' of v. In the third step - m is the smallest received node by m' and $m < m'$ - the entire cluster of m' is communicated to m.

In MapReduce terms, as input we assume for each node $v \in V$ a pair $\langle k; (v, \Gamma(v), \{v\})\rangle$ where k is an arbitrary key.

First, map every pair $\langle v; (\Gamma(v), D_v)\rangle$ onto a pair $\langle v; (\Gamma(v), D_v)\rangle$ and a pair $\langle u; \{v_{min}\}\rangle$ for every $u \in D_v$, where v_{min} is the smallest node in D_v. Every reducer receives $D_v, \Gamma(v)$ and some minimal nodes form neighbours, and therefore is able to output the pair $\langle v; (\Gamma(v), D'_v)\rangle$ where $D'_v = D_v \cup \{v_{min}\}$ and v_{min} is the smallest node among the proposals that it received. The map and reduce function from the second step are the same as in the first step. Finally, in the third step, map every pair of the form $\langle v; (\Gamma(v), D_v)\rangle$ on a pair $\langle v_{min}; D_v\rangle$, where v_{min} is the smallest node in D_v, and on pairs $\langle v; (\Gamma(v), v_{min})\rangle$ and $\langle u; v_{min}\rangle$ for every $u \in D_v$. The reduce function computes as output a pair $\langle v; (\Gamma(v), D'_v)\rangle$ where $D'_v = \bigcup D^i_v$ for each D^i_v that it received.

In analogy to [54], we define the set $GT_k(m)$ as the set of all nodes v for which $m = \min D_v$ in round k and the set $M_k = \{m \in V \mid \exists v \in V, v \in GT_k(m)\}$. We write $GT(m)$ and M to denote the sets $GT_k(m)$ and M_k after convergence.

Proposition 7.4 (Correctness): *Let $G = (V, E)$ be an undirected graph, $\mathcal{D} \subseteq 2^V$ the set of connected components in G, d an upper bound on the diameter of each connected component, and π an ordering on the nodes in V. After at most $\lceil d/2 \rceil$ rounds, $\bigcup_{m \in M_k} GT(m) = \mathcal{D}$.*

Proof: Let $\Gamma : V \to 2^V$ be the neighbour relation, that is, $\Gamma(v) = \{u | \{v, u\} \in E\}$ for every $v \in V$. We define the reflexive neighbour relation Γ^* as the relation such that $\Gamma^*(v) = \Gamma(v) \cup \{v\}$ for every v. Extend both Γ and Γ^* to sets of nodes, that is $\Gamma(S) = \bigcup_{v \in S} \Gamma(v)$ and $\Gamma * (S) = \bigcup_{v \in S} \Gamma^*(v)$. As a shorthand let $\Gamma^*_i(v) = (\Gamma^* \circ \Gamma^*_{i-1})(v)$ where $\Gamma^*_0(v) = \Gamma^*(v)$, that is, the set of every node at a distance of at most i edges from v, including v itself.

Take an arbitrary, nonempty, connected component in $D \in \mathcal{D}$, and call m the smallest node in D according to π. Clearly, after the first step we have that $m \in D_v$ for every $v \in \Gamma^*(m)$. After the second step, we have that $m \in D_v$ for every $v \in \Gamma^*_2(m)$. Therefore, after the third step we have that $\Gamma^*_2(m) \subseteq D_m$ and $m \in D_v$ for every $v \in \Gamma^*_2(m)$. In other words, D_m contains every node v from which there exists a path from m to v that has a length of at most two edges. Moreover, $m \in D_v$ for every node v at a distance of at most two edges from m. Remark that since m is the smallest node in D, every node in D_m stays in D_m during the iteration.

Assume that $\Gamma^*_i \subseteq D_m$ and $m \in D_v$ for every $v \in \Gamma^*_i$. In the first step of this round, m is communicated to every node $u \in \Gamma^*_{i+1}$ and in the second step, m is further communicated to every node $u \in \Gamma^*_{i+2}$. Finally, in the third step, we have that $\Gamma^*_{i+2} \subseteq D_m$ and $m \in D_v$ for every $v \in \Gamma^*_{i+2}$.

Since d is the longest path between nodes that can occur, eventually, after $\lceil d/2 \rceil$ rounds, there is an $m \in V$ for every $D \in \mathcal{D}$ such that $D = D_m$ and $D_v = \{m\}$ for every other node $v \in D \setminus \{m\}$. □

Proposition 7.5 (Communication Cost): *For each step, the total communication cost is bounded by* $\mathcal{O}(|E| + |V|)$.

Proof: In the first step, we have as input for every node v a set $D_v = \{v\}$ and the neighbours of each v. Clearly, $\sum_v D_v = |V|$, the total sum of neighbours is bounded by $2|E|$. Therefore, since a local minimum is sent to each neighbour, the total sum of mapper outputs in the first step is bounded by $|V| + 4|E|$. Every node adds only the smallest received node to its set D_v, that is, $\sum_v D_v \leq 2|V|$. Therefore, the total communication cost in the second step is $2|V| + 4|E|$. Again, only the smallest node is added to D_v. Therefore, we have $3|V| + 2|E|$ as the total size of stored sets D_v and edges after the second step. Next, the set $D_{\geq v}$ is computed. Since both received nodes should be smaller than v (otherwise they are ignored), we have that $\sum_v D_{\geq v} = |V|$. Therefore, the total communication cost in the third step is $2|V| + 2|E|$. Remark that this result consists of $|V|$ nodes stored in local minima, and $|V|$ nodes that are smaller than the node on which they are stored.

In round i, we assume that the input has size $2|V| + 2|E|$, $|V|$ nodes that are greater than the node where they are stored, $|V|$ nodes that are smaller, i.e., the local minima, and $|E|$ edges. Analogously to the initial case, $2|V| + 4|E|$ is an upper bound on the communication in the first step, and $3|V| + 4|E|$ is an upper bound on the total communication in the second step. Thereafter we have that the total sum of D_v for every v is $4|V|$, but only $|V|$ of these nodes are lower than the node v on which they are stored. From the remark at the end of the initial round (and this round) we have that $D_{\geq v}$ contains at most $|V|$ nodes. Therefore, the total communication in this step, as well as $\sum_v |D_v|$ after this step, is bounded by $2|V| + 2|E|$, where $|V|$ nodes are smaller than the node on which they are stored. □

In Proposition 7.4 we already proved that the proposed algorithm is correct and finishes in at most d rounds, but in practice, as we prove below, the number of required rounds is only logarithmic.

Proposition 7.6 (Number of Rounds): *The algorithm runs in a logarithmic number of rounds with high probability.*

Proof: Let G_{M_k} be the graph defined by $(M_k, \{(m, m') \in M_k \times M_k \mid \exists v \in GT_k(m), v' \in GT_k(m'), (v, v') \in E\})$. In other words, we assume G_{M_k} to be the graph of local minima, where there exists a connection between two local minima m, m' if there exists an edge in the input graph such that one node has m as its minimum and the other has m' as its minimum. Clearly, if $m \in M_k$ has no neighbors in G_{M_k}, $GT_k(m)$ is a connected component in G disconnected from other components.

Assume that m has an edge to m' in G_{M_k} and $m > m'$. In the first step of hash-greater-to-min m' is communicated to a node that has m as its minimum (from v' to v). In the next step, m is communicated to m'. Finally, in the third step, m will send to m' the set $GT_k(m)$.

Call MC_k the set of nodes defined by $\{m \in G_{M_k} \mid \exists m', (m, m') \in G_{M_k}\}$, i.e., the nodes of G_{M_k} that have an outgoing edge in G_{M_k}. Without loss of generality assume that $|MC_k| = l$ and the nodes are labeled with $1, 2, \ldots, l$. The probability that a set $GT_k(m)$ will have as minimum a node $m' \in \{l/2, l\}$ after the next round is $1/4$, since, m itself has to be in $\{1/2, l\}$ and m' has to be in $\{1/2, l\}$. Therefore, the chance that the set $GT_k(m)$ has $m' \in \{1, l/2\}$ as its minimum in the next round is $3/4$. In general, the expected number of minima in the next round is $3l/4$.

Since, in every round the number of minima decreases with a constant factor, we can conclude that the total number of required rounds is expected to be $3 \log n$. $\square$

Note on Key Complexity

When the graph contains nodes that have many edges, when the graph has only a few connected components, or when the connected components are highly skewed in their size, some reducers may need a lot of memory to store the received data. Therefore, as is, the algoritme is not very scalable. This is also concluded in [54].

Instead of mapping every node on a particular reducer, we can spread nodes on a block of reducers uniformly at random. Assume that we have r' reducers for each node and that the content of D_v for a node v is spreaded among r' pairs. Every pair has a key (v, i), where v is a node, and i denotes a particular reducer for that node. More precise, we have pairs of the form $\langle (v, i); (D_v^i, N^i(v), v_{min})\rangle$ such that $\bigcup_{i \in [1, r']} D_v^i = D_v$ and $D_v^i \cap D_v^j = \emptyset$ for every $i \neq j$. The set $N^i(v)$ has an analogous meaning to D_v^i for the edges of v. Furthermore, we assume that v_{min} is the minimum of D_v. The first step is replaced by a mapper that maps these pairs on itself and a message containing v_{min} that is sent to a random reducer of every neighbour. The reduce function accepts only the received minimum, outputs the pair containing D_v^i itself, and sends its local minimum to reducer $(v, 1)$. In the next step, the map function is the identity and the reduce function maps the received local minimum pairs onto a pair containing the global minimum for v, addressed to every reducer of v. Finally, we need a map and reduce function that compute the new D_v^i for each v and i, containing the global minimum for v. The second step of the original is replaced the same way.

The third step is replaced by a map function that computes $D_{\geq v}^i$ based on D_v^i and output a tuple containing its neighbours and minimum addressed to (v, i) itself, $D_{\geq v}^i$ addressed to (v_{min}, i), where i is randomly chosen, and v_{min} is sent to a single reducer for every node in $D_{\geq v}^i$. Then, two additional steps are required to compute the minimal nodes for each D_v and distribute them among the reducers for node v. These are analogous to the described changes for step one and two.

8 MapReduce Extensions and Alternatives

Despite the popularity of MapReduce, several problems exist that are difficult to compute in the default MapReduce environment. Examples are: joins, recursion, and graph algorithms, as we have seen in Chapters 6 and 7. Several extensions and generalizations are developed with the aim to overcome one or more of these issues. In this chapter, we give a description of some MapReduce variations that seem important to mention in our setting.

8.1 Pregel

The main issue when computing problems related to large graphs in a sequence of MapReduce steps is that, in each step, the entire graph has to be communicated over the network; meaning a lot of unnecessary communication of static data. Therefore, MapReduce is not a very satisfying model in the context of graph algorithms. Pregel is Google's own alternative to MapReduce for computing with large graphs [47]. It is a programming paradigm for Bulk Synchronous Parallel (BSP) processing [58], that consists of several components each assigned with a particular part of the graph whereupon tasks are executed in parallel within super steps. Each computation is performed on a particular node and its direct environment; i.e., the outgoing edges and target-node ids. Every node has a value that can be modified during a computation. Furthermore, messages can be sent from one node to the other (based on node ids) and are received at the target node at the beginning of the next step. Every iterative step consists of the parallel execution of a user defined computation on every node from the graph.

The description of Pregel in [47] is rather informal. Therefore, we provide a more formal description below. As input, assume a directed graph $G = (V, E)$ and a set $\mathcal{D}$ that contains for each node v a value D_v. We define a communication message as a pair of the form $\langle k; m \rangle$, where k is a node identifier and m is the content of the message, which can be anything. Let M_v be the set of messages to node v. Initially, $M_v = \{\}$ for every node v. Let f be a user defined computation function that maps a set of messages and the data associated to a node onto another set of messages and modified data:

$$f : (v, S_v, D_v, M_v) \to (D'_v, M'_v).$$

Here, S_v represents the scope of the computation and contains the ids of the outgoing nodes for v.

Each step in Pregel consists of the following two phases:

- COMPUTATION PHASE: $f(v, S_v, D_v, M_v) \to (D'_v, M'_v)$ is computed for every node $v \in V$ that is marked as active. Several instances of f are executed in parallel. After this step we have that $D_v \leftarrow D'_v$ for each node v.
- COMMUNICATION PHASE: for each $v \in V : M_v \leftarrow \{\langle v; m\rangle \in M'_u \mid u \in V\}$, i.e., every message in M'_v is sent to the appropriate destination.

Aside to communication from one node to the other, computations can request operations on the topology of the graph (i.e., removing and adding nodes and edges). Furthermore, f can utilize an aggregation mechanism. Every node can send a value m to an aggregate function, say *agg*, which combines the received values based on an associative, commutative summation:

$$z = \bigoplus(agg(m)).$$

The result of the aggregator is sent to every node in the next step. This mechanism can be used to keep every node informed about a property of the graph, e.g., its total number of nodes. Pregel also has a combiner function that can be used to group or merge messages send from within different scopes on a particular machine, reducing the real communication cost between machines.

From a logical point of view Pregel has a lot of similarities with MapReduce. In fact, the core of Pregel can be seen as a MapReduce program where the map function is fixed and bins every node, its state, and its outgoing edges. The computation is similar to the reduce phase and is performed on each bin. Communication between nodes are additional pairs outputed by the reducer and mapped into the destination bin.

Example: As an example of a computation in Pregel, consider the *PageRank problem* [14]. Let G be a graph representing pages and directed links between these pages. PageRank is the problem that asks to assign a score of importance to every page in G based on the scores of pages that are linked to it. Initially, every node has the same score. We denote the score of v with $D_v.score$. The pagerank is calculated by iteratively repeating the following equation:

$$D_v.score = \frac{(1 - \text{damp})}{|V|} + \text{damp} * \sum_{(u,v)\in E} \frac{D_u.score}{deg_o(u)}.$$

The constant *damp* represents a damping score and is usually set to 0.85.

A Pregel algorithm for computing PageRank, based on [47], is presented in Algorithm 8.1.

Algorithm 8.1: PageRank in Pregel.

```
Compute ($v, S_v, D_v, M_v$):
    if current superstep $\geq 1$ then
        sum = 0;
        foreach message $m$ in $M_v$ do
            sum += m.value;
        $D_v$.score = $\frac{1-damp}{|V|}$ + $damp$*sum;
    end
    if current superstep $< 30$ then
        foreach neighbour $u$ do
            emit $\langle u, \frac{D_v.score}{deg_o(v)} \rangle$;
    end
    else
        vote to halt;
    end
```

8.2 Distributed GraphLab

GraphLab is another variation of MapReduce that, similar to Pregel, focuses on graph computations. GraphLab is especially designed for running machine learning and data mining algorithms, initially in a multicore shared memory setting [45], but also for distributed environments [46]. The main difference with Pregel is that it is not a synchronous (in terms of super steps) message-passing system, but an asynchronous distributed shared memory abstraction [30]. It is not possible to send messages directly to other nodes. Scopes contain both the 'core' node, its incoming and outgoing edges, and its neighbour nodes and their respective states. Hence, overlap exists between the scopes of neighbour nodes.

Let $G = (V, E)$ be a directed graph, and $\mathcal{D}$ a set of data values associated to nodes and edges. We write D_v and $D_{v \to u}$ to denote the set of data associated to node v and edge (u, v) respectively. A GraphLab algorithm is defined by a user defined update function f that, given a node v and its scope S_v, is able to modify the data related to S_v and output a set of nodes. The scope of a node v, write S_v, consists of v itself, its neighbour nodes and adjacent edges, and the associated values. Formally, the update function has the following signature:

$$f : (v, S_v) \to (S_v, T).$$

Let $G = (V, E)$ be the input graph, $\mathcal{D}$ the set of data values, and $T \subseteq V$ a set of nodes. While T is not empty, for each $v \in T$:

- COMPUTATION PHASE: $(S'_v, T_v) = f(v, S_v)$. Let $T = (T \setminus \{v\}) \cup T_v$ and $S_v = S'_v$. Several instances of f are computed in parallel, asynchronously. There are no guarantees about the ordering of nodes in T other than that every node will eventually be computed.

GrapLab relies on a customizable scheduler for task assignments and consistency guarantees. Since scopes contain overlap, there are three relevant consistency models:

- *full consistency*: the computation function is executed in parallel on nodes without common neighbours. Hence, there is no scope overlap between parallel computations.
- *edge consistency*: a common neighbour is allowed, but common neighbours are not computed in parallel.
- *node consistency*: the only guarantee is that the computation function is executed only once on a node at the same time.

In general, the more strict the consistency level, the less parallelization can be utilized.

To keep track of global information, it is possible to run a synchronization process in the background similar to the aggregation feature in Pregel. In GraphLab the synchronization process is defined by a map and finalize function, which are formalized as follows:

$$\text{finalize}(\bigoplus_{v \in V} \text{map}(S_v)).$$

The map function is associative and commutative, the finalize function can be used to, for example, normalize the result. The result of the synchronization process is a global value that is accessible by the update function. Again, it is up to the programmer to determine the consistency level for this value, i.e., guaranteed consistency or tolerated inconsistency. Similar to the scheduling consistency models, consistency may cause additional delay.

An important difference between Pregel and GrabLab is that nodes in Pregel, are able to communicate with each other independent of the topology of the graph, while in GrabLab only communication between adjacent nodes is possible by utilizing scope overlap; Pregel uses a message passing system, while GraphLab uses a shared state strategy.

Example: Take the PageRank problem from the previous section. We associate a state $D_{(v,u)}$ with every edge $(v, u) \in E$. $D_{(v,u)}$ represents the weight of the edge according to the score of v. In other words, $D_{(v,u)}.weight = 1/deg_o(v)$. We also assume a state $D_v(v, u).old$ that represents the latest score of v that has been read by u. A GraphLab implementation [1] is represented in Algorithm 8.2.

8.3 Datalog and Recursion

MapReduce is an acyclic dataflow system that consists of two kinds of tasks which are executed in parallel in separated rounds. Therefore, a natural generalization of the model

[1]The implementation of PageRank in GraphLab is based on `http://select.cs.cmu.edu/code/graphlab/doxygen/html/pagerank_example.html`

Algorithm 8.2: PageRank in GraphLab.

```
Update (v, S_v):
    sum = 0;
    foreach incoming edge $e = (u, v) \in E$ do
        sum += $D_e$.weight$*u$.score;
        $e.old = u.score$;
    $v$.score = $\frac{(1-damp)}{|V|} + damp*$sum;
    foreach outgoing edge $e = (v, u) \in E$ do
        if too much difference between $e.old$ and $v.score$ then
            schedule $u$;
        end
```

from this point of view is a dataflow system where tasks have a generic, user defined form [5].

The output of a MapReduce task is delivered to other tasks only after its completion. Therefore, MapReduce does not support recursion explicitly. In this section we explorer the implementation of recursive Datalog in a distributed and parallelized environment similar to MapReduce based on [5]. In particular, we start with a description of the evaluation of Transitive closure.

8.3.1 Seminaive Evaluation and Derivations

Consider the following Datalog program:

```
T(x,y) :- E(x,y).
T(x,y) :- E(x,z), T(z,y).
```

A derivation of a tuple t is a substitution of values for variables in the body of a rule that makes the head be t [5]. There are several ways to evaluate a sequence of Datalog rules. Clearly, in the above example every tuple of the EDB (extensional database) predicate E is conveyed to T. The evaluation of the second rule, however, requires an iterative strategy. The naive approach would be to join every tuple in E with every known tuple of T, resulting in a huge number of redundant derivations. The seminaive approach [2, 5] is based on the naive approach, but only considers substitutions of the body predicates where at least one of the EDB tuples is derived in the previous round. Hence, avoiding redundant derivations.

In this chapter the seminaive approach takes an important role in that the complexity of parallel Datalog algorithms will be compared to it. The evaluation of our example program using seminaive evaluation is formalized in Algorithm 8.3. In our context Theorem 8.1 is of particular importance.

Theorem 8.1 ([5]): The number of times a tuple t is produced during seminaive evaluation is equal to the number of derivations of t.

Algorithm 8.3: Example of seminaive evaluation.

```
$T_0 := \Delta_0 := E$;
$i := 0$;
**repeat**
    $i := i + 1$;
    $T_i := T_{i-1} \cup \Delta_{i-1}$;
    $\Delta'_i := E(x, z) \bowtie \Delta_{i-1}(z, y)$;
    $\Delta_i := \Delta'_i \setminus T_i$;
**until** $\Delta_i = \emptyset$;
```

8.3.2 Nonlineair TC

A fundamental problem in the context of Datalog is the computation of the transitive closure of a directed graph [8]. In this section we discuss several approaches to perform transitive closure in the generalized MapReduce environment as described above. First note that there are several ways to formulate the transitive closure algorithm. The most common approaches are the nonlinear, left- and right-linear programs as described below:

- nonlinear TC:

  ```
  T(x,y) :- E(x,y).
  T(x,y) :- T(x,z), T(z,y).
  ```

- left-linear TC:

  ```
  T(x,y) :- E(x,y).
  T(x,y) :- E(x,z), T(z,y).
  ```

- right-linear TC:

  ```
  T(x,y) :- E(x,y).
  T(x,y) :- T(x,z), E(z,y).
  ```

Since it is a property of the linear approach to detect every path only once, in a sequential environment it is common to prefer a linear approach. In a parallel environment, however, it is common to tolerate an overhead in communication cost in return for a decrease in the number of rounds. Therefore, in a MapReduce-like environment we are tempted to prefer a nonlinear approach.

Implementation

In [5, 4] two new tasks are introduced: a join task and a duplicate-elimination task, which are executed in parallel on a set of distributed machines. Furthermore, they assume the existence of a hash function g from tuples in T on a duplicate-elimination task and a hash function h from tuples in N on join tasks. Initially, every tuple $(a, b) \in E$ is sent to a particular duplicate-elimination task $g(a, b)$. The duplicate-elimination task first checks if the received tuple (a, b) is not stored already. If not then it stores (a, b) and sends a copy to the join tasks associated to $h(a)$ and $h(b)$ respectively. Clearly, each join task receives a tuple (a, b) if the task is associated to a or b. It stores (a, b) and checks if there exists a tuple (b, c) or (c, a) and sends the newly constructed tuples (a, c) and (c, b) to the particular duplicate elimination tasks based on g.

Proposition 8.1 ([5]): *The total cost of nonlinear TC is linear in the number of derivations.*

Proof: Assume that we have nodes a, b, c such that there exists a path from a to b and from b to c. Consequently, there exists a derivation such that a, b and c substitute for x, y and z respectively. The join task $h(b)$ will receive (a, b) as well as (b, c) and therefore eventually construct the tuple (a, c) and communicate it to the duplicate elimination task $g(a, c)$. No other task receives both (a, b) and (b, c), therefore this derivation can only be made by the join task $h(b)$. The duplicate-elimination task prevents sending the same tuples to a join task again and, consequently, prevents that a same derivation is done multiple times. Clearly, the maximum number of tuples that is communicated for a particular derivation is three and. Hence, the number of tuples that are communicated is bounded by $3n$, where n is the total number of derivations. $\square$

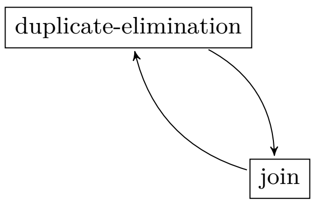


Figure 8.1: Implementation of nonlinear TC.

Figure 8.1 represents a schema of the implementation of nonlinear TC as described above. The boxes represent a collection of processes that are responsible for a certain task. Note that although the schema is cyclic the data flow is not.

8.3.3 Recursive Doubling

Linear algorithms have the unique-decomposition property, that is, every path is detected only once. The traditional nonlinear TC, however, may also derive a same path multiple times. Nevertheless, there exist more complex nonlinear TC algorithms that combine the best of both worlds, i.e., non-linear algorithms that detect every path only once. One such algorithm is called recursive doubling [5] and is described in Algorithm 8.4. Basically, P is an intensional predicate that keeps track of the pairs of nodes where there exists a path of length between 0 and $2^i - 1$, Q keeps track of the pairs of nodes where the smallest path

Algorithm 8.4: Recursive Doubling.

```
Input: $V, E$
Output: transitive closure of $E$
$Q_0 := E$;
$P_0 := \{(x,x) | x \in V\}$;
$i := 0$;
repeat
    $i := i + 1$;
    $P_i(x,y) := \pi_{x,y}(Q_{i-1}(x,z) \bowtie P_{i-1}(z,y))$;
    $P_i := P_i \cup P_{i-1}$ ;
    $Q_i(x,y) := \pi_{x,y}(Q_{i-1}(x,z) \bowtie Q_{i-1}(z,y))$;
    $Q_i := Q_i \setminus P_i$ ;
until $Q_i = \emptyset$;
```

has length 2^i. Other nonlinear transitive closure algorithms exist that have the unique-decomposition property. In [8] a study is performed on transitive closure algorithms based on length partitioning. It concludes that these algorithms are incomparable to each other, i.e., it depends on the particular situation which of the approaches gives the best result.

Implementation

The implementation of recursive doubling is slightly more difficult than the implementation of the traditional nonlinear approach. In particular, it requires two collections of join and duplicate elimination tasks; one that is responsible for P and one that is responsible for Q. Moreover, the join tasks of P are responsible for the line $\pi_{x,y}(Q_{i-1}(x,z) \bowtie P_{i-1}(z,y))$, whereas the duplicate elimination tasks are responsible for $P_i \cup P_{i-1}$. The join tasks associated to Q are responsible for $\pi_{x,y}(Q_{i-1}(x,z) \bowtie Q_{i-1}(z,y))$ and the duplicate elimination tasks for $Q_i \setminus P_i$.

A schema of the cluster implementation is given in Figure 8.2. Every box represents a cluster of processes that is responsible for the specific task.

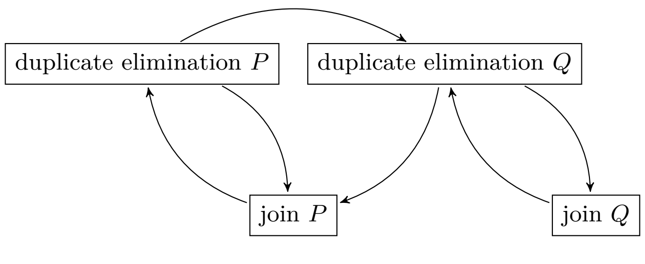


Figure 8.2: Implementation of Recursive Doubling.

8.3.4 Generalization to Datalog

Informally, this approach can be generalized to arbitrary Datalog programs by considering a collection of join and duplicate elimination tasks for every rule body. Subsequently, the derivations made by these tasks are send to the task collections associated to rule bodies that contain the derived predicate. In the case of multi-way joins this task becomes considerably harder than in the two-way join case [7].

8.4 Real-World Graph Structures

A lot of research has been conducted on the structure of real-world graphs. Some properties that occur regularly are [28]:

- small-world property: the average distance between nodes is short;
- network transitivity: two nodes that have a common neighbour are more likely to are neighbours themselves;
- community structures: the occurrence of groups of densely connected nodes; and
- power-law degree distributions: a graph typically consists of many low-degree nodes and a small number of high-degree nodes.

Pregel and GraphLab perform best when the input graph is sparse and communication occurs mainly over edges [47]. Unfortunately, many natural graphs are not sparse, or, to be precise, there always are some nodes that are connected to almost every other node (e.g., popular persons in social media, or search engines on the web). The internet graph, for example, agrees to a power-law distribution and consists of star patterns [30]: although most nodes have small out degrees, some nodes are connected to almost any other node.

Formally, a power-law degree distribution [22] is a distribution where the chance of a node to have a degree of d is $d^{-\alpha}$ up to a constant factor, where α is a positive constant. In symbols, we write:

$$\Pr[d] \propto d^{-\alpha}.$$

A high α indicates a low density graph and a large number of nodes with a low degree. A low α on its turn indicates a dense graph and many nodes with a high degree.

Partitioning such a graph is not an easy task, and since the running time of a vertex program is usually directly correlated to the number of edges, Pregel and GraphLab have highly-skewed running times and space requirements for this type of graphs.

8.5 PowerGraph

Pregel and GraphLab are vertex programs, which means that computations are performed on nodes and corresponding scopes. The main difference is that a scope in Pregel only contains the outgoing edges and neighbour-node ids, while scopes in GraphLab contain incoming and outgoing edges. Furthermore, interaction between computations is performed through message passing in Pregel, and through shared states in GraphLab.

However, scopes based on adjacent edges are highly sensitive to data skew. Huge star-like motives, as described in Section 8.4 may not even fit in the memory of a single machine. Therefore, both abstractions are infeasible for many natural graphs. Hence, Pregel and GraphLab perform best when the input graph is sparse and communication occurs mainly over edges [47].

PowerGraph [30] is very similar to GraphLab, but addresses these challenges by pulling apart the computation phase in three separated phases: gather, apply, and scatter; called the GAS model. Instead of executing the computation phase on the adjacent edges as a whole, the scatter function is executed on every adjacent edge separately. The results of these functions are merged based on an associative, commutative summation:

$$\Sigma \leftarrow \bigoplus_{v \in \Gamma(u)} gather(D_u, D_{(u,v)}, D_v).$$

The apply function takes the core functionality of the computation function and is able to update the state of the central node based on the result of the gather phase:

$$D_u^{new} \leftarrow apply(D_u, \Sigma).$$

Finally, the scatter function is similar to the gather function, but enables the user to update edge states after the apply phase:

$$D_{(u,v)} \leftarrow scatter(D_u^{new}, D_{(u,v)}, D_v).$$

PowerGraph provides two executions of the GAS model: a bulk synchronous execution and an asynchronous execution. In the bulk synchronous execution, the gather, apply, and scatter function are executed synchronously in minor steps. The computations as a whole are executed in supersteps. In the asynchronous execution, resources are utilized as soon as they become available.

Example: Again, we implement the PageRank problem. A PowerGraph implementation of PageRank, based on [30], is represented in Algorithm 8.5.

8.6 Other Extensions

Aside to GraphLab and Pregel there exist several other extensions and generalizations of MapReduce. In this section we give a brief overview of alternative programming paradigms, but there exist many more.

Dryad [36] is a programming model by Microsoft that shares with MapReduce that computations have the form of an acyclic dataflow system and that the mapping on physical machines is done automatically. Nevertheless, computation tasks can take any form and are not restricted to map or reduce tasks. An important issue in both MapReduce and

Algorithm 8.5: PageRank in PowerGraph.

```
Gather ($D_u, D_{(u,v)}, D_v$):
    return $\frac{D_v}{deg_o(v)}$;
sum (a,b):
    return $a + b$;
Apply ($D_u, acc$):
    score = $\frac{1-damp}{|V|} + damp * acc$;
    $D_u$.delta = $\frac{score-D_u.score}{deg_o(u)}$;
    $D_u$.score = score;
Scatter ($D_u, D_{(u,v)}, D_v$):
    if ($|D_u.delta|$ too big) then
        schedule $v$;
    end
    return $D_u$.delta;
```

Dryad is that many applications require the iterative execution of one or more MapReduce steps. This causes a lot of overhead communication, especially when it comes to static data. Furthermore, reaching a fixpoint state is a common termination condition that may require an additional MapReduce step, implying additional overhead. Haloop [15] is a MapReduce extension that introduces caching of invariant data and reducer outputs to improve the execution of certain iterative algorithms.

Pig [51] is a system that is build on top of Hadoop and offers the ability to programmers to write MapReduce algorithms via Pig Latin. Pig Latin is a high-level language, somewhere between SQL and low-level MapReduce, that provides the ability to write code that is much more maintainable. Pig itself is the system that translates Pig Latin algorithms into physical MapReduce plans.

8.7 Note on Complexity Measures

The complexity of algorithms in Pregel can be expressed similarly to the complexity in MapReduce. The main difference is in the fact that in Pregel the graph itself persists in the distributed file system. Therefore it does not contribute to the communication cost. Furthermore, Pregel has no shuffle phase and messages are send directly to their destination based on node ids.

GraphLab has similarities with Pregel, but scopes contain overlap. Therefore, the communication cost is less clear. For example, although a computation may be performed on the same node as the main content of its scope, some edges or nodes may need to be transfered or synchronized between machines.

9 Regular Path Queries in MapReduce

We introduced the regular path queries in Chapter 4 as a very simple navigational language for graphs. Although RPQs have several interesting properties and are certainly simple compared to the full graph query languages that are typically used today, the evaluation problem for RPQs on large graphs is still an intensive task.

The aim of this chapter is to study the evaluation of regular path queries in MapReduce and similar environments. First, we give some algorithms for the evaluation of RPQs that are already studied in the context of distributed computing. Next, we present algorithms and complexity bounds for the evaluation of RPQs in MapReduce that, to the best of our knowledge, are not yet studied in literature.

9.1 Distributed Regular Expressions

The evaluation of regular expressions in MapReduce and similar environments is rather new, but the evaluation of regular expression in a distributed shared nothing environment is not [56]. In a general distributed setting, where the graph is partitioned in large components on the available machines, it is possible to partially evaluate path queries on individual components locally. Hence, a partial result can be obtained without any communication between components.

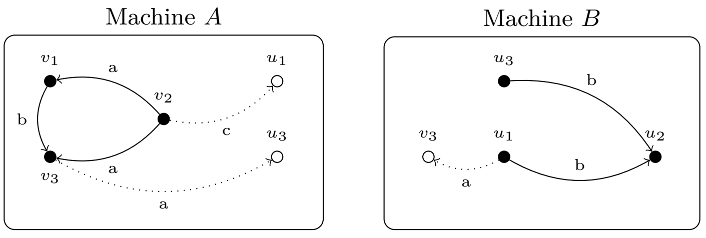


Figure 9.1: Example of a distributed graph.

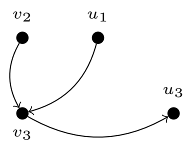


Figure 9.2: Example of an accessibility graph.

Example: Given the distributed graph from Figure 9.1, where filled nodes represent local nodes, open nodes represent virtual nodes, and dotted edges are inter-machine edges. Now, consider the query $q : (x, y) \leftarrow (x, a^+, y)$. We are able to compute parts of $q(G)$ locally. In particular we are able to derive $(v_2, v_1), (v_2, v_3)$, and (v_2, u_3) on machine A, and (u_1, v_3) on machine B.

The only paths that remain hidden within this approach are paths covering edges in multiple components. Therefore, after the partial evaluation of the individual components, the missing paths have to be computed via a coordinating machine [56, 23]. To limit the communication cost to find these missing paths, techniques such as the use of an accessibility graph, containing border nodes and inter-partition paths, can be utilized [56].

Example: Recall the previous example. An accessibility graph could take the form of the graph in Figure 9.2. Every machine sends its relevant part of the accessibility graph to the coordinating machine, where these parts are assembled. Afterwards, the relevant parts are sent back to the machines.

Clearly, there is no need to send information about the edges (v_2, v_3) or (v_3, u_3) to machine B, while the information about the edge (u_1, v_3) is required to make machine A able to derive the path from u_1 to u_3. This kind of insights are reflected by the accessibility graph. Note that edge (v_2, v_3) of the accessibility graph can be eliminated by the coordinating machine, since it is not involved in any inter-machine path.

In [24] a similar technique is utilized for the evaluation of regular reachability queries on node-labeled graphs. However, the definition of RPQs in [24] differs to ours. Graphs are node-labeled graphs instead of edge-labeled graphs. Here, a path label is defined as the concatenation of labels from the sequence of nodes in the path (excluding the start and end node). Therefore, they also use a variant of NFAs for the evaluation of regular expressions, called query automata, which are in fact node-labeled directed graphs with two special nodes: a start node and an end node. The nodes are called *states*, similar as in NFAs. The conversion from a regular expression to a query automaton can be performed in a similar way as the conversion from regular expression to NFA.

Without going into much detail, [24] uses partial evaluation by computing, in parallel on the local fragments, which nodes can reach t satisfying the regular expression r by associating a vector to each node, where every cell of the vector represents a state. Clearly, for t itself, a cell has value *true* for every state that has the same label as t; *false* otherwise. Every node that can reach t via a local path has a value *true* for its reachable states. If an arbitrary node v has an incoming edge from a node located on another site, and does not know if that other node can reach t, a variable is used to denote that a state is reachable iff the value for that state associated to v is true. When these computations are finished, every state sends its meaningful vectors (i.e., vectors with none-false values that have outgoing edges on external sites, and the vector associated to s) to a coordinator site. This coordinator site assembles the dependency relations and evaluates the resulting conjunction to construct the query answer. This strategy leads to the following bounds:

Theorem 9.1 ([24]): Regular reachability queries can be evaluated in $\mathcal{O}(|F_m||Q|^2 + |Q|^2|V_f|^2)$ time, by visiting each site once, and with the total network traffic in $\mathcal{O}(|Q|^2|V_f|^2)$, where F_m is the largest fragment of the input graph and $|V_f|$ is the total number of nodes (including virtual nodes) in the graph fragment.

A translation from this algorithm to MapReduce is given in [24] as well. They assume MR-pairs that contain entire fragments of the input graph and the query automaton. Hence, the map function handles the partial evaluation and maps its result on a single reducer, that takes the roll of the coordinating site. This reducer finishes the computation by combining the partial results into the query answer.

Edge Labeled Graphs

Based on the reachability algorithm in [24], we describe a similar approach for edge labeled graphs. Basically, a precomputation is performed to partition the input graph G into fragments and combine every fragment with the regular expression in a MapReduce pair. The first map step computes locally on each fragment the reachable table for every node, as was done in Algorithm 9.2. In a sequential environment, this algorithm is performed in time polynomial in the size of the fragment. Afterwards, it is easy to create vectors, similar to what is done in [24].

The vector created for node s and the vectors associated with nodes having ingoing edges from nodes on external sites are sent to a particular reducer. Every vector has a size of at most $\mathcal{O}(|O_f||Q|)$, where O_f is an upper bound on the number of virtual nodes in a fragment.

The total number of nodes with incoming edges from external nodes is no more than the total number of virtual nodes. Hence, this number is bounded by $\mathcal{O}(m|O_f|)$, where m is the number of fragments. For every of these nodes, including s, a vector is sent to the coordinating reducer. The total communication cost is thus bounded by $\mathcal{O}(m|O_f|^2|Q|)$. Assuming that $\mathcal{O}(m|O_f|^2|Q|)$ fits in a single machine's memory, this algorithm can be performed in a constant number of steps.

Issues

Although this approach seems interesting at first, some issues arise. For example, the input pairs are very large and the efficiency of this approach strongly depends on the ability to partition the graph. In particular, densely connected components should be grouped together on a single machine and the number of inter-machine edges should be low. Graph partitioning, however, is an NP-hard problem. Further, this approach seems to undo much of the spirit of MapReduce programming and does not fit properly into the abstraction. Therefore, we do not consider this method any further.

9.2 RPQs in Datalog

In Section 8.3, the evaluation of Datalog rules in a distributed environment was discussed, which yields another strategy to evaluating regular path queries. It is easy to translate a regular expression into a Datalog program by following the subsequent rules below. Let E be the extensional database (EDB) predicate associated to the edge relation and $a \in \Sigma$.

ε	:	`O(x,x) :- .`
a	:	`O(x,z) :- E(x,a,z).`

Let R and S be regular expressions and `R` and `S` be the intensional database (IDB) predicates associated to R and S respectively.

R^*	:	`O(x,x) :- .`
		`O(x,z) :- R(x,z).`
		`O(x,z) :- O(x,y),O(y,z).`
$R\|S$	:	`O(x,z) :- R(x,z).`
		`O(x,z) :- S(x,z).`
RS	:	`O(x,z) :- R(x,y), S(y,z).`

Example: An illustration of this construction is provided based on the example expression $(a|b)^*c^*$. First, the expression is translated into a Datalog program by following the previously given ruleset:

```
D(x,z) :- A(x,z).
D(x,z) :- B(x,z).
E(x,x) :- .
E(x,z) :- D(x,z).
E(x,z) :- E(x,y), E(y,z).
F(x,x) :- .
F(x,z) :- C(x,z).
F(x,z) :- F(x,y), F(y,z).
G(x,z) :- E(x,y), F(y,z).
```

Note that A, B, and C are the EDB predicates associated to a, b, and c respectively, and G is the output predicate. The implementation of the R^* construction is similar to the implementation of transitive closure, which is discussed in Section 8.3. Therefore,

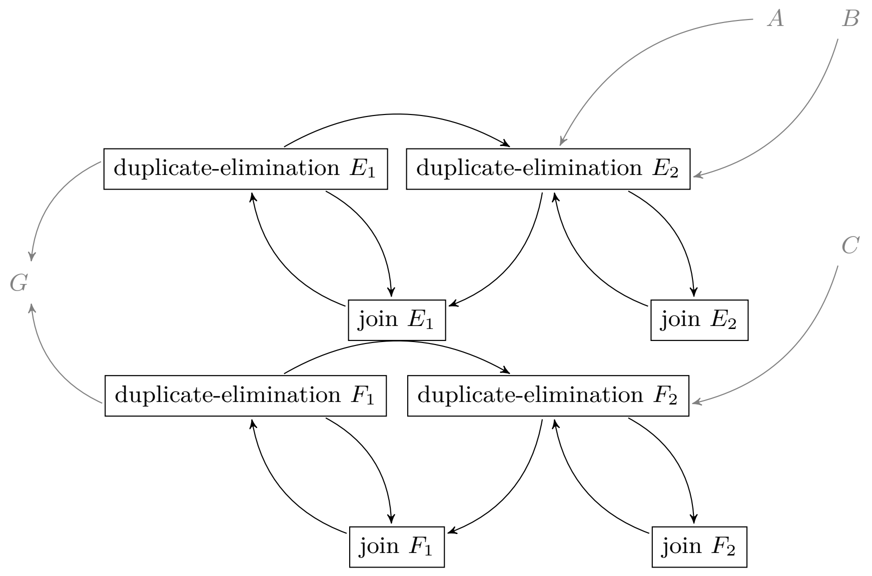

A
B
C
G
duplicate-elimination E_1
duplicate-elimination E_2
join E_1
join E_2
duplicate-elimination F_1
duplicate-elimination F_2
join F_1
join F_2

Figure 9.3: Implementation of $(a|b)^*c^*$. Boxes represent collections of tasks and arrows represent the dataflow.

the recursive doubling approach can be utilized. The Datalog construction of the union operator $R|S$ is nothing but an overall predicate that contains tuples of both the predicates related to R and S. Therefore, it suffices to view both tuples of R and S as part of the output, and if necessary, to provide these as input to a parent operation.

The direct translation of this Datalog program into a distributed algorithm leads to a collection of tasks connected as in Figure 9.3. Note that boxes represent sets of machines, and that the assignment of inputs to particular machines is based on hash functions.

We mention this method for a sake of completeness, as CRPQs are contained in the expressiveness of Datalog. However, this technique is not related to MapReduce and we have no convenient complexity measures for it.

9.3 Regular Reachability in MR

Let Σ be a finite alphabet, $G = (V, E)$ a Σ labeled directed graph, and q a regular path query over Σ of the form $ans(x, y) \leftarrow (x, r, y)$, where x and y are node variables and r is a regular expression over Σ. Let $A_r = (Q, \Sigma, \delta, s_0, F)$ be the NFA associated to r. For notational simplicity, we extend δ to a transition relation on strings $\delta : Q \times \Sigma^* \to \mathcal{P}(Q)$ such that

$$\delta(s_i, a_1 a_2 \dots a_m) = \bigcup_{s_k \in \delta(s_i, a_1)} \delta(s_k, a_2 a_3 \dots a_m).$$

Intuitively, $\delta(s_1, a_1 a_2 \dots, a_n)$ contains every state that is reachable from state s, by consuming the entire string $a_1 a_2 \dots a_n$.

Similar to the s-t-connectivity problem, we focus on regular reachability (or RPQ recognition), i.e., the decision problem that asks, given G, q, and $s, t \in V$, if there exists a path η from s to t such that $\lambda(\eta) \in L(r)$, where r is the regular expression associated to q.

The RPQ recognition problem is closely related to the s-t-reachability problem, but only paths that have an accepting run in A_r are considered. To detect if there exists a valid path from s to t, we conduct a breadth-first search from s to t, meanwhile keeping track of the states in A_r. Assume that every node v has a set of states D_v associated to it, initially the empty set. Let $D_s = \{(s_0)\}$. Repeat the following steps:

- If $D_t \cap F \neq \emptyset$, then conclude that $(s, t) \in q(G)$ and stop the iterations.
- Otherwise, combine the knowledge in D_v with the knowledge about the outgoing edges and the transition relation of A_r to communicate to neighbours of v in which states it is reachable from node s.

The algorithm is presented in Algorithm 9.1. Recall that we used $E_{v\to *}$ to denote the set of outgoing edges of v.

Algorithm 9.1: RPQ recognition.

```
Input: $\langle k; (v, D_v, E_{v\to *})\rangle$ for every $v \in V$, where $D_v := \emptyset$
Output: $\langle k; (v, D_v, E_{v\to *})\rangle$ for every $v \in V$, where $D_v$ is converged
repeat
    Map:
        foreach active $s_i \in D_v$ do
            foreach $(v, a, w) \in E_{v\to *}$ do
                foreach $s_j \in \delta(s_i, a)$ do
                    emit $\langle w; (s_j)\rangle$;
        emit $\langle v; (v, D_v, E_{v\to *})\rangle$;
    Reduce:
        Let $M$ be the set of received states;
        $D_v \leftarrow D_v \cup M$;
        mark every new state active;
until nothing changes;
```

Since the size of D_v is bounded by $\mathcal{O}(Q)$ and it can only grow throughout the iterative steps, it is easy to see that D_v, for every v will eventually converge to a fixed state. An upper bound on the complexity of this approach is given by Proposition 9.1.

Proposition 9.1: *Let G be a graph and Δ an upper bound on the degree for each node in G. The RPQ recognition problem can be solved in $\mathcal{O}(|E||Q|)$ steps.*

- ***Key complexity:*** *the amount of space per node is bounded by $\mathcal{O}(|Q| + \Delta)$; the input and output size are bounded by $\mathcal{O}(\Delta|Q|)$; and the running time is bounded by $\mathcal{O}(\Delta|Q|^2)$.*

- ***Sequential complexity:*** *the total input and output size in a single step is bounded by $\mathcal{O}(\max\{|E|, |V|\}|Q|)$; and the total running time in a single step is bounded by $\mathcal{O}(\Delta|Q|^2|V|)$.*

Proof: If $(s, t) \in q(G)$ there is a path η in G from s to t such that $\lambda(\eta) \in L(A_r)$. Therefore if $(s, t) \in q(G)$ we are guaranteed to find a path when our algorithm runs long enough. Moreover, since the maximum length of the minimal path for every node and state combination is $|Q||E|$ we are able to decide that $(s, t) \notin q(G)$ after $|Q||E|$ steps.

Let v be an arbitrary node in G. There are at most Q states from which s is able to reach v. Therefore, the size of D_v is bounded by $\mathcal{O}(|Q|)$. Since there is only communication between neighbour nodes, in a worst-case scenario every node sends every state to every outgoing neighbour. Hence, the input size of a node is at most $|Q|$ times the number of the incoming neighbours, and the cost for storing the adjacent edges.

A reducer adds every node node that it receives in D_v. This is done in time $\mathcal{O}(|Q|^2)$. The mapper is executed in time $\mathcal{O}(|Q|^2|E_{v\to *}|)$. Hence, the latter dominates the cost for a single mapper or reducer.

In a worst-case scenario every node receives every state from every neighbour. The total sum of state sizes is at most $\mathcal{O}(|V||Q|)$. Therefore, the worst-case input and output size is $\mathcal{O}(|Q|\max\{|E|, |V|\})$. The total running time is the sum of the running times associated to individual nodes. Hence, $\mathcal{O}(\Delta|Q|^2|V|)$. ☐

In the case that there is a path from s to t that matches the given regular expression, we can tighten the number of steps that is required by Algorithm 9.1 to the size of the smallest matching path.

Proposition 9.2: *Let η be the shortest path in G between s and t such that $\lambda(\eta) \in r$, where q is an RPQ and r its regular expression. The number of steps required to solve the RPQ recognition problem $\langle G, q, s, t\rangle$ is bounded by $\mathcal{O}(|\eta|)$.*

Proof: Every path of length k is detected after k steps. Hence, for $s, t \in V$, such that there is a path η from s to t in G, the number of steps to detected η is $\mathcal{O}(|\eta|)$. ☐

Reducing the output of Algorithm 9.1 to a yes/no answer can be done with a very simple single-step MapReduce algorithm that maps every $v \in V \setminus \{t\}$ on the empty set, while for t, we generate a pair $\langle k; (s \in D_t)\rangle$. The reduce step can be skipped. Instead of monitoring convergence, which is a difficult task in MapReduce, it suffices to stop the algorithm after $|E||Q|$ consecutive rounds.

9.4 RPQ Evaluation in Logarithmic Rounds

Although the memory bounds in Section 9.3 are promising, the number of MapReduce steps is rather high. Ideally, a smaller number of rounds should be obtained. Therefore, an alternative approach is examined in this section using a pointer-doubling technique. Unfortunately, in this approach the communication cost increases dramatically, and in fact

Algorithm 9.2: RPQ evaluation in logarithmic rounds.

```
Input: $\langle v; D_v\rangle$ for every $v$, where $D_v := \{(v, s_0, v, s_0)\}$
Output: $\langle v; D_v\rangle$ for every $v$, where $D_v$ converged.
repeat
    Map:
        Let $i$ be the current step number;
        foreach $(u, s, u', s', l) \in D_v$ such that $l = 2^{i-1}$ do
            foreach $z = (w', t', w, t, l') \in D_v$ do
                if $u' = w$, $u = v$, and $s' = t$ then
                    emit $\langle u; (u, s, w, t, l + l')\rangle$;
                    emit $\langle w; (u, s, w, t, l + l')\rangle$;
                end
        emit $\langle v; (v, D_v)\rangle$;
    Reduce:
        Let $M = \{m_1, m_2, \ldots, m_k\}$ be the received messages;
        Let $D_v$ be the received state;
        Remove every tuple in $M$ that has a representative in $D_v$ (disregarding the
        lengths);
        $D_v \leftarrow D_v \cup M$;
        emit $\langle v; D_v\rangle$;
until nothing changes;
```

the evaluation problem can be solved without additional cost. Therefore, the logarithmic extension of this algorithm is discussed in the context of the evaluation problem for RPQs.

As discussed previously, a linear number of rounds in the size of the input is not satisfying. Therefore, in this section a variation of Algorithm 9.1 is described where the number of rounds is only logarithmic in the size of the input. Informally, we use a pointer doubling technique to speedup the convergence process at the expense of an increased communication overhead. More particular Algorithm 9.1 is extended to keep track of not only the nodes that can reach the associated node, but also the set of nodes that are reachable from this node and their associated states. Therefore, our algorithm is able to derive paths of length n in $\log(n)$ steps.

Formally, D_v now is a set containing tuples of the form (u, s, u', s', l), where $u, u' \in V$, $s, s' \in Q$, and l a positive integer. The meaning of a tuple (u, s, u', s', l) in D_v is that from node u, starting in state s, there is a path in the input graph to node u' such that there is a matching run that reaches state s'. As before, s' is not necessarily an accepting state. Furthermore, l represents the length of the path. The complete algorithm is described in Algorithm 9.2. The initialization step is represented in Algorithm 9.3.

Although we keep track of the length of paths to filter redundant combinations when generating new paths, this was mainly done for readability; a boolean would equally well suffice. Therefore, we neglect the logarithmic cost of l in the analysis.

Proposition 9.3: *Let $G = (V, E)$ be the input graph, Δ an upper bound on the degree of the nodes in G, and q an RPQ. After $\mathcal{O}(\log(|Q||E|))$ steps of Algorithm 9.2, it holds that*

$$q(G) = \bigcup_{v \in G} \{(u, v) \mid (u, s_0, u', s') \in D_v \textit{ and } s' \in F\}.$$

- ***Key complexity:*** *the amount of space is bounded by $\mathcal{O}(|V||Q|^2 + \Delta)$; the input and output size for single mappers and reducers is bounded by $\mathcal{O}(|V|^2|Q|^2)$; and the running time is bounded by $\mathcal{O}(|V|^4|Q|^4)$.*
- ***Sequential complexity:*** *the total input and output size is bounded by $\mathcal{O}(|V|^3|Q|^2)$; the total running time is bounded by $\mathcal{O}(|V|^5|Q|^4)$.*

Proof: After the initial phase, every node has an associated set D_v containing tuples that represent paths of length one such that v is either a source or target node. Moreover, every path is associated with pairs of states that represent valid runs from one end node to the other.

Assume that in step $i > 1$, D_v contains every pair of nodes that represents a path of length at most 2^{i-1}, such that v is the source or target node and the associated states represent a valid run from one end node to the other. In the map phase, every path of length 2^{i-1} in D_v is extended with its matching paths in D_v to derive new paths of length at most 2^i. Clearly, every path of length $2^{i-1} + k$ that agrees to the transition function is derived, where k is a number between 1 and 2^{i-1}. The newly derived paths are sent to the corresponding end nodes. Hence, in the next phase every node has a tuple corresponding to every valid path of length up to 2^i in its set D_v.

Note that there are some paths that are not derived in v, more particular paths where v is not the $2^{i-1} + 1$th node. Nevertheless, every path of length between $2^{i-1} + 1$ and 2^i has a node at position $2^{i-1} + 1$ that will derive this path in step i.

The longest possible path is bounded by $\mathcal{O}(|Q||E|)$. Hence, after $\mathcal{O}(\log(|E||Q|))$ rounds every accepting tuple of nodes in $q(G)$ is detected.

The size of the state D_v associated to an arbitrary node v is bounded by $\mathcal{O}(|V||Q|^2 + |E_{*\to v} \cup E_{v\to *}|)$. Every node v outputs at most $\mathcal{O}(|V||Q|^2)$ size of messages to every other node and receives at most $\mathcal{O}(|V||Q|^2)$ size from every other node. In a worst-case scenario, every tuple in the state of a node is compared to every tuple received by that node. Hence, the running time is bounded by $\mathcal{O}(|V|^4|Q|^4)$.

The total size of node states is bounded by $\mathcal{O}(|V|^2|Q|^2 + |E|)$. Since every node may send $|V|^2|Q|^2$ size of message, the total amount of messages received or send by nodes is bounded by $\mathcal{O}(|V|^3|Q|^2)$. The worst-case total running time is bounded is bounded by $\mathcal{O}(|V|^5|Q|^4)$. $\square$

Although the shuffle overhead has been decreased by lowering the number of rounds, the communication cost has been increased. Note that the worst-case number of pairs detected after convergence is at most $|V|^2|Q|^2$ (i.e., every node and state combination is reachable from every other node and state combination). Furthermore, since every node

Algorithm 9.3: RPQ evaluation in logarithmic rounds: initialization.

```
Input: $V + E$
Output: $\langle v; D_v \rangle$ for every $v$.
Map:
    foreach $v \in V$ do
        emit $\langle v; v \rangle$;
    foreach $e = (u, a, u') \in E$ do
        emit $\langle u; e \rangle$ emit $\langle u'; e \rangle$
Reduce:
    Let $D_v = \{\}$;
    foreach $(v, a, u) \in E_{v \to *}$ do
        foreach $s' \in \bigcup_{s \in Q} \delta(s, a)$ do
            $D_v \leftarrow D_v \cup \{(v, s, u, s', 1)\}$;
    foreach $(u, a, v) \in E_{* \to v}$ do
        foreach $s' \in \bigcup_{s \in Q} \delta(s, a)$ do
            $D_v \leftarrow D_v \cup \{(u, s, v, s', 1)\}$;
    emit $\langle v; (v, D_v) \rangle$;
```

is responsible for the storage of every node that is reachable from it and for every node that can reach it, every node requires at most $\mathcal{O}(|Q||V|)$ space to store D_v. The main observations, that explains why the communication cost is much higher, is that:

- several paths may exist between a pair of nodes, and consequently, a particular pair may be discovered several times.

However, in Algorithm 9.2 at least every *distinct* path is detected only once.

Proposition 9.4: *Every path is derived at most once.*

Proof: Let $\eta = v_1 v_2 \dots v_m$ be a path in G. In every step, only paths of length more than 2^{i-1} are derived. Hence, when η is derived in step i it cannot be derived again in any later step.

First, let $m = 3$. Clearly, after the second step, η is derived. More particular, by node v_2, which is the only node that knows the path from v_1 to v_2 and the path from v_2 to v_3.

By strong induction it follows that every path is derived only once. Let $m > 3$ and let $i \in \mathcal{Z}$ be the minimal value such that $m < 2^{i-1}$ holds. Hence, η is derived in the ith step of the algorithm. Clearly, η is not derived earlier due to its length. Therefore, there is a node v_j in step i such that the concatenation of paths $\eta_1 = v_1 v_2 \dots v_j$ and $\eta_2 = v_j v_{j+1} \dots v_m$ leads to η. Since in step i, concatenations are only taken if the first path has a length of 2^{i-1}, j has to be 2^{i-1}. By induction, η_1 and η_2 are derived only once. Hence, they are communicated to both end nodes. Only the pivoting node v_j receives both paths and is able to derive η. $\square$

Algorithm 9.4: RPQ evaluation in linear rounds.

```
Input: $\langle v; (v, D_v, E_{v\to *})\rangle$ for every $v$, where $D_v := \{(v, s_0)\}$
Output: $\langle v; (v, D_v, E_{v\to *})\rangle$ for every $v$, where $D_v$ converged.
repeat
    Map:
        foreach active pair $(z, s) \in D_v$ do
            foreach $(v, a, u) \in E_{v\to *}$ do
                foreach $s' \in \delta(s, a)$ do
                    emit $\langle u; (z, s')\rangle$;
            mark $(z, s)$ as passive;
        emit $\langle v; (v, D_v, E_{v\to *})\rangle$;
    Reduce:
        Let $M = \{m_1, m_2, \dots, m_k\}$ be the received messages;
        $D_v \leftarrow D_v \cup M$;
        emit $\langle v; D_v, E_{v\to *}\rangle$;
until nothing changes;
```

9.5 Linear RPQ Evaluation in MR

A linear strategy to find the set of node tuples $q(G)$ (i.e., the evaluation problem for RPQs) is to materialize the closure of G under r by iteratively deriving paths from every node to every other node simultaneously until a fixpoint is reached. To achieve this, assume that every node knows its outgoing neighbours, and associate to every node v a set D_v that contains tuples of the form $(u, s) \in V \times Q$. To be more precise, $(u, s) \in D_v$ if there exists a path η from node u to node v such that $s \in \delta(s_0, \lambda(\eta))$.

Informally, Algorithm 9.4 traverses the graph edge by edge starting in every node from the initial state concurrently. A single step initialisation phase is required to transform the original input graph G into the required input format, and a finalization step can be used to transform the output into the required output format. To prevent redundant derivations, tuples are marked passive after they have been used to derive new tuples. Only active tuples are compared to D_v. The complexity of Algorithm 9.4 is given by Proposition 9.5.

Proposition 9.5: *Let G be the input graph, Δ an upper bound on the degree of the nodes in G, and q an RPQ. After $\mathcal{O}(|E||Q|)$ steps $q(G) = \bigcup_{v \in G}\{(u, v) \mid (u, s_i) \in D_v \text{ and } s_i \in F\}$.*

- ***Key complexity:*** *the input and output for single mappers and reducers is bounded by $\mathcal{O}(\Delta|V||Q|)$; the amount of space is bounded by $\mathcal{O}(|V||Q| + \Delta)$; and the running time is bounded by $\mathcal{O}(\Delta|V||Q|)$.*
- ***Sequential complexity:*** *the total input and output size is bounded by $\mathcal{O}(|E||V||Q|)$; the sequential running time is bounded by $\mathcal{O}(|E||V||Q|)$.*

Proof: Let $s, t \in V$ be an arbitrary pair of nodes such that $(s, t) \in q(G)$. In other words, there exists a path η from s to t such that $\lambda(\eta) \in L(r)$. Assume $\eta = v_0 v_1 \ldots v_m$ where $v_0 = s$ and $v_m = t$. Further, let $s_0, s_1, \ldots, s_m$ be a sequence of states in A_r that represents an accepting run for this path.

Initially, the reducer associated to node v_0 computes $\delta(s_0, \lambda(e))$ for every outgoing edge e. Call this step 0. In particular it computes $\delta(s_0, \lambda(e_1))$ where e_1 is the edge from v_0 to v_1. Therefore, it creates an output tuple $\langle v_0; (v_0, v_1, s_1) \rangle$ that maps on the reducer associated to node v_1. Hence, in the next step, v_1 receives a tuple (v_0, s_1).

In step i, $i > 0$, the reducer associated to v_i receives a tuple (v_0, s_i). Therefore, it is able to compute a tuple (v_0, s_{i+1}) based on its outgoing edge to node v_{i+1} and, consequently, creates output pair $\langle v_i; (v_{i+1}, v_0, s_{i+1}) \rangle$

After step $m + 1$, the reducer associated to node v_m has the tuple (v_0, s_m) in its state D_v and since $s_m \in F$, we are able to conclude that there exists a path from v_0 to v_m that is accepted by A_r.

The construction of paths starts in every node concurrently in state s_0, where after δ is used to determine subsequent states while traversing the graph. Therefore, at any time and any v, $(u, s) \in D_v$ represents a valid (not necessary accepting) run from node u to node v. Hence, only valid paths are detected.

Since for every v, D_v has a size of at most $\mathcal{O}(|V||Q|)$, the size of every record associated to v is bounded by $\mathcal{O}(|V||Q|+|E_{v\to*}|)$. A reducer associated to node v receives its state and the set of messages send to v. In a worst-case scenario, a message is received from every incoming neighbour, containing every node and state combination. Hence, the total input size for a single reducer is bounded by $\mathcal{O}(|\mathrm{V}||\mathrm{Q}| + |E_{v\to*}| + |E_{*\to v}||\mathrm{V}||\mathrm{Q}|)$. Analogous, the mapper output size for a node v is bounded by $\mathcal{O}(|V||Q| + |E_{v\to*}| + |E_{v\to*}||V||Q|)$.

Assuming streaming inputs, the worst-case space requirements for every individual node are $\mathcal{O}(|V||Q| + |E_{v\to*}|)$. Otherwise, the amount of space equals the size of the input. Clearly, from Algorithm 9.4, it takes at most $\mathcal{O}(|D_v||E_{v\to*}|)$ time to compute the map function on node v, and an analogous bound can be found for the reduce function. Hence, the running time for a single map or reduce function is $\mathcal{O}(\Delta|\mathrm{V}||\mathrm{Q}|)$.

In a worst-case scenario every node sends a message to every neighbour containing every node and state combination. Therefore, the total communication cost by messages is bounded by $\mathcal{O}(|E||V||Q|)$. Furthermore, communication caused by passing sets D_v is bounded by $\mathcal{O}(|V|^2|Q|)$. Consequently, the total input size for all reducers in a single step is bounded by $\mathcal{O}(|V|^2|Q| + |E| + |E||V||Q|)$. Assuming that the input graph has at least one edge per node, this bound is in $\mathcal{O}(|E||V||Q|)$. We have that the key running time is bounded by $\sum_{v\in V} |E_{v\to*}||V||Q|$. Hence, the sequential running time is bounded by $\mathcal{O}(|E||V||Q|)$. □

If RPQs are replaced by boolean RPQs in Proposition 9.5, it suffices to find a single matching path. Therefore, the number of rounds in the boolean case is bounded by $\mathcal{O}(|\eta|)$, where η is the shortest path such that $\lambda(\eta) \in L(r)$.

9.6 RPQs in Pregel and GraphLab

First, note that Algorithm 9.1 fits seamlessly in Pregel. Therefore, we can also use this algorithm for the recognition of RPQs in Pregel. In fact, since nodes, outgoing edges and states persist, Pregel seems a more appropriate framework to run this algorithm in practice than MapReduce, where an explicit shuffle phase is involved for all data. However, since the communication cost is dominated by 'communication' pairs, the theoretical complexity results of Proposition 9.1 remain valid in Pregel.

The evaluation algorithms fit in Pregel as well. Instead of maintaining a state via static communication from one task to its successor, as in the naive evaluation, Pregel is able to maintain a state locally, hereby decreasing the communication cost. Though, since the key complexity bounds on input and output size are dominated by the communication of non-state related messages, the asymptotic bounds remain. The total input and output size of the sequential complexity decreases to $\mathcal{O}(|E||V||Q|)$. In practice, many nodes and edges are partitioned on the same local machine. Hence, the real communication cost between machines may be much smaller. In particular, if there are c inter-machine edges, the sequential input and ouput is bounded by $\mathcal{O}(c|V||Q|)$.

The communication cost of the logarithmic approach for evaluating regular path queries in MapReduce is also dominated by messages between nodes. An important difference between this approach and the naive approach, however, is that communication is performed not only throughout edges. Furthermore, although the complexity is high, it is independent from the structure of the input graph. Hence, by using combiners on every local machine, the input and output size for single mappers and reducers is $\mathcal{O}(2|Q|^2|V|p)$, where p is the number of machines. Consequently, the sequential input and output size are bounded by $\mathcal{O}(|Q|^2|V|^2p)$.

A similar analyses can be performed in GraphLab. Since the only communication in the linear approaches is between neighbour nodes, the linear algorithms fit in GraphLab as well. The logarithmic approach can not be simulated in GraphLab in a trivial way since GraphLab uses shared states and has no mechanism to communicate directly between nodes that are note connected by an edge.

10 CRPQs in MapReduce

In this chapter algorithms are given for the evaluation of conjunctive regular path queries in MapReduce. Basically, we combine the knowledge from the previous chapters to formulate complexity bounds for the evaluation and recognition of CRPQs.

10.1 Full-CRPQ Recognition

Let q be a CRPQ consisting of $m-1$ conjunctions and let G be a directed labeled graph. We define the recognition problem for CRPQs as the problem $\bar{z} \in Q(G)$. For simplicity, assume that q is a full query. Hence, the recognition problem for CRPQs can be reduced to the recognition problem for RPQs. Formally, for every RPQ q_r check $(s,t) \in q_r(G)$, where s, t are the corresponding nodes. In particular we have the following result:

Proposition 10.1: *Let $q_1, q_2, \ldots q_m$ be regular path queries, G a directed labeled graph, and Δ an upper bound on the degree of nodes in G. Let Q be the set of states associated to the longest regular expression. Regular path recognition can be computed in MapReduce in $\mathcal{O}(m|Q||E|)$ steps.*

- ***Key complexity:*** *the amount of space per node is bounded by $\mathcal{O}(m|Q| + \Delta)$; the input and output size is bounded by $\mathcal{O}(m.\Delta|Q|)$; and the running time is bounded by $\mathcal{O}(m.\Delta|Q|^2)$.*
- ***Sequential complexity:*** *the total input and output size in a single step is bounded by $\mathcal{O}(m.\max\{|E|,|V|\}|Q|)$; and the total running time in a single step is bounded by $\mathcal{O}(m.\Delta|Q|^2|V|)$.*

Proof: Compute every RPQ concurrently based on the regular reachability algorithm from Section 9.3. $\square$

When the regular path computations are finished, it is easy to check if every path is recognized by sending a message from every reducer that finds a path to a coordinating reducer. If for some RPQ no message is received, decide that $t \notin q(G)$, otherwise decide that $t \in q(G)$.

10.2 General CRPQ Recognition

Without the fullness restriction, the recognition of CRPQs becomes a lot more difficult. With the presence of existentially quantified variables it no longer suffices to recognize the involved regular path queries, rather we have to consider the evaluation of RPQs followed by a join between the results.

In Chapter 9, mainly two algorithms where given for the evaluation of regular path queries in MapReduce: a linear approach and a logarithmic approach. Hence, analogous to Proposition 10.1 upper bounds are provided for the evaluation of multiple RPQs in parallel, based on Proposition 9.3:

Proposition 10.2: *Let $q_1, q_2, \dots q_m$ be regular path queries, G a directed labeled graph, and Δ an upper bound on the degree of nodes in G. Let Q be the set of states associated to the longest regular expression. The evaluation of $q_1.q_2, \dots, q_m$ can be computed in $\mathcal{O}(\log(|Q||E|))$ steps.*

- ***Key complexity:*** *the input and output size are bounded by $\mathcal{O}(m|V|^2|Q|^2)$; the running time is bounded by $\mathcal{O}(m|V|^4|Q|^4)$; and the space is bounded by $\mathcal{O}(m|V||Q|^2 + \Delta)$.*

- ***Sequential complexity:*** *the total input and output size per step are bounded by $\mathcal{O}(m|V|^3|Q|^2)$; and the total running time per step is bounded by $\mathcal{O}(m|V|^5|Q|^4)$.*

A similar result is gained by using the result of Proposition 9.5:

Proposition 10.3: *Let $q_1, q_2, \dots q_m$ be regular path queries, G a directed labeled graph, and Δ an upper bound on the degree of nodes in G. Let Q be the set of states associated to the longest regular expression. The evaluation of $q_1.q_2, \dots, q_m$ can be computed in $\mathcal{O}(|Q||E|)$ steps.*

- ***Key complexity:*** *the input and output size are bounded by $\mathcal{O}(m.\Delta|V||Q|)$; the running time is bounded by $\mathcal{O}(m.\Delta|V||Q|)$; and the space is bounded by $\mathcal{O}(m|V||Q| + \Delta)$.*

- ***Sequential complexity:*** *the total input and output size per step; and the running time per step are bounded by $\mathcal{O}(m|E||V||Q|)$.*

There are several possibilities to perform multi-way joins in MapReduce. In particular, a sequence of two-way joins can be computed based on the data- or reducer-centered approach from Chapter 6, or the multi-way join can be computed directly by using a more advanced algorithm; as the one described in Section 6.7. However, when m is large enough, the key and sequential complexity of the multi-way join subsume the complexity of computing the RPQs (except for the number of rounds). Bounds on a sequential multiway join on graphs are given by Proposition 6.8.

Proposition 10.4: *Let $G = (V, E)$ be a graph, and $q_1(G), q_2(G), \ldots, q_m(G)$ be relations. A multiway join between these relations can be computed in $\mathcal{O}(m)$ steps.*

- ***Key complexity:*** *the input size is bounded by $\mathcal{O}(|V|^m/\sqrt{R})$; the output, space, and running time are bounded by $\mathcal{O}(|V|^{2m}/R)$.*
- ***Sequential complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{R}|V|^m)$; and the output and running time are bounded by $|V|^{2m}$.*

Proof: For every $i \in [1, m] : |q_i(G)| \in \mathcal{O}(|V|^2)$. Hence, the proof follows directly from Proposition 6.8. □

The evaluation of CRPQs is hard. Therefore, in the following sections we also consider the evaluation problem for restricted classes of CRPQs:

- chain CRPQs; and
- acyclic CRPQs.

An illustration is provided in Figure 10.1.

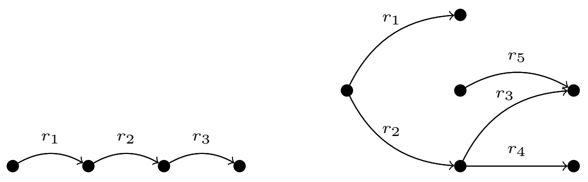


Figure 10.1: Two acyclic CRPQs, the left one is a chain CRPQ.

10.3 Evaluation of Chain CRPQs

A chain CRPQ is a query of the form:

$$q : \bar{z} \leftarrow (x_1, r_1, x_2) \wedge (x_2, r_2, x_3) \wedge \cdots \wedge (x_{m-1}, r_{m-1}, x_m).$$

A chain of RPQs can be seen as an extension of RPQs where internal nodes may be part of the output. Therefore, the evaluation of chain RPQs seems not very different to the evaluation of RPQs, except that we have to keep track of nodes that fill in the non-existentially-quantified variables. Therefore, our strategy consists of the sequential evaluation of RPQs. In particular, we evaluate a chain CRPQ q by first evaluating the first RPQ of q, and then evaluating the second RPQ based on the results of the first, and so on. Unfortunately, except when the query is a boolean query, this strategy does not take away the need to join the results afterwards.

Again, there are mainly two possibilities to implement this strategy: based on te linear approach or based on the logarithmic approach for the evaluation of RPQs. Essentially, we use the algorithms from Chapter 9, but, instead of initializing every node as a potential start node, only the nodes that are valid end nodes in the previous RPQ result are considered.

For example, take Algorithm 9.2 (i.e., the logarithmic RPQ evaluation), instead of initializing every node v with set $D_v = \{(v, s_0, v, s_0)\}$ we only initialize a selected set of nodes in this way. Let q_i be a regular path query and $q_i(G)$ be the result of q_i over G. In the initialization step for the evaluation of q_{i+1}, where $i + 1 > 1$: for every $v \in \{v \mid (u, v) \in q_i(G)\}$ set $D_v = \{(v, s_0, v, s_0)\}$ and for every other node u set $D_u = \{\}$. After each computation of a regular expression, we introduce a MapReduce step that keeps only the necessary information in the status for each node. Every reducer keeps a set D^i_v that contains the nodes that can reach v from the initial state in a final state. Hence, in a worst-case scenario every node has at most $|V|$ information from every RPQ that is already computed.

In Boolean queries there is less need for additional storage capacity in the sense that we do not need to remember the nodes that we have already visited. It suffices to know from which nodes the next RPQ can start. In this case, if an end state is reached for the last expression, the answer for the query is true.

Proposition 10.5: *Let $G = (V, E)$ be the input graph and q a Boolean chain CRPQ containing $q_1, q_2, \ldots, q_m$ as its RPQs. Let Δ be an upper bound on the degree of nodes in G and $|Q|$ be an upper bound on the number of states associated to regular expressions in q. The result of the chain can be computed in $\mathcal{O}(m|log(E)|)$ steps.*

- ***Key complexity:*** *the amount of space for each node is bounded by $\mathcal{O}(|V||Q|^2 + \Delta)$; the input and output size for single mappers and reducers is bounded by $\mathcal{O}(|V|^2|Q|^2)$; and the running time is bounded by $\mathcal{O}(|V|^4|Q|^4)$.*
- ***Sequential complexity:*** *the total input and output size is bounded by $\mathcal{O}(|V|^3|Q|^2)$; and the total running time is bounded by $\mathcal{O}(|V|^5|Q|^4)$.*

Proof: It is easy to prove the proposition by induction of m. When $m = 1$, these bounds follow directly from Proposition 9.3. Assume that for $m = i$ the bounds are correct. For $m = i + 1$ we have that we can evaluate te first i RPQs within these bounds. We use the same algorithm to compute the ith RPQ. First, initialize for every $v \in V$ the set D_v based on the paths that are detected for the previous RPQ. After this step, remove the old states. Again, the bounds of Proposition 9.3 are applicable. □

A similar approach can be used based on the linear RPQ evaluation strategie.

Proposition 10.6: *Let $G = (V, E)$ be the input graph and q a Boolean chain CRPQ containing $q_1, q_2, \ldots, q_m$ as its RPQs. Let Δ be an upper bound on the degree of nodes in G and $|Q|$ be an upper bound on the number of states associated to regular expressions in q. The result of the chain CRPQ can be computed in $\mathcal{O}(m|E||Q|)$ steps.*

- ***Key complexity:*** *the input and output for single mappers and reducers is bounded by $\mathcal{O}(\Delta|V||Q|)$; the amount of space is bounded by $\mathcal{O}(|V||Q| + \Delta)$; and the running time is bounded by $\mathcal{O}(\Delta|V||Q|)$.*

- ***Sequential complexity:*** *the total input and output size is bounded by $\mathcal{O}(|E||V||Q|)$; the sequential running time is bounded by $\mathcal{O}(|E||V||Q|)$.*

Proof: The proof is analogous to the proof of Proposition 10.5 □

10.4 Evaluation of Acyclic CRPQs

In analogy to the acyclic case in the sequential setting, we expected that the evaluation of acyclic CRPQs in a parallel setting is easier then the general case. In particular, the evaluation of multi-way joins. Like before, first the RPQs are evaluated. To take advantage from the acyclicity property, we have to generate a set of relations that is globally consistent. As we have seen in Section 3.5, this can be done by utilizing the equivalence of pair-wise consistency and global consistency, in the acyclic case, by means of a full reducer. A full reducer consists of a polynomial number of semijoins. Therefore, in the next paragraphs we consider the computation of semijoins in MapReduce.

To compute a semijoin $S \ltimes T$, first compute a two-way join $S \bowtie T$, then split the result in a second MapReduce step. For the join, we can use an algorithm as in Section 6.2. The split step is illustrated in Algorithm 10.1. Since the join step dominates in communication

Algorithm 10.1: Semijoin split.

```
Input: $\langle k; (v_1, v_2, u_1, u_2, S \bowtie T)\rangle$ for every $(v_1, v_2, u_1, u_2) \in S \bowtie T$
Output: $\langle k; (v_1, v_2, \mathrm{S})\rangle$ and $\langle k; (u_1, u_2, \mathrm{T})\rangle$ for every $(v_1, v_2, u_1, u_2) \in S \bowtie T$
Map:
    on receive $\langle k; (v_1, v_2, u_1, u_2, S \bowtie T)\rangle$;
    emit $\langle v_1, v_2; (v_1, v_2, S)\rangle$;
    emit $\langle u_1, u_2; (u_1, u_2, T)\rangle$;
Reduce:
    emit the first received pair;
    ignore the rest;
```

cost, the complexity of a semijoin is about the same as taking a join. The following proposition summarizes the complexity of computing a full reducer. In this section we assume that joins satisfy the condition of Proposition 6.3.

Proposition 10.7: *Let $T_1, T_2, \dots, T_m$ be a set of relations in an acyclic database schema. We are able to compute a global consistent set of relations based on $T_1, T_2, \dots, T_m$ in $\mathcal{O}(m^k)$ steps, where k is a constant. Let $|T_i|, |T_j|$ be the size of the two largest relations, where $i, j \in [1, m], i \neq j$.*

- ***Key complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{|T_i||T_j|/R})$; the output size, running time, and space are bounded by $\mathcal{O}(|T_i||T_j|/R)$.*
- ***Sequential complexity:*** *the total input size is bounded by $\mathcal{O}(\sqrt{R|T_i||T_j|})$ per step; the output size and running time are bounded by $\mathcal{O}(|T_i|.|T_j|)$.*

Once the relations are globally consistent, a multi-way join can be performed similar to Proposition 6.8. But, now the complexity bounds can be formulated in terms of the size of the output. Note, however, that these bounds are based on Proposition 6.4 and only apply if the relations (and intermediate relations) involved in a two-way join are are not significantly smaller then each other. Otherwise, the strategy and bounds of Proposition 6.2 should be applied, which makes the analysis a lot more complex.

Proposition 10.8: *Let $T_1, T_2, \ldots, T_m$ be globally consistent sequence of relations and $|T|$ an upper bound on the size of these relations. The iterative implementation of two-way joins to compute $T_1 \bowtie T_2 \bowtie \ldots \bowtie T_m$ can be done in $m-1$ steps. Let $t = |T_1 \bowtie T_2 \bowtie \ldots \bowtie T_{m-1}|$.*

- ***Key complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{t|T_m|/R})$; the output size, running time and space are bounded by $\mathcal{O}(t|T_m|/R)$.*
- ***Sequential complexity:*** *the input size is bounded by $\mathcal{O}(\sqrt{R.t|T_m|})$; the output size and running time are bounded by $t|T_m|$.*

Proof: Without loss of generality assume that $T_1, T_2, \ldots, T_m$ is the order based on the join forest. Now, compute the two-way joins one after the other. The first join takes one step and clearly agrees to the above bounds, based on Proposition 6.4. Since we have that the relations are consistent and the join is based on the order implied by the join forest, we have that the output of every step is always greater or equal to the size of the instances on which the join is based. Hence, the dominating step is the last one. □

11 Conclusions and Outlook

In the current big data era, in which we are overloaded with huge amounts of data, there is a large demand for alternatives to traditional querying systems. One of these alternatives are distributed environments on which huge scale computations are performed in parallel. MapReduce is the first that offers a simple abstract interface to this kind of environments in order to perform general computations, and includes automated fault tolerance. Basically, MapReduce was introduced as an answer to the practical need for robust distributed systems. Theoretical research in the subject has only recently been started. Hence, sound theoretical foundations still have to be established. Therefore, many properties of MapReduce and similar frameworks are still to be discovered.

The large size and irregular shape of the current data landscape let traditional databases and query languages fall short. Therefore, we consider graph databases and conjunctive regular path queries; graph databases, because they naturally model the semi-structured data to which we are exposed in many domains nowadays, and CRPQs, because they combine the most used part of SQL (i.e., conjunctive queries) and navigation functionality. The latter which is essential in a graph context.

First, we discussed the classical results of CQs, RPQs, and CRPQs in a sequential context, based on established theoretical research. In particular, we studied the main complexity results of CQs, RPQs, and CRPQs, as well as the acyclic counterparts, which are well known to be easier when it comes to combined complexity. This study gave us basic insights about the hardness of these languages and the traditional evaluation techniques based on automata constructions. Secondly, we provided a study of MapReduce and its theoretical properties, based on recent work. This study contains a description of the MapReduce framework, and the main models and complexity measures that are available today. We also discussed some MapReduce alternatives; on the one hand, to show that there already exist several alternatives that address some of the problems in MapReduce, and on the other hand, to illustrate that it is an active topic of research and there is still much place for improvement. Furthermore, in anticipation of the intended goal of evaluating CRPQs, we studied algorithms that have already been explored in the MapReduce literature; i.e., taking joins between relations and solving the undirected s-t-connectivity problem on graphs. Thirdly, based on the insights obtained from the domain study, we combine both parts to provide algorithms and upper bounds on the evaluation of RPQs and CRPQs in MapReduce.

However, there are also parts of literature that where not covered in this thesis. In the sequential part, we only discussed the complexity results that seemed important in our context. Furthermore, we did not consider techniques like query rewriting, although these are important in practice; rather, we assumed that queries are evaluated as is. The relations that have already been studied between MapReduce and established models like PRAM and BSP, or the relation between MapReduce and streaming models, are mentioned only briefly, as we did not expect to find directly usable results for our purpose. We also mentioned only a very small selection of MapReduce alternatives and extensions, though there exist many more. Furthermore, by a lack of lower bounds we have no information about the tightness of our bounds on the evaluation of RPQs in MapReduce. In fact, we suspect that there is still much room for improvement, especially with regard to key complexity. When it comes to the evaluation of CRPQs, the complexity is dominated by taking a multi-way join which is a very expensive operation in MapReduce. Experiments could have led to insights in the feasibility of the algorithms in practice. However, we didn't had a dataset available that enabled us to make a relevant analysis.

In general, while doing the domain study of MapReduce, it became clear that, although MapReduce is extensively used in practice, for many purposes it is far from perfect. MapReduce has many shortcomings, amongst others are the difficulties of computing joins and handling graph data – which are key operations in our context. MapReduce has revived the topic of distributed and parallel computations and triggered a wave of abstractions that try to address the shortcomings of it. Therefore, we think that MapReduce should not be considered as an end point, but rather as the impetus to further research into the subject. Abstractions like Pregel, Graphlab, and Powergraph already address some of MapReduce its shortcomings in graph processing. Hence, we suspect that many will follow.

Bibliography

[1] Apache hadoop. `http://wiki.apache.org/hadoop/`.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management.* Cambridge University Press, 2011.

[4] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Cluster computing, recursion and datalog. In *Datalog Reloaded*, volume 6702 of *Lecture Nodes in Computer Science*, pages 120–144. Springer, 2011.

[5] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*, pages 1–8. ACM, 2011.

[6] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. `arXiv:1206.4377 [cs.DC]`, June 2012.

[7] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 99–110. ACM, 2010.

[8] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 132–143, 2012.

[9] Sanjeev Arora and Boaz Barak. *Computational Complexity A Modern Approach.* Cambridge University Press, 2009.

[10] Pablo Barceló, Carlos A. Hurtado, Leonid Libkin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 3–14. ACM, 2010.

[11] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)*, 37(4), 2012.

[12] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[13] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 975–986. ACM, 2010.

[14] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th international conference on World Wide Web (WWW)*, pages 107–117. Elsevier Science, 1998.

[15] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1):285–296, 2010.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 194–204. ACM, 1999.

[17] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC)*, pages 77–90. ACM, 1977.

[18] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering (CISE)*, 11(4):29–41, 2009.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150. USENIX Association, 2004.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 35(6):85–98, 1992.

[22] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 251–262. ACM, 1999.

[23] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory (ICDT)*, pages 8–21. ACM, 2012.

[24] Wenfei Fan, Xin Wang, and Yinghui Wu. Performance guarantees for distributed reachability queries. *Proceedings of the VLDB Endowment*, 5(11):1304–1316, 2012.

[25] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, and Cliff Stein. On distributing symmetric streaming computations. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 710–719. SIAM, 2008.

[26] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 139–148. ACM, 1998.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43. ACM, 2003.

[28] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[29] Ashish Goel and Kamesh Munagala. Complexity measures for map-reduce, and comparison to parallel computing. `arXiv:1211.6526 [cs.DC]`, November 2012.

[30] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 17–30. USENIX Association, 2012.

[31] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383. Springer, 2011.

[32] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[33] Ronald L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the American Federation of Information Processing Societies (AFIPS) Conference*, pages 205–217. AFIPS, 1972.

[34] Steve Haris and Andy Seaborne, editors. *SPARQL 1.1 Query Language*. W3C Recommendation 21 March 2013. `http://www.w3.org/TR/sparql11-query/`.

[35] Jurai Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62(4):565–588, 2001.

[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72. ACM, 2007.

[37] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[38] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system - implementation and observations. In *The 9th IEEE International Conference on Data Mining (ICDM)*, pages 229–238. IEEE Computer Society, 2009.

[39] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948. SIAM, 2010.

[40] André Koschmieder and Ulf Leser. Regular path queries on large graphs. In *Proceedings of the 24th International conference on Scientific and Statistical Database Management (SSDDM)*, pages 177–194. Springer-Verlag, 2012.

[41] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 223–234. ACM, 2011.

[42] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome A. Rolia. Skewtune in action: Mitigating skew in mapreduce applications. *Proceedings of the VLDB Endowment*, 5(12):1934–1937, 2012.

[43] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Record*, 40(4):11–20, 2011.

[44] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349. AUAI Press, 2010.

[46] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8), 2012.

[47] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: System for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 135–146. ACM, 2010.

[48] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[49] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[50] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 949–960. ACM, 2011.

[51] Christopher Olston, Ravi Kumar, Benjamin Reed, Andrew Tomkins, and Utkarsh Srivastava. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110. ACM, 2008.

[52] Matthew F. Pace. BSP vs mapreduce. *Proceedings of the International Conference on Computational Science (ICCS)*, 9(0):246–255, 2012.

[53] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for mapreduce computations. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 235–244. ACM, 2012.

[54] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. `arXiv:1203.5387 [cs.DC]`, November 2012.

[55] Michael Stonebraker, Daniel Abadi, David J. Dewitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel DBMSs: Friends or foes. *Communications of the ACM*, 53(1), 2010.

[56] Dan Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, 2002.

[57] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web (WWW)*, pages 607–614. ACM, 2011.

[58] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[59] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 137–146. ACM, 1982.

[60] Moshe Y. Vardi. A call to regularity. In *Principles of Computing & Knowledge.* ACM, 2003.

[61] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[62] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pages 82–94. IEEE Computer Society, 1981.

[63] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *Proceedings of the VLDB Endowment*, 5(11):1184–1195, 2012.

Auteursrechtelijke overeenkomst