MASTERPROEF

Coordination-freeness and Parallel Evaluation of Conjunctive Queries

Auteur:
Brent CHESNY

Promotor:
Prof. dr. Frank NEVEN

Begeleider:
Bas KETSMAN

Masterproef voorgedragen tot het behalen van de graad van master in de informatica, afstudeerrichting databases

Academiejaar 2016-2017

Abstract

The interest in utilizing parallel and distributed systems to process large amounts of data has grown immensely over the past few years. However, this also brings along some interesting new challenges. In this thesis, we focus on coordination on the one hand, and efficient parallel evaluation of conjunctive queries on the other. We study the existing theoretical research regarding coordination-freeness and the CALM-conjecture and formalize some variations on the current model. Then we turn our attention to more practical evaluation strategies specifically for conjunctive queries. First, we look at optimal broadcasting strategies based on oblivious broadcasting functions. Next, we look at two algorithms in the MPC model: the single-round HyperCube algorithm which is optimal on skew-free input instances, and a multi-round algorithm based on HyperCube which is worst-case optimal even on skewed input instances. To conclude, we made an implementation of these last two algorithms and conducted several experiments to better understand their practical behaviour.

Acknowledgements

I would like to take this opportunity to express my gratitude to a number of people who helped realize this thesis. First and foremost, I would like to thank my promotor Prof. dr. Frank Neven for allowing me to study this topic and guiding me through the process of creating this work. Furthermore, I'd like to thank my supervisor Bas Ketsman for all his thoughtful remarks and insights and the proofreading of my drafts.

I also want to thank dr. Geert Jan Bex for his help with the setup of the VSC cluster for the conducted experiments.

Finally, I would like to thank my parents for giving me the opportunity to study and for their invaluable support during that time. A special mention also goes out to my friends for the interesting discussions and the necessary moments of distraction during the past five years. *Thank you!*

Diepenbeek, June 2017

In loving memory of my father

Dutch Summary Nederlandse Samenvatting

Inleiding

De laatste jaren is *big data* een van de meest populaire begrippen in allerlei industriën. Men spreekt over big data wanneer het verwerken van deze data niet meer haalbaar is op één enkele computer. Een mogelijke manier om hier mee om te gaan is de te verwerken data verdelen over meerdere computers die door een netwerk verbonden zijn en zo de berekeningen in parallel uit te voeren. Eén van de redenen waarom big data zo populair geworden is, is dat dergelijke clusters van computers de laatste tijd veel toegankelijker zijn geworden voor organisaties. Bovendien werden ondertussen ook heel wat softwaresystemen ontwikkeld die gebruik maken van parallelisme, en die het werk voor de gebruiker in deze context sterk verlichten. Voorbeelden van zulke softwaresystemen zijn Apache Hadoop [1] en het recentere Apache Spark [2, 18].

Deze parallelle manier van werken brengt echter ook heel wat uitdagingen met zich mee. Een van deze uitdagingen betreft coördinatie. Indien verschillende computers tijdens het berekenen van een query moeten coördineren kan dit een belangrijke bron van inefficiëntie zijn. Het is dus zeker interessant om eens na te denken over welke queries te berekenen zijn zonder expliciete coördinatie en welke niet. Een andere uitdaging betreft het herverdelen van data in het netwerk. Moderne big data systemen kunnen een query meestal volledig in het geheugen verwerken. De klassieke maat voor performantie, het aantal disk I/Os, wordt dus vervangen door een nieuwe bottleneck, namelijk het netwerkverkeer ten gevolge van het herverdelen van de data over de verschillende computers. Dit fenomeen leidde reeds tot heel wat onderzoek naar algoritmen die deze communicatie proberen te minimaliseren.

Coördinatie-vrijheid

Hellerstein formuleerde in 2010 het CALM-conjecture [13]. Hiermee drukte hij zijn vermoeden uit dat een query een uitvoeringsstrategie heeft zonder expliciete coördinatie die uiteindelijk consistent is, als en slechts als de query uitdrukbaar is in monotone Datalog (zonder negatie of aggregatie). Dit vermoeden was echter niet geformaliseerd, omdat dit een formeel model vereist om over gedistribueerde berekeningen te redeneren alsook duidelijke definities van de begrippen coördinatie en uiteindelijke consistentie. Hiervoor stelde Ameloot et al. [8] een model voor op basis van relationele transducers, samen met de andere nodige definities.

Relationele transducer netwerken De basis voor een relationele transducer is het *transducer schema* $\Upsilon = \langle \Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}} \Upsilon_{\text{sys}} \rangle$ dat bestaat uit een aantal database schema's voor input, output, messages, geheugen en het systeem zelf. Een *transducer staat* voor een bepaald transducer schema is vervolgens simpelweg een database instance over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$. De relationele transducer Π zelf is een verzameling van queries over het transducer schema. Deze queries vormen het mechanisme om de transducer staat aan te passen en messages uit te sturen. De overgang van een transducer staat naar een andere noemen we een lokale transitie.

Een relationeel transducer netwerk $\mathcal{T}$ kunnen we vervolgens definiëren als een netwerk van nodes $\mathcal{N}$, waarbij op elke node in het netwerk dezelfde relationele transducer draait. Verder kunnen we een gedistribueerde database instance definiëren als een functie H die elke node van het netwerk mapt op een gewone database instance. We kunnen een gewone database instance I als input geven aan een transducer netwerk door deze instance te partitioneren over het network. Zo een horizontale partitie kunnen we definiëren als een gedistribueerde database H over $\mathcal{N}$ waarvoor geldt dat $I = \bigcup_{x \in nodes(\mathcal{N})} H(x)$.

De configuratie ρ van een transducer netwerk $\mathcal{T}$ voor een gegeven gedistribueerde database instance H wordt gegeven door een staat-functie $s(x)$ die elke node mapt op een transducer staat en een buffer-functie $b(x)$ die elke node mapt op een multiset van *message*-facts. Een overgang tussen twee configuraties noemen we een globale transitie. Hierbij wordt er een arbitraire node in het netwerk actief. Deze node ontvangt vervolgens een submultiset van de messages in zijn buffer en voert hiermee een locale transitie uit. De output van een gegeven configuratie is de unie van alle output-facts over alle nodes in het netwerk:

$$out(\rho) = \bigcup_{x \in nodes(\mathcal{N})} s(x)|_{(\text{out})}.$$

Een run $\mathcal{R}$ van een transducer netwerk $\mathcal{T}$ over een gedistribueerde database H is vervolgens een reeks configuraties $\rho_1, \rho_2, \ldots$. We noemen het getal

$i \geq 1$ waarvoor $out(\rho_j) = out(\rho_i)$ voor elke $j > i$ een rustpunt voor $\mathcal{R}$. De overeenkomstige configuratie ρ_i noemen we de rustconfiguratie. Elk transducer netwerk heeft zo een rustconfiguratie vanwege het feit dat er slechts een eindig aantal output facts mogelijk zijn. We kunnen nu zeggen dat een transducer netwerk $\mathcal{T}$ consistent is als voor elke input instance I geldt dat alle runs van $\mathcal{T}$ dezelfde output hebben voor alle horizontale partities van I over $\mathcal{N}$. We zeggen dat $\mathcal{T}$ een query $\mathcal{Q}$ berekent indien de output van $\mathcal{T}$ gelijk is aan $\mathcal{Q}(I)$ voor elke instance I waarvoor $\mathcal{Q}$ gedefiniëerd is. Verder noemen we een transducer Π netwerk-onafhankelijk indien het overeenkomstige transducer netwerk $\mathcal{T}$ voor elk netwerk $\mathcal{N}$ consistent is en dezelfde query berekent. In dat geval zeggen we dat $\mathcal{Q}$ gedistribueerd berekend wordt door Π.

CALM-conjecture Nu dat we een formeel model voor gedistribueerde berekeningen hebben, moeten we enkel nog het begrip *coördinatie-vrijheid* definiëren. Volgens de definitie van Ameloot et al. zeggen we dat een transducer netwerk $\mathcal{T}$ coördinatie-vrij is indien voor elke database instance I, er een horizontale partitie H van I en een run $\mathcal{R}$ van $\mathcal{T}$ over H bestaan zodat $\mathcal{R}$ reeds een rustconfiguratie bereikt zonder ooit messages te lezen uit de messagebuffers. We noemen een transducer Π coördinatie-vrij indien voor elk netwerk het overeenkomstig transducer netwerk coördinatie-vrij is. We zeggen dat een query $\mathcal{Q}$ gedistribueerd berekend kan worden op een coördinatie-vrije manier als er een netwerk-onafhankelijke, coördinatie-vrije transducer bestaat die $\mathcal{Q}$ berekent. Dit komt erop neer dat wanneer de input data op de juiste manier verdeeld is over de servers, er totaal geen communicatie nodig is om de query te berekenen.

We kunnen nu het CALM-conjecture van Hellerstein herformuleren door gebruik te maken van bovenstaande begrippen.

Conjecture. *Een query kan gedistribueerd berekend worden door een coördinatie-vrije transducer als en slechts als hij uitdrukbaar is in Datalog.*

Ameloot et al. wisten dit conjecture te formaliseren in de volgende zin. Voor het volledig bewijs en een gedetailleerde uitleg verwijzen we naar Hoofdstuk 3 van deze thesis.

Theorem. *Volgende uitspraken zijn equivalent voor een query $\mathcal{Q}$:*

1. *$\mathcal{Q}$ kan gedistribueerd berekend worden door een coördinatie-vrije transducer;*
2. *$\mathcal{Q}$ kan gedistribueerd berekend worden door een transducer die zich onbewust is van het netwerk waarin hij zich bevindt;*
3. *$\mathcal{Q}$ is monotoon.*

Alternatieve definities en modellen In sommige gevallen is het moeilijk om te redeneren met de bovenstaande definitie van coördinatie-vrijheid, die stelt dat er *een* horizontale moet bestaan zonder te specifiëren hoe deze er moet uitzien. Daarom stellen wij een alternatieve definitie voor die wel een specifieke partitionering van de data vermeldt. Deze definitie stelt dat een query $\mathcal{Q}$ gedistribueerd berekend kan worden op een coördinatie-vrije manier indien er een netwerk-onafhankelijke transducer Π bestaat die zich niet bewust is van het netwerk waarin hij zich bevindt zodat het transducer netwerk voor Π met één enkele node $\mathcal{Q}$ steeds correct berekend wanneer die ene node de volledige input krijgt.

Verder bekijken we ook een variatie op het transducer netwerk model waarbij alle messages steeds worden ontvangen in dezelfde volgorde als dat ze werden verzonden. Het originele model beschouwt een betrouwbaar netwerk in de zin dat messages nooit verloren gaan, maar hun volgorde blijft niet behouden. Dit in tegenstelling tot de betrouwbaarheidsgaranties van het populaire TCP-protocol. We tonen aan dat deze aanpassing de eventuele coördinatie-vrijheid van een transducer behoudt.

Optimale Broadcasting Strategiën

De coördinatie-vrije oplossingsstrategiën die gebruikt worden in de formalisatie van het CALM-conjecture zijn niet echt praktisch. Hierbij broadcasten alle nodes hun lokaal fragment van de data zodat elke node de volledige dataset kan verzamelen. Dit is natuurlijk niet heel efficiënt. Daarom kijken we naar meer economische broadcasting strategiën die enkel de data broadcasten die strikt noodzakelijk is om een query correct te berekenen [14].

In een oplossingsstrategie die gebaseerd is op broadcasting moet een relationele transducer slechts twee dingen doen:

1. Bepalen welke facts naar de andere nodes in het netwerk gestuurd dienen te worden;
2. Potentieel nieuwe output facts berekenen wanneer er nieuwe data binnenkomt op een node.

De enige veranderende factor in deze strategie is de verzameling van facts die gebroadcast moet worden.

Oblivious broadcasting functies We kunnen deze verzameling formaliseren met behulp van *oblivious broadcasting functies* (OBF). Zo een OBF f is een mapping van instances naar instances zodat $f(J) \subseteq J$ voor alle instances J. We noemen $f(J)$ dan de gebroadcaste facts. De verzameling $J \setminus f(J)$ noemen we de statische facts. Niet alle OBFs leidden echter tot een correcte berekening voor een gegeven query. We kunnen de correctheid van een OBF

f t.o.v. een query $\mathcal{Q}$ karakteriseren door te stellen dat twee compatibele facts op verschillende nodes nooit allebei statisch kunnen zijn. We noemen twee facts $\mathbf{f}$ en $\mathbf{g}$ compatibel, genoteerd als $\mathbf{f} \sim_{\mathcal{Q}} \mathbf{g}$, indien ze beiden kunnen bijdragen tot een geldige valuatie van $\mathcal{Q}$.

Naast correct willen we natuurlijk ook dat een OBF zo weinig mogelijk data broadcast. Daarom definiëren we de notie van *lokale optimaliteit*. Een OBF f is lokaal optimaal indien er geen andere OBF g bestaat die slechts een subset van f broadcast voor een bepaalde instance. Ook dit kunnen we karakteriseren in termen van compatibele facts. We noemen een OBF f lokaal optimaal indien voor elke instance I en fact $\mathbf{f}$ waarvoor geldt dat $\mathbf{f} \in f(I \cup \{\mathbf{f}\})$, er een instance J en een fact $\mathbf{g}$ bestaan zodat $\mathbf{f} \sim_{\mathcal{Q}} \mathbf{g}$, $\mathbf{g} \notin I$, $\mathbf{f} \notin J$ en $\mathbf{g} \notin f(J \cup \{\mathbf{g}\})$. Dit is makkelijk in te zien. Namelijk, als er geen compatibel fact $\mathbf{g}$ bestaat dat niet gebroadcast wordt, is het niet nodig om $\mathbf{f}$ te broadcasten en kan er dus een optimalere OBF verkregen worden.

Broadcast dependency sets We introduceren *broadcast dependency sets* (BDS) als een syntactisch alternatief om de semantische OBFs mee te beschrijven. Voor we hier dieper op ingaan introduceren we eerst het concept van atomische types. Een atomisch type $\tau = (R_\tau, \varphi_\tau)$ wordt gedefiniëerd over een query $\mathcal{Q}$. Hierbij is R_τ een predicaat en φ_τ een equality type. Dit equality type is een binaire relatie tussen de variabelen in $\mathcal{Q}$ die aangeeft of twee variabelen gelijk moeten zijn of niet. We kunnen nu zeggen dat een fact $\mathbf{f}$ van het type τ is als er een valuatie h bestaat zodat $h(\mathit{atom}(R_\tau)) = \mathbf{f}$ en aan φ_τ voldoet wanneer we elke variabele x_i vervangen door $h(x_i)$. We kunnen een BDS nu in essentie definiëren als een verzameling van key-value paren (τ, T) waarbij τ een atomisch type is en T een verzameling van atomische types die we de dependency set noemen. Een BDS $\mathcal{S}$ induceert een OBF met het volgende gedrag: een fact $\mathbf{f}$ wordt gebroadcast als het type van $\mathbf{f}$ gedefiniëerd is over $\mathcal{Q}$ en als de dependency fact set $\mathit{Dep}(\mathbf{f}, \mathcal{S})$ niet aanwezig is in de lokale instance. De dependency fact set $\mathit{Dep}(\mathbf{f}, \mathcal{S})$ is gedefiniëerd als $\{V_{\mathbf{f},\tau'}(\mathit{atom}(\tau')) \mid \tau' \in T \text{ and } \tau \sim_{\mathcal{Q}} \tau'\}$ indien $(\mathit{type}(\mathbf{f}), T) \in \mathcal{S}$ en is ongedefiniëerd in het andere geval. Hierbij is $V_{\mathbf{f},\tau'}$ de valuatie van $\mathbf{f}$ uitgebreid naar τ'. Voor een BDS kunnen we eveneens syntactische condities geven voor correctheid en lokale optimaliteit. Deze condities laten ons toe een eenvoudig algoritme te beschrijven om een correcte en lokaal optimale OBF te construeren voor een gegeven query $\mathcal{Q}$.

Het HyperCube Algoritme

Het MPC model We kijken ook naar enkele andere algoritmen voor het berekenen van conjunctieve queries. Deze zijn te situeren in een meer gesynchroniseerde setting dan de broadcasting strategiën. We noemen deze setting het *Massively Parallel Computation* model. Hierin vindt de berekening plaats in een aantal *rounds*. Elke round bestaat uit twee fases:

1. **Communicatiefase**: Elke server stuurt data uit naar een ontvangt data van andere servers in het netwerk;
2. **Berekeningsfase**: Elke server voert lokaal een berekening uit op de data die zich momenteel op die server bevindt.

Dit model maakt synchronisatie expliciet, omdat de servers op het einde van elke round moeten wachten tot alle servers klaar zijn met hun berekening voordat ze aan de volgende round kunnen beginnen. Een belangrijke maat voor performantie in dit model is de maximale load van elke server. Dit is de maximale hoeveelheid data die een server kan ontvangen in één round.

HyperCube Het eerste algoritme dat we bestuderen in deze context is het HyperCube algoritme [7, 9]. Dit algoritme kan gebruikt worden om een conjunctieve query van de vorm

$$Q(x_1, \ldots, x_k) \leftarrow S_1(\bar{x}_1), \ldots, S_l(\bar{x}_l).$$

in één round te berekenen. Het algoritme kent aan elke variabele x_i een waarde p_i toe zodat $\prod_{i=1}^{k} p_i = p$. Deze waarden noemen we de *shares*. Elke server kan nu worden voorgesteld door een punt $\mathbf{y} \in \mathcal{P}$, met $\mathcal{P} = [p_1] \times \cdots \times [p_k]$. Met andere woorden, de servers worden gemapt naar punten in een k-dimensionale hypercube. Het algoritme gebruikt vervolgens k onafhankelijke hash-functies $h_i : \mathbf{dom} \mapsto [p_i]$ om elk tuple t in relatie $S_j(x_{i_1}, \ldots, x_{i_n})$ naar alle servers in de verzameling

$$\mathcal{D}(t) = \{\mathbf{y} \in \mathcal{P} \mid \forall j \in [n] : h_{i_j}(t[i_j]) = \mathbf{y}_{i_j}\}.$$

te sturen. In deze uitdrukking gebruiken we $t[i_j]$ om de projectie van tuple t op de variabele x_{i_j} aan te duiden. Tenslotte berekent elke server de query Q lokaal op de ontvangen data tijdens de berekeningsfase. Dit algoritme is optimaal over skew-free input.

Het HyperCube algoritme wordt geparameteriseerd door de keuze van de shares. Er zijn verschillende manieren om deze shares te bepalen. Een mogelijke manier is om de exacte optimale shares te bepalen m.b.v. linear programming. Het nadeel hiervan is dat deze methode meestal geen integer-waarden oplevert, wat wel vereist is voor het algoritme. De bekomen waarden kunt wel naar beneden afgerond worden maar dit leidt vaak tot suboptimale resultaten. Een andere optie is om de optimale integrale shares te bepalen via een cost-based aanpak.

Multi-round algoritme Naast het HyperCube algoritme beschouwen we ook een multi-round algoritme dat gebaseerd is op HyperCube [15]. Dit algoritme is echter worst-case optimaal, wat wil zeggen dat het de minimale verwachte maximum load behaalt zonder restricties te plaatsen op de input. Die input mag dus skew bevatten. Voor dit algoritme beschouwen we slechts een deelklasse van de conjunctieve queries, namelijk de *simple* en *connected* conjunctieve queries die een beschikken over een *tight fractional edge packing*.

Het idee achter dit algoritme is als volgt. Zij $\mathcal{Q}$ de query die we willen berekenen. We beschouwen dan alle mogelijke heavy-hitter configuraties Ψ voor $\mathcal{Q}$. Een heavy-hitter configuratie is een paar (H, δ) waarbij H een subset is van de variabele van de query en δ een threshold die we gebruiken om *heavy* en *light* waarden te onderscheiden. We noemen een waarde c light voor een variabele x als het aantal voorkomens van c op een positie van variabele x in de query kleiner is dan m/p^{δ}, waarbij m het aantal tuples in de grootste relatie is. In het andere geval noemen we c heavy. Vervolgens berekenen we voor elke Ψ in parallel de query $\mathcal{Q}$ over de subinstance die compatibel is met Ψ gebruikmakend van een gespecialiseerd algoritme afhankelijk van H. De output $\mathcal{Q}(I)$ is dan precies de unie van de berekeningen op deze subinstances.

Implementatie en Experimenten

Om een beter inzicht te krijgen in de algoritmen uit de vorige sectie werden zowel het HyperCube algoritme als het multi-round algoritme geïmplementeerd bovenop Apache Spark [2]. Bovendien implementeerde we een naïef algoritme voor het berekenen van conjunctieve queries, dat we als baseline konden gebruiken bij de experimenten. Hierbij evalueren we de joins op een traditionele manier, gebruikmakend van één round per join. Naast deze shuffle algoritmen implementeerden we twee lokale join algoritmen: een klassieke binaire hash-join en de multiway leapfrog trie-join [17]. Op die manier kunnen we hun impact bestuderen wanneer we ze combineren met de verschillende shuffle algoritmen.

Met behulp van deze implementatie werden eveneens enkele experimenten uitgevoerd. In een eerste experiment trachtten we te achterhalen of een cost-based aanpak voor het bepalen van de HyperCube shares tot een snellere uitvoeringstijd leidt dan het afronden van de fractionele shares. Uit dit experiment bleek dat de cost-based aanpak tot 26% performanter was. In een tweede experiment vergeleken we de performantie van het HyperCube algoritme en het naïeve algoritme op skew-free data. Hier bleek het HyperCube algoritme superieur, maar enkel voor queries die veel tussenliggende join resultaten genereerden. Voor queries waar dit niet het geval was bleek de naïeve aanpak efficiënter. In een laatste experiment vergeleken we hoe de verschillende algoritmen omgingen met skewed data. Hier bleek het multi-round algoritme beter dan het HyperCube algoritme op voorwaarde dat er voldoende skew was en het aantal heavy-hitters niet te groot. Ook konden we concluderen dat de leapfrog trie-join in het algemeen zeer goed werkt in combinatie met skewed data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Big Data and Its Challenges

For several years, *big data* has been one of the hottest buzzwords across all industries. But what exactly is big data? A common definition is the one given by Douglas Laney in 2012:

> *"Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."* [16]

This definition captures one of the most important aspects of big data. Namely, that we're really talking about big data when processing on a single machine is no longer feasible. One of the main reasons for the popularity of big data is that the rise of cloud computing and commodity hardware clusters has made distributed processing a lot more accessible to organizations over the past few years. Additionally, many modern data management systems have been developed to make use of the power of parallelism. Examples of such systems are Apache Hadoop [1] and the more recent Apache Spark [2, 18].

However, systems like these also present us with some interesting new challenges. One of those is regarding coordination. An inherent source of inefficiency in distributed systems are the global barriers raised by the need for synchronization during the computation of a query. It is therefore interesting to look into what kind of queries can be computed without synchronization, in a coordination-free fashion. This caused Hellerstein to formulate the CALM-conjecture [13], which suggests a link between monotonicity on one hand and eventual consistency without the need for coordination on the other hand. However, a proper treatment of this conjecture requires clear definitions of eventual consistency and coordination. These were later provided by Ameloot et al. [8], who proposed a formalization of the conjecture.

Another challenge of big data management systems is one of efficiency in general. Systems like Spark are designed to process data in-memory, because they can use a sufficient number of servers to make sure that all data fits in main memory. This is great because it generally offers a performance improvement over older systems. However, the traditional performance measure consisting of the number of disk I/Os is replaced by a new bottleneck: the communication cost of reshuffling data across the network during computation. In this thesis, we study two kinds of algorithms for efficient evaluation of conjunctive queries in particular. One is an improvement of the broadcasting strategy used in the formalization of the CALM-conjecture, which states that every monotone query can be evaluated in an eventually consistent and coordination-free manner through a naive broadcasting strategy that makes all data available to all nodes. This is however rather wasteful. The improved broadcasting strategies that we consider in this work are optimal in the sense that they broadcast no more data than strictly necessary in order to correctly compute the query [14]. The second kind of algorithm that we consider is one that can be modelled in the MPC model. In this model, a computation proceeds in rounds: each round consists of some local computation followed by global exchange of data between the servers. We first consider the HyperCube algorithm that can compute any conjunctive query in a single communication round, with optimal load on skew-free input instances [7, 9]. Next, we also look at a multi-round algorithm based on HyperCube which is worst-case optimal even on skewed input instances [15]. To really understand the practical behavior of these algorithms, we made an implementation on top of Apache Spark. We finish by performing an experimental evaluation of this implementation. In particular, we compare the algorithms to a naive baseline implementation and investigate how well these algorithms perform in combination with a modern sequential multi-way join algorithm [17].

1.2 Goals and Contributions

The goals and contributions of this thesis can be summarized as follows.

1. Study the existing theoretical research regarding coordination-freeness and the CALM-conjecture and formalize some variations on the current model.

2. Look into more practical computation strategies than those mentioned in the research about coordination-freeness, specifically for conjunctive queries.

3. Implement and experimentally verify some of these algorithms using Apache Spark.

1.3 Thesis Outline

In Chapter 2, we give an overview of some basic concepts in database theory that are used heavily throughout the rest of the text. In Chapter 3, we dive deeper into coordination-freeness and the CALM-conjecture. We then study a way to create more optimal broadcasting strategies in Chapter 4. Subsequently, we turn our attention to a more synchronized setting in which we study the single-round HyperCube algorithm and a multi-round algorithm based on HyperCube. This is covered in Chapter 5. We then detail our implementation of these algorithms in Chapter 6. Next, we discuss the experimental validation of this implementation in Chapter 7. We end this thesis by giving our conclusions and outlook on future work in Chapter 8.

Chapter 2

Preliminaries

In this chapter we give a short overview of some of the basic concepts that are used throughout this thesis [6]. We first describe the relational model for databases. Next, we introduce the notion of conjunctive queries. Lastly, we also formally define the Datalog query language.

2.1 Relational Database Model

We recall Codd's relational model. Formally, a *relational database schema* σ is a collection of relation names R, where each R has an associated arity $ar(R)$. We also assume an infinite set **dom** of data values. We call $R(x_1, \ldots, x_k)$ a *fact* if R is a relation name and $(x_1, \ldots, x_k)$ is a tuple over **dom**. We say that this fact is *over* a database schema σ if $R \in \sigma$ and $ar(R) = k$. We can then define a *relational database instance* I over σ as a finite set of facts over σ. For an instance I over σ, and a schema σ', we define $I|_{\sigma'}$ as the set of facts in I that are over σ'. Throughout this text we assume that σ is clear from the context and do not mention it explicitly.

Furthermore, we define a *query* $\mathcal{Q}$ over a schema σ to a schema σ' as a generic mapping of instances over σ to instances over σ'. By genericity we mean that for every permutation π of **dom** and every instance I, $\mathcal{Q}(\pi(I)) = \pi(\mathcal{Q}(I))$.

Example 2.1. Consider a databse schema $\sigma = \{R, S\}$, with $ar(R) = 2$ and $ar(S) = 3$. An example of a database instance over σ is

$$I = \{R(a,b),\ R(a,c),\ R(b,c),\ S(a,b,d),\ S(c,e,f)\}.$$

✦

We also formally define the notion of monotonicity.

Definition 2.2. A query $\mathcal{Q}$ is *monotone* if $\mathcal{Q}(I) \subseteq \mathcal{Q}(I \cup J)$ for all database instances I and J.

Intuitively, this means that when the instance grows, previously generated outputfacts remain valid also for the new instance.

To conclude, we introduce the following convenient notations. For a finite set S we denote by $|S|$ its cardinality, and by 2^S its powerset. Additionally, we denote $\{1, \ldots, n\}$ by $[n]$, for $n \in \mathbb{N}$.

2.2 Conjunctive Queries

Let σ be a database schema and assume a universe **var** of variables, disjoint from **dom**. A *conjunctive query* (CQ) over σ is an expression of the form

$$Q(\bar{u}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

where $n \geq 1$, $R_1, \ldots, R_n$ are relation names in σ and $\bar{u}, \bar{u}_1, \ldots, \bar{u}_n$ are tuples over **var**. Additionally, we require that $\bar{u}_1, \ldots, \bar{u}_n$ have the appropriate arities. That is, for $\bar{u}_i = (x_1, \ldots, x_k)$, we must have that $k = ar(R_i)$. Finally, each variable occurring in $\bar{u}$ must also occur at least once in $\bar{u}_1, \ldots, \bar{u}_n$. An expression of the form $R(x_1, \ldots, x_k)$ is called an *atom*. Let A be an atom of this form. We call R the predicate, denoted by $pred(A)$. We denote the variables $\{x_1, \ldots, x_k\}$ that occur in A as $Vars(A)$.

Let $\mathcal{Q}$ be a CQ of the form described above. We call $Q(\bar{u})$ the head of $\mathcal{Q}$, denoted as $head_{\mathcal{Q}}$. Similarly, we call $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ the body of $\mathcal{Q}$, denoted as $body_{\mathcal{Q}}$. By $Vars(\mathcal{Q})$ we denote the set of variables occurring in $\mathcal{Q}$. For convenience, we also use the query name as the head-predicate throughout the text. That is, for a query $\mathcal{Q}$ we assume $\sigma' = \{\mathcal{Q}\}$.

Next, we describe the semantics of conjunctive queries in terms of valuations. Formally, a *valuation* for $\mathcal{Q}$ on an instance I is a total function $V : Vars(\mathcal{Q}) \rightarrow \textbf{dom}$. The application of V on an atom $A = R(x_1, \ldots, x_k)$ results in a fact $R(a_1, \ldots, a_k)$ where $a_i = V(x_i)$ for each $i \in [k]$. We say that a valuation V is *satisfying* for $\mathcal{Q}$ over instance I if $V(A) \in I$ for all atoms A in $body_{\mathcal{Q}}$. In that case it is said that V derives the fact $V(head_{\mathcal{Q}})$. The result of $\mathcal{Q}$ on I, denoted $\mathcal{Q}(I)$, is defined as the set of facts that can be derived by satisfying valuations.

Example 2.3. Consider the database instance I from Example 2.1 and the following CQ:

$$Q(x, y, z) \leftarrow R(x, y), S(y, z, w).$$

One possible valuation for Q is given by $V : \{x \mapsto a, y \mapsto c, z \mapsto e, w \mapsto f\}$. Applying this valuation to the body atoms of Q yields $R(a, c)$ and $S(c, e, f)$, both of which are contained in I. It follows that V derives the fact $Q(a, c, e)$. The complete result of $Q(I)$ is the set $\{Q(a, c, e), Q(b, c, e)\}$.

✦

To conclude, we mention a restricted class of conjunctive queries where every conjunctive query is full and does not contain self-joins. Formally, this means that $\mathit{pred}(A) \neq \mathit{pred}(B)$ for every distinct pair of atoms A and B in body_Q, and $\mathit{Vars}(\mathit{head}_Q) = \bigcup_{A \in \mathit{body}_Q} \mathit{Vars}(A)$. In other words, every body atom has a unique relation symbol and all variables occurring in body atoms must also occur in the head.

Example 2.4. Consider the following conjunctive queries:

$$\begin{aligned} Q_1(x,y,z) &\leftarrow R(x,y), S(y,z), T(x,z) \\ Q_2(x,y) &\leftarrow R(x,y), S(y,z), T(x,z) \\ Q_3(x,y,z) &\leftarrow R(x,y), S(y,z), S(x,z). \end{aligned}$$

Q_1 is full and does not contain any self-joins. On the other hand, Q_2 is not full, while Q_3 contains a self-join. ✦

2.3 Datalog

In this section, we recall the more expressive query language Datalog with negation, denoted Datalog$^\neg$ [6]. A Datalog$^\neg$ rule φ is a tuple $(\mathit{head}_\varphi, \mathit{pos}_\varphi, \mathit{neg}_\varphi)$ where head_φ is an atom, and pos_φ and neg_φ are both sets of atoms. We call the components of this tuple the *head*, the *positive body atoms* and the *negative body atoms* respectively. We require that pos_φ must contain at least one atom, whereas neg_φ may be empty. If $\mathit{neg}_\varphi = \emptyset$ then we call φ *positive*. The variables that appear in a rule φ are denoted by $\mathit{Vars}(\varphi)$. We say that a rule φ is over schema σ if for every atom $R(u_1, \ldots, u_k) \in \{\mathit{head}_\varphi\} \cup \mathit{pos}_\varphi \cup \mathit{neg}_\varphi$, the arity of R in σ is k.

A Datalog$^\neg$ program P over schema σ is a finite set of rules over σ. Furthermore, we write $\mathit{sch}(P)$ to denote the minimal schema that P is over. We also define the internsional database schema $\mathit{idb}(P) \subseteq \mathit{sch}(P)$ as the database schema consisting of all relations that occur in the heads of the rules in P. We denote by $\mathit{edb}(P) = \mathit{sch}(P) \setminus \mathit{idb}(P)$ the extensional database schema. A Datalog$^\neg$ program P is called *positive* is all its rules are positive. It is said that P is *semi-positive* if for each rule $\varphi \in P$, the atoms in neg_φ are over $\mathit{edb}(P)$.

A *valuation* for a rule φ in P w.r.t. an instance I over $\mathit{edb}(P)$, is a total function $V : \mathit{Vars}(\varphi) \mapsto \mathbf{dom}$. The application of V to an atom $A = R(u_1, \ldots, u_k)$ in φ, denoted $V(A)$, results in the fact $R(a_1, \ldots, a_k)$ where $a_i = V(u_i)$ for each $i \in \{1, \ldots, k\}$. This is naturally extended to a set of atoms, which results in a set of facts. The valuation V is said to be satisfying for φ on I if $V(\mathit{pos}_\varphi) \subseteq I$ and $V(\mathit{neg}_\varphi) \cap I = \emptyset$. If so, φ is said to derive the fact $V(\mathit{head}_\varphi)$.

We can now give the semantics of a semi-positive Datalog$^\neg$ program P. Let T_P be the *immediate consequence operator* that maps each instance J

over $sch(P)$ to an instance $J' = J \cup A$ where A is the set of all facts derived by all possible satisfying valuations for the rules of P on J. Now consider an instance I over $edb(P)$ and the infinite sequence I_0, I_1, I_2, $\ldots$, where $I_0 = I$ and $I_i = T_P(I_{i-1})$. The output of P on instance I, denoted $P(I)$, is defined as $\bigcup_j I_j$.

Example 2.5. The following is an example of a positive Datalog$^\neg$ program which computes the transitive closure of an instance over schema $\sigma = \{E^{(2)}\}$.

$$\begin{aligned} &T(x,y) \leftarrow E(x,y).\\ &T(x,y) \leftarrow T(x,z),\ T(z,y). \end{aligned}$$

✦

Chapter 3

Coordination-Freeness

Back in 2010, Hellerstein conjectured a link between eventual consistent and coordination-free distributed computations, and expressibility in monotonic Datalog [13]. This conjecture, known as the CALM-conjecture, was not formalized however, as this requires clear definitions of the notions eventual consistency and coordination as well as a distributed computational model. Ameloot et al. [8] propose these definitions and a computational model based on relational transducers. Subsequently they formalize and proof the CALM-conjecture in their framework. In this chapter, we introduce the computational model of relational transducer networks. We then show how this model can be used to formalize the CALM-conjecture. Next, we try to find a somewhat more intuitive definition of coordination-freeness. To conclude, we take a look at a different semantics regarding message delivery for the relational transducer model.

3.1 Relational Transducers

The notion of relational transducers can be used to formalize the computation on a single node.

Definition 3.1. A *transducer schema* Υ is a tuple $\langle \Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}}, \Upsilon_{\text{sys}} \rangle$ of disjoint database schemas, representing 'input', 'output', 'message', 'memory' and 'system' respectively. We fix Υ_{sys} to always contain the two unary relations `Id` and `All`.

Definition 3.2. A *transducer state* for a transducer schema Υ is a database instance over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$.

Definition 3.3. A *relational transducer* Π over a transducer schema Υ is a collection of queries:

- $\mathcal{Q}^R_{\text{out}}$ with output schema $\{R^{(k)}\}$ for each $R^{(k)} \in \Upsilon_{\text{out}}$;

- $\mathcal{Q}^R_{\text{ins}}$ and $\mathcal{Q}^R_{\text{del}}$ both with output schema $\{R^{(k)}\}$ for each $R^{(k)} \in \Upsilon_{\text{mem}}$;
- $\mathcal{Q}^R_{\text{snd}}$ with output schema $\{R^{(k)}\}$ for each $R^{(k)} \in \Upsilon_{\text{msg}}$;

where each of these queries have the same input schema, namely $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$.

The queries that make up a relational transducer will form the mechanism to send messages and update the state of a node. Notice that we use $R^{(k)}$ to denote a relation symbol R of arity k. Henceforth, we often abbreviate 'relational transducer' as simply 'transducer'.

Next, we will describe how a transducer uses these queries to transition from one state to another. Let Π be a transducer over Υ. We call a database instance over Υ_{msg} a *message instance* for Υ. We can now define a *local transition* as a 4-tuple $(I, I_{\text{rcv}}, J, J_{\text{snd}})$, sometimes denoted as $I, I_{\text{rcv}} \to J, J_{\text{snd}}$, where I and J are transducer states, and I_{rcv} and J_{snd} are message instances such that, denoting $I' = I \cup I_{\text{rcv}}$:

$$\begin{aligned}
J|_{(\text{in,sys})} &= I|_{(\text{in,sys})};\\
J|_{(\text{out})} &= I|_{(\text{out})} \cup \bigcup_{R^{(k)} \in \Upsilon_{\text{out}}} \mathcal{Q}^R_{\text{out}}(I');\\
J|_{(\text{mem})} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{mem}}} (I|_R \cup R^+) \setminus R^-;\\
J_{\text{send}} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{msg}}} \mathcal{Q}^R_{\text{snd}}(I'),
\end{aligned}$$

with

$$\begin{aligned}
R^+ &= \mathcal{Q}^R_{\text{ins}}(I') \setminus \mathcal{Q}^R_{\text{del}}(I'); \text{ and}\\
R^- &= \mathcal{Q}^R_{\text{del}}(I') \setminus \mathcal{Q}^R_{\text{ins}}(I').
\end{aligned}$$

Intuitively, this boils down to the following. When a node receives the message facts contained in I_{rcv}, a local transition updates the current transducer state I to a new state J, while sending out the message facts contained in J_{snd}. The input and system relations remain unchanged during any local transition. The output queries produce potentially more output facts and each memory relation R is updated according to insertion and deletion queries. More specifically, all facts produced by $\mathcal{Q}^R_{\text{ins}}$ are added to R and all facts produced by $\mathcal{Q}^R_{\text{del}}$ are removed from R. There is no-op semantics when a fact is both added and removed at the same time. This means that the fact will not be added if it was not in the memory relations already, and that it will not be deleted if it was. Also notice that output facts are never removed. These local transitions are also deterministic: if $I, I_{\text{rcv}} \to J, J_{\text{snd}}$

and $I, I_{\text{rcv}} \rightarrow J', J'_{\text{snd}}$, then $J = J'$ and $J_{\text{snd}} = J'_{\text{snd}}$.

Remark 3.4. The transducer model can be parameterized by a query language $\mathcal{L}$ that indicates the language in which the queries that make up the transducer are specified. In that case we call Π an $\mathcal{L}$-transducer.

3.2 Relational Transducer Networks

We can now use the relational transducers from the previous section to formalize computing networks [8].

Definition 3.5. A *network* $\mathcal{N}$ is a finite, connected and undirected graph. We write *nodes*$(\mathcal{N})$ and *edges*$(\mathcal{N})$ to denote the nodes and edges of $\mathcal{N}$. We require that *nodes*$(\mathcal{N}) \subseteq \mathbf{dom}$. Moreover, for $x \in$ *nodes*$(\mathcal{N})$ we write *neighbor*$(x, \mathcal{N})$ to denote the set $\{y \mid (x, y) \in \textit{edges}(\mathcal{N})\}$.

Definition 3.6. A *transducer network* is a triple $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ where $\mathcal{N}$ is a network, Υ is a transducer schema and Π is a transducer over Υ.

A transducer network as defined above is called homogeneous, because copies of the same transducer Π are running on all nodes of the netwerk. We consider only homogeneous transducer networks.

We now formalize data distribution over a network.

Definition 3.7. A *distributed database instance* over a network $\mathcal{N}$ and a database schema σ is a total function H mapping each node of $\mathcal{N}$ to a regular database instance over σ.

Notice that by this definition, a data fact can be replicated over multiple nodes in the network.

3.2.1 Operational Semantics

We are now ready to define the operational semantics of transducer networks. Let $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ be a transducer network and let H be a distributed database instance over $\mathcal{N}$ and Υ_{in}. A *configuration of* $\mathcal{T}$ *on* H is a pair $\rho = (s, b)$ of functions where

- s is the *state function* that maps each $x \in$ *nodes*$(\mathcal{N})$ to a transducer state $J = s(x)$ so that $J|_{(\text{in})} = H(x)$ and $J|_{(\text{sys})} = \{\texttt{Id}(x)\} \cup \{\texttt{All}(y) \mid y \in \textit{nodes}(\mathcal{N})\}$; and
- b is the *buffer function* that maps each $x \in$ *nodes*$(\mathcal{N})$ to a finite multiset of facts over Υ_{msg}.

The state function initializes the input of each node based on the distributed database H. Additionally, it initializes the system relations `Id` and `All` to provide the transducer at each node with the identity of the node it is running on, as well as the identities of all other nodes in the network. The buffer function on the other hand, maps each node in the network to a multiset of message facts that were sent to this node but that have not been received yet.

The *start configuration* of $\mathcal{T}$ on H, denoted as $start(\mathcal{T}, H)$, is the configuration $\rho = (s, b)$ of $\mathcal{T}$ on H that defines $s(x)|_{(\text{out,mem})} = \emptyset$ and $b(x) = \emptyset$ for each $x \in nodes(\mathcal{N})$. In other words, at the start configuration we have no output or memory facts and there are no messages to receive.

Next, we define the actual computing mechanism of transducer networks. A *global transition* of $\mathcal{T}$ on H is a 4-tuple (ρ_1, x, m, ρ_2), sometimes denoted as $\rho_1 \xrightarrow{x,m} \rho_2$, where $x \in nodes(\mathcal{N})$, and $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$ are configurations of $\mathcal{T}$ on H so that

- m is a submultiset of $b_1(x)$ and there exists a message instance J_{snd} such that
$$s_1(x), set(m) \to s_2(x), J_{\text{snd}}$$
is a local transition of transducer Π;
- for each $y \in nodes(\mathcal{N}) \setminus \{x\}$ we have $s_1(y) = s_2(y)$;
- $b_2(x) = b_1(x) \setminus m$; for each $y \in neighbor(x, \mathcal{N})$ we have $b_2(y) = b_1(y) \cup J_{\text{snd}}$; and for all other nodes y we have $b_2(y) = b_1(y)$.

Here, we call x the active node and $set(m)$ the delivered messages. Intuitively, this means the following. The active node receives an arbitrary submultiset of messages from its buffer. The node then performs a local transition to update its output and memory relations, as well as send the new messages contained in J_{snd}. The states of the non-active nodes remain unchanged. The messages that were sent by the active node are added to the buffer of all neighboring nodes. A node does never send messages to itself. It is possible that $m = \emptyset$. In that case we call the global transition a *heartbeat transition*, otherwise we call it a *delivery transition*.

We can now define a *run* $\mathcal{R}$ of a transducer network $\mathcal{T}$ on a distributed database instance H as an infinite sequence of global transitions $\rho_i \xrightarrow{x_i,m_i} \rho_{i+1}$ for $i = 1, 2, 3, \ldots$, where $\rho_1 = start(\mathcal{T}, H)$. Notice that the changes made to the memory and output relations of the active node, are only visible to that node starting from the next global transition in which it is active.

Remark 3.8. The mechanism described above does not define concurrent global transitions, where more than one node is active at the same time.

These transitions can be simulated by using multiple sequential global transitions however, by letting each of the concurrent nodes become active in an arbitrary order. Because of the deterministic nature of local transitions, the nodes will behave exactly the same as during a concurrent transition.

3.2.2 Fairness

When talking about a distributed computation model, it is important to consider a notion of 'fairness' in order to guarantee the liveness of the system. This bascially means that eventually some progress has to be made and the process cannot get stuck. As for transducer networks, we call a run of $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ on some input *fair* if:

(1) every node is active in an infinite number of transitions; and

(2) if for some node a fact occurs in its message buffer in an infinite number of configurations, then this fact is delivered to that node during an infinite number of transitions.

Condition (1) means that every node becomes active once in a while. Condition (2) requires that no messages are infinitely delayed. We only consider fair runs.

3.3 Computing Queries

In this section, we describe how we can compute a query using a transducer network. For this purpose we study the notions of consistency and network-independence. But first, we will introduce terminology for transducers that will come in handy later.

- **Oblivious:** An *oblivious* transducer does not use the relations `Id` and `All` in its queries. This means that the transducer is unaware of the network context, as it does not know the node it is running on and it does not know about other nodes in the network.
- **Inflationary:** An *inflationary* transducer does never delete any facts from its memory relations.
- **Monotone:** A transducer is *monotone* if all its queries are monotone.

3.3.1 Input & Output

Next, we look at how to define the input and output of a transducer network. Let $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ be a transducer network and let I be a regular database instance over Υ_{in}. We can give I as an input to $\mathcal{T}$ by partitioning it across the network. Formally we can define a *horizontal partition* of I as a distributed database instance H over $\mathcal{N}$ for which $I = \bigcup_{x \in nodes(\mathcal{N})} H(x)$.

Now consider a configuration $\rho = (s, b)$ of $\mathcal{T}$ on input H. We can associate an output database instance $out(\rho)$ with this configuration which is defined as follows:

$$out(\rho) = \bigcup_{x \in nodes(\mathcal{N})} s(x)|_{(\text{out})}.$$

Now let $\mathcal{R}$ be a run of $\mathcal{T}$ on some input and denote the sequence of configurations of this run as $\rho_1, \rho_2, \ldots$. We call the number $i \geq 1$ such that $out(\rho_j) = out(\rho_i)$ for every $j > i$ a *quiescence point* for $\mathcal{R}$. Similarly, we call the corresponding configuration ρ_i a *quiescence configuration* of $\mathcal{R}$. Notice that only quiescence configurations can follow a quiescence configuration and that all quiescence configurations define the same output database.

Because we only consider finite input instances, only a finite number of distinct output facts are possible. Therefore, a run will surely reach a quiescence configuration at some point.

Property 3.9. *For every transducer network, on every input, every run contains a quiescence configuration.*

We can now define the *output* of a run $\mathcal{R}$ as $out(\rho_i)$ where ρ_i is a quiescence configuration of $\mathcal{R}$.

3.3.2 Consistency

Let's define what it means for a transducer network to be consistent.

Definition 3.10. A transducer network $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ is *consistent* if for all database instances I over Υ_{in} all fair runs of $\mathcal{T}$ have the same output, denoted $\mathcal{T}(I)$, on all horizontal partitions of I over $\mathcal{N}$.

We say that $\mathcal{T}$ *computes a query* $\mathcal{Q}$ over input schema Υ_{in} and output schema Υ_{out} if $\mathcal{T}$ is consistent and $\mathcal{T}(I) = \mathcal{Q}(I)$ for all instances I over Υ_{in} on which $\mathcal{Q}$ is defined.

The following example describes a consistent transducer network to compute the well-known transitive closure query.

Example 3.11. Consider the transducer schema with schema $\Upsilon_{\text{in}} = \{E^{(2)}\}$,

$\Upsilon_{\text{out}} = \{T^{(2)}\}$, $\Upsilon_{\text{msg}} = \{U^{(2)}\}$ and $\Upsilon_{\text{mem}} = \{M^{(2)}\}$, and the rules:

$$\begin{aligned}
U_{\text{snd}}(x, y) &\leftarrow E(x, y).\\
U_{\text{snd}}(x, y) &\leftarrow U(x, y).\\
\\
M_{\text{ins}}(x, y) &\leftarrow U(x, y).\\
\\
T_{\text{out}}(x, y) &\leftarrow E(x, y).\\
T_{\text{out}}(x, y) &\leftarrow M(x, y).\\
T_{\text{out}}(x, y) &\leftarrow T(x, z),\ T(z, y).
\end{aligned}$$

This transducer uses the U_{snd} rules to send its local input to all neighbors, as well as forward any incoming message facts. It also uses the M_{ins} rule to add all received messages to its memory relation M. By doing so, the complete input is accumulated at each node. Lastly, the transducer uses the T_{out} rules to compute the transitive closure based on its own local input and the accumulated memory facts. The consistency of this transducer follows from the monotonicity of the transitive closure query. This transducer is also oblivious, inflationary and monotone. ✦

Example 3.12. We also give an example of a transducer network that is not consistent. Consider the transducer schema with schema $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{U^{(1)}\}$ and $\Upsilon_{\text{mem}} = \emptyset$, and the rules:

$$\begin{aligned}
U_{\text{snd}}(x) &\leftarrow R(x).\\
\\
\texttt{block}() &\leftarrow T(x).\\
T_{\text{out}}(x) &\leftarrow \neg\texttt{block}(),\ U(x).
\end{aligned}$$

We claim that a tranducer network of at least two nodes running this transducer is not consistent. The transducer forwards its local input to its neighbors using the U_{snd} rule. The T_{out} rule subsequently outputs received messages, only if the output is still empty. If there are at least two distinct input facts in R, different runs may deliver these facts in different orders resulting in a different output. ✦

3.3.3 Network-Independence

When talking about queries, we don't just want the transducer network to be consistent. We also want that the query can be computed by the transducer regardless of the network. We therefore introduce the notion of network-independence.

Definition 3.13. Let Π be a transducer over a schema Υ. We say that Π is a *network-independent* transducer if for all networks $\mathcal{N}$, the transducer networks $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ are consistent and compute the same query $\mathcal{Q}$. We say that $\mathcal{Q}$ is *distributedly computed* by Π.

The transducer that we described in Example 3.11 is network-independent. In the following example however, we give a transducer that is not network-independent.

Example 3.14. Consider the transducer schema with schema $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{A^{(1)}, B^{(2)}\}$ and $\Upsilon_{\text{mem}} = \{S^{(2)}\}$, and the rules:

$$A_{\text{snd}}(x) \leftarrow \texttt{Id}(x).$$

$$\begin{aligned} B_{\text{snd}}(x,y) &\leftarrow A(x),\ \texttt{Id}(y).\\ B_{\text{snd}}(x,y) &\leftarrow B(x,y). \end{aligned}$$

$$S_{\text{ins}}(x,y) \leftarrow B(x,y).$$

$$\begin{aligned} \texttt{missing}() &\leftarrow \texttt{All}(x),\ \texttt{All}(y),\ x \neq y,\ \neg S(x,y).\\ T_{\text{out}}(x) &\leftarrow \neg\texttt{missing}(),\ R(x). \end{aligned}$$

Here, the transducer uses the A_{snd} rule to send its identifier to its neighbors. Doing so, the transducer can discover the edges of the network. These edge facts are flooded over the network using the B_{snd} rule, as well as stored in the memory relation S. Finally, the transducer uses the T_{out} rule to output its local input on the condition that the edges contained in the memory relation S form a complete graph. Consequently, this transducer computes the indentity query on networks that form a complete graph, and the empty query on other networks. This transducer is also non-oblivious, as it relies heavily on the relations `Id` and `All` in order to detect the network topology. ✦

3.3.4 Expressiveness

In this section, we provide some insight into why the transducer model has enough expressive power to study the distributed evaluation of queries. To do this, we present two lemmas that show that each node in a transducer network is always able to gather a local copy of all input facts on the network.

Lemma 3.15. *Let σ be a database schema. There exists a transducer schema Υ with $\Upsilon_{in} = \sigma$ and an oblivious, inflationary and monotone UCQ-transducer Π over Υ so that for every transducer network for Π, for every instance I of σ, on every horizontal partition of I, every fair run reaches a configuration where every node has a local copy of the entire instance I in its memory.*

Proof. The proof is by construction. The idea is as follows. Every node sends its local input facts to its neighbors and will forward any messages it receive. Furthermore, the node collects its local input facts and received messages in memory relations. By fairness, every node will eventually have gathered the complete input database instance in local memory.

For completeness, we provide a transducer that implements this idea. For this, we need the following transducer schema: $\Upsilon_{\text{in}} = \sigma$, $\Upsilon_{\text{msg}} = \{R^{\text{msg},(k)} \mid R^{(k)} \in \sigma\}$ and $\Upsilon_{\text{mem}} = \{R^{\text{mem},(k)} \mid R^{(k)} \in \sigma\}$. We don't include an output schema since that is not part of this construction. The following transducer rules formalize the mechanism described in this proof.

$$\begin{aligned} R^{\text{msg}}_{\text{snd}}(\bar{x}) &\leftarrow R(\bar{x}).\\ R^{\text{msg}}_{\text{snd}}(\bar{x}) &\leftarrow R^{\text{msg}}(\bar{x}). \end{aligned}$$

$$\begin{aligned} R^{\text{mem}}_{\text{ins}}(\bar{x}) &\leftarrow R(\bar{x}).\\ R^{\text{mem}}_{\text{ins}}(\bar{x}) &\leftarrow R^{\text{msg}}(\bar{x}). \end{aligned}$$

□

Note that we have already used the idea outlined above in Example 3.11.

Lemma 3.16. *Let σ be a database schema. There exists a transducer schema Υ with $\Upsilon_{in} = \sigma$ and an UCQ$^\neg$-transducer Π over Υ so that for every transducer network for Π, for every instance I of σ, on every horizontal partition of I, every fair run reaches a configuration where every node has a local copy of the entire instance I in its memory, and a 'ready'-flag is set to true. Moreover, this flag only becomes true after the node has the entire instance I in its memory.*

Proof. Again, the proof is by construction. The idea is as follows. Each node x sends its local input facts over relation $R^{(k)} \in \sigma$ to every other node, with an additional last component containing its node identifier which we call the *tag*. When a node y receives a tagged input fact, it removes the tag and stores the fact in its memory relation. For each fact that y receives from x, y also sends an acknowledgement back to x. Node x subsequently checks if node y has acknowledged all input facts of x. If so, then x sends out the fact $\texttt{done}(x, y)$. If y has received $\texttt{done}(x, y)$ for all other nodes x, the it knows it has collected all input data on the network and created the 'ready'-flag.

Next, we specify the transducer schema needed for this construction: $\Upsilon_{\text{in}} = \sigma$, $\Upsilon_{\text{msg}} = \{R^{\text{msg},(k+1)}, R^{\text{ack},(k+2)} \mid R^{(k)} \in \sigma\} \cup \{\texttt{done}^{(2)}\}$ and $\Upsilon_{\text{mem}} = \{R^{\text{mem},(k)}, R^{\text{ackMem},(k+2)} \mid R^{(k)} \in \sigma\} \cup \{\texttt{doneMem}^{(2)}, \texttt{notDone}^{(1)}, \texttt{missing}^{(0)}\}$ $\cup\{\texttt{started}^{(0)}, \texttt{ready}^{(0)}\}$. We don't include an output schema since that is not part of this construction.

Finally, we give the necessary transducer rules to implement the idea above. First, for each $R^{(k)} \in \sigma$, we have the following rules to let all nodes send their tagged input facts to all other nodes, store them in their memory relations and reply with the necessary acknowledgements:

$$R^{\text{msg}}_{\text{snd}}(\bar{u}, x) \leftarrow R(\bar{u}), \texttt{Id}(x).$$
$$R^{\text{msg}}_{\text{snd}}(\bar{u}, x) \leftarrow R^{\text{msg}}(\bar{u}, x).$$

$$R^{\text{mem}}_{\text{ins}}(\bar{u}) \leftarrow R(\bar{u}).$$
$$R^{\text{mem}}_{\text{ins}}(\bar{u}) \leftarrow R^{\text{msg}}(\bar{u}, x).$$

$$R^{\text{ack}}_{\text{snd}}(\bar{u}, x, y) \leftarrow R^{\text{msg}}(\bar{u}, x), \texttt{Id}(y).$$
$$R^{\text{ack}}_{\text{snd}}(\bar{u}, x, y) \leftarrow R^{\text{ack}}(\bar{u}, x, y).$$

$$R^{\text{ackMem}}_{\text{ins}}(\bar{u}, x, y) \leftarrow R^{\text{ack}}(\bar{u}, x, y), \texttt{Id}(x).$$

Next, we use the `started` relation to remember if the first transition already happened:

$$\texttt{started}_{\text{ins}}() \leftarrow .$$

Then, on each node x, we compute the relation `notDone` which contains all nodes that have not yet acknowledged all input facts of x. For each $R^{(k)} \in \sigma$, we have a rule of the form:

$$\texttt{notDone}_{\text{ins}}(y) \leftarrow R(\bar{u}), \texttt{Id}(x), \texttt{All}(y), \neg\texttt{Id}(y), \neg R^{\text{ackMem}}(\bar{u}, x, y).$$

Also, since this relation is recomputed during every transition, we need to make sure to delete all facts in it:

$$\texttt{notDone}_{\text{del}}(y) \leftarrow \texttt{notDone}(y).$$

When a node x notices that another node y has acknowledged all its input facts, it sends out $\texttt{done}(x, y)$:

$$\texttt{done}_{\text{snd}}(x, y) \leftarrow \texttt{started}(), \texttt{Id}(x), \texttt{All}(y), \neg\texttt{Id}(y), \neg\texttt{notDone}(y).$$
$$\texttt{done}_{\text{snd}}(x, y) \leftarrow \texttt{done}(x, y).$$

These done messages must be stored at the addressed node:

$$\texttt{doneMem}_{\text{ins}}(x, y) \leftarrow \texttt{done}(x, y), \texttt{Id}(y).$$

Finally, when a node y has received done from all other nodes, the ready flag can be set:

$$
\begin{aligned}
\texttt{missing}_{\text{ins}}() &\leftarrow \texttt{Id}(y), \texttt{All}(z), \neg\texttt{Id}(z), \neg\texttt{doneMem}(z, y).\\
\texttt{missing}_{\text{del}}() &\leftarrow \texttt{missing}().\\
\texttt{ready}_{\text{ins}}() &\leftarrow \texttt{started}(), \neg\texttt{missing}().
\end{aligned}
$$

□

We can now make the following conclusion.

Theorem 3.17 ([8]). *Let $\mathcal{L}$ be a query language containing $UCQ^{\neg}$. Every query expressible in $\mathcal{L}$ can be distributedly computed by an $\mathcal{L}$-transducer.*

Proof. Let $\mathcal{Q}$ be a query expressible in $\mathcal{L}$ over input schema σ and output schema σ'. We specify an $\mathcal{L}$-transducer that is able to compute $\mathcal{Q}$. We know from Lemma 3.16 that we can construct a transducer that collects the complete input at every node using the $UCQ^{\neg}$ query language. We can construct this transducer over input schema σ, but it does not produce any output. We can extend this transducer by defining the output schema $\Upsilon_{\text{out}} = \sigma'$, and by adding output queries that compute $\mathcal{Q}$ once the 'ready'-flag is set to true, based on the collected input facts. □

Moreover, since monotone queries are important in the context of the CALM-conjecture, which we will describe in more detail in the next section, we can observe that we only need oblivious transducers to compute them.

Theorem 3.18 ([8]). *Let $\mathcal{L}$ be a query language containing UCQ. Every monotone query expressible in $\mathcal{L}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer.*

Proof. Let $\mathcal{Q}$ be a monotone query expressible in $\mathcal{L}$ over input schema σ and output schema σ'. We can use the same strategy as in the proof of Theorem 3.17, but instead of using the construction from Lemma 3.16 we use Lemma 3.15 to construct an oblivious transducer that gradually collects the entire input at each node. The difference here is that, by monotonicity of $\mathcal{Q}$, we don't need to wait until the entire input is available before outputting results. We can output increasingly more facts when more input becomes available, without ever outputting incorrect facts. □

3.4 The CALM-conjecture

In 2010, Hellerstein conjectured the following link between eventual consistent and coordination-free distributed computations, and expressibility in monotonic Datalog.

Conjecture 3.19 (CALM-conjecture [13])**.** *A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in monotonic Datalog.*

In the previous sections we have seen how we can model distributed execution strategies. It remains to be shown what it means for an execution strategy to be coordination-free, as this seems to be a key part of the CALM-conjecture. We have already seen an example of an execution strategy that does require coordination in Lemma 3.16, where the nodes need to coordinate to make sure every node has all necessary data. This coordination often leads to a situation where nodes are simply waiting for each other before continuing with the actual computation, which we call a global synchronization barrier. It may be clear that these barriers are a major source of inefficiency in distributed z programs.

One possible definition of coordination-freeness would be to not allow any communication at all. However, it seems like this approach would be too drastic. Ameloot et al. [8] came up with an interesting formal definition of coordination-freeness which does not prohibit communication as a whole, but only requires the computation to be successful without communication on specific horizontal partitions.

Definition 3.20. (Coordination-freeness) Let Π be transducer over schema Υ and $\mathcal{T}$ be a transducer network for Π. We say that $\mathcal{T}$ is *coordination-free* if for every database instance I over Υ_{in}, there exists a horizontal partition H of I and a run of $\mathcal{T}$ on H in which a quiescence configuration is reached by only doing heartbeat transitions. We call transducer Π coordination-free if for every network its corresponding transducer network is coordination-free. Furthermore, a query $\mathcal{Q}$ can be distributedly computed in a coordination-free manner if there exists a network-independent, coordination-free transducer Π that computes $\mathcal{Q}$.

This basically means that when the data is distributed in the right way, no communication is needed to compute the query. An example of a coordination-free transducer was already given in Example 3.11, where we showed a transducer to compute the transitive closure. When every node is given the full input, this transducer can correctly compute the transitive closure with only heartbeat transitions.

Coordination-freeness is undecidable for FO-transducers. However, we can define a syntactic class of transducers that is always coordination-free.

Proposition 3.21. *Let $\mathcal{L}$ be a query language. Every network-independent, oblivious $\mathcal{L}$-transducer is coordination-free.*

Proof. Let Π be a network-independent, oblivious transducer over schema Υ that distributedly computes a query $\mathcal{Q}$. First, consider a single-node network where this single node is always given the full input. In this case there can only be heartbeat transitions. It then follows that for every input instance I over Υ_{in}, a quiescence configuration containing $\mathcal{Q}(I)$ is always reached by only doing heartbeat transitions. Next, consider an arbitrary network $\mathcal{N}$, any instance I over Υ_{in}, and the horizontal partition which places the full input at every node. Because Π is oblivious, a node cannot detect that it is on a multi-node network unless it receives a message. Therefore, every node will initially act the same as on a single-node network when performing only heartbeat transitions and will already output $\mathcal{Q}(I)$. Moreover, because Π is network-independent, the nodes will never output more than $\mathcal{Q}(I)$ when they receive messages later on. $\square$

Using the models and definitions described above, Ameloot et al. [8] subsequently reformulated Hellerstein's conjecture in a formal way.

Conjecture 3.22. *A query can be distributedly computed by a coordination-free transducer if and only if it is expressible in Datalog.*

It is clear that the if-side of this formulation holds. Namely, every query in Datalog is monotone. It then follows from Theorem 3.18 that there exists an oblivious transducer which computes this query. Moreover, according to Proposition 3.21, this transducer is coordination-free.

The only-if side on the other hand may does not hold in the strict sense, mainly because there exist monotone queries that are not expressible in Datalog. However, it is clear that the key aspect of the conjecture is the monotonicity, as is confirmed by the following theorem.

Theorem 3.23 ([8]). *Let $\mathcal{L}$ be a query language. Every query that is distributedly computed by a coordination-free $\mathcal{L}$-transducer is monotone.*

Proof. Let Π be a transducer over schema Υ that is coordination-free. Let $\mathcal{Q}$ be the query that is distributedly computed by Π. Let I and J be two database instances over Υ_{in} with $I \subseteq J$. We show that $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. Consider an arbitrary transducer network $\mathcal{T}$ with at least two nodes, and a fact $f \in \mathcal{Q}(I)$. By definition of coordination-freeness, there exists a horizontal partition H of I such that $\mathcal{T}$ has a run $\mathcal{R}$ in which $\mathcal{Q}(I)$ is already computed in a prefix consisting of only heartbeat transitions. Let x denote the node that output f during run $\mathcal{R}$ and let y denote a node different from x. Now consider the horizontal partition H' with $H'(x) = H(x)$ and $H'(y) = H(y) \cup (J \setminus I)$. Let n be the number of heartbeat transitions that x went through before outputting f. Consider a prefix of a run of $\mathcal{T}$ on H' where we initially do n heartbeat transitions on node x. Because of the deterministic nature of local transitions, node x will go through the same states as in run $\mathcal{R}$, and therefore output f again. We can then extend this prefix to a full fair run of $\mathcal{T}$ of H'. We know that f will be output on any partition of J and during every fair run because $\mathcal{T}$ is consistent. Moreover, since Π is network-independent, this is true for every transducer network $\mathcal{T}$ for Π so we can conclude that $f \in \mathcal{Q}(J)$. □

We can summarize the CALM-conjecture in the following equivalence theorem.

Theorem 3.24. *Let $\mathcal{L}$ be a query language containing UCQ. For every query $\mathcal{Q}$ expressible in $\mathcal{L}$, the following are equivalent:*

1. *$\mathcal{Q}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer;*
2. *$\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer;*
3. *$\mathcal{Q}$ is monotone.*

Proof. (3) $\Longrightarrow$ (2) follows from Theorem 3.18. (2) $\Longrightarrow$ (1) follows from Proposition 3.21. (1) $\Longrightarrow$ (3) follows from Theorem 3.23. □

3.5 Coordination-freeness: Alternative Definition

In this section, we take another look at the definition of coordination-freeness. In some cases, it is hard to reason with the definition that was proposed by Ameloot et al, which states that there should exist *a* horizontal partition, without specifying what this partition should looks like. Therefore, it may be interesting to investigate an alternative, maybe more intuitive, definition that mentions a specific way of partitioning the data.

Definition 3.25. (Coordination-freeness, alternative) A query $\mathcal{Q}$ can be distributedly computed in a coordination-free manner if there exists a network-independent, oblivious[1] transducer Π over a schema Υ so that a single-node network $\mathcal{T}$ for Π always correctly computes $\mathcal{Q}$ when it receives the full input. In other words, $\mathcal{T}(I) = \mathcal{Q}(I)$ for every database instance I over Υ_{in} on which $\mathcal{Q}$ is defined.

We claim that this definition is equivalent to the definition that Ameloot et al. used, which we provided in the previous section.

Proposition 3.26. *Definition 3.25 is equivalent to Definition 3.20.*

Proof. First, we show that if a query can be distributedly computed in a coordination-free manner according to Definition 3.20, it can also be distributedly computed in a coordination-free manner according to Definition 3.25. Assume a query $\mathcal{Q}$ that can be distributedly computed in a

[1]Otherwise a transducer could detect whether or not it is in a multi-node network. Consequently, it could compute the query perfectly fine on a single-node network, but require that it receives a message before outputting on a multi-node network. Therefore, it wouldn't guarantee coordination-freeness.

coordination-free manner according to Definition 3.20. Therefore, there exists a network-independent, coordination-free transducer Π that computes $\mathcal{Q}$. This means that for every network $\mathcal{N}$ the corresponding transducer network $\mathcal{T}$ for Π is coordination-free. Let's denote the single-node transducer network for Π as $\mathcal{T}_1$. Since all transducer networks for Π are coordination-free, this one is as well. Therefore, as stated in Definition 3.20, there exists a horizontal partition H of I and a run of $\mathcal{T}_1$ on H in which a quiescence configuration is reached by only performing heartbeat transitions. The output at this quiescence configuration, $\mathcal{T}_1(I)$, is exactly $\mathcal{Q}(I)$ for every input database I, seeing as Π computes $\mathcal{Q}$ and Π is network-independent. We note that H is the horizontal partition that places the full input at the only node in the network, as this is the only possible horizontal partition on a single-node network. We can thus conclude that Π satisfies the condition stated in Definition 3.25, and therefore that $\mathcal{Q}$ can be distributedly computed in a coordination-free manner according to Definition 3.25.

Next, we show that if a query can be distributedly computed in a coordination-free manner according to Definition 3.25, it can also be distributedly computed in a coordination-free manner according to Definition 3.20. Assume a query $\mathcal{Q}$ that can be distributedly computed in a coordination-free manner according to Definition 3.25. Therefore, there exists a network-independent, oblivious transducer Π over a schema Υ so that a single-node network $\mathcal{T}$ for Π always correctly computes $\mathcal{Q}$ when it receives the full input. Let $\mathcal{N}$ be an arbitrary network and $\mathcal{T}$ the corresponding transducer network for Π. We show that for every database instance I, there exists a horizontal partition H of I and a run of $\mathcal{T}$ on H in which a quiescence configuration is reached by only performing heartbeat transitions. We claim that the horizontal partition H that places the full input at every node satisfies this condition. Indeed, according to Definition 3.25, Π always correctly computes $\mathcal{Q}(I)$ on a single-node network. Therefore, a quiescence configuration containing $\mathcal{Q}(I)$ is always reached by doing only heartbeat transitions. Now consider the network $\mathcal{N}$ and the horizontal partition H. Since Π is oblivious it cannot detect that it is on a multi-node network, unless it receives a message. So, by only doing heartbeat transitions initially, it will act the same as on the single-node network and output the entire $\mathcal{Q}(I)$. Also, because Π is network-independent, it cannot output more than $\mathcal{Q}(I)$. We conclude that $\mathcal{T}$ is coordination-free, and therefore that $\mathcal{Q}$ can be distributedly computed in a coordination-free manner according to Definition 3.20. □

3.6 In-order Message Delivery

A possible variation of the transducer networks model involves the order in which messages are received. The earlier described model considers a reliable network in the sense that messages never get lost, but their order is not preserved. This is in contrast with, for example, the reliability guarantees of the popular TCP protocol. Therefore, in this section we study what the effect is of adding ordering in the aforementioned model.

Recall the original operational semantics from Section 3.2.1 and more specifically the *buffer function* b that maps each $x \in \mathit{nodes}(\mathcal{N})$ to a finite multiset of facts over Υ_{msg}. In order to accommodate the in-order message delivery, we need to modify the buffer function. A multiset of facts won't be sufficient to define the in-order delivery semantics, so we say b maps each $x \in \mathit{nodes}(\mathcal{N})$ to a *queue* of facts over Υ_{msg}. This way we require that the fact that was added to the buffer first, is also read from the buffer first.

Aside from this change, we also need to modify the notion of a global transition. Under the in-order delivery semantics, m can no longer be just an arbitrary submultiset of the message buffer $b_1(x)$. We now define m to be a multiset of facts constructed by popping zero or more facts from the queue $b_1(x)$.

We now show that these changes preserve coordination-freeness.

Proposition 3.27. *Every transducer that distributedly computes a query $\mathcal{Q}$ in a coordination-free manner under the original operational semantics, also distributedly computes $\mathcal{Q}$ in a coordination-free manner under the in-order delivery semantics.*

Proof. Let Π be a transducer that distributedly computes a query $\mathcal{Q}$ in a coordination-free manner under the original operational semantics. By definition of coordination-freeness, Π is coordination-free for every network $\mathcal{N}$ and its corresponding transducer network $\mathcal{T}$. Let $\mathcal{T}$ be an arbitrary transducer network for Π. Since $\mathcal{T}$ is coordination-free, there exists a horizontal partition H of I, for every database instance I, and a run of $\mathcal{T}$ on H in which a quiescence configuration is reached by only performing heartbeat transitions. Since no messages are received during heartbeat transitions ($m = \emptyset$), m is trivially a valid multiset of facts to read under the in-order delivery semantics. The same horizontal partition H and run of $\mathcal{T}$ on H are therefore evidence of the coordination-freeness of $\mathcal{T}$ under the in-order delivery semantics. Moreover, since every fair run of $\mathcal{T}$ under the in-order semantics is also a fair run under the original semantics, it follows that $\mathcal{T}$ is also consistent under the in-order semantics. Therefore we know that $\mathcal{Q}$ can also be distributedly computed by Π in a coordination-free manner when messages are received in the order they were sent in. $\square$

The converse of Proposition 3.27 is not necessarily true, as the following example demonstrates.

Example 3.28. We give a transducer that is coordination-free under the in-order delivery semantics, but is not consistent under the original operational semantics. The transducer has a single binary input relation E, and a ternary output relation T. During its first transition, every node outputs all triangles that can be found in its local facts. That is, it outputs all facts $T(x, y, z)$ such that $E(x, y)$, $E(y, z)$ and $E(z, x)$, with $x \neq y$, $y \neq z$, $z \neq x$, are in its local instance. Then it broadcasts its local facts to all other nodes, followed by a `done`-message. Whenever a node received such a `done`-message, it recomputes the output as it did in the first transition on its local instance as well as the facts it received from other nodes.

Intuitively, it is clear that this transducer correctly computes the (monotone) triangle query $\mathcal{Q}_{\triangle}(I)$ under the in-order delivery semantics. Indeed, the transducer will never read a `done`-message before it has read all the facts that were sent by the node that sent this `done`-message (because of the in-order delivery). As a result, there is always a `done`-message left in the buffer after a node has read all data facts sent by other node. Reading this `done`-message will trigger a recompute of the query on the complete input I, resulting in the complete output $\mathcal{Q}_{\triangle}(I)$. It is also coordination-free, because each node can already compute the full output in the first (heartbeat) transition when the full input is placed at every node.

Under the original operational semantics however, this transducer may not always compute the same output. It is possible that the nodes read all `done`-messages before the actual input facts sent by the other nodes. This may lead to a situation where the last recompute of the output happens before all input facts are available to a node, resulting in an incomplete output.

For completeness, we specify a transducer Π that implements this idea. The transducer schema Υ is as follows: $\Upsilon_{\text{in}} = \{E^{(2)}\}$, $\Upsilon_{\text{out}} = \{T^{(3)}\}$, $\Upsilon_{\text{msg}} = \{U^{(2)}, \texttt{done}^{(0)}\}$, $\Upsilon_{\text{mem}} = \{S^{(2)}, \texttt{started}^{(0)}\}$. The rules for the transducer are:

$U_{\text{snd}}(x, y) \leftarrow E(x, y).$
$U_{\text{snd}}(x, y) \leftarrow U(x, y).$

$S_{\text{ins}}(x, y) \leftarrow E(x, y).$
$S_{\text{ins}}(x, y) \leftarrow U(x, y).$
$\texttt{started}_{\text{ins}}() \leftarrow .$

$T_{\text{out}}(x, y, z) \leftarrow \neg\texttt{started}(), S(x, y), S(y, z), S(z, x), x \neq y, y \neq z, z \neq x.$
$T_{\text{out}}(x, y, z) \leftarrow \texttt{done}(), S(x, y), S(y, z), S(z, x), x \neq y, y \neq z, z \neq x.$

✦

Although the example above shows that not every transducer that is coordination-free under the in-order delivery semantics is coordination-free under the original semantics, we can show that the expressiveness under both semantics are the same. More specifically, we show that every coordination-free transducer under the in-order delivery semantics has an equivalent transducer that is coordination-free under the original semantics. In order to proof this statement, we first notice the following.

Proposition 3.29. *Every query that is distributedly computable by a coordination-free transducer under the in-order delivery semantics is monotone.*

Proof. The proof this proposition follows the exact same reasoning as the proof of Theorem 3.23. □

We can now state the following.

Proposition 3.30. *Every network-independent, coordination-free transducer under the in-order delivery semantics has an equivalent network-independent transducer that is coordination-free under the original semantics.*

Proof. Let Π be a transducer that is coordination-free under the in-order delivery semantics. Let $\mathcal{Q}$ be the query that is distributedly computed by Π. According to Proposition 3.29, we know that $\mathcal{Q}$ is monotone. We also know that every monotone query can be distributedly computed by a coordination-free transducer (under the original semantics), following Theorem 3.24. Consequently, there exists a transducer Π' that is coordination-free under the original semantics and distributedly computes $\mathcal{Q}$, making it equivalent to Π. □

3.6.1 In-order Semantics and Obliviousness

Proposition 3.31. *Every network-independent, oblivious transducer under the in-order semantics is coordination-free.*

Proof. This proposition follows from the proof of Proposition 3.21, since the same reasoning can be used here. □

As is the case under original semantics, the argument given in the proof of Proposition 3.31 is still valid when the transducer reads on the system relation `Id` since it can still not determine that it's on a multi-node network. Therefore, every network-independent transducer that reads only `Id` is still coordination-free under the in-order semantics.

Chapter 4

Optimal Broadcasting Strategies

The coordination-free evaluation strategies that we presented in the previous chapter aren't all that useful in practice. They mostly rely on all nodes broadcasting their local fragment of the data in order to collect the full dataset at every node. This is not desirable in a real world application. In this chapter, we take a look at a more economical and hence more practical broadcasting strategy [14]. First of all, we formally define broadcasting by means of broadcasting functions. Next, we introduce the notion of broadcast dependency sets which will finally lead us to an algorithm allowing to create broadcasting functions that satisfy an optimality property, which we will define later. In particular, we focus on broadcasting strategies for the evaluation of full conjunctive queries without self-joins as described in Section 2.2.

4.1 Oblivious Broadcasting Functions

In an execution strategy based on broadcasting, a relational transducer only has to do two things:

1. Determine which facts to send to the other nodes in the network;
2. Compute potentially new output facts whenever new data arrives.

This strategy works because we only consider conjunctive queries, which are monotone. We can thus simply recompute the query when a message arrives from another node. Consequently, this strategy is also coordination-free. The only changing factor in this strategy is the set of facts which should be broadcast. We'll formalize this aspect by means of oblivious broadcasting functions.

Definition 4.1. An *oblivious broadcasting function (OBF) f* is a generic mapping that maps instances to instances such that $f(J) \subseteq J$ for all instances J.

An OBF specifies which of a node's local input facts should be broadcast. We therefore call $f(J)$ the broadcast facts. The remaining set of facts $J \setminus f(J)$ are called the static facts. We use the term 'oblivious' in the sense that the set of broadcast facts $f(J)$ is not dependent on what other nodes on the network are doing. This means that $f(J)$ doesn't change when messages arrive.

We can model the output of a broadcasting algorithm based on an OBF f in the following way. Let $\mathcal{Q}$ be a conjunctive query, I a database instance, H a distribution of I and $\mathcal{N}$ a network. We can denote by $B(f, H) = \bigcup_{c \in \mathcal{N}} f(H(c))$ the set of all broadcast facts. We can then define the output of the algorithm $eval(\mathcal{Q}, f, H) = \bigcup_{c \in \mathcal{N}} \mathcal{Q}(H(c) \cup B(f, H))$ as the union of result of $\mathcal{Q}$ on the local instance extended by the broadcast facts over all nodes.

Definition 4.2. An OBF f is *correct* for a conjunctive query $\mathcal{Q}$ when $\mathcal{Q}(I) = eval(\mathcal{Q}, f, H)$ for every instance I and all distributions H of I. When f is correct for $\mathcal{Q}$, we also say that f is an OBF *for* $\mathcal{Q}$.

In order to characterize this correctness, we need a way talk about facts that can possibly contribute to a satisfying valuation together. We say that two facts $\mathbf{f}$ and $\mathbf{g}$ are *compatible w.r.t.* $\mathcal{Q}$, denoted as $\mathbf{f} \sim_{\mathcal{Q}} \mathbf{g}$, if they can be assigned to two atom from the body of $\mathcal{Q}$ under one valuation. In other words, there exists a valuation V for $\mathcal{Q}$ and atom $A, B \in body_{\mathcal{Q}}$ so that $V(A) = \mathbf{f}$ and $V(B) = \mathbf{g}$. We can characterize the correctness of an OBF in the sense that two compatible facts located at different nodes can never both be static. The reason for this is that if they are, the valuation V that witnesses their compatibility will not always be considered at any node. This would mean that the fact $V(head_{\mathcal{Q}})$ would never be derived. A formal characterization is given in [14].

Example 4.3. Consider the query $Q(x, y, z) \leftarrow A(x, y), B(y, x), C(x, z)$ and the database instance $I = \{A(1, 2), B(2, 1), B(2, 2), C(1, 3)\}$. The facts $A(1, 2)$ and $B(2, 1)$ are compatible w.r.t. $\mathcal{Q}$ because of the possible valuation $V : \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$. The same goes for facts $A(1, 2)$ and $C(1, 3)$. The facts $A(1, 2)$ and $B(2, 2)$ are an example of two incompatible facts. ✦

Aside from being correct, we would also like that an OBF transmits as little data as possible. For two OBFs f and g, we say that f is included in g, denoted $f \subseteq g$, if and only if $g(J) \subseteq g(J)$ for every instance J. We can use this to define local optimality.

Definition 4.4. An OBF f for a conjunctive query $\mathcal{Q}$ is *locally optimal* if and only if for every other OBF g for $\mathcal{Q}$, $g \subseteq f$ implies $f = g$.

In other words, this means that there exists no subdivision of f which broadcasts only a strict subset of the facts broadcast by f. Again, we can characterize local optimality in terms of compatible facts. We can say that f is locally optimal is for every instance I and fact $\mathbf{f}$ for which $\mathbf{f} \in f(I \cup \{\mathbf{f}\})$, there exists an instance J and a fact $\mathbf{g}$ such that $\mathbf{f} \sim_{\mathcal{Q}} \mathbf{g}$, $\mathbf{g} \notin I$, $\mathbf{f} \notin J$ and $\mathbf{g} \notin f(J \cup \{\mathbf{g}\})$. This holds because if there is no compatible fact $\mathbf{g}$ that is not broadcast, there is no reason for us to broadcast $\mathbf{f}$ and thus a more optimal OBF can be obtained.

4.2 Broadcast Dependency Set

Next, we introduce the concept of broadcast dependency sets (BDS) as a syntactical alternative to specify the more semantical OBFs. A BDS is a set of key-value pairs where the key is an equality type and the value is a list of dependencies for this type. Before we can formally define a BDS however, we need to define what these equality types are, as they are an important aspect of the BDS formalism.

Let $\mathcal{Q}$ be the conjunctive query $A_0 \leftarrow A_1, \ldots, A_n$. Since we only consider full conjunctive queries without self joins, we know that each body atom is uniquely identified by its predicate $pred(A_i)$. For a predicate R, denote by $atom(R)$ the unique atom $A \in body_{\mathcal{Q}}$ for which $pred(A) = R$.

Definition 4.5. For a finite set of variables X, a *partial equality type over X* is a pair of binary relations $\varphi = (E_\varphi, I_\varphi)$. We require that $E_\varphi \cup I_\varphi \subseteq X \times X$, E_φ is an equivalence relation, and I_φ is irreflexive and symmetric. When $E_\varphi \cup I_\varphi = X \times X$, we call φ a *complete equality type.*

Intuitively, an equality type represents equalities and inequalities between the variables in X. Furthermore we can also view each equality type as a formula:

$$\varphi = \bigwedge\{x = y \mid (x, y) \in E_\varphi\} \wedge \bigwedge\{x \neq y \mid (x, y) \in I_\varphi\}.$$

We can now link these equality types to the atoms in our query.

Definition 4.6. A *partial atomic type* over $\mathcal{Q}$ is a pair $\tau = (R_\tau, \varphi_\tau)$ where R_τ is a predicate and φ_τ is a partial equality type over $Vars(atom(R_\tau))$. We say τ is a *complete atomic type* when φ_τ is complete.

We provide a shorthand notation $Vars(\tau) = Vars(atom(R_\tau))$ for the variables over which τ is defined, and $atom(\tau) = atom(R_\tau)$ for the atom over which τ is defined. For readability, we will always denote a partial atomic type with τ and a complete atomic type with ω. Finally, we denote by $PTypes(\mathcal{Q})$ and $Types(\mathcal{Q})$ respectively the set of all partial atomic types and all complete atomic types over $\mathcal{Q}$.

Example 4.7. Consider the query $Q(x, y, z) \leftarrow A(x, x), B(x, y, z)$ and the set of variables $X = \{x, y, z\}$ and the following examples of equality types over X:

$$\varphi_1 = (\begin{array}{|c|c|}\hline \mathrm{x} & \mathrm{x} \\ \mathrm{y} & \mathrm{y} \\ \mathrm{z} & \mathrm{z} \\ \mathrm{x} & \mathrm{z} \\ \mathrm{z} & \mathrm{x} \\ \hline \end{array}, \begin{array}{|c|c|}\hline \mathrm{x} & \mathrm{y} \\ \mathrm{y} & \mathrm{x} \\ \mathrm{y} & \mathrm{z} \\ \mathrm{z} & \mathrm{y} \\ \hline \end{array}), \quad \varphi_2 = (\begin{array}{|c|c|}\hline \mathrm{x} & \mathrm{x} \\ \mathrm{z} & \mathrm{z} \\ \mathrm{x} & \mathrm{z} \\ \mathrm{z} & \mathrm{x} \\ \hline \end{array}, \emptyset).$$

Or, in their alternative form:

$$\varphi_1 := x \neq y \wedge y \neq z \wedge x = z$$

$$\varphi_2 := x = z$$

We can use these equality types to define the complete atomic type $\omega = (B, \varphi_1)$ and the partial atomic type $\tau = (B, \varphi_2)$. ✦

We say that a fact $\mathbf{f}$ is of type τ or satisfies τ, denoted $\mathbf{f} \models \tau$, when there is a valuation $V_{\mathbf{f}}$ from the variables in $atom(\tau)$ onto $Adom(\mathbf{f})$ such that $V_{\mathbf{f}}(atom(\tau)) = \mathbf{f}$ and the formula associated with the partial type evaluates to true when each variable x_i is substituted by $V_{\mathbf{f}}(x_i)$. Notice that this valuation is unique because we only consider full conjunctive queries. By $type(\mathbf{f})$ we denote the unique atomic type satisfied by $\mathbf{f}$ when it exists. However, $type(\mathbf{f})$ is not always defined since atomic types are defined w.r.t. a query $\mathcal{Q}$. For example, when $\mathbf{f} = R(a, b)$ (with $a \neq b$) and $atom(R) = R(x, x)$, then there is no τ with $\mathbf{f} \models \tau$.

We can also talk about compatibility in the context of atomic types. We say that two partial atomic types τ, τ' are *compatible w.r.t.* $\mathcal{Q}$, denoted $\tau \sim_{\mathcal{Q}} \tau'$, when there are two facts $\mathbf{f}$ and $\mathbf{g}$ with $\mathbf{f} \models \tau$ and $\mathbf{g} \models \tau'$ such that $\mathbf{f} \sim_{\mathcal{Q}} \mathbf{g}$. Furthermore, we say that τ implies τ', denoted $\tau \models \tau'$, if for all facts $\mathbf{f}$, $\mathbf{f} \models \tau$ implies $\mathbf{f} \models \tau'$. Define $Types(\tau) = \{\omega \in Types(\mathcal{Q}) \mid \omega \models \tau\}$ as the set of all atomic types ω that imply τ. For a set of partial atomic types T, we define $Types(T)$ as a shorthand for $\bigcup_{\tau \in T} Types(\tau)$.

Example 4.8. Recall the query and atomic types from Example 4.7. The fact $B(a, b, a)$ satisfies both ω and τ. We can also see that $\omega \models \tau$. Now define $\omega' = (A, x = x)$. We can see that $\omega \sim_Q \omega'$ because $\omega \models B(1, 2, 1)$ and $\omega \models A(1, 1)$. Furthermore, $B(1, 2, 1)$ is compatible with $A(1, 1)$ due to the valuation $V : \{x \mapsto 1, y \mapsto 2, z \mapsto 1\}$. ✦

Before moving on, we mention another notational convenience. For two sets of variables X and Y, and a partial atomic type τ, we say that $X \subseteq_\tau Y$ if for every $x \in X$ either $x \in Y$ or there exists an $y \in Y$ such that $(x, y) \in E_{\varphi_\tau}$. In other words, X is a subset of Y when taking the equalities in E_{φ_τ} into account. For example, let τ be an atomic type with $(x, y) \in E_{\varphi_\tau}$. We then have $\{x, y, z\} \subseteq_\tau \{y, z\}$.

We can now define what a broadcast dependency set actually is.

Definition 4.9. A *broadcast dependency set (BDS)* for a conjunctive query $\mathcal{Q}$ is a set $\mathcal{S}$ of key-value pairs (τ, T), where $\tau \in \mathit{PTypes}(\mathcal{Q})$ is a key, and $T \in 2^{\mathit{PTypes}(\mathcal{Q})}$ is a *dependency set*, such that:

1. $(\tau, T) \in \mathcal{S}$ and $(\tau, T') \in \mathcal{S}$ implies $T = T'$;
2. $\tau, \tau' \in \mathit{Keys}(\mathcal{S})$ implies $\mathit{Types}(\tau) \cap \mathit{Types}(\tau') = \emptyset$;
3. $(\tau, T) \in \mathcal{S}$ implies $\mathit{Vars}(\tau') \subseteq_{\tau'} \mathit{Vars}(\tau)$ for every $\tau' \in T$.

The elements of $\mathcal{S}$ are called *dependencies*.

The conditions in this definition require that every key can have at most one value in $\mathcal{S}$; every complete atomic types implies at most one of the keys in $\mathcal{S}$; and the set of variables of $\mathit{atom}(\tau')$ is a subset of the variables of $\mathit{atom}(\tau)$ when taking the equalities into account, for each $\tau' \in T$.

Next, we explain how a BDS $\mathcal{S}$ represents an OBF. We identify three cases:

- $\mathit{type}(\mathbf{f})$ is undefined:
  In this case, we keep $\mathbf{f}$ static because $\mathbf{f}$ can never participate in a satisfying valuation, so there's no point in broadcasting it.

- $\mathit{type}(\mathbf{f}) = \tau$ and $\tau \notin \mathit{Keys}(\mathcal{S})$:
  In this case, we always broadcast $\mathbf{f}$.

- $\mathit{type}(\mathbf{f}) = \tau$ and $\tau \in \mathit{Keys}(\mathcal{S})$:
  Notice that the pair $(\tau, T) \in \mathcal{S}$ is unique because of conditions (1) and (2) in the definition above. Also recall that there is a valuation $V_{\mathbf{f}}$ such that $V_{\mathbf{f}}(\mathit{atom}(\tau)) = \mathbf{f}$. By condition (3) of the definition of a BDS, we can also use this valuation for every $\mathit{atom}(\tau')$ for each $\tau' \in T$. Indeed, for each $y \in \mathit{Vars}(\tau') \setminus \mathit{Vars}(\tau)$, we have a variable $x \in \mathit{Vars}(\tau)$ with $(x, y) \in E_{\tau'}$. So we can define the following extended valuation:

  $$V_{\mathbf{f},\tau'}(y) = \begin{cases} V_{\mathbf{f}}(y) & \text{if } y \in Vars(\tau) \\ V_{\mathbf{f}}(x) & \text{if } y \notin Vars(\tau) \text{ and } (x, y) \in E_{\tau'} \end{cases}$$

  Now, we broadcast $\mathbf{f}$ when the local instance does not contain all facts $V_{\mathbf{f},\tau'}(\mathit{atom}(\tau'))$ for each $\tau' \in T$ and $\tau \sim_{\mathcal{Q}} \tau'$. We call this set of facts the *dependency fact set*, which we formally define as $\mathit{Dep}(\mathbf{f}, T) = \{V_{\mathbf{f},\tau'}(\mathit{atom}(\tau')) \mid \tau' \in T \text{ and } \tau \sim_{\mathcal{Q}} \tau'\}$. Notice that if $\mathit{Dep}(\mathbf{f}, T) = \emptyset$, the fact $\mathbf{f}$ is kept static. For convenience we will also define $\mathit{Dep}(\mathbf{f}, S)$ as $\mathit{Dep}(\mathbf{f}, T)$ if there is a $(\tau, T) \in \mathcal{S}$ for which $\mathit{type}(\mathbf{f}) = \tau$. If not, we consider $\mathit{Dep}(\mathbf{f}, S)$ as undefined.

We can summarize this as follows.

Definition 4.10. For a conjunctive query $\mathcal{Q}$ and a BDS $\mathcal{S}$ for $\mathcal{Q}$, $\mathcal{S}$ induces an OBF $f_{\mathcal{S}}$ that maps every instance J to the set $f_{\mathcal{S}}(J)$ of facts $\mathbf{f} \in J$ for which $\mathit{type}(\mathbf{f}) \in \mathit{Types}(\mathcal{Q})$ and, $\mathit{Dep}(\mathbf{f}, \mathcal{S})$ is is undefined or $\mathit{Dep}(\mathbf{f}, \mathcal{S}) \not\subseteq J$.

Example 4.11. Consider the query $Q(x, y, z) \leftarrow A(x, y, z), B(x, y, z), C(z, z)$ and the following partial atomic types:

$$\begin{aligned}
\tau_B &= (B, \mathit{true}),\\
\tau_A^{x=y} &= (A, x = y),\\
\tau_A^{y=z} &= (A, y = z),\\
\tau_A^{\neq} &= (A, x \neq z \wedge y \neq z),\\
\tau_B^{\neq} &= (B, x \neq z \wedge y \neq z).
\end{aligned}$$

Using these partial types, one possible BDS for Q is $\mathcal{S} = \{(\tau_B, \{\tau_A^{x=y}, \tau_A^{y=z}\}), (\tau_A^{\neq}, \{\tau_B^{\neq}\})\}$. We consider the following database instance to illustrate how the induced OBF $f_{\mathcal{S}}$ works:

$$\begin{aligned}
I = \{&A(1,2,3),\ B(1,2,3),\ A(1,1,2),\ B(1,1,2),\\
&A(1,2,2),\ B(1,2,2),\ C(3,4),\ C(3,3)\}.
\end{aligned}$$

We can see that $f_{\mathcal{S}} = \{A(1,1,2),\ A(1,2,2),\ C(3,3)\}$. Indeed, these facts do not match a key in $\mathcal{S}$ but their type is defined, so they are broadcast. The fact $C(3,4)$ is kept static as its type is undefined. The other four facts do match a key in $\mathcal{S}$, but their dependency fact set is contained in I. For example, the fact $\mathbf{f} = B(1,1,2)$ matches τ_B but $\mathit{Dep}(\mathbf{f}, \{\tau_A^{x=y}, \tau_A^{y=z}\}) = \{A(1,1,2)\} \subseteq I$. ✦

It is important to notice that not every valid BDS for a query $\mathcal{Q}$ induces an OBF that is correct for $\mathcal{Q}$. The following lemma gives syntactic conditions for an OBF $f_{\mathcal{S}}$ to be correct for a query.

Lemma 4.12. *Let $\mathcal{Q}$ be a conjunctive query and let $\mathcal{S}$ be a BDS for $\mathcal{Q}$. Then $f_{\mathcal{S}}$ is correct for $\mathcal{Q}$ if and only if there are no complete atomic types ω_1, ω_2, and pairs $(\tau_1, T_1), (\tau_2, T_2) \in \mathcal{S}$, with $\omega_1 \sim_{\mathcal{Q}} \omega_2$, $\omega_1 \models \tau_1$, $\omega_2 \models \tau_2$ such that $\omega_1 \notin \mathit{Types}(T_2)$ and $\omega_2 \notin \mathit{Types}(T_1)$.*

We refer to [14] for a complete and formal proof.

Similarly, we can give syntactic conditions for an OBF $f_{\mathcal{S}}$ to be locally optimal for a query.

Lemma 4.13. *Let $\mathcal{Q}$ be a conjunctive query and let $\mathcal{S}$ be a BDS for $\mathcal{Q}$. Then $f_{\mathcal{S}}$ is correct for $\mathcal{Q}$ if and only if $\mathcal{S}$ satisfies the following conditions:*

1. *for $(\tau, T) \in \mathcal{S}$ and $\omega \in \mathit{Types}(T)$, $\omega \sim_{\mathcal{Q}} \tau$ implies $\omega \models \tau'$ for some $\tau' \in \mathit{Keys}(\mathcal{S})$;*

2. *for every $\omega \in Types(Q) \setminus Types(Keys(\mathcal{S}))$, there is a partial atomic type $\tau_1 \in Keys(\mathcal{S})$ and an $\omega_1 \in Types(\tau_1)$ such that $\omega \sim_Q \omega_1$ and $Vars(\omega_1) \not\subseteq_{\omega_1} Vars(\omega)$; and*

3. *for $(\tau_1, T_1), (\tau_2, T_2) \in \mathcal{S}$, where $\omega_1 \in Types(\tau_1)$, $\omega_2 \in Types(\tau_2)$, and $\omega_1 \sim_Q \omega_2$, we require that $\omega_1 \in Types(T_2)$ implies $\omega_2 \notin Types(T_1)$.*

Intuitively this means the following. Condition (1) requires that every atomic type implying a partial type in a dependency set in $\mathcal{S}$ must also imply a key in $\mathcal{S}$. Indeed, when an atomic type does not imply a key, every local fact of this type is always broadcast. The atomic type can thus be removed from every dependency set it appears in. When condition (2) is not satisfied for an atomic type ω, we can modify $\mathcal{S}$ to broadcast less while preserving correctness by adding $(\omega, \{\tau \mid \tau \sim_Q \omega, \tau \in Types(Keys(\mathcal{S}))\})$. Finally, condition (3) is the syntactic equivalent of the characterization of local optimality for OBFs that we mentioned earlier in Section 4.1. Again, we refer to [14] for a complete and formal proof.

4.3 Algorithm for Constructing a BDS

In this section, we provide an algorithm that was presented in [14], which can be used to construct a BDS that induces an OBF for a conjunctive query.

The algorithm BDS-Build takes a conjunctive query $\mathcal{Q}$ and a sequence of partial atomic types $\mathcal{R}$ as input. The reason a sequence is used here instead of a set is to keep te algorithm deterministic. The algorithm proceeds as follows. It loops over the sequence $\mathcal{R}$ and considers each type $\tau \in \mathcal{R}$ as a key for $\mathcal{S}$. It then finds the set T of all keys that are already in $\mathcal{S}$, and compatible with τ. If for any of these $\tau' \in T$, $Vars(\tau') \subseteq_{\tau'} Vars(\tau)$ doesn't hold, then τ won't be added to $\mathcal{S}$. However if it does hold for all of them, then we add the key-value pair (τ, T) to $\mathcal{S}$.

The detailed algorithm is depicted in Algorithm 1.

Example 4.14. We demonstrate a run of the BDS-Build-algorithm on the query $Q(x, y, z, w) \leftarrow A(x, y, z), B(x, y, z), C(z, w)$. Consider the partial atomic types $(\tau_A, true)$, $(\tau_B, true)$ and $(\tau_C, true)$, and let $\mathcal{R} = (\tau_A, \tau_B, \tau_C)$. Then, BDS-Build starts with $\mathcal{S} = \emptyset$ and adds $(\tau_A, \emptyset)$ in the first iteration. In the second iteration, it adds $(\tau_B, \{\tau_A\})$. Nothing is added in the last iteration though, because $Vars(\tau_A) \not\subseteq_{\tau_A} Vars(\tau_C)$. ✦

Correctness and local optimality The OBF induces by the BDS created with BDS-Build is always correct for the given query, but only locally optimal when the sequence $\mathcal{R}$ contains either exactly:

- all complete types for the query; or

Algorithm 1 BDS-Build

Input: A conjunctive query $\mathcal{Q}$ and a sequence of partial types $\mathcal{R}$
Output: A BDS that induces an OBF for $\mathcal{Q}$

```
1: $\mathcal{S} = \emptyset$
2: for each $\tau \in \mathcal{R}$ do
3:     addPair = $true$
4:     values = $\emptyset$
5:     for each $\tau' \in Keys(\mathcal{S})$ with $\tau' \sim_{\mathcal{Q}} \tau$ do
6:         values = values $\cup\{\tau'\}$
7:         if $Vars(\tau') \not\subseteq_{\tau'} Vars(\tau)$ then
8:             addPair = $false$
9:         end if
10:    end for
11:    if addPair then
12:        $\mathcal{S} = \mathcal{S} \cup \{(\tau, \text{values})\}$
13:    end if
14: end for
15: return $\mathcal{S}$
```

- all open types for the query. An atomic type is considered *open* when is poses no restrictions. In other words, when the formula associated with its equality type always evaluates to true.

Notice that in Example 4.14 the considered partial atomic types are exactly all open types for Q. The BDS constructed in that example in thus locally optimal. For other arbitrary sequences of types, the OBF is not necessarily locally optimal.

Example 4.15. We give an example of a run of BDS-Build that creates a BDS that is not locally optimal. Recall query Q and the partial atomic types from Example 4.14. Now let $\mathcal{R} = (\tau_A, \tau_C)$. Then, BDS-Build will create the BDS $\mathcal{S} = \{(\tau_A, \emptyset)\}$. This is BDS is not locally optimal, as it will always broadcast all B-facts. This is not necessary when the local instance already contains the compatible A-facts. In contrast, the BDS created in Example 4.14 will only broadcast a fact $B(x, y, z)$ if the local instance does not already contain $A(x, y, z)$. ✦

Complexity The algorithm runs in exponential time in the size of $\mathcal{Q}$ when using a sequence of complete types, and in polynomial time in the size of $\mathcal{Q}$ when using strictly open types. This is easy to see. The outer loop iterates over all keys in $\mathcal{R}$ and therefore runs $|\mathcal{R}|$ times. The inner loop iterates over the number of keys already in $\mathcal{S}$, which can be atmost $|\mathcal{R}|$ as well. Also, each iteration of the inner loop which checks the variable containment takes polynomial time, since an atomic type has a size that is polynomial in the size of $\mathcal{Q}$. The algorithm thus always runs in polynomial time in the size of $\mathcal{R}$. However the number of complete atomic types is exponential in the size of $\mathcal{Q}$, resulting in an exponential running time in the size of $\mathcal{Q}$ when using complete types.

Chapter 5

HyperCube

In this chapter, we take a look at some alternative ways to compute conjunctive queries. We first introduce a more synchronous setting than the one used in the previous chapters, that models computation as a sequence of rounds. We then look at the single-round HyperCube algorithm [7, 9], which is optimal on skew-free inputs. Lastly, we study a multi-round algorithm based on HyperCube which is worst-case optimal, even for skewed data [15]. As in the previous chapter, we limit ourselves to full conjunctive queries without self-joins.

5.1 The MPC Parallel Model

The algorithms that will be discussed in this chapter can all be modelled using the *Massively Parallel Computation* model [9]. In this setting, computation takes place on a cluster of p machines using a shared-nothing architecture. The computation itself proceeds in *rounds*, each of which consists of two distinct phases:

- **Communication-phase:** Each server sends data to and receives data from other servers in the network.

- **Computation-phase:** Each server performs computations locally, on the data that is currently present at the server.

This model makes synchronization explicit, since at the end of each round the servers have to wait for all machines to finish their local computations before moving on to the next round.

Initially, the input data is partitioned evenly across all p servers. That is, if the input data consists of m tuples, each server stores m/p input tuples. At the end of the execution, the output consists of the union of the local outputs of the p servers.

An important measure of performance in this model is the *maximum load* at each server. This is defined as the maximum amount of data that a server can receive during any round. An ideal algorithm uses a single round and distributes data evenly without replication achieving a maximum load of m/p. Since this is rarely possible, query evaluation algorithms often need to use more round, have an increased maximum load, or even both.

5.2 The HyperCube Algorithm

Consider the triangle query $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$. A traditional way of computing this join would be using two rounds. The first round would compute the join $U(x, y, z) \leftarrow R(x, y), S(y, z)$ using a parallel hashjoin. The second round would join this intermediate relation $U(x, y, z)$ with $T(z, x)$, again using a parallel hashjoin. The problem with this strategy however, is that the intermediate relation U may be much larger than the input, resulting in a very costly shuffle in between the rounds.

The single-round HyperCube algorithm [7, 9] solves this problem. We will first demonstrate the algorithm by means of an example. Then we will provide a generalization.

Example 5.1. Recall the triangle query $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$. The algorithm first organises the p available servers in a 3-dimensional cube (one dimension per variable). Let's denote the sizes of these dimensions as p_x, p_y and p_z. We call these the *shares*. Each server now has an address defined as a distinct point in $\mathcal{P} = [p_x] \times [p_y] \times [p_z]$. Since we only have p servers in total, we require that $p_x \cdot p_y \cdot p_z \leq p$. A possible choice could therefore be $p_x = p_y = p_z = p^{1/3}$. Next, we also define three independent hash-functions h_x, h_y and h_z, which map values from the domain **dom** to $[p_x], [p_y]$, and $[p_z]$ respectively. During the communication phase, we subsequently send each tuple $R(a, b)$ to all servers that match the address $(h_x(a), h_y(b), \alpha)$ for every $\alpha \in [1, p_z]$. Notice that each tuple is replicated p_z times. Similarly, each tuple $S(b, c)$ is sent to all servers that match $(\alpha, h_y(b), h_z(c))$ for every $\alpha \in [1, p_x]$ and each tuple $T(a, c)$ is sent to all servers that match $(h_x(a), \alpha, h_z(c))$ for every $\alpha \in [1, p_y]$. During the computation phase, each server will perform a local computation of Q on the data it has received. The correctness of this algorithm follows from the fact that any output tuple $Q(a, b, c)$ will be computed on the server with address $(h_x(a), h_y(b), h_z(c))$, since all necessary input facts will be sent to that server. ✦

We can generalize the HyperCube algorithm as follows for any full conjunctive query:

$$Q(x_1, \ldots, x_k) \leftarrow S_1(\bar{x}_1), \ldots, S_l(\bar{x}_l).$$

In the remainder of this chapter we will denote by m_j the number of tuples in relation S_j. The algorithm assigns to each variable x_i a share p_i such that $\prod_{i=1}^{k} p_i = p$. Each server is now represented by a distinct point $\mathbf{y} \in \mathcal{P}$, with $\mathcal{P} = [p_1] \times \cdots \times [p_k]$. In other words, the servers are mapped to points in a k-dimensional hypercube. The algorithm makes use of k independent hash-functions $h_i : \mathbf{dom} \mapsto [p_i]$ to send each tuple t of relation $S_j(x_{i_1}, \ldots, x_{i_n})$ to all servers in the set:

$$\mathcal{D}(t) = \{\mathbf{y} \in \mathcal{P} \mid \forall j \in [n] : h_{i_j}(t[i_j]) = \mathbf{y}_{i_j}\}.$$

In this expression, we use $t[i_j]$ to denote the projection of tuple t onto the variable x_{i_j}. Lastly during the computation phase, each server computes query Q locally on the received facts.

We already mentioned in the introduction of this chapter that the HyperCube algorithm is optimal over skew-free input. We can now formalize that notion in the following sense.

Definition 5.2. Let $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of shares. If for every relation S_j and every tuple t over $A \subseteq \bar{x}_i$ the frequency of t in S_j is at most $m_j / \prod_{x_i \in A} p_i$, we say that the input is skew-free w.r.t. $\mathbf{p}$.

Consequently, Beame et al. [10] formalized the maximum load over skew-free instances:

Proposition 5.3 ([10])**.** *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of shares for the HyperCube algorithm. If the input is skew-free w.r.t. $\mathbf{p}$, then with high probability the maximum load per server is*

$$\tilde{O}\Big(\max_j \frac{m_j}{\prod_{i:x_i \in S_j} p_i}\Big).$$

5.3 Determining HyperCube Dimensions

Notice that a different choice of shares $p_1, \ldots, p_k$ gives a different parameterization of the HyperCube algorithm, which can have a big impact on the maximum load. It is therefore crucial to choose the best shares in order to achieve an optimal load. In particular, the load is optimal when the expected maximum load per server is as small as possible. In this section, we will look at two ways of choosing the shares for the HyperCube algorithm.

5.3.1 Exact Fractional Shares

The first method is to calculate the exact shares that minimize the maximal load by means of linear programming. We can use the analysis from Proposition 5.3 to do this. The idea is to construct an optimization problem that minimizes the maximum load L under the constraints that $L \geq m_j / \prod_{i:x_i \in S_j} p_i$ for every input relation S_j. Additionally we require that $\prod_i p_i \leq p$.

We can transform this problem into a linear program by taking the logarithm with base p on both sides of the constraints. Let $\lambda = log_p L$ and $e_i = log_p p_i$ for all $i \in [k]$. We call e_i the share exponent since $p_i = p^{e_i}$. The linear program then takes the following form:

$$\begin{aligned} \textbf{minimize} \quad & \lambda \\ \textbf{subject to} \quad & \sum_{i \in [k]} e_i \leq 1 \\ & \forall j \in [l] : \sum_{x_i \in S_j} e_i + \lambda \geq log_p(m_j) \\ & \forall j \in [k] : e_i \geq 0 \\ & \lambda \geq 0 \end{aligned} \tag{5.1}$$

Now let $\mathbf{e} = (e_1, \ldots, e_k)$ be the optimal solution to Equation 5.1 and let $\mathbf{p} = (p_1, \ldots, p_k)$ with $p_i = p^{e_i}$. The HyperCube algorithm achieves an optimal maximum load with shares $\mathbf{p}$, assuming the input is skew-free w.r.t. $\mathbf{p}$.

While the shares that we calculate through this method are optimal, it also has one big disadvantage. These optimal shares are in general fractional. In practice however, we can only use integral shares. We could simply round down the fractional shares to integral shares, but this approach could be highly suboptimal as it may leave many servers unused.

Example 5.4. Again consider the triangle query $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$. Let $m_R = m_S = m_T$ and $p = 48$. The lineair program in Equation 5.1 will find the shares $p_x = p_y = p_z = 3.634$. Rounding this down gives $p_x = p_y = p_z = 3$. This means that only $p_x \cdot p_y \cdot p_z = 27$ servers will be used by the HyperCube algorithm, leaving 21 servers unused. ✦

5.3.2 Estimating Optimal Integral Shares

We also investigate another approach that was first mentioned by Chu et al [11]. This approach involves enumerating all possible integral share configurations with the number of used servers less than the total number of available servers. For each of these possible share configurations, we calculate the expected maximum load (again, using Proposition 5.3). The algorithm then chooses the share configuration with the lowest maximum load. If two share configurations yield the same maximal load, we prefer the one with more equal dimension sizes to reduce possible skew during shuffle. For example, assume both x and y are join attributes in a relation $R(x, y)$. We would then prefer $(p_x = 2, p_y = 2)$ over $(p_x = 1, p_y = 4)$. In both cases the relation would be partitioned into 4 partitions, but in the former case this partitioning would be based on both the values for x and y, while in the latter case it would only be based on the value of y. The details of this algorithm are shown in Algorithm 2.

Algorithm 2 Cost-based estimation of optimal integral shares [11]

Input: The total number of available servers p

Output: An optimal integral share configuration $\mathbf{p}$

```
1:  optimalWL = $\infty$
2:  $\mathbf{p}$ = null
3:  for each integral share configuration $\mathbf{p}'$ do
4:      if workload($\mathbf{p}'$) < optimalWL then
5:          optimalWL = workload($\mathbf{p}'$)
6:          $\mathbf{p} = \mathbf{p}'$
7:      else if workload($\mathbf{p}'$) = optimalWL and max($\mathbf{p}'$) < max($\mathbf{p}$) then
8:          optimalWL = workload($\mathbf{p}'$)
9:          $\mathbf{p} = \mathbf{p}'$
10:     end if
11: end for
12: return $\mathbf{p}$
```

5.4 Multi-Round Algorithm Based on HyperCube

Aside from the single-round HyperCube algorithm, we also study a multi-round algorithm based on HyperCube, which is worst-case optimal [15]. This means that the algorithm achieves the minimal expected maximum load while placing no restrictions on the input, which thus may have skew. In this text, we only focus on the algorithm for a specific subclass of conjunctive queries. The concepts needed to identify this subclass are introduced in Sections 5.4.1 and 5.4.2. We then describe some building blocks for the algorithm in Sections 5.4.3 and 5.4.4. The algorithm itself is discussed in Section 5.4.5.

5.4.1 Simple Connected Queries

We focus on a subset of conjunctive queries called *simple and connected queries*. We first define what it means for a query to be simple.

Definition 5.5. A conjunctive query $\mathcal{Q}$ is called *simple* if:

(i) $\mathcal{Q}$ has only binary atoms; and

(ii) there are no distinct atoms $R, S \in \mathit{atoms}(\mathcal{Q})$ for which $\mathit{Vars}(R) \subseteq \mathit{Vars}(S)$.

Notice that these are exactly the conjunctive queries whose query graph is a simple graph (hence the name). Next, we define connectedness for conjunctive queries.

Definition 5.6. A conjunctive query is called *connected* when its query graph is connected, meaning that for every pair of variables $x, y \in \mathit{Vars}(\mathcal{Q})$ there is a sequence of atoms $S_1, \ldots, S_n$ such that $x \in \mathit{Vars}(S_1)$, $y \in \mathit{Vars}(S_n)$ and for every $i \in [1, n-1] : \mathit{Vars}(S_i) \cap \mathit{Vars}(S_{i+1}) \neq \emptyset$.

To conclude this section, we also introduce the concept of *residual queries*. Let $\mathcal{Q}$ be a conjunctive query. For a set of variables $X \subseteq \mathit{Vars}(\mathcal{Q})$, we define the residual query $\mathcal{Q}[X]$ as the query obtained by removing the variables in X from $\mathcal{Q}$ and adjusting the arities of the relations accordingly.

Example 5.7. Consider the following queries:

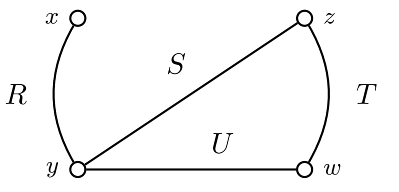
$$Q_1(x, y, z, w) \leftarrow R(x, y), S(y, z), T(z, w), U(y, w)$$

$$Q_2(x, y, z, w) \leftarrow R(x, y), S(y, z), T(z, w), U(z, z).$$

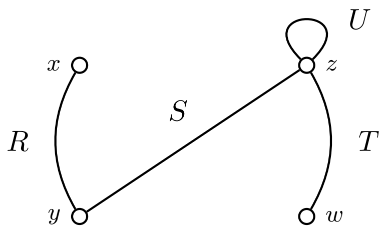
Notice that Q_1 is both simple and connected. On the other hand, Q_2 is not simple because of the atom $U(z, z)$. However, it is connected. The query graph is depicted in Figure 5.1 for reference. Now let $X = \{y, z\}$. The residual query $Q_1[X]$ takes the form:

$$\bar{Q_1}(x, w) \leftarrow \bar{R}(x), \bar{S}(), \bar{T}(w), \bar{U}(w).$$

✦



(a) The query graph of Q_1.



(b) The query graph of Q_2.

Figure 5.1: The query graphs from Example 5.7.

5.4.2 Tight Fractional Edge Packings

Next, we introduce the notion of fractional edge packings.

Definition 5.8. Let $\mathcal{Q}$ be a conjunctive query and $f : \mathit{atoms}(\mathcal{Q}) \mapsto \mathbb{R}^+$ be a mapping from the atoms of $\mathcal{Q}$ to non-negative weights. We say that f is a *fractional edge packing* if

$$\sum_{a:x \in \mathit{Vars}(a)} f(a) \leq 1, \text{ for every } x \in \mathit{Vars}(\mathcal{Q}) \tag{5.2}$$

where a is over $\mathit{atoms}(\mathcal{Q})$.

Moreover, we call f tight if the inequality in Equation 5.2 is an equality. For the multi-round algorithm we only consider queries that have a tight fractional edge packing.

Example 5.9. Reconsider query Q_1 from Example 5.7. There exists a tight fractional edge packing for this query, namely $f = \{R \mapsto 1, S \mapsto 0, T \mapsto 1, U \mapsto 0\}$. It is easy to see that $\sum_{a:x \in \mathit{Vars}(a)} f(a) = 1$ for every $x \in \{x, y, z, w\}$. ✦

5.4.3 Semi-join Decompositions

In the context of distributed query evaluation, it is often possible to reduce relations using semi-joins. Doing so can dramatically decrease the amount of data that needs to be shuffled over the network. For this purpose, we take a look at semi-join decompositions.

Definition 5.10. Let $\mathcal{Q}$ be a query. A *semi-join decomposition* for $\mathcal{Q}$ is a minimal subset of atoms $\mathcal{V} \subseteq \mathit{atoms}(\mathcal{Q})$ such that every atom $a \notin \mathcal{V}$ is contained in some atom $b \in \mathcal{V}$, more precisely $\mathit{Vars}(a) \subseteq \mathit{Vars}(b)$.

Given a semi-join decomposition $\mathcal{V}$ for a query $\mathcal{Q}$, the reduced query $\mathcal{Q}_\mathcal{V}$ is defined as the query consisting of all atoms in $\mathcal{V}$. If S is an atom in $\mathcal{V}$, we will denote its corresponding occurence in $\mathcal{Q}_\mathcal{V}$ by $S^\mathcal{V}$. A semi-join reduction then consists of a set of $|\mathcal{V}|$ queries of the form

$$S^\mathcal{V}(\bar{x}) \leftarrow S(\bar{x}), R_1(\bar{x_1}), \ldots, R_k(\bar{x_k})$$

where S is an atom in $\mathcal{V}$ and R_i are all atoms with $\bar{x_i} \subseteq \bar{x}$.

Using this concept, we can easily compute a query $\mathcal{Q}$ in two steps:

1. Compute the semi-join reduction to obtain the reduced relations $S^\mathcal{V}$.
2. Compute the reduced query $\mathcal{Q}_\mathcal{V}$ on these reduced relations.

This may seem pointless, especially since the reduced query of a simple query $\mathcal{Q}$ is always $\mathcal{Q}$ itself. However, this method will come in handy later when computing residual queries, which are not necessarily simple.

An easy way of obtaining a semi-join decomposition is by iteratively removing body atoms from $\mathcal{Q}$ whose variables are contained in other atoms. The remaining atoms form a semi-join decomposition for $\mathcal{Q}$. This semi-join decomposition is not necessarily unique, but one always exists.

Example 5.11. Consider the query $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(y)$. A semi-join decomposition $\mathcal{V}$ for Q is the set $\{R(x, y), S(y, z)\}$. The semi-join reduction then consists of the following queries:

$$R^{\mathcal{V}}(x, y) \leftarrow R(x, y), T(y)$$
$$S^{\mathcal{V}}(y, z) \leftarrow S(y, z), T(y).$$

The reduced query $Q_{\mathcal{V}}$ takes the form:

$$Q_{\mathcal{V}}(x, y, z) \leftarrow R^{\mathcal{V}}(x, y), S^{\mathcal{V}}(y, z).$$

✦

5.4.4 Heavy-Hitter Configurations

The last building block we need for the algorithm is the notion of heavy-hitter configurations. Heavy hitters are values whose frequency in a relation is much higher than other values. Heavy-hitter configurations give us a mechanism to partition relations based on these heavy hitters. Before we continue though, we introduce some notation to talk about the value frequencies.

Consider a query $\mathcal{Q}$, a value $c \in \mathbf{dom}$ and a variable $x \in \mathit{Vars}(\mathcal{Q})$. We denote by $\text{freq}_S(c, x)$ the frequency of the value c in the column of relation S that corresponds with the position of variable x in the atom a with $\mathit{pred}(a) = S$. Furthermore, by $\text{freq}(c, x)$ we denote the maximal frequency over all relations where x is incident to in $\mathcal{Q}$. More specifically, this means that $\text{freq}(c, x) = \max_R\{\text{freq}_R(c, x)\}$, where R is over all relations incident to variable x in $\mathcal{Q}$.

We can now turn our attention to the heavy-hitter configurations itself.

Definition 5.12. A *heavy-hitter configuration* Ψ for a query $\mathcal{Q}$ is pair (H, δ) where H is a subset of the variables in $\mathcal{Q}$ and $\delta \in [0, 1]$ a threshold value separating heavy from light values.

Given a heavy-hitter configuration $\Psi = (H, \delta)$ and an instance I, a value c is called *light* for variable x if $\text{freq}(c, x) \leq m/p^\delta$, where m is the number of tuples in the largest relation in I. Otherwise, we call c *heavy.* This definition is relation independent, so a variable x that occurs in k distinct atoms may have up to $k \cdot p^\delta$ heavy hitters.

Further, we denote by $I|_\Psi$ the subset of I containing all facts that are compatible with Ψ. For an atom $S(x_1, \dots, x_k)$ in $\mathcal{Q}$, we call a fact $S(c_1, \dots, c_k)$ compatible with Ψ if for all variables x_i and their corresponding values c_i: $\text{freq}(c_i, x_i) > m/p^\delta$ if $x_i \in H$ and $\text{freq}(c_i, x_i) \leq m/p^\delta$ if $x_i \notin H$.

Example 5.13. Let $p = 4$, and consider the query $Q(x, y, z) \leftarrow R(x, y), S(y, z)$ over instance

$$I = \{R(a, d), R(b, d), R(c, e), S(d, a), S(e, b), S(c, d)\}.$$

Let $\delta = 1/2$ be our threshold value. We know that $m = 3$, so we have $m/p^{\delta} = 3/2$. We can thus distinguish the following heavy-hitter configurations:

$$\begin{aligned} I_{(\{y\},\delta)} &= \{R(a,d), R(b,d), S(d,a)\}, \\ I_{(\emptyset,\delta)} &= \{R(c,e), S(e,b), S(c,d)\}. \end{aligned}$$

For all other $H \subseteq \mathit{Vars}(\mathcal{Q})$ we have $I_{(H,\delta)} = \emptyset$. ✦

Lastly, we introduce notation to specify subinstances where certain values are fixed. Let $\mathcal{Q}$ be a query, I an instance and $X \subseteq \mathit{Vars}(\mathcal{Q})$. Also let $\mathbf{t} = (c_1, \ldots, c_n)$ be a sequence of values from **dom** where each value corresponds to one variable in $X = \{x_1, \ldots, x_n\}$. We denote by $I|_{\mathbf{t},X}$ the subinstance of I consisting of tuples that are compatible with the values in $\mathbf{t}$ for their corresponding variables in X. More specifically, $I|_{\mathbf{t},X}$ consists of all relations $R^{I|_{\mathbf{t},X}}$ defined as follows: if $R(y_1, \ldots, y_k)$ is an atom in the query, then $R^{I|_{\mathbf{t},X}} = \{R(d_1, \ldots, d_k) \in I \mid \forall x_i, y_j : y_i = x_j \implies d_i = c_j\}$.

Example 5.14. Recall the instance I from Example 5.13. We can express the subinstance of I with value c on the position of variable x as follows:

$$I|_{c,x} = \{R(c,e), S(d,a), S(e,b), S(c,d)\}.$$

✦

5.4.5 Multi-Round Algorithm

We can now finally discuss the algorithm itself [15]. As said before, this algorithm only works on a subset of the conjunctive queries. We quickly recap the conditions that we place on a query in order for this algorithm to work correctly. Let $\mathcal{Q}$ be a query. We require that:

(i) $\mathcal{Q}$ is simple and connected;

(ii) $\mathcal{Q}$ has a tight fractional edge packing; and

(iii) heavy-hitter configurations do no generate isolated variables.

For conditions (i) and (ii) we refer to Sections 5.4.1 and 5.4.2. Condition (iii) implies that for every heavy-hitter configuration $\Psi = (H, \delta)$, the reduced query $\mathcal{Q}_{\mathcal{V}}$ of the semi-join decomposition of $\mathcal{Q}[H]$ is also simple.

For simplicity, we further assume that all heavy hitters are known by all servers. If this is not the case, they can be computed and broadcast to all servers in one additional round.

The main idea of the algorithm is as follows. We consider all heavy-hitter configurations $\Psi = (H, \delta)$ where H ranges over all subsets of the heavy variables in our query while using $\delta = 1/|\mathit{Vars}(\mathcal{Q})|$. We then compute $\mathcal{Q}$ in parallel over all the Ψ-compatible subinstances using a specialized algorithm depending on H. The output of our query on an instance I is then exactly the union of the computations on these subinstances: $\mathcal{Q}(I) = \bigcup_{H \subseteq \mathit{Vars}(\mathcal{Q})} \mathcal{Q}(I|_{(H,\delta)})$. For convenience, we denote $L = \mathit{Vars}(\mathcal{Q}) \setminus H = |\mathit{Vars}(\mathcal{Q}[H])|$. We identify the following four cases:

All variables are light:
In this case $H = \emptyset$. The subinstance $I|_\Psi$ then consists of all light tuples. The algorithm simply computes $\mathcal{Q}$ on $I|_\Psi$ using the regular HyperCube algorithm using a dimension size of $p^{1/|\mathit{Vars}(\mathcal{Q})|}$ for every variable.

All variables are heavy:
In this case $H = \mathit{Vars}(\mathcal{Q})$. The subinstance $I|_\Psi$ is very small since there are only a limited number of heavy values. More specifically, $|I|_\Psi| \leq O(p)$. The algorithm can therefore send all those facts to a single server and compute $\mathcal{Q}$ locally on $I|_\Psi$ there.

Exactly one variable is light:
In this case $|L| = 1$. Let's denote this single light variable by x. We split the subinstance $I|_\Psi$ in $I|_{\Psi_1}$ and $I|_{\Psi_2}$ where $\Psi_1 = (H, 2/|\mathit{Vars}(\mathcal{Q})|)$ and $\Psi_2 = (H \cup \{x\}, 2/|\mathit{Vars}(\mathcal{Q})|)$. Notice that $I|_\Psi = I|_{\Psi_1} \cup I|_{\Psi_2}$, because all heavy hitters remain heavy under the new threshold. We now compute $\mathcal{Q}(I|_{\Psi_1})$ and $\mathcal{Q}(I|_{\Psi_2})$ in parallel over all servers. For $\mathcal{Q}(I|_{\Psi_1})$, we use the HyperCube algorithm with dimension size p for variable x, and 1 for all other variables. For $\mathcal{Q}(I|_{\Psi_2})$, we just proceed as in the previous case since all values are heavy.

At least two variables are light:
In this case $|L| \geq 2$. For convenience, denote $J = I|_\Psi$. The idea behind this case is as follows. The number of values for the heavy-hitter variables are by definition limited. This means that for each of these heavy-hitter tuples $\mathbf{h}$, we can compute $\mathcal{Q}$ on the $\mathbf{h}$-compatible subinstances in parallel using a fraction of the available p servers. More specifically,

$$\mathcal{Q}(J) = \bigcup_{\mathbf{h}} \mathcal{Q}(J|_{\mathbf{h},H})$$

where $\mathbf{h}$ ranges over all heavy hitter tuples. We can allocate $p' = c \cdot p^{|L|/|\mathit{Vars}(\mathcal{Q}))|}$ servers to each heavy hitter tuple $\mathbf{h}$, where $c = 1/d(\mathcal{Q})^{|H|}$. By $d(\mathcal{Q})$ we denote the degree of the query graph.

It remains to show how to compute $\mathcal{Q}(J|_{\mathbf{h},H})$ using these p' servers for any heavy hitter tuple $\mathbf{h}$. Consider an atom $R(x, y)$ in $\mathcal{Q}$. When x and y are both heavy, we can view $R(x, y)$ as a boolean value which we can broadcast to all servers. If just x is heavy, R becomes a unary relation $R(y)$. Similarly, if y is heavy, R becomes the unary relation $R(x)$. To compute $\mathcal{Q}(J|_{\mathbf{h},H})$

it now suffices to compute the residual query $\mathcal{Q}[H]$, which can be done as follows:

1. Let $\mathcal{V}$ be a semi-join decomposition for $\mathcal{Q}[H]$. Compute the semi-join reductions for $\mathcal{V}$. This can be done in two rounds because each atom in $\mathcal{Q}[H]$ has arity at most two.

2. Compute the reduced query $\mathcal{Q}_{\mathcal{V}}$ of $\mathcal{Q}[H]$ on the semi-join reductions computed in the previous step using the HyperCube algorithm. Since all values in the semi-join reductions are light, we can use a dimension size of $p^{1/|\mathit{Vars}(\mathcal{Q}[H])|}$ for every variable.

Chapter 6

Implementation

In this chapter, we discuss our implementation of the algorithms described in Chapter 5. We first introduce Apache Spark, on top of which our implementation was made. We then discuss the implementation details more thoroughly.

6.1 Apache Spark

As a basis for the implementation of our system, we chose to use Apache Spark [2]. There are several reasons for this choice. Its flexibility makes it easy to build new platforms on top of it. Furthermore, it has the ability to cache intermediate data in memory for later use, which is specifically useful for multi-round algorithms. These arguments also attracted a great deal of interest from the community. Over the past few years, Spark's popularity grew immensely while it emerged as the next generation big data processing engine. It improves over the traditional Hadoop system in several areas. First of all, it is much faster because of its extensive support for in-memory computing. Moreover, it is very easy to use due to its powerful APIs that are available in different programming languages. Lastly, it can be used for a variety of workloads including machine learning, data stream processing and graph processing. In this section, we discuss some of the core concepts of Spark that are needed to understand how it can be used as a basis for the implemented algorithms.

6.1.1 Architecture

At a high level, Spark uses a master/slave architecture with one central coordinating node and many distributed workers nodes. This coordinator is called the *driver*. It communicates with the so-called *executors* that are running on the worker nodes. The driver and executors together form the Spark application. A Spark application can be launched on a cluster of compute nodes using a cluster manager. Spark is packaged with a built-in *standalone* cluster manager. However, it also supports the use of Hadoop's YARN cluster manager as well as Apache Mesos. A schematic overview of the architecture of a Spark cluster is shown in Figure 6.1.

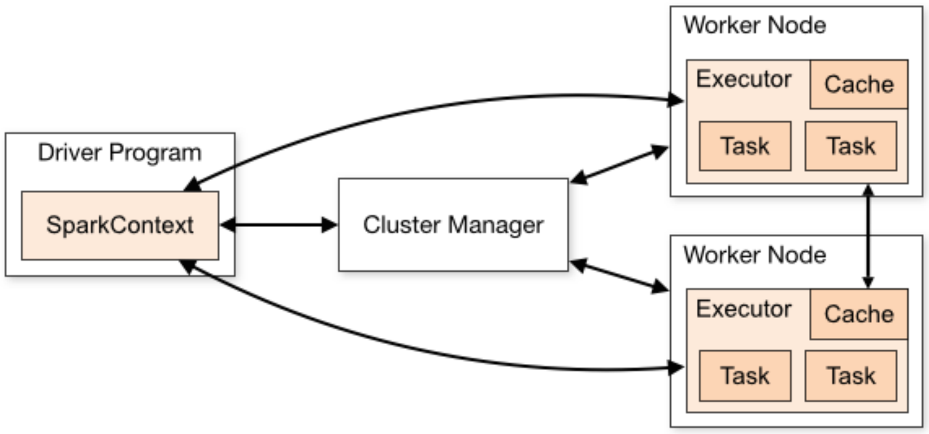


Figure 6.1: A schematic overview of the Spark cluster architecture. [2]

Driver Program

The driver program contains and runs the user's *main* function. It is also responsible for creating a `SparkContext` object that provides access to Spark. Furthermore, it defines distributed datasets on the cluster and initializes parallel operations on these datasets. Once the driver program terminates, the application is finished.

During its runtime, the driver program has two main duties. The first one is to convert the user program in to separate *tasks*. It accomplishes this as follows. Each Spark application has a common high-level structure: it constructs distributed datasets from some input and then performs operations on these datasets to derive new datasets or collect and save data. These operations implicitly form a directed acyclic graph (DAG). The driver converts this logical graph into a physical execution plan. This allows Spark to perform some optimizations, such as pipelining several map operations and combining them into one. The physical execution plan ultimately consists of a set of stages, which in turn consist of multiple tasks. These tasks can be packaged and are sent to the executors.

The second duty of the driver is to schedule these tasks on executors. When an executor is started, it registers itself with the driver. The driver thus has a complete view of the application's executors at all times. Given a physical execution plan it will look at the current situation of the executors and try to schedule each task in an appropriate location based on the data on which the task operates. A task also has the possibility to cache data for later use, so the driver also keeps an overview of the cached data which it uses to schedule future tasks that access the same data.

Executors

Spark executors are the processes responsible for running the individual tasks in a Spark application. They are launched at the start of the application and typically last for its entire duration. However, it is also possible that they fail during execution. In that case, the Spark application will just continue to run and the driver will reschedule the tasks of the failed executor if necessary.

Their responsibility is twofold. As mentioned already, they run the tasks that make up an application and can return results to the driver program. Secondly, they provide in-memory storage for the distributed datasets that are cached by the user. The fact that this data is cached directly in the executors is particularly convenient because this means that tasks can run right alongside the data that they use.

Cluster Manager

Now that we have discussed the driver program and the executors, you may be wondering how they are initially launched and managed during the application lifetime. This is the responsibility of the cluster manager. Spark ships with a built-in standalone cluster manager that makes it very easy to setup a Spark environment on a set of machines, without the need of a dedicated cluster manager. The cluster manager is however a pluggable component of Spark, which allows it to run on different external managers such as YARN and Mesos. This is particularly useful when you want to make use of other components from the Hadoop ecosystem, like the distributed filesystem HDFS.

6.1.2 Resilient Distributed Datasets

We already vaguely mentioned the use of distributed datasets. In this section, we will clarify how Spark handles these. The core data abstraction in Spark are the resilient distributed datasets (RDDs) [18]. An RDD in Spark is simply an immutable distributed collection of objects. Each RDD is split into a number of *partitions*, which may be computed on different nodes of the cluster.

Spark provides two ways to create RDDs. The first way is to load an external dataset from file, using the `textFile()` method of the `SparkContext` object:

```
JavaRDD<String> lines = sc.textFile("path/to/my/file.txt");
```

This is obviously the most common way to create RDDs for processing massive datasets. However, it is also possible to create an RDD from an already existing collection of objects in the driver program. For this you can make use of the `parallelize()` method of the `SparkContext` object:

```
JavaRDD<String> lines = parallelize(Arrays.asList(
                                        "hello", "world"));
```

Once an RDD has been created, it supports two kinds of operations: *transformations* and *actions*. Transformations are operations that transform an RDD into a new RDD (recall that RDDs are immutable). Examples of transformations are `map` and `filter` operations. Actions on the other hand are operations that return a result to the driver program or write it to storage. An example of an action is the `count` operation.

It is important to understand which type of operation you are performing, because Spark treats transformations and actions very differently. Transformations are computed lazily only when the resulting RDD is used in an action. This means that when we call a transformation on an RDD (for example, a `map` operation) it is not executed immediately. Instead, Spark records that this operation has been requested in the DAG structure that we already mentioned in the previous section. Spark uses this lazy evaluation mechanism to reduce the number of passes that need to be made over the data by grouping operations together.

It is also worth mentioning that RDDs have explicit support for key/value pairs, which are a common data type required for many operations in Spark.

For completeness, we provide a list of the most common operations that are used on RDDs. Table 6.1 shows the most common transformations for regular RDDs, while Table 6.2 shows those specifically for use with key/value RDDs. Similarly, Table 6.3 shows the most common actions for regular RDDs, while Table 6.4 shows the most common actions for key/value RDDs.

6.2 Implementation Details

The goal of our implementation is to experimentally verify the performance of the algorithms from Chapter 5. We therefore implemented the HyperCube algorithm as well as the multi-round algorithm in Apache Spark. Additionally, we implemented a naive approach as a baseline to compare the other algorithms to. In this naive approach, we evaluate the joins in a traditional way, using one round per join.

In addition to these different shuffle algorithms, we also implemented two different local join algorithms to study their impact on performance when combined with the various shuffle algorithms.

Transformation Name	**Description**
`map(`*func*`)`	Apply a function the each element in the RDD and return an RDD of the result.
`flatMap(`*func*`)`	Apply a function the each element in the RDD and return an RDD of the contents of the iterators returned. Often used to convert lines into a set of words.
`filter(`*func*`)`	Return an RDD consisting of only elements that pass a certain condition.
`distinct()`	Remove duplicates from an RDD.
`union()`	Return an RDD containing elements from two other RDDs.
`intersection()`	Return an RDD containing only elements found in two other RDDs.
`subtract()`	Remove the contents of one RDD from another.

Table 6.1: A list of common transformations on regular RDDs.

Transformation Name	**Description**
`reduceByKey(`*func*`)`	Combine values with the same key using a given reduce function.
`groupyKey()`	Group values with the same key
`mapValues(`*func*`)`	Apply a function to each value of a pair without changing the key
`keys()`	Return an RDD consisting of just the keys.
`values()`	Return an RDD consisting of just the values.

Table 6.2: A list of common transformations on key/value RDDs.

Action Name	**Description**
`collect()`	Return all elements in the RDD to the driver program.
`count()`	Return the number of elements in the RDD.
`countByValue()`	Return the number of times each element occurs in the RDD.
`foreach(`*func*`)`	Apply a given function for each element in the RDD.

Table 6.3: A list of common actions on regular RDDs.

Action Name	**Description**
`countByKey()`	Count the number of element for each key.
`collectAsMap()`	Collect the result to the driver program as a map for easy lookup.

Table 6.4: A list of common actions on key/value RDDs.

6.2.1 I/O

Our implementation takes a file as input. The first thing that is specified in this file are the input relations. The user lists the relation names, the relation arities and the locations on the file system where the associated data is stored. Each of these relations take up one line in the input file. A double line break separates these relation definitions from the actual queries that should be computed. A user can specify multiple queries separated by line breaks. Each query consists of a head atom and a comma-separated list of body atoms. The head and body are separated by `:-`. Listing 6.1 shows an example of an input file for the program.

```
R, 2, input/R.txt
S, 2, input/S.txt
T, 2, input/T.txt

Q(x,y,z) :- R(x,y), S(y,z), T(z,x)
```

Listing 6.1: Example of an input file for the program.

6.2.2 Shuffle Algorithms

The main shuffle logic is contained in the `QueryEngine`. This is an abstract class with three concrete implementations: `NaiveEngine`, `HyperCubeEngine` and `MultiRoundEngine`. Each of these have an `execute()` method that takes a `Query` object representing the query it should evaluate. This way of working makes it possible to easily extend the system with other shuffle algorithms if needed.

Naive

The `NaiveEngine` starts by loading the relation corresponding to the first two body atoms of the query into an RDD. These are the two relations that will be joined first. Next, it performs a `map` operation on this RDD to transform it in a key/value RDD. For each tuple, we take the value of the join attribute as the key, and the complete tuple as the value. This key/value RDD subsequently gets repartitioned so that all pairs with the same key end up at the same server. Ultimately, the values will be extracted from this key/value RDD and a local join algorithm will be applied on the data received at each server, yielding an RDD of intermediate results. Which local algorithm is used depends on the configuration. We refer to Section 6.2.3 for more information regarding the available algorithms. For each of the remaining body atoms in the query, the `NaiveEngine` will load in the corresponding relation and merge it with the intermediate RDD and repeat the process of mapping, repartitioning and performing the local join. Once all body atoms have been processed the intermediate result is also the final query result, which is written to file again.

This approach is very naive in the sense that it only considers the left-deep join tree. It is very well possible that using another order to process the join yields far better performance. In practice, query engines often use some sort of cost model to estimate the most efficient join order.

HyperCube

The `HyperCubeEngine` first calculates the HyperCube dimensions that will be used according to one of the algorithms described in Section 5.3. It then loads all input data in one large RDD, which once again gets transformed into a key/value RDD by using a `map` operation. In contrast to the naive approach however, every tuple can now be mapped to more than one key/value pair. As a key we now choose every point in the hypercube representing the address of the server where a tuple should be sent to. We still use the complete tuple as the value. The independent hash-functions that we use to determine the keys are instances of the Murmur3 hashfunction from the Google Guava library[1], each with a different seed value. After the `map` operation, the key/value RDD gets repartitioned based on the keys, so that all tuples with the same key end up at the same server. Once that's done, the tuples will be extracted from the key/value RDD and the local join algorithm will be applied to compute the query. The result of this computation is already the final query result, since the HyperCube algorithm arranges the tuples in such a way that all tuples that can participate in a satisfying valuation together will be placed at the same server.

Multi-round

Since the multi-round algorithm assumes that all heavy hitters are knows to all servers beforehand, the `MultiRoundEngine` starts by loading all input relations in RDDs and determining the heavy hitters. This is done by mapping each tuple to its individual values and then performing a `countByValue()` action. Each of these counts is subsequently compared to the threshold. All values that exceed the threshold are retained and broadcasted across all servers.

Once this preprocessing phase is finished, a list of all possible heavy-hitter configurations is generated and each of these configurations are processed separately. For the heavy-hitter configurations that are processed using a special case of the HyperCube algorithm (i.e. those with no light variables and at most one light variable), the algorithm is executed in a similar way as in the `HyperCubeEngine`. The case with at least two light hitters requires a little more attention. First, the number of servers that should be allocated to each heavy-hitter tuple is determined. Subsequently, all heavy-hitter tuples

[1]`https://github.com/google/guava`

are processed in parallel. For each heavy-hitter tuple the compatible subinstances are created using `filter()` operations. The semi-join reduction of the residual query is then executed on these subinstances. Lastly, the HyperCube algorithm is applied to compute the residual query result. We then compute the cross product with the heavy-hitters that satisfy the query to obtain the actual output for the original query. This output is then written to disk.

6.2.3 Local Join Algorithms

The shuffle algorithms above only deliver the data to the right servers. After that, the servers still need to compute the query locally on the fragment of data that is available to them. To do this, we implemented two local join algorithms: the classic binary hash-join [12] and the multiway leapfrog trie-join [17].

Hash-join

The first local join algorithm that we implemented is a binary hash-join which proceeds as follows. First, it creates a hash-table of the smaller input relation, where the keys of the hash-table are the values of the join attribute. Once this hash-table is built, we scan the larger relation. For each tuple we look up the value of the join attribute in the hash-table to find possible matching tuples from the smaller relation. Each matching tuple is then joined and the resulting tuple is placed in the output.

Leapfrog Trie-join

The second local join algorithm is the leapfrog trie-join which is a multiway join algorithm. This means that it can be used to join more than two relations at once. In the case of conjunctive queries, it essentially enumerates all satisfying assignment for the body atoms.

The main building block of the leapfrog trie-join is a variant of sort-merge join called the *leapfrog join*, which is capable of simultaneously joining multiple unary relations. For the purpose of this discussion, we assume that these relations consist of only natural numbers. However in practice, these relations can be over any ordered domain. These unary relations are represented in sorted order by lineair iterators (one for each relation) exposing the following interface:

`int key()`	Returns the key at the current iterator position.
`next()`	Move iterator to the next key.
`seek(int seekKey)`	Positions the iterator at a least upper bound for `seekKey`. That is, the least `key` $\geq$ `seekKey`. If no such key exists, the iterator should be moved to the end.
`bool atEnd()`	Returns true when the iterator is at the end of the relation.

To achieve optimal complexity, the `key()` and `atEnd()` methods are required to take $O(1)$ time, and the `next()` and `seek()` met hods are required to take $O(log\,N)$ time where N is the size of the relation.

The leapfrog join is itself implemented as an instance of the linear iterator interface. The algorithm uses an array of pointers to iterators, one for each relation. The join tracks the smallest and largest keys at which iterators are positioned, and repeatedly moves the iterator at the smallest key to a least upper bound for the largest key, until all iterators are positioned at the same key. This is called *leapfrogging*, hence the name of the join. Figure 6.2 illustrates this process. We refer to [17] for a more detailed description of the algorithm, including pseudo-code.

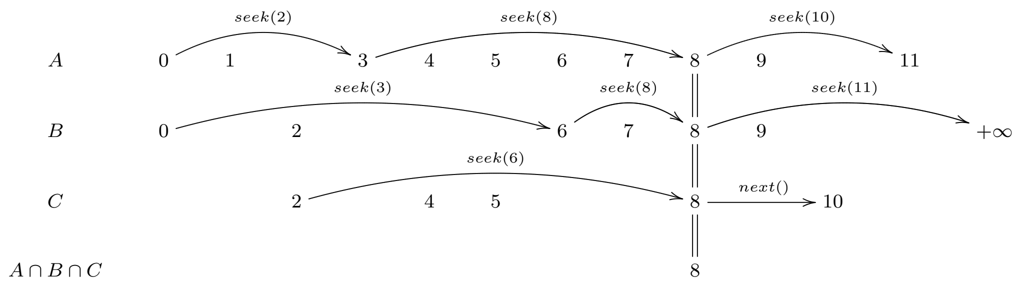


Figure 6.2: An example of a leapfrog join of three relations A, B and C. Initially, the iterators are positioned at the smallest key in the relation. The iterator for A performs `seek(2)` because of the key of the iterator for C, landing it at 3. The iterator for B then performs `seek(3)`, landing at 6. This process continues until all iterators are at the same key, namely 8, which belongs to the output of the join. [17]

To accommodate for relations of arity > 1, we need to extend the linear iterator interface. We can do this by presenting relations such as $R(x_1, \ldots, x_k)$ as tries with each tuple $(a_1, \ldots, a_k) \in R$ corresponding to a unique path in the trie from the root to a leaf. An example of this can be seen in Figure 6.3. We do not need to store the relation as a trie though. It suffices to store the relation in a tree-like search structure and presenting the data itself via a trie iterator interface. This interface provides the same methods as the linear iterator interface presented above, as well as the following two new methods:

`open()`	Proceed to the first key at the next depth.
`up()`	Return to the parent key at the previous depth.

Both these methods are required to take $O(log\ N)$ time. For our implementation, we used the `TreeSet` data structure that is built into Java (which is based on red-black trees internally) to store our relations and provide the trie iterator methods within the required complexity bounds.

$R(x, y)$	
3	4
3	5
4	6
4	8
4	9

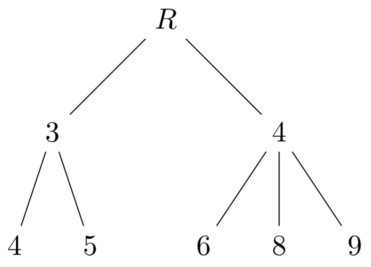


Figure 6.3: An example of a binary relation $R(x, y)$ represented as a trie.

It now remains to be shown how the leapfrog trie-join algorithm can be implemented based on the unary leapfrog join and the trie iterator interface. At initialization, the leapfrog trie-join is provided with a trie iterator for each relation of the join. It constructs an array of leapfrog join instances, one for each variable. The leapfrog join for a variable x is then given an array of pointers to trie iterators, one for each atom in which x appears. For example, in the join of relations $R(x, y)$, $S(y, z)$ and $T(z, x)$, the leapfrog join for y is given pointers to the trie iterators for R and S. Notice that there is only one instance of the trie iterator for R, which is shared by the leapfrog joins for x and y.

The leapfrog joins use the linear iterator portion of the trie iterator interfaces, while the `open()` and `up()` trie navigation methods are used only by the trie-join algorithm itself. The algorithm also uses a variable *depth* to track the current variable for which an assignment is being sought. This variable is initially set to -1 to indicate the trie-join is positioned at the root the trie.

The trie-join itself is implemented as an instance of the trie-iterator interface. The linear iterator portions of the trie iterator interface (namely `key()`, `atEnd()`, `next()`, and `seek()`) are delegated to the leapfrog join for the current variable. The implementation of the `open()` method increases the *depth* variable and calls the `open()` method of all trie iterators in the leapfrog join at the current depth. It also calls the initialization method of the leapfrog join at the current depth. The implementation of the `up()` method simply calls the `up()` method of all trie iterators in the leapfrog join at the current depth and then decreases the *depth* variable again. Again, we refer to [17] for pseudo-code.

The trie-join algorithm is parameterized by the order in which a satisfying assignment is being sought for the variables. Choosing a good variable ordering is crucial for performance in practice. However, implementing an optimizer to choose a variable ordering is out of the scope of this thesis. Therefore we have choosen to always use the order in which the variables appear in the query.

6.3 Usage

The easiest way to launch the program is by using the `spark-submit` utility included with your Spark installation:

```
./spark-submit  --master $master_url \
                --class hyperspark.Main hyperspark.jar  \
                --input input/query.txt  \
                --output output \
                --servers 54
```

Aside from specifying the URL of the master node in your Spark cluster and the main class of our implementation, it is also required to provide an input file, an output directory and the level of parallelism. This is done by using the `--input`, `--output` and `--servers` flags.

There are also a number of other flags to run the program with the desired configuration settings. A complete overview of all possible flag is shown below.

`--help, -h`	Show usage information.
`--engine, -e`	Specifies the query engine to be used: `naive`, `hypercube` (default) or `multiround`.
`--join, -j`	Specifies the join algorithm that will be used locally: `hashjoin` or `triejoin` (default).
`--shares, -s`	Specifies the algorithm used to compute the HyperCube shares when the HyperCube engine is used: `rounddown` or `optimal` (default).
`--input, -i`	File containing the query and input files.
`--output, -o`	Path where the output will be stored.
`--servers, -p`	Number of servers used for execution.

Chapter 7

Experiments

In this chapter, we discuss the experimental validation of our implementation. Part of these experiments are based on the experiments conducted by Chu et al. [11]. We first list the objectives of our experiments. Then we describe our experimental setup. Lastly, we discuss the results of the different experiments in more detail.

7.1 Objectives

Before proceeding with the actual experiments, it is important to list the insights we wish to obtain. We will try to find an answer to the following questions.

1. Does using a cost model (see Section 5.3.2) to estimate the maximum load to find the ideal dimensions for the HyperCube algorithm lead to better performance than when rounding down the optimal fractional shares?

2. How does the HyperCube algorithm compare to a naive pairwise evaluation of the joins?

3. What impact does the use of a leapfrog trie-join have as a local join algorithm during HyperCube as opposed to a standard hash join?

4. How does the multi-round worst-case optimal algorithm (see Section 5.4.5) compare to the standard HyperCube algorithm when applied on instances with different levels of skew?

In addition to these objectives, we will also compare our findings with those formulated by Chu et al. [11] where approriate.

7.2 Setup

The experiments were conducted on the infrastructure of the Flemish Supercomputer Center (VSC). We have used the compute nodes featuring two 10-core "Ivy Bridge" Xeon E5-2680v2 CPUs (2.8 GHz, 25 MB level 3 cache) with 64 GB of RAM. The nodes are linked to a QDR Infiniband network. All nodes have a small local disk (mostly for swapping and the OS image) and are connected to shared filesystem for main storage.

Because of Spark's architecture and ability to perform one task per CPU core, we identify each core as a *server* in the context of the algorithms described in Chapter 5. For example, when using 3 20-core compute nodes, we will say that $p = 60$, even though there are only 3 physical machines.

We used Spark 2.0.2 in standalone mode. The specific Spark configuration settings used for the experiments can be found in Table 7.1.

Setting Name	**Value**
`spark.executor.cores`	6
`spark.executor.extraJavaOptions`	-XX:+UseG1GC
`spark.local.dir`	$TMPDIR
`spark.files.fetchTimeout`	1800
`spark.network.timeout`	1800
`spark.worker.timeout`	30000
`spark.rpc.retry.wait`	30000
`spark.storage.blockManagerHeartBeatMs`	30000

Table 7.1: Spark configuration settings used for the experiments.

7.3 Experiment 1: Determining HyperCube Shares

The goal of this first experiment is to determine how the methods for obtaining HyperCube dimensions that we described in Section 5.3 behave and compare to each other.

Queries & Input

For this experiment, we used the following queries and input relations.

- **Triangle-query:**

  $$Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$$

  The first query is the well-known triangle query. For the input relations R, S and T we use three different instances of a Twitter dataset [5] representing a follower-followee relationship. Each of these relations contain 5,940,253 tuples.

- **CLIQUE-QUERY:**

  $$Q(x, y, z, w) \leftarrow R(x, y), S(y, z), T(z, w), U(w, x), V(x, z), W(y, w)$$

  The second query is similar the TRIANGLE-QUERY, but searches for 4-cliques instead of triangles. This requires significantly more joins. For the input relations, we use six instances of the same Twitter relation used for the TRIANGLE-QUERY.

- **GUARDED-QUERY:**

  $$Q(title, user, score) \leftarrow R(title, user, score), P(user), A(title)$$

  The third query is a guarded query. The guard relation R contains 12,940,720 tuples about movie ratings. The guarded relations are much smaller, leading to a fairly small output. Relation P contains 99,333 tuples representing premium users, while relation A contains 2,997 tuples representing action movies. The query thus computes all tuples of premium users rating an action movie. The data used for this query is based on the MovieLens dataset [3].

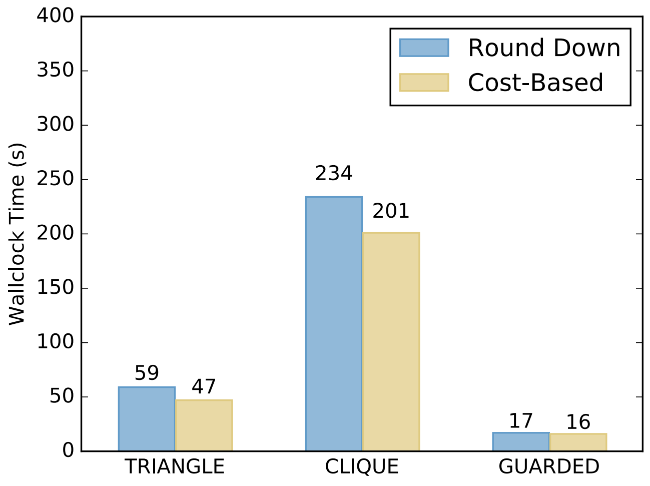
The TRIANGLE-QUERY and CLIQUE-QUERY were also used by Chu et al. [11]. However, we used a slightly different dataset that is 5 times larger.

Results

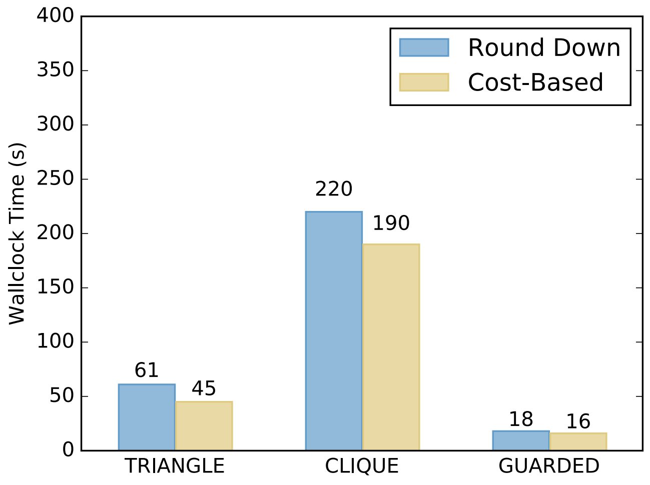
We computed the queries above using both methods to obtain HyperCube shares on 50, 60 and 70 servers. The leapfrog trie-join was used as local join algorithm for all tests. The wallclock times of these computations can be found in Figure 7.1a, 7.1b and 7.1c respectively. These timings include evaluating the query and writing the result to discuss. They do not include the initial data distribution. Initially, the data is distributed randomly but evenly across the available servers.

For the TRIANGLE-QUERY, we can observe an increase in performance upto 26.2% for cluster sizes 50 and 60 using the cost-based approach. This can be attributed to the fact that the round-down approach uses only 27 of the available servers, whereas the other approach uses nearly double. For a cluster size of 70, both approaches derive the same HyperCube shares leading to similar performance.

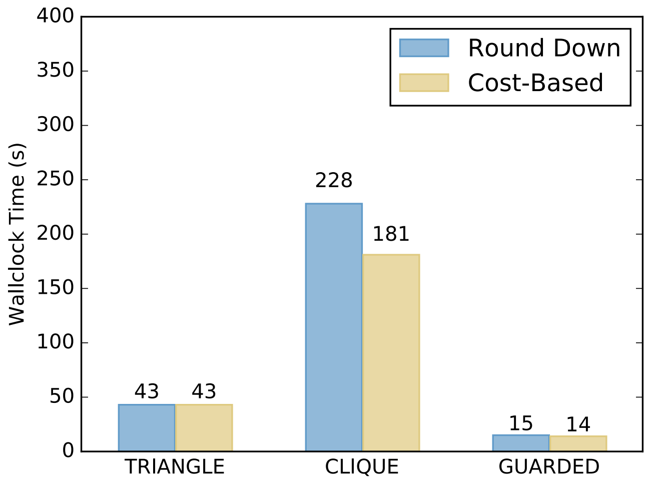
For the CLIQUE-QUERY, we see a similar performance of the round-down approach across all three cluster sizes that were tested. This is because the HyperCube shares get rounded down to the same numbers. When looking at the cost-based approach, we see that its performance increases as the cluster size grows showing a better usage of available resources. For all cluster sizes, we can see improvements of 13.6%, 14.1% and 20.6% respectively in comparison with the round-down approach.



(a) $p = 50$



(b) $p = 60$



(c) $p = 70$

Figure 7.1: Visual representation of the results of Experiment 1.

For the GUARDED-QUERY, we observe that both approaches reach a very similar performance. That is because they both assign all servers to a single variable, meaning that all servers are used in both cases. That single variable is different for both approaches though. The round-down approach assigns all servers to the variable *score*, meaning that relations P and A are both broadcasted. The cost-based approach on the other hand assigns all servers to the variable *user*, meaning that only the relation A is broadcasted. This leads to a small increase of performance over the round-down approach.

Table 7.2 shows a summary of the percentual increases in performance when using the cost-based approach rather than the round-down approach.

Lastly, it is worth discussing the efficiency of calculating the shares using the cost-based approach. For all tests that we performed, the algorithm was able to compute the best integral shares in less than a couple hundred milliseconds. This makes the algorithm also very practical.

Conclusion
We can conclude that the performance of the cost-based approach is better than that of the round-down approach in most cases, with an average performance gain of 13.2% over all our experiments. The exact gain in performance is of course dependent of the number of available servers and the number of variables in the query.

	p = 50	**p = 60**	**p = 70**
TRIANGLE-QUERY	20.3%	26.2%	0%
CLIQUE-QUERY	14.1%	13.6%	20.6%
GUARDED-QUERY	5.8%	11.1%	6.6%

Table 7.2: Summary of percentual increases in performance when using the cost-based approach rather than the round-down approach.

7.4 Experiment 2: Skew-free Data

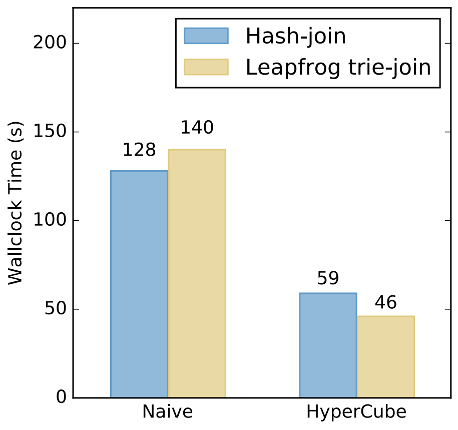
With this second experiment, we compare the HyperCube algorithm with the naive pairwise evaluation of the joins. We also analyse what impact on performance the leapfrog trie-join has as a local join algorithm instead of a standard hash-join.

Queries & Input
For this experiment, we used the same queries and input relations as for the previous experiment in Section 7.3.

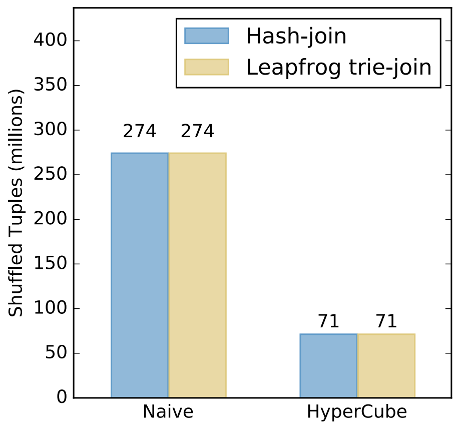
Results
For this experiment, we ran all queries using 54 servers. For the runs of

the HyperCube algorithm, we used the cost-based approach to determine the HyperCube dimensions. Additionally, all the input relations that were used are skew-free.

Figure 7.2 shows the results for the TRIANGLE-QUERY. The first observation we can make is that using the HyperCube algorithm leads to a huge increase in performance, taking less than half the wallclock time of the naive shuffle algorithm. The main reason for this is that it avoids the costly shuffle of the huge intermediate result of joining R with S. This can be seen in Figure 7.2b. The HyperCube implementation shuffles a total of around 71 million tuples, while the naive implementation shuffles nearly four times as many tuples, at 274 million. We can also make an interesting observation regarding the local join algorithms. When using the HyperCube algorithm, the leapfrog trie-join is more performant than the hash-join. On the other hand, the hash-join is more efficient when using the naive shuffle algorithm. Both join algorithms have their disadvantages. The leapfrog trie-join needs to sort the input data, while the hash-join algorithm still generates a significant amount of intermediate results locally. The amount of data that needs to be sorted is rather high when using the naive shuffle algorithm for the TRIANGLE-QUERY. It seems that the cost of this sorting phase outweighs the amount of intermediate tuples that are generated locally by the hash-join, leading to a better performance for the hash-join algorithm. In the case of the HyperCube algorithm however, the amount of data that requires sorting is much smaller resulting in a better performance of the leapfrog trie-join.



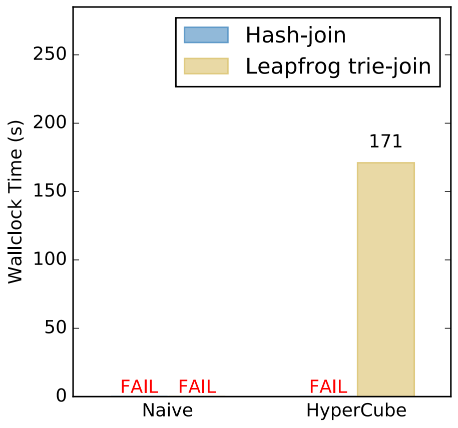
(a) Wallclock Time



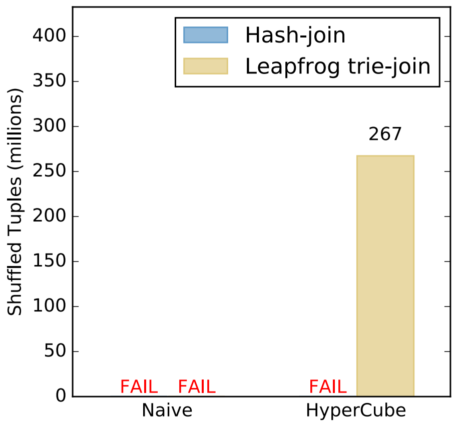
(b) Number of shuffled tuples

Figure 7.2: Visual representation of the results of Experiment 2 on the TRIANGLE-QUERY.

The results for the CLIQUE-QUERY, shown in Figure 7.3, seem to confirm our conclusions from the TRIANGLE-QUERY. It is an exaggerated case of the TRIANGLE-QUERY in the sense that it generates even more intermediate results. In fact it generates so many intermediate tuples that both runs of our naive implementation ran out of memory. So did the run of HyperCube with the hash-join algorithm. However, this is because our hash-join implementation materializes all intermediate results. It is worth mentioning that an alternative implementation of the hash-join that pipelines multiple joins would not suffer from these memory issues. This experiment underlines the strength of the HyperCube algorithm for queries that have large intermediate join results.



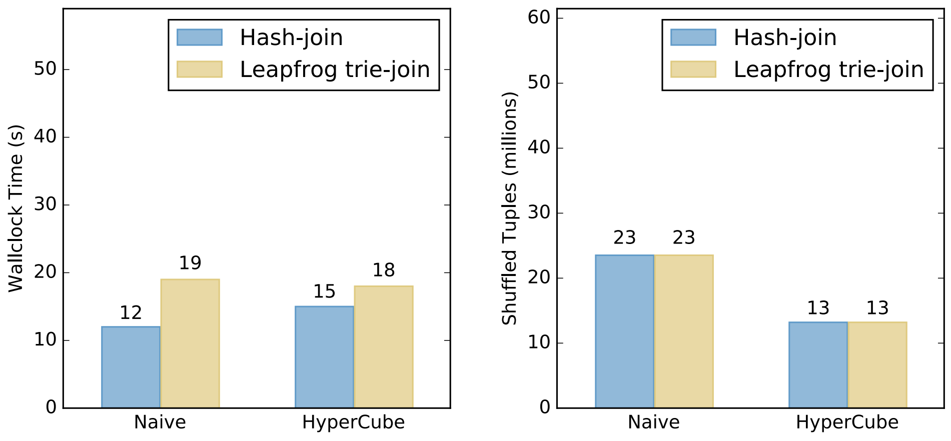
(a) Wallclock Time



(b) Number of shuffled tuples

Figure 7.3: Visual representation of the results of Experiment 2 on the CLIQUE-QUERY.

The GUARDED-QUERY, whose results can be found in Figure 7.4, is very different from the TRIANGLE-QUERY and CLIQUE-QUERY because it does not generate a lot of intermediate results. In fact, every join reduces the amount of data in the pipeline. This is due to the fact that the GUARDED-QUERY is guarded. The number of tuples in an intermediate join result can therefore never exceed the size of the guard relation. A consequence of this can be seen in Figure 7.4b. The number of tuples that have to be shuffled with the naive implementation is still slightly more than with the HyperCube implementation, but the difference isn't so significant as with the other two queries. This also translates to very similar timings for the two implementations, as seen in Figure 7.4a. We can conclude that there is no real advantage in not having to compute the intermediate results explicitly, as long as these results are fairly small. We can also see that for both the naive and HyperCube implementations the hash-join algorithm outperforms the leapfrog trie-join. The reason for this is most likely also the lack of large intermediate results. Subsequently there's not much advantage in using a multiway join algorithm instead of a tree of binary hash-joins. Moreover, the leapfrog trie-join still has the overhead of sorting.



(a) Wallclock Time **(b)** Number of shuffled tuples

Figure 7.4: Visual representation of the results of Experiment 2 on the GUARDED-QUERY.

Conclusion

We can conclude that the HyperCube implementation clearly outperforms the naive implementation for queries with large intermediate join results. However, there's no real gain in performance in case the size of the intermediate results is the same order of magnitude as the size of the input. A multiway join algorithm like the leapfrog trie-join helps to enhance the performance of the HyperCube algorithm in the former case, but should not be used in the latter case where it will only impact performance in a negative way.

These conclusions are similar to those formulated by Chu et al. [11], who also observed the advantage of using the HyperCube algorithm to shuffle data for queries that generate large intermediate results. Furthermore, they also observed faster runtimes when using a traditional shuffle algorithm for queries that have smaller intermediate results.

7.5 Experiment 3: Skewed Data

In this last experiment, we concentrate on skewed input data. More specifically, we compare the performance of HyperCube algorithm and the multi-round algorithm when applied on skewed instances.

Queries & Input
All relations in this experiment are based on the Twitter dataset [5] that we also used in the previous experiments. However, we modified this dataset to introduce skew. To do this, we looped over the original, skew-free dataset and replaced each value with a heavy-hitter value with a predefined probability p. Doing so, we could control the number of heavy-hitters that were introduced, as well as the percentage of tuples in the dataset that actually contain these heavy-hitter values. We used this process to introduce different levels of skew, which are described below.

As for the actual query that we used, we reconsider the TRIANGLE-QUERY that was also used in the other experiments:

$$Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$$

However, we ran this query using six different levels of skew.

Level	**Description**
`S1`	Only skew in relation R. We introduced 1 heavy-hitter for the first attribute that occurs in 50% of the tuples, resulting in 1 heavy-hitter in total.
`S2`	Skew in all relations. We introduced 1 heavy-hitter for the first attribute of each relation where the heavy-hitter occurs in 50% of the tuples, resulting in 3 heavy-hitters in total.
`S3`	Skew in all relations. We introduced 1 heavy-hitter for the first attribute of each relation where the heavy-hitter occurs in 70% of the tuples, resulting in 3 heavy-hitters in total.
`S4`	Skew in all relations. We introduced 1 heavy-hitter for the first attribute of each relation, where the heavy-hitter occurs in 90% of the tuples, resulting in 3 heavy-hitters in total.
`S5`	Skew in all relations. We introduced 1 heavy-hitter for each of the attributes of each relation, where each heavy-hitter occurs in 50% of the tuples, resulting in 6 heavy-hitters in total.
`S6`	Skew in all relations. We introduced 2 heavy-hitters for the first attribute of each relation, where each heavy-hitter occurs in 40% of the tuples, resulting in 6 heavy-hitters in total.

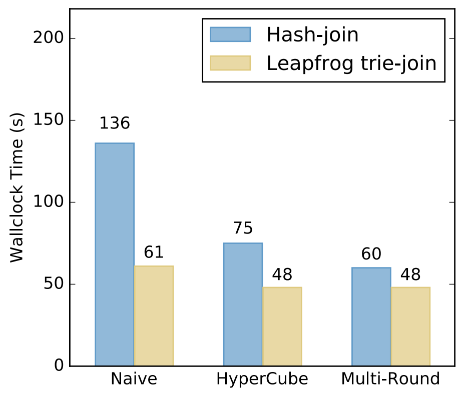
Results
Before we dive into the actual results, it is important to make a remark

regarding the multi-round implementation. The algorithm assumes that all servers know the heavy-hitters beforehand. Since this is not the case in our experiments, our implementation determines the heavy-hitters in an extra preprocessing round. This preprocessing step is not included in the timings for this experiment. Furthermore, we used 54 servers for all queries in this experiment.

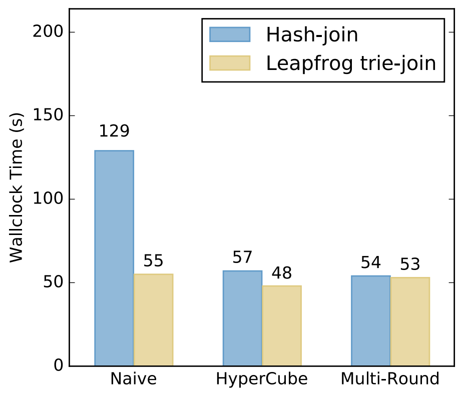
The result for the TRIANGLE-QUERY are shown in Figure 7.5. When looking at the results for skew level `S1`, we see a small increase in performance for the multi-round implementation in comparison with the standard HyperCube implementation when using the hash-join algorithm locally. However, when using the leapfrog trie-join this increase is non-existent. Moreover, when looking at the naive implementation we see that the leapfrog trie-join performs much better than the hash-join. Therefore, it seems that the leapfrog trie-join is the preferred choice as local join algorithm when working with skewed instances. The results for the other skew levels confirm this conclusion. The use of the multi-round algorithm on the other hand does not really provide an advantage over the HyperCube algorithm. Next, we look at the results for skew levels `S2`, `S3` and `S4`. For skew level `S2`, we see that the multi-round implementation performs slightly worse than the HyperCube implementation. However, we do see a clear increase in performance when using the multi-round implementation as opposed to the the HyperCube implementation for skew levels `S3` and `S4`, which have more skew. This seems to indicate that there needs to be a significant amount of skew in order for the multi-round algorithm to have a positive effect. Lastly, we consider the results for skew levels `S5` and `S6`, which have more heavy-hitter values than the other skew levels. We see that the multi-round implementation performs significantly worse than the standard HyperCube implementation. Upon further investigation, it was clear that the case that handles the heavy-hitter configurations with at least two light-hitters forms the bottleneck here. The reason for this is that this step heavily relies on the computation of subinstances. This requires multiple applications of Spark's `filter()` transformation, which in turn requires multiple passes over the same data. The time taken by handling this case exploded when increasing the number of heavy-hitter values from 3 to 6. Another thing that stood out was that there was a lot more variation in the timing between multiple runs of the multi-round implementation. A possible explanation for this is the higher number of tasks that are generated by Spark, and the final running running time is of course dependent on how the cluster manager schedules all these tasks.

Conclusion

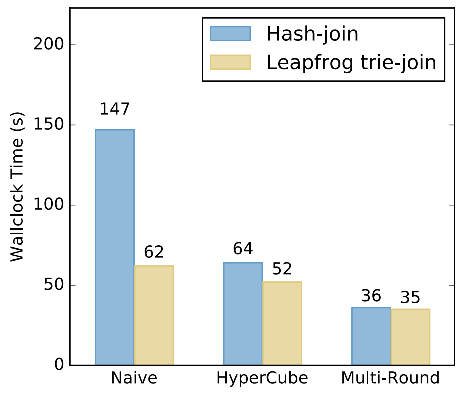
In conclusion, we can say that the multi-round implementation performs better than the standard HyperCube implementation on skewed instances only when a few conditions are satisfied. Firstly, the skew should be significant



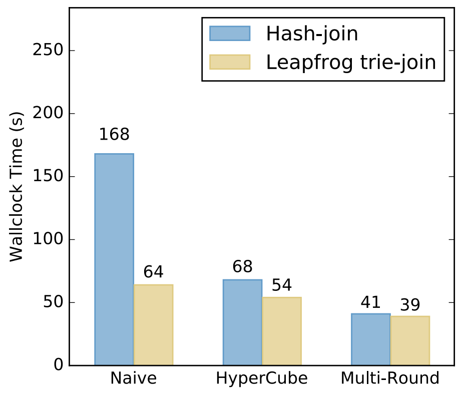
(a) Results for skew level S1.



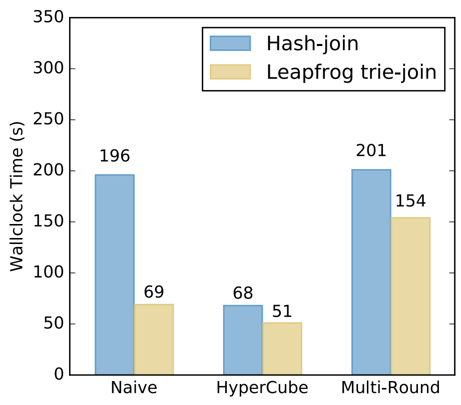
(b) Results for skew level S2.



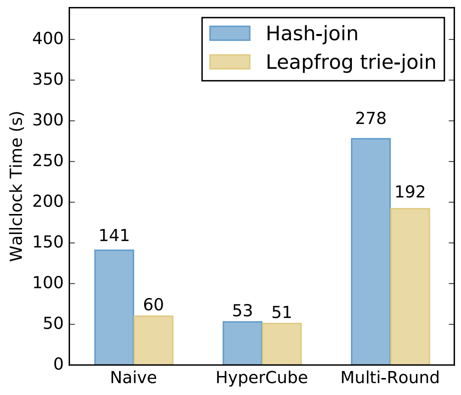
(c) Results for skew level S3.



(d) Results for skew level S4.



(e) Results for skew level S5.



(f) Results for skew level S6.

Figure 7.5: Visual representation of the results of Experiment 3 on the TRIANGLE-QUERY.

enough. If not, the increase in performance does not outweigh the overhead of the multi-round algorithm. Secondly, there should not be too many heavy-hitter values. Furthermore, we can conclude that the leapfrog trie-join algorithm works very well in combination with skewed input instances.

Bear in mind that these results should only be viewed in the context of our Spark implementation. It is very well possible that an alternative implementation would yield better results, even in cases with a higher number of heavy-hitters. For example, consider the repeated application of the *filter* operator to split an instance into several subinstances. This kind of splitting behaviour could be implemented in its own operator that does not rely on other high-level operators in the framework. This way the operator could be optimized to only loop over the data once, instead of once for every created subinstance. Something similar has been done in the Myria [4] system, which has a built-in operator for the HyperCube shuffle.

Chapter 8

Conclusion

In this thesis, we studied the theoretical concept of coordination-freeness and took a closer look at efficient ways to evaluate conjunctive queries in a parallel fashion. More specifically, we focussed on the CALM-conjecture and a formalization thereof based on the computational model of relational transducer networks. We subsequently proposed an alternative definition of coordination-freeness and showed that it is equivalent to the definition originally proposed by Ameloot et al [8]. To conclude our study of coordination-freeness, we also proposed a possible variation of the transducer networks model in which the order of sent messages is preserved.

However, because the computation strategies used in this theoretical study mostly rely on broadcasting all data, we turned our attention to more economical computation strategies. On one hand we considered more optimal broadcasting strategies, expressed using OBFs that satisfy a semantical optimality property, which broadcast no more data than strictly necessary in order to correctly compute the query. To construct these OBFs, we studied the concept of broadcast dependency sets which are a syntactical alternative to OBFs. This allowed for the introduction of a simple algorithm that can be used to construct an optimal broadcasting strategy for a given query. On the other hand, we studied the single-round HyperCube algorithm which is optimal for skew-free input instances, as well as a multi-round algorithm based on HyperCube which is worst-case optimal even for skewed input instances. We made an implementation of these algorithms on top of Apache Spark to experimentally verify them. Additionally, we implemented a naive algorithm for conjunctive query evaluation to use as a baseline throughout the experiments, as well as two join algorithms that are used locally by each server during the computation. More specifically, we wanted to find out the best way to calculate which dimensions to use in the HyperCube algorithm and how the performance of the algorithms compare to each other in practice. We also wanted to find out how the local join algorithms that were used affected this performance. Our experiments showed that the HyperCube algorithms performs really well on skew-free instances for queries with large intermediate join results. Furthermore, the multi-round algorithm outperforms the Hyper-Cube algorithm on skewed instances on condition that the skew is significant enough and that there are not too many heavy-hitter values. However, the bad performance of queries with more heavy-hitter values seems to be caused by how we used Spark's API to split the data in different subinstances.

This inefficiency gives rise to some possible future research. For instance, it would be very interesting to see the performance of an implementation that does not rely on the higher level abstractions provided by the Spark API. This would allow for a more fine-tuned implementation of the different steps in the multi-round algorithm, possibly leading to better performance regardless of the number of heavy-hitter values. Furthermore, it might be interesting to extend the implementation of the multi-round algorithm to also include queries with non-tight edge packings as described in the original paper [15].

Bibliography

[1] Apache Hadoop. `http://hadoop.apache.org`. Accessed: 2017-05-15.

[2] Apache Spark. `http://spark.apache.org`. Accessed: 2017-04-04.

[3] Movielens Data Set. `https://grouplens.org/datasets/movielens/`. Accessed: 2017-03-24.

[4] Myria. `http://myria.cs.washington.edu`. Accessed: 2017-05-10.

[5] Twitter Data Set. `https://an.kaist.ac.kr/traces/WWW2010.html`. Accessed: 2017-01-22.

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

[7] Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[8] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, 2013.

[9] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284, 2013.

[10] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. *CoRR*, abs/1401.1872, 2014.

[11] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78, 2015.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (international edition).* Pearson Education, 2002.

[13] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[14] Bas Ketsman and Frank Neven. Optimal broadcasting strategies for conjunctive queries over distributed data. In *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, pages 291–307, 2015.

[15] Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. *Accepted for publication at PODS*, 2017.

[16] Douglas Laney. The importance of 'big data': A definition. https://www.gartner.com/doc/2057415/importance-big-data-definition, 2012. Accessed: 2017-05-15.

[17] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.