Beheer van heterogene opslagtechnologieën
voor snelle datatoegang in 'Big Data'-omgevingen

Management of Polyglot Persistent Environments
for Low Latency Data Access in Big Data

Thomas Vanhove

Promotoren: prof. dr. ir. F. De Turck, dr. G. Van Seghbroeck
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2017 - 2018

ISBN 978-94-6355-084-0
NUR 988, 995
Wettelijk depot: D/2018/10.500/2

UNIVERSITEIT
GENT

Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Informatietechnologie

Leden van de examencommissie:

prof. dr. ir. Filip De Turck (promotor)
Universiteit Gent - imec
dr. Gregory Van Seghbroeck (promotor)
Universiteit Gent - imec

prof. dr. ir. Daniël De Zutter (voorzitter)
Universiteit Gent
prof. dr. ir. Frank Gielen
Universiteit Gent
prof. dr. Guy De Tré
Universiteit Gent
prof. dr. ing. Erik Mannens
Universiteit Gent - imec
ir. Werner Van Leekwijck
Universiteit Antwerpen
dr. Anthony Liekens
IO Lab

Proefschrift tot het behalen van de graad van
Doctor in de ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2017-2018

Dankwoord

Je zou denken dat 5,5 jaar een lange tijd is, maar nu ik hier de laatste woorden van dit boek neerschrijf, lijkt het alsof augustus 2012 toch niet zo veraf is. Ik ben dankbaar dat ik dit hoofdstuk mag afsluiten, maar tegelijk betekent dat ook dat ik afscheid moet nemen van mensen die ik in de voorbije jaren mocht leren kennen. Alvorens ik dat doe, wil ik iedereen bedanken die in mindere of meerdere mate een invloed gehad heeft op dit boek of op mijn leven in het algemeen gedurende mijn tijd aan de Universiteit Gent. Daarnaast zijn er een paar mensen die ik uitdrukkelijk wens te bedanken:

De eerste personen die ik wil bedanken zijn mijn promotoren Filip De Turck en Gregory Van Seghbroeck. Filip, bedankt om mij de mogelijkheid te geven om aan een doctoraat te beginnen. Bedankt om keer op keer weer beschikbaar te zijn voor een gesprek of feedback en om mij de flexibiliteit te geven om dit doctoraat tot een goed einde te brengen. Gregory, bedankt om samen met mij vorm te geven aan mijn doctoraat en om een klankbord te zijn voor mijn ideeën. Ik denk dat het maar zelden voorkomt dat je ook samen met één van je promotoren een spin-off kan oprichten en leiden. Bedankt om dit avontuur met mij te willen aangaan en ik kijk uit naar wat de komende jaren met Qrama zullen brengen. Naast mijn promotoren zijn er ook nog een paar mensen die niet kunnen ontbreken omdat zonder hun dit werk en de voorbije 5 jaren er heel anders hadden uitgezien: Tim Wauters en Bruno Volckaert. Bedankt om vanaf dag één klaar te staan voor mij, om papers na te lezen, en mij tot op vandaag te ondersteunen. Mijn woordenschat is te beperkt om uit te drukken hoe dankbaar ik ben voor jullie steun. Bruno, jouw bizarre humor en de talrijke verhalen tijdens de autoritten van en naar Leuven zijn legendarisch. Ik hoop er nog veel te mogen horen. En sorry voor de San Francisco fietstocht!

Er zijn ook nog een paar iconen die op zich een vermelding verdienen. Onze vakgroep Informatietechnologie en onderzoeksgroep IDLab Gent worden gerund door de bewonderenswaardige Bart Dhoedt en Piet Demeester. Zij zijn de drijfveer die zorgen voor de constante vooruitgang en groei die onze groepen kennen. Daarnaast zit je als doctoraatsstudent vaak met praktische vragen, zeker in die eerste en laatste maanden van het doctoraat, maar ik kan mijn hand ervoor in het vuur steken dat je het antwoord steeds bij Martine en Davinia kan vinden. Ik vraag mij ten zeerste af wat iedereen zou doen moesten jullie niet elke dag paraat staan voor iedereen met een kleine of grote vraag. Bedankt! Zonder een technisch team is een internettechnologie-onderzoeksgroep maar weinig waard, dus ook veel dank gaat uit naar Brecht, Joeri, Bert, Vicent en Simon van het A-team. Thanks guys!

Bedankt aan Bernadette en Joke van de financiële administratie en, last but not least, Sabrina om steeds onze werkplek proper te houden en voor de spontane gesprekken in de wandelgangen.

Een dankwoord is niet compleet zonder de schijnwerper even op de bureaus 2.21 en 200.012 te zetten. Bram, Jerico, Jeroen, Kristof, Laurens, Leandro, Maxim, Merlijn, Niels, Olivier, Philip, Piet, Sander, Stefano, Steven B, Steven VC, Thijs, Tim en Wim ... dankjewel! Bedankt om het waard te maken om elke dag, goed of slecht, naar de bureau te komen. Ik zal nog lang met een lach terugkijken op de vele fijne bureau-momenten en de lunches waarbij we de een of andere existentiële levensvraag van Merlijn in onze richting geslingerd kregen. Like, is this even real life? Het is ook fijn om jouw thesisbegeleiders als collegas te leren kennen. Bedankt Femke en Femke voor jullie begeleiding in dat laatste masterjaar en voor de verschillende gezellige board game momenten daarna. Soms was het noodzakelijk om ook eens weg te stappen van de bureau en het is dan leuk om collega's te hebben waarbij je stoom kan aflaten bij het *koffiemachien*. Bedankt hiervoor Thomas en Wannes. Bedankt ook aan alle andere collega's om deze ervaring zo uniek te maken!

Als we het dan toch over unieke ervaringen hebben, dan moet ik even stilstaan bij het Tengu verhaal. Tengu is in de loop van mijn doctoraat zijn eigen leven gaan leiden en dat is allemaal dankzij het Tengu-team en zijn supporters. Gregory, Jeroen, Merlijn, Sander, Leandro, Tim, Bruno, Filip, bedankt om Tengu te maken tot wat het vandaag al geworden is. Sinds oktober 2016 hebben ook Sebastien, Mathijs, Dixan en Michiel hun steentje bijgedragen via Qrama. Bedankt om deel uit te maken van dit avontuur en bedankt voor de schitterende sfeer die jullie elke dag weer opnieuw maken. Het is een voorrecht om samen met jullie in de tech start-up wereld aan Tengu te bouwen.

Soms was het gemakkelijk om te vergeten dat er nog een leven was buiten het doctoraat en Qrama, maar gelukkig zijn er vrienden waar je beroep op kan doen om alles eventjes achter te laten. Bros (Nicolas, Thibaut, Lino en Ewout), bedankt voor de maandelijkse hangouts, jaarlijkse oudejaarsavonden en vele andere momenten die we mochten delen met of zonder onze wederhelften (Teresa, Elke, Inne en Inge). De talrijke ontmaskeringen van wannabe dictators, het bestrijden van virussen op wereldschaal en UNO met punten zijn ook momenten uit de vele board game nights (aka `REDACTED`) die ik koester dankzij vrienden zoals Bart, Melissa, Nicolas en Teresa. Allemaal heel oprecht bedankt! Ik voeg hier graag een speciale vermelding voor Maxim toe, want het is buitengewoon om een collega een vriend te kunnen noemen. Ik heb je leren kennen als de grappenmaker van de bureau, maar daarachter schuilt iemand waar je echt op kan rekenen en die voor zijn vrienden door het vuur zou gaan. Bedankt voor de Tomorrowlands, het doorsturen van de templates voor vanalles en nog wat en alle schitterende momenten die we mochten beleven.

Zonder mijn familie had dit verhaal waarschijnlijk ook een andere wending gekend. Bedankt mama en papa om mij te stimuleren in alles wat ik onderneem en om er te zijn als het soms niet volledig uitdraait zoals ik gedacht had. Ik kijk enorm op naar jullie en alles wat jullie gerealiseerd hebben en kan alleen maar hopen om

daar ooit een fractie van te bereiken. Nic, ik denk niet dat je beseft wat jij voor mij betekend hebt in deze laatste jaren en ik denk ook niet dat ik het ooit in woorden zal kunnen uitdrukken. Ik ben heel dankbaar om een broer en beste vriend zoals jou te hebben. Bedankt ook aan Filip, Sabine, Mémé, Michiel en An-Sofie voor de zovele gezellige momenten.

Tot slot wil ik nog een heel speciaal iemand in mijn leven bedanken: Stephanie. Bollie, ik prijs mijzelf gelukkig dat ik jou al bijna 10 jaar in mijn leven heb. Dit laatste jaar heeft veel van mijn tijd ingenomen, maar zonder klagen of zagen was je er steeds om mij te ondersteunen in de moeilijke momenten en zo veel stress van mij weg te nemen. Bedankt om mijn leven in deze voorbije 10 jaar beter te maken en ik kan al niet wachten op wat de rest van ons leven in petto heeft. Ik hou van jou!

Gent, januari 2018
Thomas Vanhove

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

API	Application Programming Interface
AWS	Amazon Web Services

B

BSON	Binary-encoded JSON

C

CEP	Complex Event Processor
CoT	City-of-Things
CPU	Central Processing Unit
CQL	Cassandra Query Language

E

(E)ER	(Enhanced) Entity-Relationship
ESB	Enterprise Service Bus
ETL	Extract-Transform-Load

F

FIRE Future Internet Research and Experimentation

G

GB Gigabyte
GCE Google Cloud Environment
GENI Global Environment for Network Innovations
GHz Gigahertz

H

HDFS Hadoop Distributed File System
HPCC High Performance Computing Cluster
HQL Hibernate Query Language

I

IaaS Infrastructure-as-a-Service
ILP Integer Linear Programming
IoT Internet-of-Things

J

JSON JavaScript Object Notation

M

MB Message Broker

N

NCCS NASA Center for Climate Simulation

NoSQL Not-only SQL

O

OGM Object-Grid Mapping

ORM Object-Relational Mapping

OSGi Open Services Gateway initiative

OSN Online Social Network

P

PaaS Platform-as-a-Service

R

RAM Random-Access Memory

RDBMS Relational Database Management System

REST Representational State Transfer

S

SLA Service Level Agreement
SQL Structured Query Language

T

TB Terabyte

Z

ZB Zettabyte

Samenvatting
– Summary in Dutch –

Nu er meer en meer apparaten verbonden zijn met het Internet is de hoeveelheid data die gegenereerd wordt in de laatste jaren drastisch toegenomen. Recente schattingen tonen aan dat er elke minuut bijna 2MB aan data gecreëerd wordt voor elke persoon op Aarde. Tegelijkertijd hebben de vooruitgang in cloudverwerking en verlaagde opslagkosten ervoor gezorgd dat bedrijven op een goedkope manier deze data kunnen verzamelen en opslaan. Dit heeft tot gevolg dat er tegen 2020 ongeveer 40 miljard TB aan data zal opgeslagen zijn. Om deze hoeveelheden data binnen een redelijke termijn te verwerken zijn er nieuwe analysetechnologieën nodig die gebruik maken van computerclusters. Het nadeel aan deze batch raamwerken is dat de resultaten van de analyse pas beschikbaar zijn na de volledige verwerking van alle input. Dit maakt deze systemen ongeschikt voor real time toepassingen. Een tweede generatie aan analysetechnologieën focust zich op stream analyses die data verwerken van zodra deze in het systeem komt. De algoritmes om inzichten te genereren in de data worden echter alsmaar complexer en sinds kort is er een nood ontstaan voor hybride verwerkingsoplossingen die data zowel in real time als in batch kunnen verwerken.

De Lambda architectuur is zo een hybride oplossing die een batch- en streamtechnologie combineert in een twee-lagige architectuur. Data wordt eerst verwerkt door de online laag, ondersteund door een streamtechnologie, en op regelmatige tijdstippen wordt dezelfde data verwerkt door de offline laag. Dit laat een systeem toe om snel inzichten te verwerven op recente data, zonder de mogelijkheid te verliezen om historische data te raadplegen. Het beheer van de synchronisatie tussen de twee lagen is van cruciaal belang, zeker bij het bijwerken en vervangen van de resultaten. Als dit niet goed beheerd wordt, dan kan er informatie verloren gaan of kan er data meerdere keren opgeslagen worden in het systeem. Op dit moment zijn er geen oplossingen beschikbaar die garanties kunnen bieden dat dit niet zal gebeuren. Zelfs bij de oplossing die aangebracht wordt door de originele bedenker van de Lambda architectuur, Nathan Marz, kan er redundante informatie opgeslagen worden. Dit maakt de gehele oplossing ongeschikt voor applicaties die precieze resultaten vereisen.

De eerste uitdaging die dit proefschrift aangaat, is het ontwerpen en implementeren van een Lambda architectuur die zich nooit in een (tijdelijke) staat van verlies of redundantie van informatie bevindt. Dit doel wordt bereikt door data een label toe te kennen vooraleer deze verwerkt wordt. Dit label laat het systeem toe

om groepen data te volgen doorheen de verwerking en zo relevante resultaten bij te werken en te vervangen. De voorgestelde implementatie is ook zelfherstellend: wanneer redundantie of het verlies van informatie manueel wordt geïntroduceerd in het systeem, herstelt het zich altijd na een paar iteraties van de offline laag.

Los van deze beheersuitdagingen in hybride oplossingen, zijn verwerkingstechnologieën in het algemeen ook strengere eisen gaan stellen aan de opslagtechnologieën om zo sneller data te kunnen wegschrijven of uit te lezen. Er kon niet meer aan deze eisen voldaan worden met een klassiek relationeel databasemanagementsysteem (RDBMS) en zo zag een nieuwe generatie opslagtechnologieën het licht: NoSQL. Deze nieuwe technologieën zijn ontwikkeld om horizontaal te schalen om zo betere prestaties te kunnen leveren. Nieuwe applicaties kunnen zo ontwikkeld worden met deze nieuwe generatie in het achterhoofd, maar oudere applicaties zijn gebonden aan hun RDBMS en kunnen dus niet onmiddellijk gebruikmaken van deze nieuwe technologieën. Dit proefschrift stelt een systeem voor dat oudere applicaties toelaat om de voordelen van NoSQL technologieën te benutten. Het is wel belangrijk om op te merken dat veel van deze oudere applicaties in een productieomgeving draaien en zodoende moeten code-aanpassingen en onderbrekingen vermeden worden. Het transformatiealgoritme dat wordt voorgesteld, maakt gebruik van de Lambda architectuur. In de offline laag wordt een momentopname van de database getransformeerd aan de hand van een canonisch model, terwijl updates van de momentopname via de online laag worden verwerkt. Dit zorgt ervoor dat er geen onderbrekingen nodig zijn in de diensten die door de applicatie geleverd worden. Daarboven kunnen ook code-aanpassingen in de applicatie vermeden worden door de applicatie nog steeds in originele query-taal databewerkingen te laten uitvoeren. Deze queries worden dan door de online laag vertaald naar de correcte nieuwe technologie. De resultaten tonen aan dat een oudere applicatie baat kan hebben bij het overschakelen naar een NoSQL technologie, maar ook dat de continue vertaling van de online laag voor een minimale overheadkost zorgt die binnen de interactiviteitsgrenzen ligt.

De populariteit van de NoSQL oplossingen en de grote hoeveelheid aan oplossingen hebben ervoor gezorgd dat er een specialisatie is opgetreden binnen de NoSQL technologieën. Ze kunnen onderverdeeld worden in vier categorieën (key-value, document, column-oriented, graph) die elk beter of minder geschikt zijn in bepaalde situaties. Meer dan ooit is het dus belangrijk om een juiste opslagtechnologie te kiezen. Recente applicaties maken steeds meer gebruik van verschillende datatypes. Die verschillende datatypes kunnen dus beter elke ondergebracht worden in een verschillende technologie in de plaats van allemaal tesamen in één opslagtechnologie. Een applicatie die verschillende data types in verschillende opslagtechnologieën opslaat wordt polyglot persistent genoemd. Natuurlijk wordt het beheer van zo een polyglot persistente omgeving complexer naarmate het aantal ondersteunde technologieën oploopt. Dit proefschrift brengt een oplossing aan die een applicatie beschermt voor de complexiteit van polyglot persistence via een abstractielaag. Daarenboven wordt het schema van de data stores geoptimaliseerd om zo de overheadkost van de abstractielaag te mitigeren en maximale winst te halen uit de polyglot persistente omgeving. De optimalisatie wordt gebaseerd op

de extractie van frequente query-patronen en past het schema zodanig aan dat deze queries sneller kunnen uitgevoerd worden. Het toepassen van deze optimalisaties gebeurt via het eerder vermelde transformatiealgoritme.

De bijdragen van dit proefschrift zijn ook toegepast op realistische gevallenstudies van het smart city project City-of-Things (CoT). In CoT wordt er data verzameld van verschillende sensoren die verspreid in een stad zijn opgesteld. Gebruikers kunnen toegang krijgen tot deze data, maar ook tot een omgeving dat hun toelaat om aangepaste analyses te doen op die data. Het transformatiealgoritme wordt in deze context gebruikt worden om data aan te leveren in het aangevraagde formaat, terwijl de backend van het CoT systeem geoptimaliseerd kan worden met de eerder beschreven schema optimalisaties op basis van frequente queries van gebruikers. De Lambda architectuur is de drijvende kracht achter de transformaties, maar kan ook aan de gebruikers aangeboden worden om over de data te analyseren.

Aangezien het werk van dit proefschrift zich afspeelt in big data omgevingen, zijn de hierboven beschreven oplossingen geëvalueerd in experimentele setups met big data technologieën. Het manueel opzetten van deze omgevingen met geclusterde technologieën is een tijdrovende en complexe taak. De laatste contributie van dit doctoraat is daarom het ontwerp en de implementatie van het Tengu platform. Tengu laat onderzoekers toe om snel en geautomatiseerd big data omgevingen op te zetten voor experimenten. Het Tengu platform is in de voorbije jaren in verschillende projecten gebruikt geweest en, na een valorisatietraject, heeft dit ook geleid tot een spin-off van de Universiteit Gent en imec.

Summary

Over the past decade, the amount of data that is being generated has increased drastically as more and more devices are connected to the Internet. Estimates say that close to 2MB of data is created per minute for every person on Earth. At the same time, advances in cloud processing and decreasing storage costs have allowed companies to capture and store this data in a cheap way. This has led to a situation where, by 2020, worldwide data storage will take up 40 billion Terabytes. Processing these large volumes of data requires new analysis frameworks that use the computing power of clustered servers to generate insights within a reasonable time. The drawback of these batch frameworks is that analysis results are only available after the process for the entire batch of input data is completed, making them unfit for systems with (near) real-time constraints. A second generation of processing frameworks, centered around stream analysis, process data as soon as it is captured and scale across clusters to cope with higher velocity streams. However, as algorithms for generating insights grow more complex, a need arises for hybrid solutions that are able to process data in (near) real-time while also being able to analyze historic data.

The Lambda architecture is a hybrid framework that combines a batch and streaming technology in a two-layered architecture. Captured data is immediately processed by an online layer, powered by a streaming technology, and at regular intervals the data is processed more thoroughly by the offline layer. This allows a system to quickly generate (near) real-time superficial insights, which are updated over time by more precise results. Managing the synchronization between the on- and offline layer, i.e., when results are updated or replaced, is critical. If not managed well, the system is vulnerable to (temporary) states of information redundancy and loss. Currently, there are no solutions that guarantee a loss-less and non-redundant Lambda implementation. Even the proposed solution by the creator of the Lambda architecture concept, Nathan Marz, condones a temporary state of partial redundancy, making the solution unusable for applications that require correct results. The first challenge this dissertation tackles is the design and implementation of a Lambda architecture without any (temporary) information loss or redundancy. By tagging data as soon as it enters the system, it can be traced throughout the on- and offline layers and storage. As the system is aware which tags are currently being processed it can update or replace relevant information in the on- and offline data stores in such a way that it eliminates the possibility of redundant or lost information. The proposed solution is also fault-tolerant, i.e., when information loss or redundancy is introduced manually through human fault

or otherwise, the stored information self-corrects after a couple of batch execution runs.

Regardless of the management challenges in hybrid solutions, the processing frameworks have put stringent demands on storage solutions requiring them to provide fast read and/or write access in order to keep up with the processing capabilities. These demands have long exceeded the means of classic relational database management systems (RDBMS) and therefore spawned a new generation of storage solutions, commonly identified as NoSQL. These technologies are designed to scale and provide better performance in doing so. While new applications can be designed with this new generation of technologies in mind, legacy applications that are tightly bound to their classic RDBMS cannot. A second challenge that is addressed in this dissertation is the design of a system that would allow legacy applications to benefit from the advantages of NoSQL data stores. The devised solution must consider that many of these applications run in a production environment and as such, changes to the application's code and downtime of the application must be minimized or even eliminated completely. A transformation algorithm is developed that uses the Lambda architecture to transform a snapshot of the data store schema and data through a canonical model in an offline layer, while updates after the snapshot are processed by the online layer. In other words, the data store of the application can be transformed without any downtime. An additional benefit is that, after the snapshot is transformed, the online layer can continue to transform queries of the application in its original query language, i.e., no code changes are needed on the application's side. Results show that the transformation to a NoSQL data store can hugely benefit query latency, depending on the chosen data store, but also that the continuous transformation can be done with limited overhead, well within interactivity bounds.

The popularity of NoSQL data stores and the amount of possible solutions for data storage led to a specialization of the NoSQL technologies. Divided into four categories (key-value, document, column-oriented, graph), each is better or less suited for certain data types or uses. Thus, the correct choice of a data store is is the utmost importance for the query performance of an application. Considering the amount of different data types one application handles has gone up as a direct consequence of the increasing volumes of data generated, different data types within one application would benefit from using different data store technologies. An application that stores its data in different storage technologies is referred to as being polyglot persistent. However, supporting a polyglot persistent storage environment with different technologies, and their different query languages accordingly, quickly gains in complexity as the number of supported technologies increases. In this dissertation a solution is proposed that shields the complexity of polyglot persistence from the application through an abstraction layer. The overhead of the abstraction layer is mitigated through the optimization of the data schema in order to maximize the benefit of the polyglot persistent environment. An algorithm is defined to mine frequent query patterns used by applications or users and optimizing the schema to facilitate those queries. Enacting the changes on the data schema is done by the transformation algorithm mentioned previously.

The contributions of this dissertation are applied in real-life use cases as part of the smart city project City-of-Things (CoT). As a smart city testbed, CoT gathers data from a variety of sensors spread across a city environment. Users of the testbed get access to the sensor data as well as an automated analytics environment. The transformation algorithm is used to provide users with data in the correct data format, while the schema optimization for polyglot persistent environments is applied to the backend of CoT ensuring data delivery to users is optimized. The Lambda implementation powers the transformation approaches, but can also be applied to custom analytics by the users of the testbed.

As the work of this dissertation is set in big data environments, many experimental setups were used for the evaluation of the proposed solutions. Setting up these big data experimental environments manually is a time consuming and inconvenient task. As many of the used technologies are clustered in nature, the setup only becomes more complicated. A final contribution of this PhD is therefore the design and implementation of a platform, Tengu, to facilitate the experimentation of big data solutions. The platform provides automated setups of big data environments for researchers that are pre-configured to custom specifications. The Tengu platform has contributed to several projects and, ensuing a valorisation process, has led to a spin-off company of Ghent University and imec.

1 Introduction

"Success is a science; if you have the conditions, you get the result."

–Oscar Wilde (1854 - 1900)

1.1 The Impact of Big Data

The cloud has gained a lot of popularity in these last years and is an ever growing concept. One of the main reasons for companies to move or develop their applications in the cloud is the elastic scalability provided in a cloud setting. It allows for applications to be scaled up or down (i.e., use more or less computational resources) without any downtime. Additionally, this scalability is often delivered on-demand resulting in a *pay-per-use* model allowing the companies to cut costs instead of investing in their own hardware. Not only have computing costs gone down drastically with this pay-per-use model, but storage costs as well. In 1980 1 GB of storage would have cost $700,000$\$, but by 2014 this cost has been reduced to about 0.03 \$/GB [1]. This means the cost of storage has fallen by over 99.99999% in just about 35 years. As these storage costs continue to decline while more and more devices are connected to the internet, more data than ever is being generated and captured by companies worldwide. Our digital universe, the collection of all data stored in the world, is continuously expanding at ever increasing rates. By 2020 the digital universe is predicted to contain over 40 ZB (1 Zettabyte = 1 billion Terabyte) of data [2]. That is over $5,200$ GB of data, or about 30 hours of 4 K footage (4096 x 2160), per person alive in 2020.

This new computing paradigm, trying to process all this data, has dawned the age of big data analysis. There still remains some confusion over what makes a data set big data and many definitions have tried to capture exactly what it encompasses. One definition considers a data set to be big data when traditional processing and storage solutions no longer suffice, but require parallel software running in clusters of tens, hundreds or even thousands of servers [3]. However, the concept of big data is still evolving and definitions therefore become dated rather quickly. A definition that is now gaining in general acceptance is the definition by means of the 3 V's by Gartner. It defines big data as a data set that adheres to at least two of the following characteristics:

- **Volume:** a data set needs to be large enough. The threshold for a data set to be large enough is also still growing as hardware for both storage and processing is still improving. In general, a data set is large enough when a cluster of computing power is needed to process it in a reasonable time.
- **Velocity:** the rate at which data is generated or arrives in the data set must be significantly high, again so that a cluster is required to process it in (near) real-time.
- **Variety:** the data in the data set is unstructured or is a combination of different data formats making it hard to process.

The benefit of this definition is that it can be extended with new V's as the domain evolves. For example, in recent years a new V, **Veracity**, has been introduced and it indicates that incoming data cannot always be trusted. Examples of big data projects in the academic domain can be found in the Human Genome Project [4] and NASA Center for Climate Simulation (NCCS) [5], while social network sites, such as Facebook and Twitter, internet retailer Amazon, and search engine Google, are great examples of companies in the private sector dealing with big data.

In order to process these big data sets, many computational frameworks have been developed. First came the batch frameworks, designed to process large data sets in parallel (**Volume**). The best-known batch framework is MapReduce, originally developed by Google, but made popular by its open-source implementation Apache Hadoop [6]. Another more recent and increasingly popular batch framework is Spark [7]. Spark is proven to execute certain programs up to 100 times faster than Hadoop in memory or 10 times on disk. However, as data sets started growing faster because of higher arrival rates (**Velocity**), batch frameworks could no longer provide insights within a reasonable time limit. Specific types of applications, such as sensor networks, social media, and network monitoring created the need for (near) real-time or stream processing [8]. The most notable examples are Apache Storm [9] and Heron [10]. Storm provides a continuously running topology made up of singular nodes, called bolts, thus creating custom analysis

streams. Heron works in a very similar way and is considered to be the successor to Storm. Both were orginally developed at Twitter and later made available under an open source license. A final category are the hybrid frameworks, which do not introduce new computational frameworks but rather combine existing batch and streaming technologies. An example of such a hybrid solution is the Lambda architecture [11].

These computational frameworks, aimed at large scale processing, also have put stringent demands (e.g., fast access) on the data storage solutions they use. Over the past decade, these demands have exceeded the capabilities of classic relational database management systems (RDBMS). Therefore, new storage systems have been designed to scale horizontally, providing read/write operations distributed over many servers [12]. Many of these new technologies can be categorized as NoSQL, which stands for *Not only SQL*. Contrary to the classic relational databases, they provide easy scaling and performance advantages in specific scenarios, depending on the chosen NoSQL data store [13]. Additionally, they provide a more flexible or even schema-less data model, allowing rapid changes in the model. The popularity of these data stores can be measured by the sheer amount of solutions available. However, this does not mean relational databases no longer have a role to play in the big data domain [14]. An example is Google's F1 hybrid database [15] supporting their AdWords business, an ecosystem with hundreds of concurrent applications and thousands of users sharing the database, over 100TB in size in 2013.

The amount of possible solutions for data storage led to a specialization of these data stores in order to distinguish themselves from each other. A NoSQL data stores can be divided in one of 4 categories: key-value stores, document stores, column-oriented stores, and graph databases, where each category is more or less suitable for different types of data or for different applications [16]. Thus, a correct choice in data store is paramount for the optimal performance of the application. However, as previously mentioned, applications often work with different types of data (**Variety**), e.g., e-commerce platforms. Combined with the plethora of NoSQL solutions available it would be beneficial for an application to use multiple data stores simultaneously for the different data types it handles. This is often referred to as polyglot persistence and is considered to be the most profound consequence of the NoSQL domain [17].

The above paragraphs only provide a summary of the available technologies for data processing and storage. Figure 1.1 gives a high-level overview of relevant technologies and company solutions in the big data landscape for purposes such as analysis, storage, but visualization and infrastructure among others as well. It visualizes the impact the big data paradigm has had on data processing and storage approaches: the way applications work with data has irrevocably changed allowing them to create deeper insights faster onto data sets.

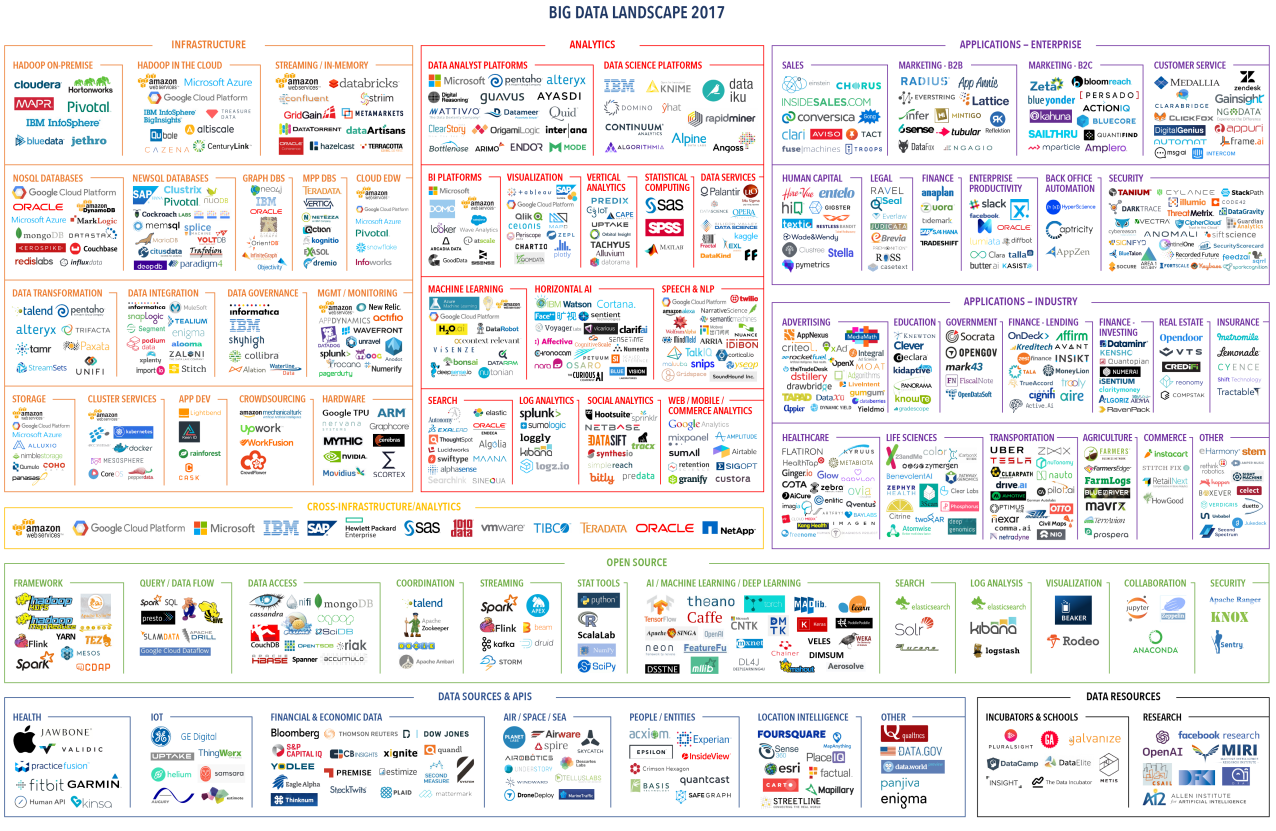

Figure 1.1: Technology overview of the big data landscape in 2017 [18].

1.2 Problem Statement

In the previous section polyglot persistence was introduced as a situation where an application stores data in more than one data store technology based on data type, usage, or other qualifying factors. The ultimate goal of polyglot persistence is to optimize data access for applications through minimizing query latency by storing data in the most optimized technology. However, this optimized data access does come with a significantly higher complexity. For newly created applications, this means that an application needs to support different query languages in order to communicate with the different storage technologies. Additionally, each data store technology needs to be managed for optimal configuration (e.g., scaling, replication). For legacy applications, there is an additional dimension to this complexity as data is already stored in a, potentially sub-optimal, data store. In order to get a legacy application into polyglot persistence, existing data needs to be transformed and migrated to the new data stores on top of changing the application to support the different querying languages, which is made even harder in a live environment where no downtime of the application is allowed.

This dissertation aims to address the data management complexity in a polyglot persistent environment for both new and legacy applications. More specifically, answers to the following research questions are formulated:

1. **How can the complexity of a polyglot persistent environment be mitigated for applications?**

Applications should be able to benefit from lowered query latency provided by polyglot persistence without the perceived complexity. In other words, this dissertation aims to find a method to shield applications from the complexity of a polyglot persistent environment while increasing data access speed.

2. **How can data be transformed between data store technologies in an automated and extensible way?**

While the answer to Research Question 1 would aid new applications, applying it to legacy applications with a, potentially sub-optimal, data store would still incur a high impact on the stored data and the application itself. A solution for legacy applications in polyglot persistence should therefore include a transformation of data from the original, single data store towards other data store technologies. Furthermore, this solution should be automated, i.e., require minimal user interaction, and be extensible with new data store technologies as to be future-proof.

3. **Can the impact of the transformation on the application changes be eliminated?**

When the data of an application is transformed and moved to another data store technology this results in a necessary code change for the application to support the

new data model, if any, and the new querying language. Eliminating this necessary change would allow for a faster turnover towards polyglot persistence for legacy applications.

4. **Can the polyglot persistent data model be optimized based on how data is retrieved by an application?**

The data model of a data store is designed and built in such a way that it optimizes data access for a specific application. This should also apply to a polyglot persistent environment where parts of the data model are contained within disparate storage technologies, as it could potentially further improve data accessing times for an application. Therefore, the solution to Research Question 1 should also contain a dynamic aspect where it takes into account information from the application on how a data is retrieved to tailor the data model to the application.

1.3 Research Contributions

This section aims to provide an overview of the contributions of this dissertation with regard to the research questions posed in Section 1.2.

A use case in big data where applications would touch upon the principles of polyglot persistence are hybrid frameworks, such as the Lambda architecture. The architecture was introduced by Nathan Marz as a hybrid solution for both off- and online analysis [11]. The online analysis provides (near) real-time insights on streaming data sets, while the offline component generates deeper insight over the entire history of the data. The results from the layers are stored in disparate storage solutions effectively putting an application, using the results, in a polyglot persistent environment. However, managing this entire hybrid environment becomes significantly more complex. As the same data is always processed twice, once in the online and once in the offline layer, and stored in two different data stores, this leaves the system vulnerable to (temporary) information loss and redundancy. The original creator, Nathan Marz, acknowledges this challenge but never provides a true solution to managing this synchronization complexity and tolerates a state of temporary redundancy [11]. The first contribution of this dissertation therefore is:

1. **A technology-independent framework of a true Lambda architecture with no (temporary) information loss or redundancy.**

While this contribution might not answer to any of the research questions directly, it does provide a good use case for polyglot persistent applications and the basis of an approach well-suited for the transformation of data.

The next contribution formulates an initial answer to Research Question 1 as follows:

2. **Definition and design of a canonical model that acts as a generic representation of a data schema, independent of underlying technologies.**

The canonical model acts as a shielding layer between the application and the data stores supplying the application with a technology-independent overview of its data schema. As such, the heterogeneity of polyglot persistence is shielded from the application. The question remains how an application is then able to communicate with the different data sources to retrieve data. The answer can be found in the third contribution of this dissertation:

3. **An extensible transformation algorithm between data storage technologies supporting continuous transformation.**

The architecture for the transformation of a data store's schema and data combines both the first and second contribution: it builds on the Lambda architecture, where a snapshot of the data store is transformed offline and live queries in the online layer, using the canonical model to map the data schema between data store technologies. Additionally, by keeping the online layer of the Lambda architecture active, queries can be continuously transformed from the original query language to the transformed data stores. In other words an application is able to communicate with the polyglot persistent data storage with one query language. For a legacy application this can be the query language of the single data store. This continuous transformation provides an answer to Research Questions 1 and 3: applications can communicate with a polyglot persistent environment, aided by the canonical model, in a single query language, which for legacy applications eliminates need for changes in the code.

The definition of the transformation algorithm answers Research Question 2 as it defines an extensible approach to transformation between data store technologies aided by the canonical model. The goal of the dissertation was not to build an implementation of the transformation algorithm for all data store technologies and all their features, but rather design an extensible approach for transformation through a canonical model. This framework ought to give enough flexibility to extend support to a majority of data storage solutions and their feature sets.

None of the aforementioned contributions consider input from the application querying the data store, but rather build on information provided by data stores themselves. As applications use, i.e., query, data stores in unique ways, this information could be used to optimize and tailor the canonical model. Specifically, this dissertation looks into potential relations between data inside a data model based on live queries by the application in an attempt to optimize or eliminate complex queries. This yields the following and final contribution as a response to Research Question 4:

4. **A framework for the detection of potential relations between data through sequential pattern mining on live queries.**

1.4 Dissertation Outline

This dissertation bundles several publications that have been written over the course of this PhD. These publications were chosen as they represent a thread of research topics that have shaped the work performed over the past years. This chapter serves as an introduction to several of these topics. Other publications that have not been included as a part of this dissertation are listed in Section 1.5. Aside from research publications, this PhD has also lead to a spin-off company of Ghent University and imec. More information on the company can be found in Section 1.6.

Chapter 2 presents a technology-independent implementation of the Lambda architecture. It provides a solution to the synchronization issue between the off- and online layer, and completely removes any state of temporary loss or redundancy of information. By tagging data as soon as it enters the environment, it can be traced throughout the on- and offline analysis and storage. As the system is aware which tags are currently being processed it can clear relevant information in the on- and offline data stores and manage the synchronization between those data stores eliminating the possibility of redundant and lost information.

In Chapter 3 an approach for the transformation between two data store technologies is introduced. The big data paradigm introduced new technologies and ways to store data, but many legacy applications are tightly bound to their current storage solution and not equipped to deal with these new technologies. The solutions provided in this chapter allow such applications to finally benefit from this new generation of data stores. The chapter describes an extensible transformation algorithm between two data store technologies that is capable of transforming both a data store schema and the actual data. A new canonical model is introduced that abstracts specific data storage technologies and allows for the extensibility of the proposed approach. Applying the transformation to legacy data stores results show that the schema and data can be correctly transformed and, depending on the choice of storage technology, significant gains can be made in query latency. Additionally, by using the Lambda architecture implementation from Chapter 2, the transformation solution in this chapter does not impose any changes to the legacy application. The work is a continuation of the work described in Appendix A, which has been added to this dissertation for completeness sake.

Chapter 4 builds on the advancements made in Chapter 3. Whereas Chapter 3 introduces direct transformation between two data store technologies, this chapter tackles challenges in a polyglot persistent environment and assumptions that are made in the previous chapter. The previously defined canonical model is extended with a technology layer, effectively giving the model a notion of the concept of polyglot persistence. It also allows the canonical model to act as an abstraction of the complex polyglot storage environment. However, as with any abstraction layer, it introduces an overhead to the general operations, so assurances are needed that

the overhead is significantly smaller than the benefit gained of using it. First of all, there is the benefit of using the correct technology, or set of technologies, which was demonstrated in Chapter 3. Secondly, the transformation algorithm optimized the transformed data store by relying on selective denormalization for defined relations in the data schema, ultimately benefiting the query latency. The assumption was that these relations were correctly defined in the data schema, but in many realistic cases these relations are not or cannot be defined. This can be due to omission when building the data schema for a data store, or because the data store technology does not support the explicit expression of a relation in its data schema, which is often the case in NoSQL data stores. This chapter introduces an optimization approach that learns these relations based on the queries from the applications. The detected relations are then added to the canonical model after which the changes are reflected in the storage environment. Combined the abstraction of polyglot persistence provided by the new canonical model is leveraged to optimize the data schemas of all involved data store technologies.

Finally, Chapter 5 combines the contributions from the previous chapters into real-life use cases applied in the City-of-Things (CoT) project. City-of-Things is a smart city testbed that gathers data from different sensors spread out across the city of Antwerp (Belgium). This chapter describes the architecture of the CoT platform and provides several use cases on how users are able to interact with the platform. Users can get direct access to the sensor data but also to an analysis environment in which they can build custom analysis components and applications on the data. The contributions from Chapter 3 provides users access to data in whichever format available. The transformation can be used to prepare the data in the right format before it is sent to the users. Based on how users interact with the data, the back-end of the CoT platform can also be optimized by the solutions detailed in Chapter 4. As mentioned before, the solutions in Chapters 3 - 4 make use of the generic implementation of the Lambda architecture defined in Chapter 2, but the implementation can also be provided to users directly that wish to use it in their analysis.

The work in Chapters 2 - 5 is set in big data environments. The setup of such environments in practice is often cumbersome and manual task complicated by the clustered nature of many of the technologies available. Early on in this PhD the complexity of these environments put a strain on experiments because of the discrepancy between the setup and the execution effort. As a result the Tengu experimentation platform was created to automatically set up big data environments for researchers. Since its creation the Tengu platform has contributed to several other dissertations, research publications, and different projects with Ghent University and its industry partners [19, 20]. Appendix B provides further insight into the Tengu platform and its capabilities.

1.5 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of international conferences. The following list provides an overview of the publications during my PhD research.

1.5.1 A1: Journal publications indexed by the ISI Web of Science "Science Citation Index Expanded"

1. Femke De Backere, **Thomas Vanhove**, Emanuel Dejonghe, Matthias Feys, Tim Herinckx, Jeroen Vankelecom, Johan Decruyenaere, Filip De Turck. *Platform for Efficient Switching between Multiple Devices in the Intensive Care Unit.* Published in Methods of Information in Medicine, Schattauer, Volume 54, Issue 1, Pages 5-15, January 2015. doi:10.3414/ME13-02-0021.

2. **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Managing the Synchronization in the Lambda Architecture for Optimized Big Data Analysis.* Published in IEICE Transactions on Communications (ToC), Volume 99, Issue 2, Pages 297-306, February 2016. doi:10.1587/transcom.2015ITI0001.

3. Femke Ongenae, **Thomas Vanhove**, Femke De Backere, Filip De Turck. *Intelligent task management platform for health care workers.* Published in Informatics for Health & Social Care, Taylor & Francis, Volume 42, Issue 2, Pages 122-134, February 2017. doi:10.3109/17538157.2015.1113178.

4. **Thomas Vanhove**, Merlijn Sebrechts, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Data Transformation as a means towards Dynamic Data Storage and Polyglot Persistence.* Published in International Journal of Network Management (IJNM), Wiley-Blackwell, Volume 27, Issue 4, July 2017. doi:10.1002/nem.1976.

5. **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Sequential Pattern Mining for Data Storage Optimization in Polyglot Persistent Environments.* Submitted to the Journal of Network and Computer Applications, September 2017.

6. **Thomas Vanhove**, Merlijn Sebrechts Gregory Van Seghbroeck, Tim Wauters, Philip Leroux, Filip De Turck. *City of Things: Smart Cities beyond Open Data.* Submitted to IEEE Communications Magazine, IEEE, September 2017.

1.5.2 P1: Proceedings included in the ISI Web of Science "Conference Proceedings Citation Index - Science"

1. **Thomas Vanhove**, Philip Leroux, Tim Wauters, Filip De Turck. *Towards the Design of a Platform for Abuse Detection in OSNs using Multimedial Data Analysis.* In proceedings of the IFIP/IEEE Symposium on Integrated Network Management (IM), Ghent, Belgium, Pages 1195-1198, May 2014.

2. **Thomas Vanhove**, Jeroen Vandensteen, Gregory Van Seghbroeck, Tim Wauters, Filip De Turck. *Kameleo: Design of a new Platform-as-a-Service for Flexible Data Management.* In proceedings of the IFIP/IEEE Network Operations and Management Symposium (NOMS), Krakow, Poland, Pages 1-4, May 2014. doi:10.1109/NOMS.2014.6838331.

3. **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Filip De Turck. *Live Datastore Transformation for optimizing Big Data applications in Cloud Environments.* In proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM), Ottawa, Canada, Pages 1-8, May 2015. doi:10.1109/INM.2015.7140270.

4. **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Filip De Turck, Brecht Vermeulen, Piet Demeester. *Tengu: an Experimentation Platform for Big data Applications.* In proceedings of the IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), Columbus, OH, USA, Pages 42-47, June 2015. doi:10.1109/ICDCSW.2015.19.

5. Merlijn Sebrechts, **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration.* In proceedings of the IEEE International Conference on Mobile Services (MS), San Francisco, CA, USA, Pages 42-47, June 2016. doi:10.1109/MobServ.2016.31.

6. Merlijn Sebrechts, Sander Borny, **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Model-driven Deployment and Management of Workflows on Analytics Frameworks.* In proceedings of the IEEE International Conference on Big Data (BIG DATA), Washington, DC, USA, Pages 2819-2826, December 2016. 10.1109/BigData.2016.7840930.

7. Leandro Ordonez-Ante, **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Filip De Turck. *Interactive Querying and Data Visualization for Abuse Detection in Social Network Sites.* In proceedings of

the International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, Pages 104-109, December 2016. doi:10.1109/ICITST.2016.7856676.

8. Leandro Ordonez-Ante, **Thomas Vanhove**, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, Filip De Turck. *Dynamic Data Transformation for Low Latency Querying in Big Data Systems.* Submitted to CloudCom 2017, Hong Kong, December 2017.

1.5.3 C1: Other publications in international conferences

1. Pieter Bonte, Femke Ongenae, Jelle Nelis, **Thomas Vanhove**, Filip De Turck. *User-friendly and Scalable Platform for the Design of Intelligent IoT Services: a Smart Office Use Case.* In proceedings of the International Semantic Web Conference (ISWC), Kobe, Japan, Pages 1-4, October 2016.

1.6 Spin-offs

A final contribution that is important to mention in the context of this dissertation is Qrama. Qrama is a spin-off company of Ghent University and imec, founded by Thomas Vanhove and Gregory Van Seghbroeck, that has commercialized the Tengu platform. Tengu, detailed in Appendix B, was created as a toolset to automate big data experimentation for researchers. During the course of this PhD it became clear that Tengu is an answer to a much larger problem outside the world of academia. As a company Qrama brings a commercial version of Tengu to the big data market.

1. Qrama, Besloten Vennootschap met Beperkte Aansprakelijkheid, **Thomas Vanhove**, Gregory Van Seghbroeck. *BE0660.556.043*, July 2016.

References

[1] M. Komorowski. *A History of Storage Cost*, March 2014. http://www.mkomo.com/cost-per-gigabyte-update (Last Visited January 9, 2018).

[2] J. Gantz and D. Reinsel. *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east*. IDC iView: IDC Analyze the Future, 2007:1–16, 2012.

[3] A. Jacobs. *The Pathologies of Big Data*. Commun. ACM, 52(8):36–44, August 2009. Available from: http://doi.acm.org/10.1145/1536616.1536632, doi:10.1145/1536616.1536632.

[4] F. S. Collins, M. Morgan, and A. Patrinos. *The Human Genome Project: Lessons from Large-Scale Biology*. Science, 300(5617):286–290, April 2003. doi:10.1126/science.1084564.

[5] J. L. Schnase, G. Tamkin, D. Fladung, S. Sinno, and R. Gill. *Federated observational and simulation data in the NASA Center for Climate Simulation Data Management System Project*. In Proceedings of the iRODS User Group Meeting 2011: Sustainable Policy-Based Data Management, Sharing, and Preservation, pages 17–18, 2011.

[6] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[8] J. Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 2010.

[9] *Apache Storm*. http://storm.apache.org/ (Last Visited January 9, 2018).

[10] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. *Twitter Heron: Stream processing at scale*. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 239–250. ACM, 2015.

[11] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[12] R. Cattell. *Scalable SQL and NoSQL Data Stores*. SIGMOD Rec., 39(4):12–27, May 2011. Available from: http://doi.acm.org/10.1145/1978915.1978919, doi:10.1145/1978915.1978919.

[13] Y. Li and S. Manoharan. *A performance comparison of SQL and NoSQL databases*. In Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on, pages 15–19. IEEE, 2013.

[14] C. Nance, T. Losser, R. Iype, and G. Harmon. *NoSQL vs RDBMS: Why there is room for both*. In SAIS 2013 Proceedings, 2013.

[15] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. *F1: A Distributed SQL Database That Scales*. Proc. VLDB Endow., 6(11):1068–1079, August 2013. Available from: http://dx.doi.org/10.14778/2536222.2536232, doi:10.14778/2536222.2536232.

[16] A. Haseeb and G. Pattun. *A review on NoSQL: Applications and challenges.* International Journal of Advanced Research in Computer Science, 8(1), 2017.

[17] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1 edition, August 2012.

[18] M. Turck. *Firing on All Cylinders: The 2017 Big Data Landscape*, April 2017. http://mattturck.com/bigdata2017/ (Last Visited January 9, 2018).

[19] S. Latre, P. Leroux, T. Coenen, B. Braem, P. Ballon, and P. Demeester. *City of things: An integrated and multi-technology testbed for IoT smart city experiments*. In Smart Cities Conference (ISC2), 2016 IEEE International, pages 1–8. IEEE, 2016.

[20] P. Bonte, F. Ongenae, J. Nelis, T. Vanhove, and F. De Turck. *User-friendly and scalable platform for the design of intelligent IoT services: a smart office use case*. In ISWC2016, the 15th International Semantic Web Conference, pages 1–4, 2016.

2

Managing the Synchronization in the Lambda Architecture for Optimized Big Data Analysis

T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck.

Published in IEICE Transactions on Communications, February 2016.

⋆ ⋆ ⋆

This chapter focuses on the implementation of the Lambda architecture. It was originally created by Nathan Marz as a hybrid concept that combines both offline and online analysis by processing data superficially in a fast speed layer first, and more thorough in a slower batch layer later. The ultimate goal is to allow applications faster access to insights on data without the loss of precision in the calculations. However, as data is processed twice, a synchronization challenge is introduced: once the data is analyzed by the batch layer, the corresponding information needs to be removed in the speed layer without introducing redundancy or loss of information. In this chapter we propose a new approach to implement the Lambda architecture concept independent of the technologies used for offline and online computing. A solution is provided to manage the complex synchronization introduced by the Lambda architecture and techniques to provide fault tolerance. The proposed solution is evaluated by means of detailed experimental results.

2.1 Introduction

Our digital universe is continuously expanding and predicted to contain 40 ZB (1 Zettabyte = 1 billion Terabyte) of data by the year 2020 [1]. Retrieving valuable information from these data sets through conventional methods becomes nearly impossible if time constraints apply. Moreover, most of these data sets consist of unstructured data, making the processing even more complex. A popular approach in the big data domain is to precompute views with big data processing technologies and let applications or users query this view instead of the entire data set. An important distinction is made in semantics: the entries in the original big data set are referred to as *data*, whereas the entries in the precomputed views are referred to as *information* [2]. Information is thus derived from data through the algorithms implemented in big data processing technologies.

These technologies can be divided in two types: batch processing, and stream processing. The best known batch processing approach is Map-Reduce, originally developed by Google [3], but made popular by its open-source implementation in Apache Hadoop [4]. Other popular solutions include Spark [5] and Flink [6]. The stream processing on the other hand, satisfies the processing needs of applications that generate data streams, such as sensor networks, social media, and network monitoring tools [7]. While batch processing analyzes an entire data set, stream processing does the analysis on a message to message basis. Important streaming analysis frameworks are Storm [8], S4 [9], and Samza [10].

The power of batch processing comes from the ability to access an entire data set during the computation, e.g., creating the opportunity for the detection of relations in the data. The drawback of batch processing is that all resulting information only becomes available after the execution is complete. This process can take hours or even days during which recent data is not taken into account. While stream processing lacks the overview of batch processing, it does allow for a (near) real-time analysis of data as it arrives in the system. The Lambda architecture is built upon a hybrid concept where during a batch analysis execution, in a batch layer, newly arriving messages are analyzed by a stream analysis technology, or speed layer [2]. This effectively harnesses the power of both approaches, giving an application a complete historic informational overview through the batch layer, stored in batch views, and (near) real-time information through its speed layer, stored in speed views. As soon as data is processed in the batch layer, the information is stored in a batch view and the corresponding information is removed from the speed view.

The Lambda architecture is clearly a very powerful concept, but it does pose several implementation challenges. First, as information is stored in two different views, the synchronization between batch and speed layer is key to providing applications and/or users with the correct information. If this is overlooked or

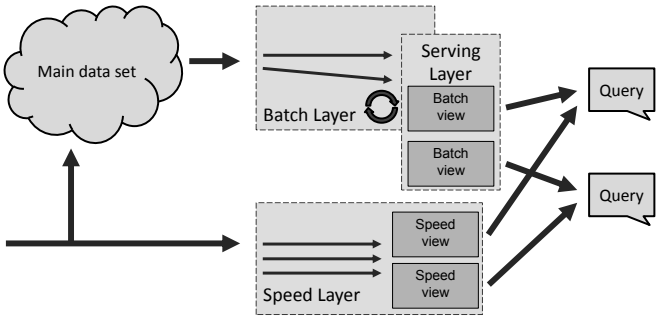

Figure 2.1: Conceptual overview of the Lambda architecture

ill-handled, information could be lost or redundantly stored for a period of time. Second, storing information across different data stores leaves the system in a state of polyglot persistence, creating the need for the aggregation of information from both the batch and speed views every time a query is sent by the application or users.

This chapter proposes a general implementation of the Lambda architecture concept without dependencies on the technologies used in the batch/speed layers or views. A proof of concept has been implemented as part of the Tengu platform, formerly known as Kameleo [11]. The chapter focuses on providing a generic solution for the synchronization challenges that arise during the implementation of the concept, but also proposes a solution for the aggregation challenge.

The remainder of this chapter is structured as follows: Section 2.2 discusses the Lambda architecture in depth. In Section 2.3 the synchronization challenge is discussed in detail and a solution is proposed. Section 2.4 explains how the system handles failures in the different layers. The implementation of the synchronization solution is detailed in Section 2.5. The experimental setup and results are provided in Section 2.6. In Section 2.7 initial steps are detailed towards a solution for the aggregation of polyglot persistent views. Finally, the conclusions are presented in Section 2.8.

2.2 Lambda architecture: overview and challenges

The aim of each data system is to answer queries for applications or users on the entire data set. Mathematically, this can be represented as follows [2]:

$$query \quad = \quad function(all\ data)$$

While in the era of Relational Database Management Systems (RDBMS) it was still possible to query the entire data set in real time, this is no longer the case

with big data sets [12]. Therefore, in big data analysis systems queries are already partially precomputed and stored in views to limit the applications' query latency. Expressed in terms of functions, this gives us the following:

$$\begin{aligned} view &= function(all\ data) \\ query &= function(view) \end{aligned}$$

It is here that Marz also makes a distinction between data and information [2]. Data is the rawest information from which all other information is derived and is perceived to be true within the system, in this case the main big data set. A big data system thus becomes the function that analyzes data through a programmed algorithm and stores the resulting information in a view. Queries thus no longer access data, but information stored inside views. According to Marz, these big data systems need to achieve several properties:

- **Robustness and fault tolerance:** a data system needs to behave correctly even in the event of software, machine, but also human failures.
- **Low latency reads and updates:** data or information needs to be available when an application or user needs it.
- **Scalability:** a data system needs to maintain a stable performance with increasing or decreasing load.
- **Generalization:** a data system needs to be applicable to a wide range of applications.
- **Extensibility:** the potential to add functionality with a minimal cost.
- **Ad hoc queries:** unanticipated information in a data set needs to be accessible.
- **Minimal maintenance:** limit the implementation complexity of the components.
- **Debuggability:** a data system needs to provide information allowing to trace how output was construed.

The Lambda architecture is built in layers each satisfying a subset of these properties.

As stated before, a big data system precomputes views on a big data set to reach reasonable latency query times. This is achieved by the first layer of the Lambda architecture: the batch layer. The results of the batch layer are stored in batch views, managed by the serving layer. Most of the above-stated properties

are already fullfilled by these two layers. The final property, concerning the low-latency reads and updates, is accomplished with the final layer: the speed layer [1]. It provides the analysis of data as soon as it enters the system and stores it in a speed view. Queries by applications or users then combine the information that is stored in the batch and speed view. A query on a big data set, analyzed by the Lambda architecture, can thus be described as follows:

$$\begin{aligned} batch\ view &= function(all\ data) \\ speed\ view &= function(speed\ view, new\ data) \\ query &= function(batch\ view, speed\ view) \end{aligned}$$

Figure 2.1 gives a conceptual overview of all the above discussed layers of the Lambda architecture.

The batch layer thus continuously recomputes the main big data set, which in time grows, causing the execution time to increase accordingly. This execution time can be limited by using an incremental function to compute the batch view:

$$batch\ view = function(batch\ view, new\ data)$$

However, in order to guarantee the robustness and fault tolerance, a recomputational algorithm needs to always exist.

As soon as data is processed by the batch layer, the derived information that will be stored in a batch view has a duplicate in the speed view. The corresponding information in the speed view thus needs to be removed to make sure no redundant information is present in the system. While this keeps the data store for the speed views relatively small, i.e., it only contains the most recent information of the system, it does expose a critical part of the system. If the synchronization between batch and speed layer is incorrect, the entire system is vulnerable to missing or redundant information. Marz suggests to maintain two sets of speed views and alternately clearing them, which introduces redundancy. This chapter proposes a general solution in Section 2.3 without information redundancy.

A second challenge arises with the final function to answer a query:

$$query = function(batch\ view, speed\ view)$$

To answer a query, information from both the batch and speed view is needed. The idea where applications store their information in a mix of data stores to take advantage of the fact that different data stores are suitable for storing different information, is referred to as polyglot persistence [13]. Support for polyglot persistent applications is still a very active research topic [14, 15]. Initial steps towards a

[1] This layer is called the real time layer by Marz, but in practice it is often more near real time than true real time. To avoid confusion, in this chapter it is referred to as the speed layer.

general solution for the aggregation challenge in the Tengu platform are disclosed in Section 2.7.

While the Lambda architecture is regarded as a promising concept in both academia [16, 17] and industry [18, 19], some critique is expressed as well [20]. Kreps points out that maintaining two code bases (for batch and speed layer) is a complex and painful issue. While this is true in some form, their proposed alternative, the Kappa architecture, limits the information that can be retrieved from the big data set. This new proposal eliminates the batch layer and only uses the speed layer to analyze the entire data set message by message. However, this way an algorithm can no longer benefit from an overview of the entire data set. For example, suppose an application analyzes the chat messages between social network users for the detection of cyber bullying [21]. In the speed layer a message is analyzed on its own, but in the batch layer a more accurate analysis is possible because the algorithm has the context of the entire chat history. In the next sections a solution for the synchronization challenge in the Lambda architecture is given without compromising on the information stored in the views.

2.3 Synchronization

The most important aspect of the synchronization between the batch and speed layer happens when the batch layer finishes its computation. A delicate operation follows where the soon to be redundant data needs to be removed from the speed view before it is entered in the batch view. If too much information is removed from the speed view, the system enters a temporary state with missing information. If too little information is removed, the system enters a temporary state where redundant data is processed in the queries. Both states are temporary, because it is fixed after another execution of the batch layer algorithm, although other information might then be missing or redundant.

Nathan Marz proposes a solution where two parallel speed views are used to store the most recent information [2]. As he points out, this leaves the system in a redundant state, but it is considered to be an acceptable price for a general solution. The goal of this chapter is to design a general solution without redundancy or information loss. In order to do so, a precise answer is needed to the following question: which information needs to be deleted once a batch layer run has finished? The system thus needs to know which data was processed by the batch layer and what the corresponding information is in the speed view.

The proposed approach is as follows: tagging data as soon as it enters the system allows for this traceability of when the data entered the system, and thus what corresponding information can be removed. As soon as data arrives, it is tagged by a current tag T_n. The data is stored with the big data set, but still marked with the tag T_n. It is also analyzed by the speed layer, which stores the

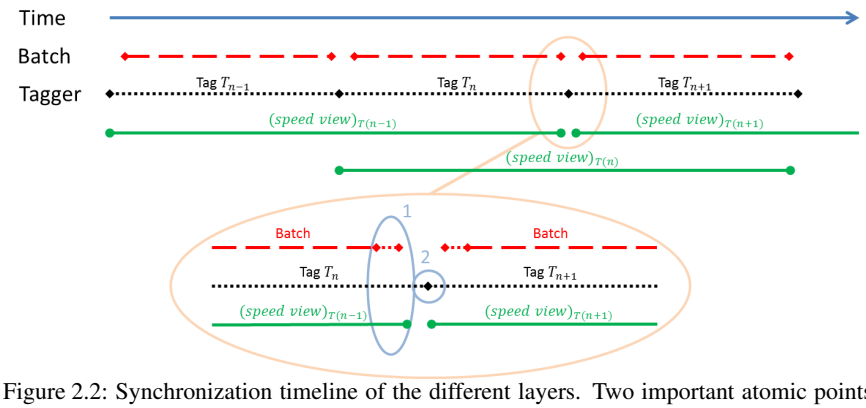

Figure 2.2: Synchronization timeline of the different layers. Two important atomic points are identified: 1) batch view update - speed view clearance 2) tag switching.

resulting information in a view specifically for all information with the tag T_n, $(speed\ view)_{Tn}$. As soon as the batch layer finishes its current execution, the following happens: the system switches to a new tag $T_{(n+1)}$ for all new incoming data. The information, resulting from the batch layer execution, is pushed into the batch view. The corresponding information in the speed view can be easily cleared with the tag that came before T_n, $(speed\ view)_{T(n-1)}$. Then the new batch data set becomes the union of all data with the T_n tag, $data_{Tn}$, and the previous batch data set:

$$batch\ data \quad = \quad data_{Tn} \cup batch\ data \tag{2.1}$$

At this point, the batch layer starts a new execution and the entire walkthrough described above is repeated. Similar to the solution proposed by Marz, parallel speed views are used, but now clearly marked with a tag that marks the information that is contained within them as to avoid redundant or missing information. A query now becomes:

$$query = \quad function(batch\ view, (speed\ view)_{T(n)}, ..., (speed\ view)_{T(n-i)}) \tag{2.2}$$

Figure 2.2 depicts the lifetime of different events and services in relation to each other in a normal running Lambda architecture implementation. The directional line on top represents the time moving from left to right. The batch layer execution time is portrayed by the dashed line. The dotted tagger line shows which tag is given to a new message that enters the system at a given time. Finally, the lifetime of the speed views is represented by solid lines and the name of the tag it stores. The sequence clearly shows how a speed view exists for two batch runs before being cleared.

Figure 2.2 also shows two atomic points that will need to be addressed in the implementation:

1. **Batch view update - speed view clearance:** during this operation the system is vulnerable for responding to queries with redundant or missing information. If a query were to enter the system between the update of the batch view and the clearance of the corresponding information in the speed view, the response of the query will contain redundant or missing information, depending on the order of the previously mentioned operations.

2. **Tag switch:** a message cannot enter the system while no or multiple tags are active. If a message is not tagged, the system will ignore it and data is lost. If a message is tagged multiple times with different tags, redundant data is introduced into the system.

Important to note is the difference in impact both points have: the tag switch concerns data, while the update/clearance works in the context of information. Recovering a system from faulty information is possible through a complete recomputation of the data set. However, recovering from faulty data is a whole lot more complex since all derived information is false as well.

Note that in this section no assumptions have been made as to which technologies are used to implement the proposed tagging solution. Tagging can be implemented in different ways: a tag can be directly inserted into a message or it can be indirectly associated with the message. The proof of concept of the tagging solution for the synchronization challenge uses the indirect approach and is presented in Section 2.5.

2.4 Failure handling

An important property of a big data system is its robustness and fault tolerance as outlined in Section 2.2 above. In the following subsections failure scenarios of the different parts of the platform are discussed and how they can be handled.

2.4.1 Batch layer failures

If the execution of the batch layer fails, there are several possibilities to handle the failure. First, a simple restart of the execution can be done with the same data set as before. The batch and speed view still contain the correct information for applications and users, and the current tag needn't change. A repeatedly failing algorithm does require human intervention as the cause might be a faulty implementation.

A second possibility is to handle the failure similar to a correct end of the batch layer: a new tag is used to tag future incoming messages, but the previous tags are not wiped from the speed view as they were not yet analyzed by the batch

layer. Otherwise this would cause temporary information loss. Data tagged with the previous tags is added to the data set that will be analyzed by the batch layer. In other words, while the batch layer needs to restart, the data set is expanded to take into account more recent data. This method is limited in the number of failures it can handle due to the increasing number of concurrent tags and the possibility of an overflow of the tag value. As with the previous method, the information in the batch and speed views remains available for applications and users. Figure 2.3 depicts this method of failure handling for the batch layer. The proof of concept, detailed in Section 2.5, handles a batch layer failure with a simple restart.

2.4.2 Speed layer failures

A failure of the speed layer has less impact on the entire data system compared to a batch layer failure because the information displayed in the speed view is only a fraction of the total data set. That being said, the goal is to eliminate redundant and missing information completely.

Failure handling is mostly dependent on how a streaming big data analysis platform handles the failures. If the analysis of one message fails, it is important the chosen technology has guaranteed message processing or checkpointing, i.e., each data message is fully processed without fault. If an entire machine or cluster fails, data in transit should be recovered or re-analyzed. For example, in an implementation with Kafka and Storm, Storm provides guaranteed message processing, but it also needs to keep an offset of messages it already consumed from Kafka. Both technologies combined can therefore recover from a variety of failures.

2.4.3 View failures

A view failure results in partial information not being available for applications and users. A failure of the speed view has a limited impact as it only contains the most recent information of the system, while a failure of the batch view would

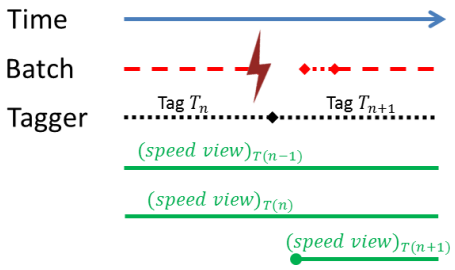

Figure 2.3: Batch layer failure handling

cause most of the historical information to be unavailable. Therefore, it is important to use distributed and replicated data stores for the views of both layers. In the NoSQL (Not Only SQL) domain most data stores are of a distributed nature and support some form of replication. The amount of replicas depends on the critical nature of the application. A careful consideration is required in this trade-off between storage cost and availability.

While a view failure can cause a temporary unavailability or redundancy of information, the layered approach of the Lambda architecture allows the system to recover without human intervention. A recomputational algorithm in the batch layer always starts with the original main data set, meaning errors in a batch or speed view are overruled in the next iteration. This property is shown extensively in the results in Section 2.6.

2.4.4 Data and communication failures

Query latency In Section 2.3 it was mentioned that the operation updating the batch view and deleting the corresponding information in the speed views needs to be atomic. During this time a read lock needs to be enforced on the different views as to insure no missing or redundant information is used to answer the query. If an error occurs during one of the steps in the operation, a rollback can make sure the views are not corrupted.

Equation 2.2 also defines a query in the Lambda architecture as a function that aggregates data from different views. Both the read lock and the aggregation will cause a certain query latency.

Tagging The impact of missing or redundant data compared to information was already briefly discussed in Section 2.3. An error in the tagging or switch between tags could cause this missing or redundant data. Recovery from such a failure entails much more than an information failure and the system will be unable to recover from this without manual intervention.

Data persistence Finally, data persistence is an important feature to make sure no data or information is lost. For example, assume a message is the last message to be tagged with tag T_n. All the $data_{Tn}$ needs to be merged with the previous batch data set, as defined in Equation 2.1. There needs to be a guarantee that all data with tag T_n is present in $data_{Tn}$, i.e., even the last message to be tagged with T_n needs to be present and not get lost in the network. This is closely related to guaranteed message processing discussed in Section 2.4.2.

2.4.5 Human failure

A final important failure is the realistic possibility that a human error will occur in the system. Here the importance of the main data set is again featured. The main data set contains unaltered data and is expected to be true, within the Lambda

architecture system. This assumption allows the system to recover from any human error in the different layers. For example, if a faulty implementation in any layer causes faulty information to be stored in the views, a fix of the faulty code allows the entire system to recover after a couple of iterations. This emphasizes the need for a re-computational algorithm in the batch layer. While an incremental batch algorithm can be used to limit the execution time of the batch layer, a re-computational algorithm needs to exist to recover from human-introduced errors, such as faulty implementations.

2.5 Implementation details

The proposed Lambda architecture implementation is implemented as part of the Tengu platform, previously known as Kameleo [11]. The Tengu platform was originally developed for the automated setup of big data technologies on experimental testbeds. Figure 2.4 shows an overview of all used technologies in the proof of concept implementation and how they are chained together.

The first technology a message encounters when it enters the system is the WSO2 Enterprise Service Bus (ESB). It allows for advanced communication between services by routing messages in a bus architecture using a vast array of protocols. For this reason the ESB was favored over a Message Broker (MB) or a Complex Event Processor (CEP) as those would limit the amount of control the system had over the messages and services. The WSO2 ESB was chosen over other candidates, such as UltraESB, Mule, and Talend, for its performance and maturity [22–24]. It is the intelligent controller-like component that coordinates the execution of the different services, i.e., the batch and speed layer, and their views. The ESB also maintains the current active tag corresponding to an active topic in Apache Kafka [25].

After retrieving the tag in the ESB, the message is sent to a Kafka topic corresponding to the received tag. The tag is hence never attached to the incoming message, but indirectly associated with the message through a topic in Kafka. From

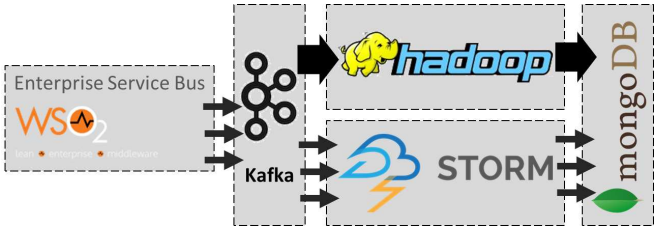

Figure 2.4: Technology overview of the implemented Lambda architecture proof of concept.

this topic the message is ingested by a speed technology, analyzed and stored in a speed view. In the proof of concept Storm [8] is used as a speed technology, while the speed views are stored in MongoDB [26]. Storm contains a topology that is responsible for a specific tag, i.e., a Kafka topic. This topology analyzes the messages and stores them in the MongoDB collection related to the tag. The union defined in Equation 2.1 is performed using all data in the Kafka topic as $data_{Tn}$. The batch layer, implemented with Hadoop [4] in this proof of concept, performs an analysis and stores the information in a batch view, a specific collection in MongoDB.

Important to note is that the implementation of the tagging system is done by the WSO2 ESB and Kafka. While Hadoop, Storm and MongoDB are used in this proof of concept, they are merely services of the ESB through which the messages are analyzed and stored. As a consequence they can be replaced by similar technologies such as Spark, Samza and Cassandra. Additionally, many technologies can already act as a consumer of Kafka messages, but if not, an extension of the WSO2 ESB can still provide the necessary communication.

In Figure 2.2 two critical points were also identified concerning the update of the batch view and simultaneous removal of the corresponding information in the speed view, and the switch between active tags. Both operations are required to be atomic to prevent data/information loss or redundancy.

The tag is stored local to and managed by the ESB, making every operation transactional. For each message the ESB reads the value of the tag and sends the message to the corresponding topic. If a call is made to change the tag, the value is updated with an atomic operation. A message can therefore never continue without a tag or with multiple tags.

The switch between views after a completed batch layer iteration is handled by inserting a read lock on the views. This can cause somewhat of a query latency if a query is on hold during the switch. A solution for this latency can be to cache the information during the transition, but this is outside the scope of this chapter and considered part of future work.

2.6 Evaluation results

The Tengu platform is deployed on the iLab.t Virtual Wall infrastructure [27]. These experimental testbeds consist of over 300 nodes spanning different generations of hardware setups. For the tests in this chapter generation 3 nodes were used: 2x Hexacore Intel $E5645$ (2.4GHz) CPU, 24GB RAM, 1x250GB harddisk, 1-5 gigabit nics. Eight nodes were used in the following setup interconnected with a 1 Gigabit connection:

- 2 hadoop nodes

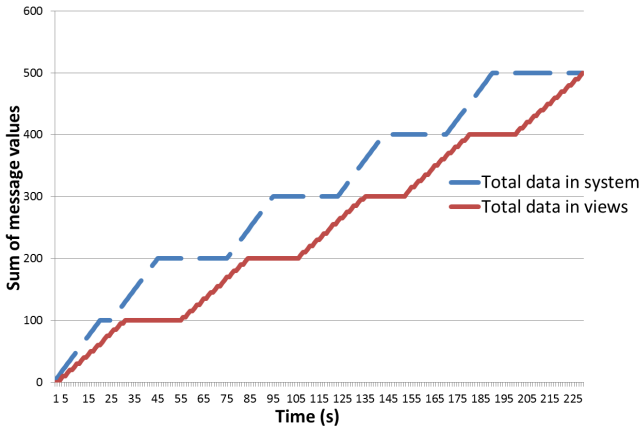

Figure 2.5: Normal progress of the active Lambda architecture implementation

- 2 storm nodes
- 1 WSO2 Enterprise Service Bus node
- 1 MongoDB node
- 1 Zookeeper node
- 1 Kafka node

In the following subsection the results are detailed to show the correctness and regenerative capabilities of the Lambda architecture implementation, especially in the context of information redundancy and information loss. Next, insight is given as to where information is stored among the different views in a normal run of the system.

2.6.1 View failure

The most important part of the synchronization challenge consists of eliminating redundant information and information loss. The first results in Figure 2.5 show the normal progress of data sizes in the Lambda architecture. For each tag 20 messages were injected into the system through a REST API, one every second, where each message had a specific value. The WSO2 ESB supports a variety of message formats but for this test JSON messages were used:

```
{
  'value': '5'
}
```

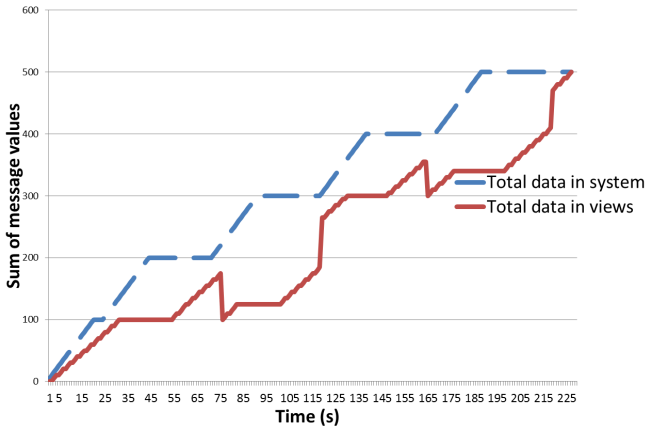

Figure 2.6: Regenerative progress of the active Lambda architecture implementation with data loss in views

The algorithm in the batch and speed layer were tasked with calculating the total sum of the message values. The dashed line shows the sum of all message values injected in the system at any given point. The solid line shows the aggregated sum that is available in all the views, both batch and speed. The sum calculation in the speed layer is slowed as to clearly differentiate the two graph-lines from each other. As can be seen in Figure 2.5 the solid line can never drop down, as this would indicate information loss, or be higher than the dashed line, as this would indicate information redundancy.

Information loss in the views is introduced in the second graph, depicted in Figure 2.6. Loss is introduced twice in the speed view at around 65 and 165 seconds. The regenerative property of the Lambda architecture is shown at around 115 seconds and 215 seconds. This is when the batch layer has recomputed the main data set and the lost information is restored in the batch view.

Figure 2.7 shows the regenerative measures of the implementation after redundancy is introduced to the speed views. The solid line clearly surpasses the dashed line in the graph, indicating the presence of information redundancy. The redundancy is however not present in the main data set, meaning that after a batch iteration the redundant information is deleted from the views, again displaying the correct total sum.

Both graphs clearly show the regenerative capabilities of the implemented Lambda architecture in situations with varying information inconsistencies. The time in which the system returns to a consistent state depends on the execution time of the batch layer. In Section 2.3, Figure 2.2 illustrates that speed views exist for two batch layer runs before being cleared, meaning that in a worst case scenario an inconsistent state is maintained during two batch layer runs before being

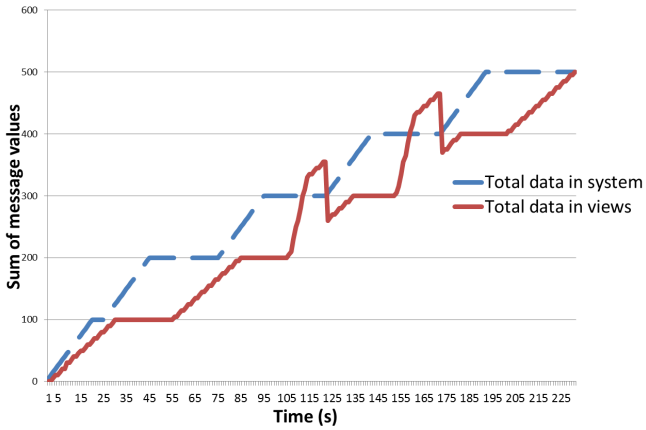

Figure 2.7: Regenerative progress of the active Lambda architecture implementation with data redundancy in views

resolved. The batch layer execution time can be shortened through use of an incremental algorithm, but as mentioned in Section 2.2 a re-computational algorithm is still required to achieve fault tolerance and robustness. An inconsistent state in the batch view can be resolved after one batch run, but only with a re-computational algorithm.

2.6.2 Information transition from speed to batch views

As information is moved between different views a lot in the Lambda architecture, the graph displayed in Figure 2.8 shares some insight as to where information is stored during a normal run of the Lambda architecture implementation. Important to note is that messages are now continuously sent to the system and have ever increasing values, hence the exponential curve of the total data sum. The speed layer is also no longer slowed down in these tests. First, speed view 1, marked by the dotted line, is filled with information until it reaches a plateau at around 25 seconds. This plateau occurs as the Storm topology is swapped for a new topology to start processing the new tag, i.e., ingest the new topic from Kafka. Once the new topology is active at around 50 seconds, it quickly catches up to the total expected sum by filling up speed view 2, indicated by the small dashed line, until it reaches the next plateau. Again the Storm topologies are switched, but speed view 1 is also cleared as the information is now contained within the batch view, marked by the dashed-dotted line. Now speed view 1 can again be used to store information and the entire above described process repeats itself. A maximum of two concurrent tags are thus active at any given time.

Based on the graph in Figure 2.8 some improvements can be made: the plateau

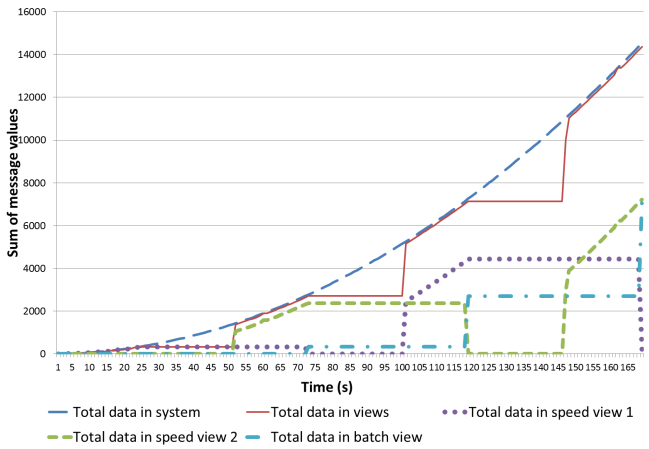

Figure 2.8: The total data in the Lambda architecture in time with respect to the different views

could be reduced by having two parallel Storm topologies, as with the speed views. This has the additional benefit that the old topology can continue generating information next to the new one. The single topology setup of this proof of concept can cause additional delay because the system waits for the topology to be entirely finished before swapping. For a simple task, like calculating a sum, Storm is fast enough and no additional delay is caused, but with more complex algorithms the time for data to be processed by the topology increases, heightening the possibility of additional delay. In a production environment it is therefore highly recommended to work with two parallel Storm topologies.

2.7 Aggregation

In Section 2.2 a query in the Lambda architecture is defined as a function over the different views. An application that stores data or information in a mix of data stores to take advantage of the fact that different data stores are suitable for storing different data is referred to as a polyglot persistent application [13]. While the work of Sadalage and Fowler focuses on dividing the data set based on data type and/or model, the polyglot persistence in the Lambda architecture splits information based on time, derived from the tag the data got when it entered the system. Both Equation 2.2 and Figure 2.1 show the need for aggregation, as an answer to a query consists of multiple queries to different data stores. The nature of the aggregation depends on the nature of the information stored in the views and the nature of the query. For example, two integers can be added in a sum, but could equally well be concatenated.

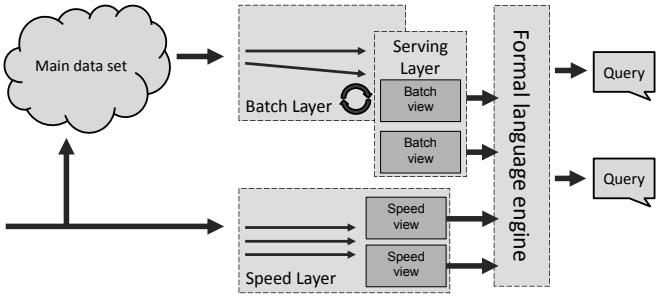

Figure 2.9: Lambda architecture with a formal language for the aggregation of information

Data abstraction layers, such as Hibernate OGM [28], Kundera [29], and DataNucleus [30], help applications with polyglot persistence by providing general access to their data stores, usually through a unified querying language. Although these data abstraction layers shield applications from underlying data storage technologies, they lack the ability to intelligently combine information from several data stores and return it. The application is thus still responsible for combining information from the different views and not effectively protected from data model changes.

If this responsibility is to be moved away from the application, it needs to be re-introduced in a new layer between the application and the data stores. As mentioned before, the nature of the aggregation is specific to the query the application sends, so user input is required. However, users often also lack the insight into the different technologies to correctly write the code for information retrieval. A definition of the aggregation through a technology independent data flow could prove to be a solution in this case.

A proposed approach is to define this data flow through a formal language. The formal language would allow users, lacking any programming skills or technology specific knowledge, to define an algorithm answering their query through a flow of operations and other queries on the different underlying data stores. Once an aggregation is created through the formal language, an engine can translate it into code and technology-specific queries for different data stores. Figure 2.9 shows how the formal language fits in with the Lambda architecture. Initial steps towards a definition and implementation of this formal language are ongoing and will be reported on in future work.

2.8 Conclusion and future work

The Lambda architecture is a powerful concept for big data systems. However, it does pose several implementation challenges. This chapter proposes a general

implementation of the concept, independent of the technologies used for different layers and views. It focuses on a solution for the synchronization challenge between the batch and speed layer through a tagging system. A solution is proposed, tagging messages when they enter the implemented Lambda architecture system, and a proof of concept is implemented in the Tengu platform. Results show that the proof of concept works correctly in regard to eliminating information loss and redundancy, and that when manually introduced, it is able to recover automatically. The information transition between batch and speed view also indicated a delay where no new information was posted in the views during the transition of topologies. A solution is suggested where two parallel topologies exist in the Storm cluster.

Another challenge was identified as the aggregation of information from batch and speed views to answer queries from applications or users. This chapter discusses the initial steps that have already been taken towards a general solution in the Tengu platform. The implementation itself will be reported on in future publications.

Acknowledgement

This work was partly carried out with the support of the AMiCA (Automatic Monitoring for Cyberspace Applications) project, funded by IWT (Institute for the Promotion of Innovation through Science and Technology in Flanders) (120007).

References

[1] J. Gantz and D. Reinsel. *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east*. IDC iView: IDC Analyze the Future, 2007:1–16, 2012.

[2] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[3] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[4] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[6] *Apache Flink*. http://flink.apache.org/ (Last Visited January 9, 2018).

[7] J. Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 2010.

[8] *Apache Storm*. https://storm.apache.org/ (Last Visited January 9, 2018).

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. *S4: Distributed Stream Computing Platform*. In Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pages 170–177, Dec 2010. doi:10.1109/ICDMW.2010.172.

[10] *Apache Samza*. https://samza.apache.org/ (Last Visited January 9, 2018).

[11] T. Vanhove, J. Vandensteen, G. Van Seghbroeck, T. Wauters, and F. De Turck. *Kameleo: Design of a new Platform-as-a-Service for Flexible Data Management*. In Proceedings of the 2014 IEEE/IFIP Network Operations and Management Symposium (NOMS 2014), 2014.

[12] A. Jacobs. *The Pathologies of Big Data*. Commun. ACM, 52(8):36–44, August 2009. Available from: http://doi.acm.org/10.1145/1536616.1536632, doi:10.1145/1536616.1536632.

[13] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[14] A. Maccioni, O. Cassano, Y. Luo, J. Castrejón, and G. Vargas-Solar. *NoX-peranto: Crowdsourced Polyglot Persistence*. Polibits, 50:43–48, 2014.

[15] S. Prasad and S. Avinash. *Application of polyglot persistence to enhance performance of the energy data management systems*. In Advances in Electronics, Computers and Communications (ICAECC), 2014 International Conference on, pages 1–6. IEEE, 2014.

[16] W. Fan and A. Bifet. *Mining big data: current status, and forecast to the future*. ACM sIGKDD Explorations Newsletter, 14(2):1–5, 2013.

[17] S. Perera and S. Suhothayan. *Solution Patterns for Realtime Streaming Analytics*. In Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, pages 247–255, New York, NY, USA, 2015. ACM. Available from: http://doi.acm.org/10.1145/2675743.2774214, doi:10.1145/2675743.2774214.

[18] HPCC Systems. *Lambda Architecture and HPCC Systems*. White Paper, February 2014.

[19] *MapR*. https://goo.gl/SBdQEW (Last Visited January 9, 2018).

[20] J. Kreps. *Questioning the Lambda Architecture*. Online article, July 2014. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html (Last Visited January 9, 2018).

[21] T. Vanhove, P. Leroux, T. Wauters, and F. De Turck. *Towards the design of a platform for abuse detection in OSNs using multimedial data analysis*. In Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on, pages 1195–1198. IEEE, 2013.

[22] D. Abeyruwan. *ESB Performance Round 6.5*. Technical report, WSO2, January 2013. http://wso2.com/library/articles/2013/01/esb-performance-65/.

[23] A. C. Perera and R. Linton. *ESB Performance Round 7*. Technical report, AdroitLogic, October 2013. http://esbperformance.org/display/comparison/ESB+Performance.

[24] S. Anfar. *ESB Performance Round 7.5*. Technical report, WSO2, February 2014. http://wso2.com/library/articles/2014/02/esb-performance-round-7.5/.

[25] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[26] K. Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[27] *iLab.t Virtual Wall*. http://doc.ilabt.imec.be (Last Visited January 9, 2018).

[28] *Hibernate OGM*. http://hibernate.org/ogm/ (Last Visited January 9, 2018).

[29] *Impetus Kundera*. https://github.com/impetus-opensource/Kundera (Last Visited January 9, 2018).

[30] *DataNucleus*. http://www.datanucleus.org/ (Last Visited January 9, 2018).

3

Data Transformation as a means towards Dynamic Data Storage and Polyglot Persistence

T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck.

Published in International Journal of Network Management, July 2017.

⋆ ⋆ ⋆

Legacy applications have been built around the concept of storing their data in a relational data store. However, the new generation of storage solutions in NoSQL provide better scaling and performance in a big data context. While the data sets these applications have collected could benefit from these new technologies, schema and data transformations between data store technologies are needed in order to get legacy applications into new storage technologies. This usually infers downtime and code changes in the application to support the new data store technology. This chapter details a transformation approach through a canonical model. It makes use of the Lambda architecture, as detailed in Chapter 2, to ensure no application downtime is needed during the transformation process, and after the transformation the application can continue to query in the original query language, thus requiring no application code changes.

3.1 Introduction

Relational data stores are an important building brick for legacy applications in their data storage strategy. However, with growing data sets in the age of big data analytics, applications' demands have exceeded the capabilities of classic relational database management systems (RDBMS). With this new paradigm for large scale processing, fast access to the data is necessary. Many new systems have been designed aimed to scale horizontally, providing read/write operations distributed over many servers [1]. Many of these new systems can be categorized as NoSQL, which stands for "Not only SQL". Contrary to the classic relational databases, they provide easy scaling and performance advantages in specific scenarios, depending on the chosen NoSQL data store [2]. Additionally, they provide a more flexible or even schema-less data model, allowing rapid changes in the model. The popularity of these data stores can be measured by the sheer amount of solutions available. However, this does not mean relational databases don not have a role to play in the big data story [3]. An example is Google's F1 hybrid database [4], a scalable distributed SQL database built on top of their globally distributed and synchronously replicated database, Spanner [5]. Google uses this database to support their AdWords business, an ecosystem with 100s of concurrent applications and 1000s of users sharing the database, over 100TB in size in 2013.

The amount of possible solutions for data storage led to a specialization of these data stores in order to distinguish themselves from each other, making different data stores more suitable for different types of data or for the different use of data. Thus, a correct choice in data store is paramount for the optimal performance of the application. However, as applications tend to evolve with frequent updates and changing user numbers, the optimal choice of data store may change over the course of the application's lifespan. The concept of dynamic storage allows the stored data to be stored in the optimal format for the application at all times, transforming the format when necessary, i.e., when certain requirements are no longer met. Along with this, applications often work with different types of data, e.g., e-commerce platforms. Another interesting concept would therefore be to have the application use multiple data stores simultaneously, instead of forcing all data into one solution. This is often referred to as polyglot persistence [3, 6, 7]. In the case of a network monitoring platform, device information can be stored in a classic RDBMS, while logging data might be a better fit for a document data store or a search server such as ElasticSearch.

Introducing dynamic storage and/or polyglot persistence in existing legacy applications requires a transformation of existing data stores or parts thereof. On the one hand there is the cost of transforming the data format, but on the other hand many application changes may be necessary as well to support the new format. Additionally, with applications having to meet specific service-level agreements

(SLA), this migration and/or transformation has to occur with as limited downtime as possible, preferably eliminating the downtime entirely in a best case scenario. This high migration and transformation cost discourages application developers to change data stores in live applications.

Based on the previous paragraphs, three main obstacles can be defined that currently hamper dynamic storage and polyglot persistence:

1. Migration of data to the cloud (or between clouds)
2. Transformation of data formats
3. Alteration of application code

This chapter reports on advances that have been made into overcoming these obstacles as well as contributing to a new approach of data transformation in such a way that the downtime of the applications is eliminated, without additional development and implementation effort [8]. The proposed solution aims to tackle the issues concerning polyglot persistence, i.e., enable applications to access and store data in different data stores simultaneously, and allow for dynamic changeovers between supported data stores based on monitoring information or the intervention of the customer or administrator. The proposed solution makes use of a new open-source Platform-as-a-Service (PaaS) called Tengu [9]. The Tengu platform provides researchers a time-saving approach for building big data analysis frameworks through automated installation, configuration and integration of big data analysis and storage technologies [9, 10].

The remainder of this chapter is structured as follows: Section 3.2 reviews related work in the different domains that already contribute to the solution of the previously stated obstacles. The approach and general workflow of the transformation are discussed in Section 3.3, while Section 3.4 describes the transformation algorithm. In Section 3.5 the implementation of the algorithm and workflow on the Tengu platform are detailed. Section 3.6 discusses the evaluation of the implementation through performance testing. Finally, the chapter is concluded in Section 3.7 and offers several leads towards future work.

3.2 Related Work

Early work on data transformation [11, 12] led into what are now called Extract-Transform-Load (ETL) processes. These software processes are commonly used in data warehouses where they extract data from often different data sources, transform the data in the correct format, and load the transformed data into the data warehouse. Research in ETL has focused on modeling, efficiency, and facilitation of construction [12]. While the approach and algorithm described in this chapter

show several similarities to ETL processes, they are vastly different. ETL facilitates data transformation between two data points, data source and data warehouse, where both data schemas are known. If a change is made in the data schema of the data source or the data warehouse, changes will need to be made in the ETL process. The proposed transformation in this chapter works between two data points where only one data schema is known, the source data store, which is then transformed into a data schema representation of the new data store.

For each of the ETL subprocesses (extract, transform and load) a research domain exists. Extract and load have been heavily researched as part of data migration and has become even more apparent with the complexity introduced by clouds and big data [13]. Data migration obstacles have been solved in several ways using high-performance networks [13], workload-aware strategies [14], and cost-minimizing approaches [15]. In this research domain, several solutions have also been proposed for live data migration, i.e., a migration where a live application needs to be supported without downtime [16, 17]. Other migration tools allow data from an RDBMS to be analyzed by big data processing tools, such as Hadoop. Apache Sqoop provides a framework to transfer data between an RDBMS and Hadoop [18]. The RDBMS data can thus be used in a big data analysis process after which the results can be migrated back to the RDBMS.

Another important research domain related to data transformation is that of schema matching and mapping [19, 20]. It is a process that identifies if two data schemas are semantically similar and describes the transformations for data to be represented in the other schema. This research domain is closely related to ETL as it aids in the creation of the transformation subprocess. This work leverages the input of two data schema and maps the transformation between them, contrary to the work in this chapter. Other work in the transformation research focuses on data transformation, more specifically between SQL and NoSQL data stores. However, compared to the work in this chapter, it is often limited in the support of data store technologies (e.g., only supporting column data stores) [21, 22]. The transformation approach and algorithm in the chapter is aimed towards flexibility and extensibility, in theory able to support any data store technology.

Finally, direct transformation tools between two specific data store technologies exist as well, such as Mongify for SQL to MongoDB [23]. They are able to transform data stores from one specific data technology to another. Compared to the approach in this chapter these tools are limited as they only provide between two specific data stores with no easy way of extending the support to other data store technologies. Moreover, these tools are often built with custom code and therefore do not scale well when working with legacy or production data stores in general. Another example of a direct transformation tool is present in Cassandra, using Apache Sqoop [24]. It capitalizes on the similarities between SQL and CQL to import and export data between Cassandra and a classic RDBMS. While

technically transforming data between two different technologies, this approach provides data migration functionality. In contrast to the contributions in this chapter, the Cassandra transformation tool also does not provide any optimizations in its supported technologies to decrease query latencies.

Several of the previously mentioned research topics have already seen applications in the industry. Table 3.1 gives an overview of how the major cloud providers, Amazon, Google and Microsoft, overcome the obstacles described in the previous section: migration of data, transformation of data and if alteration is required in the application code. The 'X' marks that no tool is made available by the cloud provider for a specific obstacle. At first sight all providers supply tools for the migration of data from and towards their platform, both offline and online. Online tools allow for the migration of data over the internet while offline tools are organized processes of sending physical disks to the providers. Amazon outperforms Google and Microsoft as it not only provides tools to overcome migration, but transformation and alteration as well. However, when taking a closer look at the schema conversion tool offered by Amazon it is mostly restricted to data stores with SQL-like querying languages for both transformation and the alteration of application code [1]. The tool can tweak the SQL schema of a source data store and alter the SQL query in the application code in order to reflect the changes made to the schema. Compared to the work in this chapter the AWS schema conversion tool is limited as it only supports SQL related data stores. Furthermore, this transformation still requires changes in the application, although these are executed automatically by the tool. While Google and Microsoft have no tools for a full transformation and alteration, Microsoft Azure does provide a tool for schema matching/mapping.

Table 3.1: State of the art in the domain of migration, transformation and alteration of application code as used at Amazon, Google and Microsoft.

	Amazon AWS	Google Cloud	Microsoft Azure
Migration (Online)	Database Migration Service	Storage Transfer Service	Azure Migration Wizard
Migration (Offline)	Amazon Snowball - AWS Import/Export Disk	*Third party support*	Import/Export service
Transformation	Schema Conversion Tool	X	*Limited*
Alteration	Schema Conversion Tool	X	X

One of the goals of transformation in this chapter is to support polyglot persistence for legacy applications. A lot of research has gone into solutions that shield the complexity of having to deal with multiple query languages through abstract data layers, such as Hibernate ORM/OGM [25] and Apache Drill [26].

[1] https://aws.amazon.com/dms

These abstract data layers provide access to different datastores without the need for the application or developer to be aware of the complexities of the datastore. Most of these provide only limited or no support for the migration of data between supported datastores, but do allow applications to store data in different parallel datastores depending on the type of data. Many of these abstract data layers however require applications to use the abstract data layer's querying language, which in some cases is the SQL standard, but in others a custom dialect (e.g., Hibernate Query Language (HQL)). The abstract data layers effectively shield the data store complexity of polyglot persistence, but only for new applications. Legacy applications with big data sets still have no out of the box solution to help them benefit from these new paradigms.

3.3 Data transformation framework

This section describes the approach and workflow of the data transformation as a means to achieve dynamic storage and polyglot persistence for applications. First an architecture for the transformation is proposed that overcomes the aforementioned obstacles and avoids application downtime. Next, the approach of the actual transformation is discussed. Finally, the architectural principles for the data transformation are applied in a practical workflow for the transformation process.

3.3.1 Architecture

A straightforward solution for the transformation would be to take a snapshot of the source data store (D_{src}), transform the snapshot, and load it into the transformed data store (D_{trans}). No queries would be allowed during the transformation process, effectively shutting down any data store operations by applications. However, in production environments it is important that any live application supported by the data store, encounters no or minimal impact on their operations. Queries submitted by the application after the snapshot of D_{src} was taken, could still be executed on D_{src}, but in order for D_{trans} to contain the latest data and/or reflect the latest changes to its data and structure, queries that insert new or modify existing data need to be transformed as well. The transformation process can therefore be divided in two parts: the transformation of a snapshot of D_{src} and the transformation of the data inserted or modified by queries arriving after the snapshot of D_{src} was taken. The specific time when the snapshot of D_{src} is taken, is indicated by T_{snap}. Important to note is that all queries will still be executed on D_{src} during the following transformation to support any live applications.

Transforming the D_{src} snapshot into D_{trans} can be regarded as a batch job. It has access to the entire data set, i.e., the snapshot of D_{src}, and processes the transformation of this entire data set. Once this batch job is finished D_{trans} still

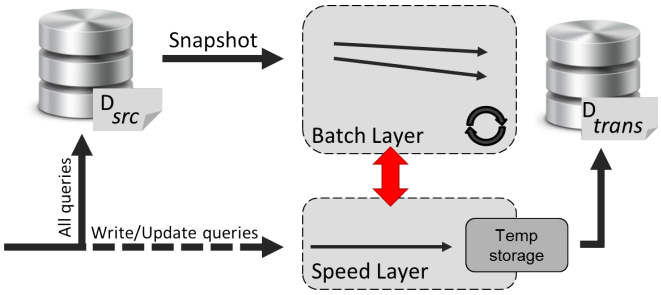

Figure 3.1: General overview of the described architecture with a batch layer and parallel streaming layer

requires to be updated with the new and adjusted data that is contained within the queries that arrived after T_{snap}. An obvious choice would be to rerun the batch job for these queries. However, during this second transformation new queries would possibly still arrive as well, requiring yet another run of the batch job. Depending on the arrival rate of the queries, the batch job would run on an ever reducing set of queries, decreasing the performance of th ese runs because of a static overhead [27]. In a worst case scenario, it would never reach a consistent state. A better solution would be to use a streaming analysis component, transforming the queries in parallel to the batch transformation as soon as they arrive. The additional benefit of this streaming layer is the continuous query transformation it can provide after the changeover to D_{trans} is complete. Continuous query transformation is a situation where the application would be able to query D_{trans} in the query language of D_{src} through the live transformation in the streaming layer. This effectively eliminates any changes to the application.

Such a two-layered hybrid solution is often referred to as the Lambda architecture, a term coined by Nathan Marz [28]. The concept leverages the computing power of batch processing with the responsiveness of a real time computation system. However, the solution described in this chapter defers from this concept in an important way. In the original Lambda architecture the batch layer continuously reanalyzes an increasing big data set, whereas the proposed solution uses the batch layer for one iteration only, i.e., the transformation of the snapshot of D_{src}. The Tengu platform provides a generic implementation of the Lambda architecture, independent from the technologies used for the different layers [10]. The solution described in this chapter will therefore be deployed on the Tengu platform.

Figure 3.1 shows a general overview of the proposed architecture. The batch layer uses a snapshot to transform the structure and data present in D_{src} at T_{snap}, while the streaming layer transforms queries that add new data or transform existing data or structure. The latter transformations are stored in sequence until the batch layer is finished, after which the queries are executed on the newly created

D_{trans}. Again, all queries arriving after T_{snap} are still being executed on D_{src} as well, since the latest data needs to be readily available for the application during the transformation. Once the batch layer is finished and while the stored queries from the streaming layer are executing on D_{trans}, a changeover process is started. This changeover process stops all queries from being sent to D_{src} and completes the changeover to D_{trans}. Figure 3.2 shows the sequence diagram of all the architectural components.

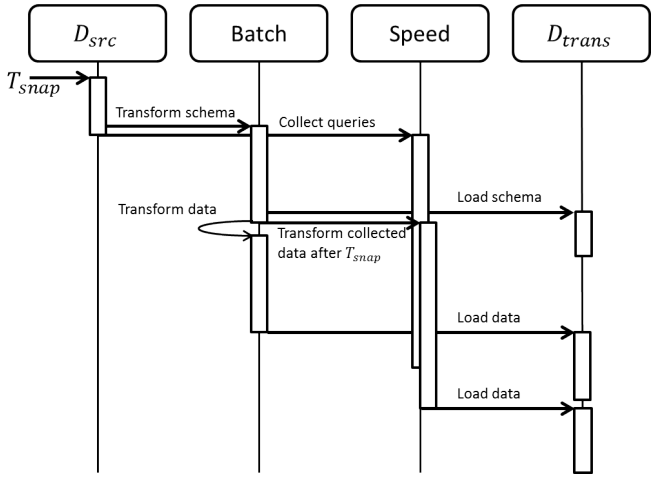

Figure 3.2: Sequence diagram detailing the functionality of the architectural components during the transformation

3.3.2 Transformation approach

Two main approaches can be identified when looking at the actual transformation of a data store: direct transformation and transformation through a centralized data model. The first is fairly straightforward as one data store is directly mapped onto another. Unique properties of a certain data store can be mapped onto specific traits of the other entirely. However, for each new supported data model, this approach would require a new implementation for transforming the new data model into each of the already supported models. For example, when a transformation is needed between SQL and Cassandra, a direct transformation can be implemented in both directions, but when support for MongoDB is required, a transformation also needs to be implemented for both SQL and Cassandra. The amount of effort to support new data store technologies would only grow exponentially. Using a centralized data model would solve this issue by first transforming the structure and data of each data store to the data model, after which it is transformed into the new data store. Supporting new data stores would then only require a transformation

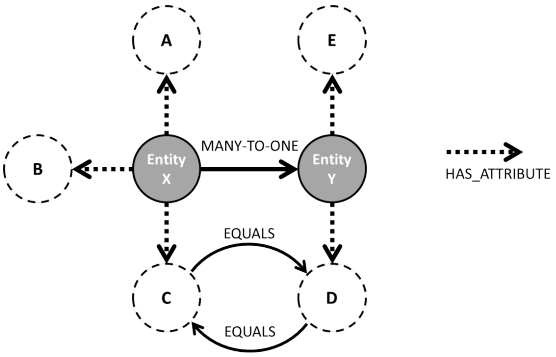

Figure 3.3: Canonical model for the structure of a data set.

towards and from the centralized data model. While this solution does support the extensibility of additional data stores being added, it also has several drawbacks. Firstly, the solution requires an extra transformation for every conversion between data stores introducing additional overhead. Secondly, while transforming to the centralized data model, it is not possible to assume anything about the unique characteristics of D_{trans} as the destination data store is not yet known at that point.

Within the centralized data model, two possibilities exist: an abstract or a canonical model. An abstract model can represent the most common characteristics shared by several data stores, while the canonical model aims to support every specific characteristic of each supported data store. Although the abstract data model allows a general representation of the data store's structure and data, not all unique characteristics of the data stores can be supported and any related advantages are also lost. With this in mind, the approach with a canonical model is preferred. The complexity in developing such a solution is mostly contained in the first stage. Once the canonical model is in place, adding support for new data stores is significantly easier. Even if this approach performs worse time-wise, compared to a direct transformation, the architecture proposed in Section 3.3.1 still allows for the application to operate with minimal impact. That is, during the transformation, D_{src} is still the main data store, i.e., it still processes all the queries from the application, while the streaming layer transforms any queries that update or insert data in the data store.

The canonical model can be represented through a directional graph, clearly showing the relations between elements of the data model. This graph representation also allows reasoning on the data model as it mimics the properties of an ontology, a model describing a domain in classes with properties and relations [29]. The domain in this case is the data model of D_{src} and the reasoning allows for insights as the data model of D_{trans} is built. Figure 3.3 represents an example of a canonical model for the structure of a data set. The central element in this canoni-

cal model is the Entity. It represents a subject and is built up by different Attributes. An Entity also keeps information about its identifiers and Attributes through a relation $HAS\_ATTRIBUTE$. Another type of relation, $EQUALS$, indicates that two attributes contain the same information. Relations between Entities can also be represented with a specific type, such as ONE_TO_ONE, $MANY\_TO\_ONE$, and $MANY\_TO\_MANY$. While the data is not mentioned in Figure 3.3, it can be regarded as a combination of singular pieces of information, related to attributes as part of an entity (e.g., a row in an SQL table, or a document in MongoDB).

In a previous publication the canonical model was represented in an EER-like model [8, 30]. This approach was however later found to be too constricting for the canonical model. The current representation of the canonical model as a directional graph allows for extensive reasoning similar to ontologies which was not possible with the EER-like model. This will aid in the detection of relationships in the data schema and the transformation from the canonical model to D_{trans}. Moreover, while there is no way to prove the soundness and completeness of this representation, it is based on the relational algebra and is much easier to extend if a new type of relationship would be needed.

3.3.3 Workflow

This section summarizes the typical workflow of a transformation by the framework. The transformation process can be described in four steps:

1. **Initiate transformation:** the transformation is initiated, based on monitoring data or by request. A snapshot is taken from D_{src} and passed on to the batch layer. Until the handover, the final step, D_{src} acts as the main data store for the application(s), i.e., all queries are still passed on to this data store. However, all queries that insert or update data in the data store are also forwarded to the streaming layer as soon as the snapshot is initiated. Currently, queries that alter the schema of D_{src} are not allowed during the transformation process.

2. **Transform schema:** before the data can be transformed, the batch layer transforms the structure or schema of D_{src}. The streaming layer is only collecting queries, but not yet transforming them, as information is needed about the transformed schema of the data store.

3. **Transform data:** based on the transformed schema of D_{src}, a new data store, D_{trans}, is set up. Data from the D_{src} is transformed in the batch layer, while recent queries that were collected in the streaming layer are transformed as well. However, resulting transformed queries from the streaming layer are only inserted in D_{trans} after the transformed data from the batch layer has been inserted into D_{trans}.

4. **Handover:** as soon as the data from the snapshot is transformed and put into D_{trans}, the handover is initiated. All queries are then redirected to D_{trans} with respect to any queries still in queue at the streaming layer.

At this point, the application still queries in the language of D_{src} which leads to the following possible scenarios:

- The application maintains the original language and every query is translated by the streaming layer. The application thus remains dependant on the proposed architecture with a minimal overhead introduced by the continuous transformation.
- The application was prepared for this transformation and changes its querying language to that of D_{trans}.
- The application communicates to the data store through an abstract data layer, such as Hibernate ORM/OGM, PlayORM or Apache Drill.

It is clear that in order to eliminate the need for the application to change, the continuous transformation of the queries is required. In practice, this translates to the transformation of data retrieval queries, such as SELECT-queries in SQL. Section 3.4.3 details the transformation of data retrieval queries in order to eliminate the need for applications' redesigns.

3.4 Transformation algorithm

3.4.1 Schema queries

An overview of the transformations to and from MySQL, Cassandra and MongoDB is given below. The transformations of MySQL and Cassandra have been detailed in a previous publication [8], but since the canonical model has changed from an EER-like model to graph representation, the implementation has been completely redone. However, the transformation rules below are still valid and similar to the ones described in [8], therefore only a summary of the transformations for MySQL and Cassandra is given.

3.4.1.1 SQL Transformations

The Structured Query Language (SQL) [31] is a language for managing relational databases based on relational algebra. SQL is a standard of both the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) [31]. The popularity of SQL has spawned many dialects for its different implementations, such as MySQL, Microsoft SQL Server and PostgreSQL. The transformation detailed below only uses elements from the SQL standard.

To canonical The following schema shows how the different data structures from SQL are mapped onto the canonical data model:

Table 3.2: Transformation schema from SQL to canonical model

SQL	Canonical
Table	Entity
Column	Attribute
Foreign keys	Relations

The first two transformations are straightforward: a table is a collection of columns, similar to an entity with its attributes. In SQL the relationships are defined through foreign keys, primary keys, and table use. Three types of relationships exist: one-to-one, many-to-one and many-to-many. The canonical model has an explicit representation of these relationships, therefore the objective of the transformation is detecting the context of the foreign keys as defined by the SQL standard [31] and translating to the correct relation type.

From canonical Similar to the transformation towards the canonical model, entities are translated into tables with columns based on the attributes. These transformations are the exact opposite of those listed in Table 3.2. The relations are implemented using the foreign keys according to the SQL standard [31].

3.4.1.2 Cassandra Transformations

Cassandra is a column-oriented data store originally developed at Facebook [32]. While showing many similarities with classic databases, it does not support a full relational data model. It is aimed at large-scale implementations across hundreds of physical servers with high-availability services.

To canonical Columns, grouped in column families, are the building blocks of a Cassandra data store, similar to columns and tables in SQL respectively. This similarity is continued in the translation towards the canonical model, where column families become entities and columns become attributes.

Table 3.3: Transformation schema from Cassandra to canonical model

Cassandra	Canonical
Column family	Entity
Column	Attribute
Index column families	Relations

Important to note is that there is no explicit way to infer relations from the Cassandra data model, due to the lack of support for a relational data model. There are however several indicators that relations are present in the Cassandra data model: *index column families*. These column families contain duplicate columns from two or more column families involved in the relationship and are identified by a primary key that spans multiple columns, also referred to as a *composite key*, containing the primary key data of the related column families. This denormalization eliminates the need for join-like queries (cfr. SQL) optimizing the data store for read performance. While their presence is a good indicator, composite keys are also used for other purposes, such as column family sorting. This makes it significantly difficult to define a generally automated way of detecting relations in Cassandra. Currently, the automated detection of relations from Cassandra is not supported unless the following naming is used for the index column family "$< entity\ x > \_ < entity\ y > \_index$".

From canonical Translating into Cassandra from the canonical data model, entities are transformed into column families with columns defined by the attributes. As mentioned before, the Cassandra data model does not allow for the explicit representation of relations, but it is possible to represent them through index column families with composite keys. Relations in the canonical model trigger the creation of these additional index column families containing data from the entities involved when translating into Cassandra. The transformations in Table 3.3 are the exact reverse.

3.4.1.3 MongoDB Transformations

An additional NoSQL data store was added to the list of supported data stores: MongoDB is a document-based data store [33]. It stores data as a key paired with a document containing key-value pairs, key-array pairs, or even nested documents. MongoDB accepts JSON documents as data for its collections, represented as binary-encoded JSON (BSON) behind the scenes.

Listing 3.1: Example of a document in JSON

```
{
 "id": 00001,
 "device_id": 00023,
 "device":
  {"device_id": "00023", "platform": "cisco", "name
      ": "router"},
 "network_info":[00001, 00002, 00003, 00004]
}
```

Listing 3.1 shows an example of a JSON document with several different elements. First and foremost, a document has fields linked to values. A field can have a single value with familiar data types, such as integers and strings, but also contain an array of values (e.g., ”network info” in Listing 3.1) or even embedded documents (e.g., ”device” in Listing 3.1). MongoDB is praised for its flexibility as collections do not impose any data model on the stored JSON documents. This lack of data model has some significant effects on the complexity of the transformation.

To canonical In SQL and Cassandra a dump of the data store includes the schema or model, i.e., every tuple (or row) of data is defined by a certain amount of attributes (or columns). This means the structure of the data is known without even looking at the data itself. MongoDB has a flexible data schema in its collections, i.e., collections do not enforce document structure. It is therefore impossible to know all the attributes of the documents without looking at the documents themselves. In order to derive the canonical model from a MongoDB data store all data in the collections needs to be checked. For each document in each collection a list of all the keys needs to be made that represent the attributes of the entity in the canonical model. It is clear that checking only one document for each collection does not suffice as documents may also include different keys within the same collection. It is to be expected that iterating over the entire data set in MongoDB as to acquire the canonical model will have negative impact on the transformation time compared to SQL or CQL. The schema in Table 3.4 details the transformation to the canonical model.

Table 3.4: Transformation schema from MongoDB to canonical model

MongoDB	Canonical
Collection	Entity
Document field	Attribute
Embedded Document	One-to-one relation Many-to-one relation
Document referral array Embedded document array	One-to-many relation Many-to-many relation

The flexible schema also limits the possibility of accurately defining relations between documents and/or entities. References to other documents can be made through document referral, but this is not explicitly mentioned in the document as is the case with foreign keys in SQL. Therefore there is no way to automatically detect a relation based on a singular document reference in a field. Another way of defining relations between collections however is through embedded documents. If a field in a document contains an embedded document this can be indicative of a

one-to-one or a many-to-one relationship. Additionally, arrays in documents containing multiple references to other documents or containing embedded documents can indicate one-to-many or many-to-many relations. Note that both many-to-one as one-to-many relations are mentioned here. The flexibility of the data model allows us to represent this relationship in two ways: denormalized with redundant data stored for low read query latency, or through the array data structure for hierarchical data sets. It also becomes clear that a many-to-many relation actually is two one-to-many relations between two entities directly. Thanks to the array data structure in MongoDB, no additional entity is needed to represent the relationship as is the case in SQL.

From canonical As for the transformation towards MongoDB, the flexible data schema simplifies the process. Documents contain keys based on a subset of the attributes defined in the canonical model and are added to their collection based on the entity. If a collection does not yet exist, one is made automatically in MongoDB. No schema transformation is therefore needed as all information is derived from the data. The data is transformed in JSON documents and pushed in MongoDB.

Table 3.5: Transformation schema from canonical model to MongoDB

Canonical	MongoDB
Entity	Collection
Attribute	Document field
One-to-one relation	Document integration
One-to-one relation Many-to-one relation	Document referral Embedded document
One-to-many relation Many-to-many relation	Document referral array Embedded document array

As mentioned before, relations between documents and collections can not be explicitly expressed in MongoDB. Similar to Cassandra, a denormalization of the canonical model can be used to indicate these relations, or similar to SQL, references can indicate a relation based on an id. Depending on the application requirements a choice can be made to either normalize or denormalize the MongoDB data store. For example, in a hierarchical data set it would be wise to normalize the data store and work with references, but, if read performance is a non-functional requirement, embedding sub-document information in documents yields less queries. The schema in Table 3.5 details the transformation from the canonical model.

It is important to note that previous subsections describe the transformation to and from the canonical model for three specific technologies, but that the algorithm

is inherently technology-independent. If a new technology were to be supported, a transformation similar to the ones above should be implemented. Once this is done, transformations to and from each already supported technology are possible.

3.4.2 Data insertion queries

The previous section discusses the transformation of the schema of a data store, but the data in the snapshot of D_{src} and new data received after T_{snap} needs to be transformed as well based on the created canonical model. In order to maintain the flexibility and extensibility of the implementation, data insertion queries are transformed to an intermediate state called *tuples*. These tuples contain all the key-value pairs contained within the insertion queries. From these key-value pairs queries are made up for D_{trans}. Below is an example for an INSERT query from SQL (D_{src}) in Listing 3.2 transformed into MongoDB (D_{trans}) in Listing 3.3:

Listing 3.2: Example of an INSERT query in SQL

```
INSERT INTO log (l_id, registrationtime, eventcount, device_id)
 VALUES ("2583301", "2010/10/28 20:22:10", "1", 30601);
```

Listing 3.3: Example of an INSERT query in MongoDB

```
db.log.insert({l_id: "2583301", registrationtime:
    "2010/10/28 20:22:10", eventcount: "1" ,
    device_id: 30601})
```

The data that is now injected into MongoDB may not yet complete. The column "device_id" in SQL, and corresponding document field in MongoDB, has been defined as a foreign key as part of a many-to-one relationship. On the one hand, if the goal is to create a hierarchical data store, a document reference would be sufficient. On the other hand, if query performance is the goal, a more thorough solution would be to store the "device" information as an embedded document. Listing 3.4 shows the query that updates the document in the "log" collection with an embedded document.

Listing 3.4: Example of embedding a document in MongoDB

```
db.log.update({device_id: 30601}, { $set: {device: {
    id: 30601,platform: "Cisco",location: "Brisbane
    ",customer: "Thomas"}}})
```

As the aim of this chapter is to decrease query latency for legacy applications and data stores, the implementation of the algorithm uses the embedding of documents instead of document referral.

3.4.3 Data retrieval queries

As mentioned in Section 3.3.3, once the handover to D_{trans} is complete the application is still querying in the language of D_{src}. There are several solutions to resolve this, but this chapter's premise is to eliminate application redesign. This means continuous transformation needs to be implemented, i.e., the continuous translations of queries in the query language of D_{src} into queries of D_{trans}. The data insertion queries have been handled in Section 3.4.2 and in this section the data retrieval queries will be detailed. Listing 3.5 shows a standard SELECT query in SQL.

Listing 3.5: Example of a SELECT query in SQL

```
SELECT log.l_id, device.location FROM log INNER JOIN device
  ON log.device_id=device.id
```

Similar to all queries, a transformation is made towards a generic representation. From this generic representation a data retrieval query is made for D_{trans} (e.g., MongoDB). The complexity in these selections comes from the joins of different entities, but Section 3.4.2 detailed that embedded documents were used for the representation of relations. The corresponding MongoDB query is written in Listing 3.6.

Listing 3.6: Example of a SELECT query in MongoDB

```
db.log.find({ },{ l_id: 1, device.location: 1 })
```

Since the document is embedded, it is clear that less calculations are needed to reach the same results. Section 3.6 shows the impact of the simplified querying.

3.5 Implementation details

3.5.1 Technology choice and motivation

As mentioned in Section 3.3.1, an implementation of the Lambda architecture is used as part of the Tengu platform [10]. As this implementation is technology-independent concerning the different layers, a decision needs to be made as to which technologies are used.

The technology used for the batch layer needs to be able to transform a data store from a legacy application efficiently. The MapReduce model, introduced by Google [34], is one of the best known programming models for Big Data analysis with Hadoop as the implemented open-source framework. In the previous implementation of the transformation algorithm MapReduce on Hadoop was used [8]. However, Spark is considered to be the successor of Hadoop MapReduce with execution times 10 up to 100 times faster through in-memory computing [35].

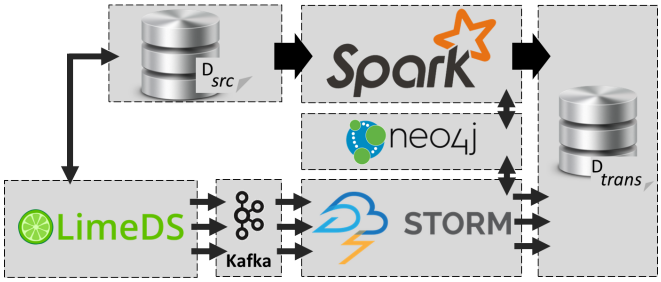

Figure 3.4: Instantiation of the framework with all the implemented technologies.

For the streaming layer, many of the previously mentioned technologies for the batch layer have (near) real-time streaming variants. Although many support streaming, for as many this has never been the sole focus. Storm on the other hand, is an analysis framework entirely built around the idea of (near) real-time analysis of streams. It was originally developed at Twitter® and is now part of the Apache project. This aside, implementing Storm as part of this proof-of-concept clearly shows both layers can be entirely different and independent technologies. Both Spark and Storm use Java, which means code is reusable across both layers.

An overview of all the integrated technologies is shown in Figure 3.4. Aside from the aforementioned analysis technologies, several supporting technologies are mentioned as well. The LimeDS framework allows for wiring different data-driven services together in an easy way [36]. In this implementation it is responsible for directing queries to D_{src} and the streaming layer. It also manages the synchronization between the batch and streaming layers. A second supporting technology integrated in the implementation is Apache Kafka [37]. Kafka is a publish-subscribe messaging system implemented as a distributed commit log. Storm and Kafka naturally work well together as both have strong guarantees on message processing. Finally, the canonical model is stored in a graph data store. In this particular implementation Neo4j was chosen for its maturity, extensive documentation, solid performance, and supporting community [38].

3.5.2 Transformation algorithm

The pseudo code in in Algorithm 3.1 can be clearly divided in two parts: the schema transformation and the data transformation. During the schema transformation, schema queries from the D_{src} snapshot are translated into the canonical model and stored in Neo4j. From this representation, the schema is built for D_{trans}. Once the D_{trans} schema is ready, the transformation of the data in the D_{src} snapshot is started. All data queries are first translated to a generic tuple representation based on the canonical model, and then matched on the D_{trans} schema.

read snapshot D_{src}
for each schema query in the snapshot **do**
    transform to canonical model
end for
store schema in canonical model
detect relations in canonical model
for each entity in canonical model **do**
    transform to D_{trans}
end for
load schema in D_{trans}
for each data query in the snapshot **do**
    transform to tuple in canonical model representation
end for
for each tuple **do**
    transform to D_{trans}
end for
load data in D_{trans}

Algorithm 3.1: Pseudo code describing the schema and data transformation in the batch layer

The additional step of translating to a generic representation is necessary for the data as well as to maintain code independence between data store technologies.

The code also contains several for-loops, but these loops are distributed and executed across the entire Spark cluster in parallel. This is especially important for the transformation of the data contained within the snapshot, as the schema information in a snapshot is negligibly small in most cases compared to the data. Each slave in the cluster gets a small subset of the D_{src} snapshot and transforms this subset towards the canonical model, and on from the canonical model to D_{trans}.

Parsing the query language of D_{src} in the transformation to the canonical model is done through ANTLR [39]. ANTLR generates a parser/lexer in Java, based on a grammar file containing a description of the structure of the language to be parsed. In this chapter grammar files were used for MySQL and the Cassandra Querying Language (CQL). There is no grammar for MongoDB as it uses JSON to store the documents in its collections and no data model is forced upon the documents.

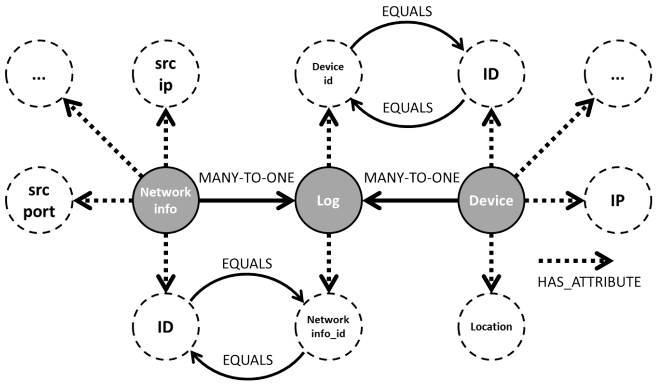

Figure 3.5: Partial canonical model of the network logging data store.

3.6 Evaluation

3.6.1 Experimental setup

The implemented instantiation of the architecture was deployed on the Virtual Wall. The iLab.t Virtual Wall facility [2] is a generic test environment for advanced network, distributed software and service evaluation, and supports scalability research. The Virtual Wall contains 300 nodes with varying hardware specifications. The server specifications in these experiments were as follows: Dual CPU (Quad core) with 12GB of RAM and 1x225GB disk. Four physical nodes were used for a dedicated 3-worker Spark cluster, two nodes for the Apache Storm cluster, and single-node instances for MySQL, MongoDB, Neo4j and a Cassandra data store.

3.6.2 Use Case description

This use case shows the application of the transformation algorithm on a data store containing network logging information. Currently, the network monitoring platform uses a relational data store in MySQL to save information, but the query performance is no longer sufficient for real-time querying and feedback as responses take several minutes. The aim is to lower query latency by transforming the MySQL relational datastore (D_{src}) into one of the supported data store technologies, whichever yields better results. As a reference Figure 3.5 shows a partial canonical model after the transformation from MySQL. Three entities can be identified: device, network_info and log. Device contains information about a certain network device, such as a router, while network info stores information about the logged package, containing amongst others a source/destination ip and port, and a

[2] http://ilabt.iminds.be/

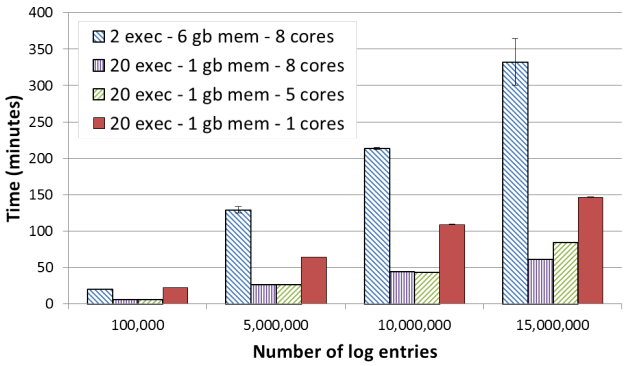

Figure 3.6: Graph showing the transformation time of an SQL snapshot data store to MongoDB for different Spark configuration parameters. They express the amount of parallel executors that are used and how much memory and how many cores are available to each executor.

protocol. An example of the log can be seen in Listing 3.1. The sizes of the data sets used for the evaluation contain 100.000, 5 million, 10 million and 15 million logs.

3.6.3 Results

Spark resource tuning Spark has a large number of configuration parameters that influence resource utilization with drastic results on execution performance. Figure 3.6 shows the execution times for the transformation of the SQL snapshot described in Section 3.6.2 with varying data set sizes expressed as a number of logs and for different configuration parameters. The impact of the configuration parameters can be clearly seen in the graph. For example, a snapshot containing 5 million logs is transformed in around 2 hours (129 minutes) with 2 executors, each having access to 6 GB of memory and 8 cores, while the same snapshot only requires around 27 minutes of execution time with 20 executors with 1 GB and 8 cores. Both executions do however use the maximum memory resources available in the entire cluster, taking into account the standard limitations defined by the Spark cluster, but the distribution of the resources also factors in. Spark thrives on in-memory computing, but for the computation of the transformation algorithm it clearly does not require 6 GB of memory per executor. One GB is enough for 20 parallel executors to outperform the previous configuration given the size of the snapshot. Allocating too much memory to an executor often results in excessive garbage collection delays which can be clearly seen in these results. Moreover, when working with a larger number of executors the standard deviation remains smaller because delays in a specific executor can be easily caught by the other

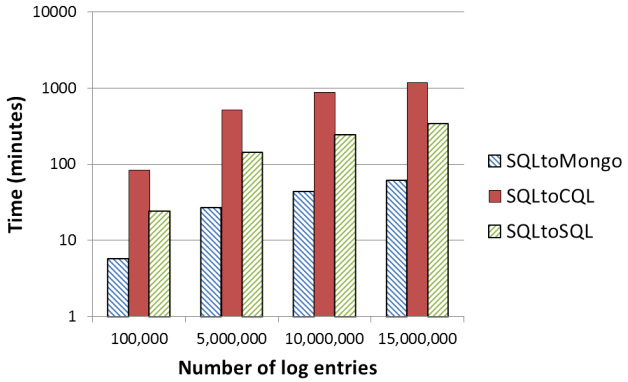

Figure 3.7: Graph showing the transformation time of an SQL snapshot data store to MongoDB, Cassandra (CQL) and SQL with 20 Spark executors each with 1GB of memory and 8 cores.

remaining executors.

Additionally, the amount of cores also influences the parallelism in Spark. When using HDFS with Spark it is recommended to not use more than 5 cores per executor as HDFS does not deal well with excessive amounts of concurrent threads [40]. Limiting the amount of cores per executor to 5 keeps the execution time at an average of around 27 minutes. So even though the parallelism is decreased, the execution time remains the same. However when a larger number of logs is considered, e.g., 15 million, the 8 cores execution outperforms the 5 cores configuration. For this data set size it seems HDFS is still able to scale, but with growing data sets it is important to take account of this parameter.

Decreasing the number of cores even further to only 1 core per executor, increases the transformation time back to around 1 hour (64 minutes) for 5 million logs as only a limited amount of tasks are allowed to execute in parallel. Figure 3.6 clearly shows the impact of Spark resource tuning on the transformation time. The influence of these parameters on memory management in Spark has also been extensively researched in previous publications [40, 41]. In general, increasing memory size in a Spark cluster will lower the execution time linearly until the entire data set is able to be loaded in memory. Increasing the memory size further will introduce garbage collection delays as seen in Figure 3.6. For the algorithm in this chapter with the described use case a solid configuration was found for 20 executors each with 1 GB of memory and 8 cores. However, when scaling to larger data stores these parameters need to be optimized continuously in order to achieve the best performance.

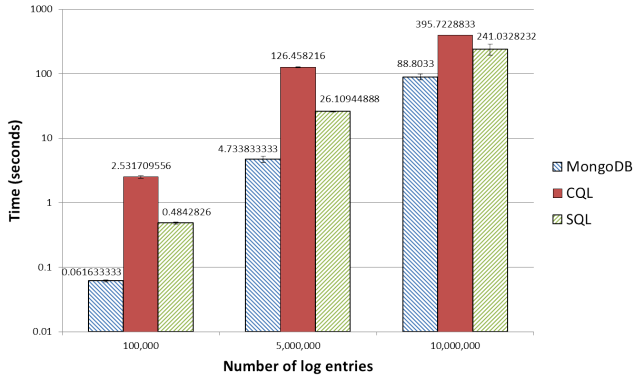

Figure 3.8: Query latency for JOIN-like query in different data stores: SQL, Cassandra (CQL) and MongoDB.

Snapshot transformation performance Figure 3.7 shows the execution times for the transformation of the SQL snapshot described in Section 3.6.2 to MongoDB, Cassandra (CQL) and SQL with varying data set sizes expressed as a number of logs. The transformation towards SQL is used to check the correctness of the algorithm as we expect to get an exact copy of the snapshot. All execution times show a linear trend with the increasing data set size, but the execution time of a transformation towards to MongoDB is significantly smaller compared to SQL and CQL. As described in Section 3.4.1.3, MongoDB is praised for its flexibility because collections do not impose any data model on the stored documents. While in a generic transformation, information from the canonical model is used to construct a data model for D_{trans} and to make sure data adheres to this data model, this is less so for MongoDB as no strict data model is required. The canonical model is stored in Neo4j, so in order to retrieve this information a connection to this data store is needed. MongoDB limits the number of connections that are required during its transformation from the canonical model, therefore lowering the total execution time. Moreover, the amount of queries generated while transforming to CQL is higher compared to MongoDB and SQL because of the denormalization of data stored in the index column families representing the many-to-one relations as mentioned in Section 3.4.1.

Query latency performance The ultimate goal of the transformation is to reduce query latency by transforming schema and data to a different data store technology. Figure 3.8 shows the average query latency for a JOIN query in SQL requesting all the logs joined with the information from the devices. This is a very expensive operation in SQL causing an exponential growth of the execution time with growing data sets. The JOIN query in SQL can however be translated into a selection query in MongoDB, still returning the same data, as all data from the

Table 3.6: Average execution time of different Storm bolts for the transformation of SQL queries, through the SQLMapBolt, into MongoDB, Cassandra (CQL) and SQL.

	SQLMapBolt	MongoReduceBolt	CQLReduceBolt	SQLReduceBolt
Time	37.568 ms	14.458 ms	80.133 ms	79.112 ms

many-to-one relation is embedded in the documents. The denormalization that was introduced by the transformation pays off in query latency as only one entity of a data store technology needs to be consulted to retrieve the same data. Similarly, in Cassandra the same data can be retrieved by querying the index column family that represents the many-to-one relationship. However, not using a partition key to retrieve data from a column family is a heavy operation in Cassandra, eliminating it from consideration for this use case.

Continuous transformation performance After the transformation of the snapshot and the handover, the application still queries in the language of D_{src}. Considering the use case, while there is a significant time gain transforming the data store to MongoDB or Cassandra, the overhead of transforming queries from the application needs to be limited in order to benefit from this transformation. The evaluation of the continuous transformation was performed on a two node Apache Storm cluster with standard configuration where each bolt was assigned a single worker in the cluster. Table 3.6 shows the average transformation time of a single query in every step of the transformation. For example, in a transformation from SQL to MongoDB, a query would pass through the SQLMapBolt, mapping the query onto the canonical model, after which it will be reduced towards MongoDB by the MongoReduceBolt. This yields a total overhead of 52.026 ms (37 ms and 14 ms respectively). Considering the query performance from Figure 3.8, it is clear the results of the transformation approach in this chapter benefits the application's query latency, increasing the general performance.

3.6.4 Discussion

The results in Section 3.6.3 clearly show the ability of the proposed algorithm to transform schema and data of a data store into a technology that yields better query latency performance as well as support for continuous transformation of application queries within a reasonable time frame. While several limitations to the current system exist, this section discusses those limitations and provides possible solutions on how to mitigate them.

The approach of the proposed algorithm in this chapter, detailed in Section 3.3.2, is theoretically slower than the direct approach as it requires one additional transformation to or from the canonical model. Direct transformations are however less

extensible towards future technologies as support for a new data store technology requires an entirely new implementation to transform to and from each existing technology. The Schema Conversion Tool provided by Amazon, discussed in Section 3.2, can be regarded as a bundle of direct transformations between dialects of SQL. This tool may achieve a faster performance compared to the approach described in this chapter, but contrary to the proposed approached only SQL dialects are currently supported and extending the tool would require an entirely new code base. Moreover, the additional latency introduced by the described approach in this chapter is alleviated by the use of the Spark platform. Spark allows for in-memory computing yielding faster execution times but its clustered architecture also allows for scaling towards specific time constraints with minimal effort [41].

A second limitation is that relations between entities in the canonical model are determined by the explicit and implicit use of certain data structures in the data schema (e.g., foreign keys in SQL, composite keys in Cassandra, arrays in MongoDB). However, it is conceivable that the implicit use of these data structures may not always be found, especially in NoSQL data stores such as Cassandra and MongoDB. Given that specific situation the algorithm would currently only detect the entities for its canonical model with no relations between them. While still being able to transform these entities to another data store technology, it might not yield a better query latency performance. An interesting extension of the algorithm would therefore be an automated detection of relations in the canonical model based on the read queries effectively optimizing the data schema based on its use. For example, SQL retrieval queries with JOIN operations indicate a relationship even if foreign keys were not defined. For NoSQL stores this needs to be derived from the sequence of queries that are often requested in succession. These chains of queries indicate the potential existence of a relationship between the entities. The extension of the algorithm to automatically detect relationships in the canonical model based on querying behavior is deferred to future work.

Finally, the current algorithm is not equipped to deal with queries that alter the data schema of D_{src} while the transformation process is in progress. While creating the data schema of D_{trans} any changes to the schema of D_{src} would potentially create inconsistencies while adding the data to D_{trans}. It was therefore decided to deny any queries that alter the schema until the handover is completed. The continuous transformation could then deal with the schema altering queries which will reflect in both D_{trans} as in the canonical model.

3.7 Conclusion and future work

This chapter introduces an approach and algorithm for schema and data transformation as a means to support dynamic data storage and polyglot persistence. The approach uses an intermediate canonical model to ensure the flexibility and

extensibility of the implementation towards future supported technologies. In order to support a new data store technology, one only needs to implement a transformation towards and from the canonical model. In previous work, support for SQL and CQL were already discussed, but the implementation has been revised as part of the newly changed canonical model. The chapter also introduces support for MongoDB, a NoSQL document data store. The transformation algorithm is implemented as a Lambda architecture with a batch and speed layer in order to support live applications without downtime and the need for code changes. A network monitoring platform is considered as a use case and shows a significant performance increase after the transformations to both CQL and MongoDB. The overhead introduced for the continuous transformation is limited to a maximum of around 100 ms. The time to transform a snapshot heavily depends on D_{src} and the chosen D_{trans} and is influenced by the strictness of the data models.

For future work, now a transformation algorithm has been defined and implemented, an interesting application would be to fully support dynamic data storage with regard to supported implementations, i.e., an automated system that stores data in the most optimal format at any given time. Additionally, while data relations are now inferred from defined uses of structures in a data store technology (e.g., foreign keys, composite keys, arrays), the best way to learn the relations in a data set is through its use. Future work will also focus on detecting relations in the canonical model based on reading queries. These changes will be reflected in the transformed data store with the ultimate goal of increasing query performance even further.

Acknowledgment

The work in this chapter has partly been funded by the iMinds SEQUOIA research project.

References

[1] R. Cattell. *Scalable SQL and NoSQL Data Stores*. SIGMOD Rec., 39(4):12–27, May 2011. Available from: http://doi.acm.org/10.1145/1978915.1978919, doi:10.1145/1978915.1978919.

[2] Y. Li and S. Manoharan. *A performance comparison of SQL and NoSQL databases*. In Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on, pages 15–19. IEEE, 2013.

[3] C. Nance, T. Losser, R. Iype, and G. Harmon. *NoSQL vs RDBMS: Why there is room for both*. In SAIS 2013 Proceedings, 2013.

[4] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. *F1: A Distributed SQL Database That Scales*. Proc. VLDB Endow., 6(11):1068–1079, August 2013. Available from: http://dx.doi.org/10.14778/2536222.2536232, doi:10.14778/2536222.2536232.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. *Spanner: Google&Rsquo;s Globally Distributed Database*. ACM Trans. Comput. Syst., 31(3):8:1–8:22, August 2013. Available from: http://doi.acm.org/10.1145/2491245, doi:10.1145/2491245.

[6] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1 edition, August 2012.

[7] R. Sellami and B. Defude. *Using Multiple Data Stores in the Cloud: Challenges and Solutions*. In A. Hameurlain, W. Rahayu, and D. Taniar, editors, Data Management in Cloud, Grid and P2P Systems, volume 8059 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin Heidelberg, 2013. Available from: http://dx.doi.org/10.1007/978-3-642-40053-7_8, doi:10.1007/978-3-642-40053-7_8.

[8] T. Vanhove, G. Van Seghbroeck, T. Wauters, and F. De Turck. *Live datastore transformation for optimizing big data applications in cloud environments*. In Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on, pages 1–8. IEEE, 2015.

[9] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester. *Tengu: An Experimentation Platform for Big Data Applications.* In ICDCS Workshops, pages 42–47. IEEE, 2015.

[10] T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck. *Managing the Synchronization in the Lambda Architecture for Optimized Big Data Analysis*. IEICE Transactions, 99-B(2):297–306, 2016.

[11] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. *EXPRESS: a data extraction, processing, and restructuring system*. ACM Transactions on Database Systems (TODS), 2(2):134–174, 1977.

[12] P. Vassiliadis. *A survey of Extract–transform–Load technology*. International Journal of Data Warehousing and Mining (IJDWM), 5(3):1–27, 2009.

[13] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. M. Ruwart. *A Technique for Moving Large Data Sets over High-performance Long Distance Networks*. In Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/MSST.2011.5937236.

[14] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai. *Workload-aware Live Storage Migration for Clouds*. SIGPLAN Not., 46(7):133–144, March 2011. Available from: http://doi.acm.org/10.1145/2007477.1952700, doi:10.1145/2007477.1952700.

[15] L. Zhang, C. Wu, Z. Li, C. Guo, M. Chen, and F. Lau. *Moving Big Data to The Cloud: An Online Cost-Minimizing Approach*. Selected Areas in Communications, IEEE Journal on, 31(12):2710–2721, December 2013. doi:10.1109/JSAC.2013.131211.

[16] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. *Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration*. Proc. VLDB Endow., 4(8):494–505, May 2011. Available from: http://dl.acm.org/citation.cfm?id=2002974.2002977.

[17] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. *Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms*. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM. Available from: http://doi.acm.org/10.1145/1989323.1989356, doi:10.1145/1989323.1989356.

[18] K. Ting and J. J. Cecho. *Apache Sqoop Cookbook*. " O'Reilly Media, Inc.", 2013.

[19] E. Rahm and P. A. Bernstein. *A survey of approaches to automatic schema matching*. the VLDB Journal, 10(4):334–350, 2001.

[20] Z. Bellahsene, A. Bonifati, E. Rahm, et al. *Schema matching and mapping*, volume 57. Springer, 2011.

[21] J. Schildgen, T. Lottermann, and S. Dessloch. *Cross-system NoSQL Data Transformations with NotaQL*. In Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR '16, pages 5:1–5:10, New York, NY, USA, 2016. ACM. Available from: http://doi.acm.org/10.1145/2926534.2926535, doi:10.1145/2926534.2926535.

[22] Y.-T. Liao, J. Zhou, C.-H. Lu, S.-C. Chen, C.-H. Hsu, W. Chen, M.-F. Jiang, and Y.-C. Chung. *Data adapter for querying and transformation between SQL and NoSQL database*. Future Generation Computer Systems, 65:111 – 121, 2016. Special Issue on Big Data in the Cloud. Available from: http://www.sciencedirect.com/science/article/pii/S0167739X16300085, doi:http://dx.doi.org/10.1016/j.future.2016.02.002.

[23] *Mongify*. http://mongify.com/ (Last Visited January 9, 2018).

[24] *Apache Sqoop*. http://sqoop.apache.org/ (Last Visited January 9, 2018).

[25] A. Leonard. *Hibernate OGM at Work*. In Pro Hibernate and MongoDB, pages 51–120. Springer, 2013.

[26] M. Hausenblas and J. Nadeau. *Apache drill: interactive ad-hoc analysis at scale*. Big Data, 1(2):100–104, 2013.

[27] N. Marz. *The Mathematics behind Hadoop-based systems*, December 2009. http://nathanmarz.com/blog/the-mathematics-behind-hadoop-based-systems.html (Last Visited January 9, 2018).

[28] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[29] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge acquisition, 5(2):199–220, 1993.

[30] P. P.-S. Chen. *The Entity-relationship Model - Toward a Unified View of Data*. ACM Trans. Database Syst., 1(1):9–36, March 1976. Available from: http://doi.acm.org/10.1145/320434.320440, doi:10.1145/320434.320440.

[31] *ISO/IEC 9075-1:2011 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. Technical report, ISO/IEC, 2011. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681.

[32] A. Lakshman and P. Malik. *Cassandra: a decentralized structured storage system*. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[33] K. Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[34] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[36] *LimeDS*. https://hub.docker.com/r/ibcndevs/limeds/ (Last Visited January 9, 2018).

[37] J. Kreps, N. Narkhede, J. Rao, et al. *Kafka: A distributed messaging system for log processing*. NetDB, 2011.

[38] J. Webber. *A programmatic introduction to Neo4j*. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, pages 217–218. ACM, 2012.

[39] T. Parr and R. Quong. *ANTLR: A Predicated-LL(k) Parser Generator*. Software - Practice and Experience, 25(7):789–810, 1995.

[40] S. Ryza. *How-to: Tune Your Apache Spark Jobs (Part 2)*, March 2015. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/ (Last Visited January 9, 2018).

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pages 2–2. USENIX Association, 2012.

4

Sequential Pattern Mining for Data Storage Optimization in Polyglot Persistent Environments

T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck.

Submitted to the Journal of Network and Computer Applications, September 2017.

⋆ ⋆ ⋆

The transformation algorithm from Chapter 3 proposes an approach for data stores from legacy applications to be transformed into modern data store technologies without any downtime or changes required for the application itself. This chapter builds on that idea by introducing the concept of polyglot persistence into the canonical model of the transformation approach. In this way the canonical model acts as an abstraction of the complex polyglot persistent environment. However, abstractions always introduce an overhead cost which needs to be mitigated. In order to minimize the impact of this overhead, this chapter details an approach that uses the transformation and canonical model to optimize the storage environment. By detecting relations between data, based on queries sent by an application, the canonical model is optimized specifically to eliminate complex queries across different storage entities. The described approach is limited to basic retrieval queries, such as 'select-from-(join)-where'-queries in MySQL and equiva-

Table 4.1: Query latency for a JOIN query in MySQL with $5,000,000$ records and a similar query in Cassandra and MongoDB where the relation is implemented with duplicated data [3].

	MySQL	Cassandra	MongoDB
Average query latency (s)	26.109	126.458	4.734

lents in CQL and MongoDB. Experiments with a MySQL host language show that the canonical model can be optimized in real-time.

4.1 Introduction

The introduction of the Not-only SQL (NoSQL) domain caused a paradigm shift in the data storage domain [1]. NoSQL is an encompassing term for all storage technologies that differ from the classic relational database management systems (RDBMS) in such a way that they no longer require data to adhere to a strict data model at all times. Additionally, they often relax the traditional ACID (Atomicity, Consistency, Isolation, Durability) properties on which RDBMS are designed into BASE (Basically Available, Soft state, Eventual consistency) properties. This allows these technologies to often better scale horizontally compared to RDBMS solutions, which makes them suitable for use in a big data environment which often stores data in a clustered way, i.e., across multiple physical servers.

The popularity of the NoSQL movement can be measured by the sheer amount of storage technologies available. Most of these technologies can be divided into four major categories [2]: key-value stores, document stores, column-oriented stores, and graph databases. Each category is tailored to certain data types and uses. For example, Cassandra is a column-oriented data store optimized for writes, but it relinquishes the responsibility of fast reads to the creators of the data model. This usually entails a highly denormalized data model and duplicated data, contrary to the beliefs in RDBMS. Therefore, Cassandra is often used in heavy write-systems, such as monitoring and social media analytics, where the amount of writes is significantly larger than the amount of reads.

It is clear that the popularity of the domain has also increased the complexity of data storage in general. In the RDBMS era storage planning involved the creation of a relational data model, whereas now, before a data model even needs to be created, a technology needs to be chosen based on how the data will effectively be used. The choice of technology has therefore become crucial in building the applications of today as it influences the performance of data access for these applications. Table 4.1 shows the impact of a technology choice on the latency of a complex JOIN-query in SQL and similar queries in Cassandra and MongoDB.

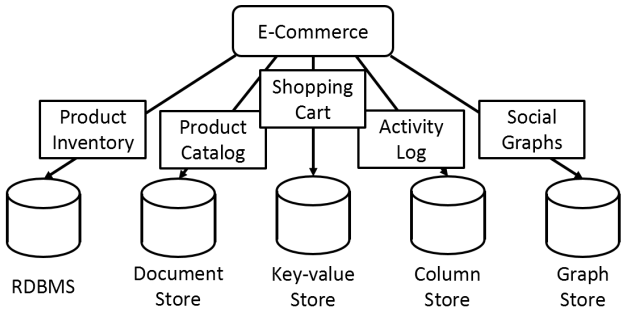

Figure 4.1: Example of polyglot persistence in an e-commerce platform architecture [4].

One of the current challenges in the big data domain is that there is a high variety of data types to be processed. An example of such an application is an e-commerce platform [4]. It contains data about products, purchases, and finances but also stores information on recommendations and advertisements. Not only is this data presented in different formats, it is also used in different ways. For example, product data does not change very often and therefore does not require a lot of writes, but it is being read a lot. To store all e-commerce data into one database technology would be detrimental to the application performance as the data is used in different circumstances [4]. A better approach is to store specific parts of the application data in different database technologies optimized for their use. Figure 4.1 shows an e-commerce platform supporting various data store technologies where the product catalog data is stored in a document store and the product inventory in a classic RDBMS. The state where one application stores its data or information in different database technologies is referred to as polyglot persistence [5]. It is considered to be one of the most important novelties of the NoSQL domain as it inherently changes the way applications handle data.

Although a polyglot persistent application benefits from better data access performance, assuming the choice for the underlying technology was done meticulously, managing this application becomes significantly more complex. The application communicates with several database technologies, each with different underlying data models (if any) and varying query languages. Additionally, as data is distributed across different technologies, aggregations need to be made to access related data. New applications can be built specifically for a polyglot persistent data environment, but, while legacy applications would clearly also benefit from this new paradigm, introducing polyglot persistence would require profound changes to the application. An abstraction layer, shielding applications from the technology-specific data stores, would lower the complexity, but should do so without introducing too much overhead in order not the diminish the gains brought by polyglot persistence. Secondly, specific functionality and characteristics should

remain intact as to not limit the expressiveness of the querying language of the application.

This chapter proposes an approach for an optimized polyglot persistent environment for both new and legacy applications. A canonical model acts as a part of an abstraction layer that shields applications from the intricacies of polyglot persistence. Additionally, an algorithm is detailed to detect relations between data based on usage independent of the different database technologies. The relational information is stored in the canonical model and serves the purpose of optimizing the data model across database technologies, for example through data duplication. The goal is to increase data access performance for applications by allowing access to a polyglot persistent environment and optimizing the data model specifically for the considered applications in order to mitigate the overhead introduced by the abstraction layer.

The work described in this chapter is complimentary to previously published work by the authors in which a transformation approach for data stores was defined based on a custom designed canonical model [3]. The transformation approach allows legacy applications to communicate with the canonical model without requiring any changes to the code or query language. The novel contributions of this chapter are threefold: the concept of the canonical model is extended to represent the polyglot persistent environment, a sequential pattern mining approach is introduced and implemented to automatically detect relations between data, and a data schema optimization algorithm for polyglot persistence is presented. Combining previous work and the new contributions in this chapter allows both new and legacy applications to benefit from a polyglot persistent environment albeit shielded from the inherent complexity.

The chapter is structured as follows: in Section 4.2 a motivation for the proposed approach is formulated together with details on the architecture. Section 4.3 presents the implementation of several components of the architecture. The evaluation of the sequential pattern mining approach is detailed in Section 4.4 while a discussion of the obtained results can be found in Section 4.5. Section 4.6 discusses related work on polyglot persistence and data store optimization techniques. Finally, in Section 4.7 the conclusions are formulated as well as planned future work.

4.2 Data Schema Optimization in Polyglot Persistence

4.2.1 Query abstraction and outline

An abstraction layer allows applications to remain unaware of the intricacies of their polyglot persistent storage. The abstraction layer ideally translates queries from applications into technology-specific query languages, aggregates results if

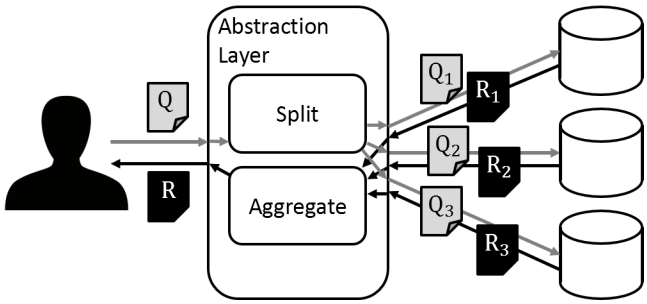

Figure 4.2: High-level overview of an abstraction layer approach to polyglot persistence where the application query Q is split into several technology-specific queries (Q_1, Q_2, Q_3). Results (R_1, R_2, R_3) are then aggregated and returned to the application (R).

needed, and returns them to the application in the expected format. Figure 4.2 shows a high-level overview of such an abstraction layer. A query is split up into technology-specific queries after which the results from those different queries is aggregated and returned to the user. The aggregation of query results is outside of the scope of this chapter. A very important property of this abstraction layer is the overhead it introduces on querying performance. If x_{As} is the average query time of application A with a single data store, x_{Ap} is the average query time of the application A in a polyglot persistent environment, and O_A is the overhead introduced by the abstraction layer for application A, then the following should apply:

$$O_A < \Delta x \quad \text{where} \quad \Delta x = x_{As} - x_{Ap} \tag{4.1}$$

If Equation 4.1 does not hold true, the query performance benefit of having the application use polyglot persistent storage is negated. Therefore, it is important that the most appropriate technologies are chosen for the different data formats and that the data schemas of all the chosen data stores are optimized and tailored towards the queries of the application, as to yield a maximal value for Δx. Previous work of the authors already covered the selection of the most appropriate technology for certain data formats and defined a transformation that allowed applications to switch between storage technologies without any required changes to the application [3]. The scope of this chapter focuses on database schema optimization in polyglot persistence. Specifically, it presents algorithms for data schema optimization based on relations between several entities within or across multiple data store technologies.

While relations are less explicitly defined inside data models as a result of the

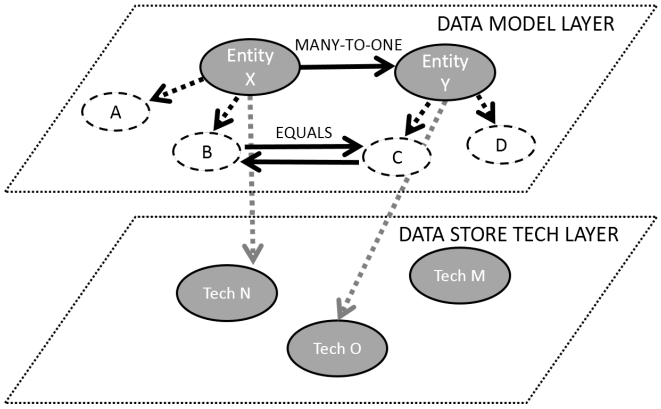

Figure 4.3: Layered canonical model for the schema and technology mapping of a data set.

NoSQL domain, it is indisputable that often data is still related. For example, in the e-commerce platform in Figure 4.1 the inventory data and product catalog are stored in different data stores, but several use cases exist where data from both stores is required. An application, or the abstraction layer, would then need to query both data stores and aggregate data from both technologies. Performance-wise it would be best to avoid these types of queries if the data store technology is not equipped to efficiently deal with them. In other words, the data store schema needs to be optimized by modeling relations in such a way that they limit the amount of entities that need to be consulted, ultimately limiting the amount of aggregation that needs to be performed.

However, as relations are less explicitly defined in NoSQL, discovering them is an important step before the schema optimization can be applied. Even in RDBMS, relations are often forgotten or not correctly defined in the data model (e.g., through foreign keys). Therefore, relations between data will need to be derived from context. One way to discover relations is to monitor incoming queries from the applications, learn the way the application uses the data stores and optimize the data schema to that specific use.

In the following sections an extension is defined for the canonical model from [3] to support a polyglot persistent environment. Next, based on the relations defined in the canonical model, an approach is detailed in Section 4.2.3 to select which relations need to be denormalized in the storage environment when certain constraints apply. Finally, as relations can or may not be explicitly defined in the data schema, Section 4.2.4 presents a relation detection system based on query use.

4.2.2 Canonical model

The canonical model described in [3] is designed to represent a technology-independent data model and and used in the abstraction layer during the transformation of one data store technology to another. The schema of the original data store is transformed to the canonical model and from there transformed to the destination technology. A live transformation process allows the application to continue querying in the original query language even though another data store technology is being used. The canonical model holds information on Entities who are defined by Attributes. Relations can be defined between Entities which are used to optimize the schema for the destination technology according to a set of rules. Relations can be of the type $ONE\text{-}TO\text{-}ONE$, $MANY\text{-}TO\text{-}ONE$ or $MANY\text{-}TO\text{-}MANY$ and are currently derived from data schema implementations based on the characteristics of specific data store technologies (e.g., foreign keys).

In previous work, the canonical model has had no knowledge of the concept of a specific data store technology. However, in polyglot persistence it is imperative that a query is directed to the most appropriate data store technologies. Figure 4.3 shows a new layer, the data store tech layer, in the canonical model representing the different technologies available in a polystore. It maps Entities in the data model layer unto specific data stores in the tech layer. The lowest granularity is the Entity, i.e., different entities can be stored in the same storage technology, but one entity can only be stored in one technology.

4.2.3 Data schema optimization procedure

Relation implementations in databases have been extensively researched by Mitoma et al. [6]. In the article, different ways are described to implement relations in data stores between an Entity$X$ and an Entity$Y$, where a relation implementation is defined as a structured collection of data item occurrences that establish a relation in the data store. Twelve defined implementations are divided into three categories of four implementations each: duplications, aggregations, and associations. Duplications store duplicate copies of all values in Entity$Y$ (or Entity$X$) that are related to values in Entity$X$ (or Entity$Y$) within the same vector. This can be done through vectors with fixed length, and the associated null value padding, or variable length vectors. The second category implements the relation as an aggregation, which again stores all values of Entity$Y$ (or Entity$X$) that are related to values in Entity$X$ (or Entity$Y$) within the same vector, but without duplication. The vectors can again be of fixed or variable length. In the final category, association, a relation is implemented by storing all related values in Entity$X$ and Entity$Y$ in separate records, referring to each other using mechanisms provided by the data store technologies, or pointers. Many of the defined implementations were not allowed at the time because of restrictions or a non-redundancy objective in the

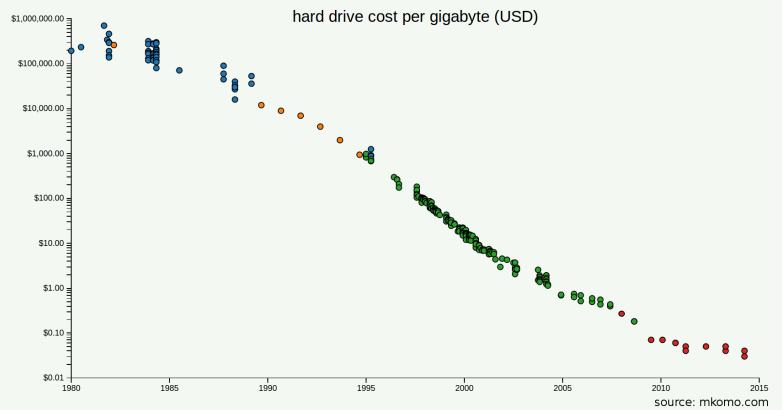

Figure 4.4: Storage cost per gigabyte between 1980 and 2014 [7].

RDBMS. However, in a NoSQL domain these implementations can be considered as many of these restrictions have been lifted through flexible data schemas.

In a virtually limitless storage environment the access cost can be minimized by duplicating all the data for all the relations. However, in a realistic storage environment choices need to be made as to which relation implementations yield the biggest benefit. To that end Mitoma et al. [6] also provided an integer programming solution to decide the best implementation for a relation, minimizing total access cost in such a way that total storage cost does not exceed a defined schema storage space capacity. It provides an interesting base for deciding which relation implementation needs to be chosen, but it has some assumptions that do no longer hold up in the current domain. For example, while storage in 1975 was a significant cost to deal with, in 2017 it has become almost negligible in comparison. Figure 4.4 shows a flow of the storage cost per gigabyte between 1980 and 2014 based on data retrieved by Komorowski [7]. It shows that the cost per gigabyte has decreased from over 700.000 \$/GB to about 0.03 \$/GB. However, even though data schemas in NoSQL no longer prohibit duplicate data and storage cost has decreased dramatically, it would still not be a good idea to duplicate data for each existing relation in the data schema since there is still a writing cost to be taken into account. When data is written to the data store, it needs to be propagated to all the replications making write and update operations more expensive. In order to limit the cost of writing operations to the data store, replicated data must be kept in bounds. Instead of limiting the optimization by storage space, the solution is adjusted to be bounded by a replication factor. However, a difference is made between explicit and implicit replication. Implicit replication is replication with the purpose of fault-tolerance and high-availability. Explicit replication is replication introduced to increase performance and simplify access to data. Implicit replica-

tion is often linked to the inner working of data store technologies and, aside from some configuration parameters, is not highly customizable. Explicit replication on the other hand is customizable and in control of an application or user. Therefore, in this chapter the integer programming formulation only considers explicit replication.

Another extension that was made to the work of Mitoma et al. is the addition of a new dimension: the technologies in the polyglot persistent environment. The access cost for similar queries in different NoSQL technologies can be vastly different and therefore needs to be taken into account when optimizing the schema. This can be done by adding a new dimension to the variables representing all the supported technologies of the polyglot persistent environment.

Combining the extensions listed above, the integer programming formulation becomes the following:

Given:

$R =$ number of relations in the data schema to be implemented (i.e., number of sequences detected)

$T =$ number of technologies supported by the polyglot persistent environment

$F_j =$ frequency of occurrence for relation j

$A_j =$ number of alternative implementations for relation j

$AC_{ijt} =$ access cost of implementation i in technology t for relation j

$ERC_{ijt} =$ explicit replication cost of implementation i in technology t for relation j

$ER_{max} =$ maximum number of explicit replications allowed for data

$$d_{ijt} = \begin{cases} 1, & \text{if implementation } i \text{ in technology } t \text{ for relation } j \text{ is chosen.} \\ 0, & \text{otherwise.} \end{cases}$$

$$\text{Minimize} \quad \sum_{j=1}^{R} \sum_{i=1}^{A_j} \sum_{t=1}^{T} d_{ijt} \cdot F_j \cdot AC_{ijt}$$

$$\text{Subject to} \quad \sum_{j=1}^{R} \sum_{i=1}^{A_j} \sum_{t=1}^{T} d_{ijt} \cdot ERC_{ijt} \leq ER_{max}$$

$$\sum_{i=1}^{A_j} \sum_{t=1}^{T} d_{ijt} = 1 \text{ (for all } j = 1, 2, ..., R)$$

One final difference compared to the work of Mitoma et al. is that the frequency of occurrence of a certain relation is enclosed within the access cost. How-

ever, this frequency is a constant for a certain relationship within this Integer Linear Program (ILP) formulation as it is not influenced by a certain implementation of that relation. In order to represent the solution in a clear way, the frequency has therefore been extracted from the access cost AC and is shown as F_j for any relation j.

4.2.4 Detection of (cross-technology) relations

The procedure detailed in Section 4.2.3 optimizes a data schema based on relational information between Entities from the canonical model, but, as previously mentioned, relations may not have been implemented into the data schema of a specific data store technology. Therefore relations need to be detected based on the actual use by the application.

There are two types of relations that need to be detected. The first type is explicitly defined as is the case with classic RDBMS. This definition can already be stated in the data model (e.g., foreign keys in SQL), but it can also be a part of the actual query through operations such as JOIN in SQL. In this section only the latter is considered as in the former case the relation would already be present in the canonical model [3]. The second type is implicitly defined as in query languages that do not support explicit definitions of relations, but do so by aggregating the results of a series of queries. For example, in the e-commerce example data on product inventory and product catalog is stored in two different technologies, but is often needed together to inform a customer of the specifications of a certain product and how many of it are still in stock. If enough customers come to the website on a daily basis, these two subsequent queries, one to the product inventory and the other to the product catalog, are often fired to the data stores. It is clear that the second type, implicit relations, are much more difficult to detect compared to the explicitly defined relations. It would require an approach where patterns are detected in subsequent queries over time.

Applying such an approach to support each individual query language requires a tremendous amount of effort tailoring it to the specifics of each language. The representation of queries in the canonical model, a technology-independent representation of the data model, allows for a generalist approach. Retrieval of data in the canonical model is represented as a GET-query. Listing 4.1 details an example of such a GET-query.

Listing 4.1: Example of a generic GET-query in the canonical representation

$$\{\texttt{GET}, E, P, S\}$$

It consists of the word GET to denote that it represents a GET-query, followed by a two arrays and a map. The first array (E) refers to one or more entities involved

in the query and in the second array (P) the projection of the query detailed. The final component is a selection map (S) that stores tuples of an attribute and another attribute or value. Currently, this displays and is limited to the required equality between the two for the proof-of-concept that is developed in this chapter. In a full implementation it would be advisable to expand this tuple to a triple, where the third element represents the logical comparison between the two previous elements.

Explicit relation The detection of explicit relations in the canonical representation of queries is straightforward. If a GET-query contains multiple entities in E, there is a clear indication that those entities contain data that is related. As soon as a certain threshold occurrence is detected for a certain combination of entities, it would be beneficial for the performance to add this relation to the canonical model and optimize the schema towards these type of queries. The evaluation in Section 4.4 will decide upon a specific threshold when this optimization step should be triggered. Listing 4.2 shows an example of a query where two entities, Entity$X$ and Entity$Y$, are explicitly mentioned. Attributes also contain an extra reference to clearly mark the Entity they belong to.

Listing 4.2: Example of a GET-query in the canonical representation with a possible explicit relation

```
{GET,[entityX, entityY],[entityX.attributeA],
    [(entityY.attributeB,entityX.attributeC)]}
```

Implicit relation Implicit relations can not be discovered through one single GET-query, but can only be derived from a sequence of two or more GET-queries. In Listing 4.3 an example of a sequence of GET-queries is detailed. The first GET-query is designed to retrieve one or several values from one entity (e.g., a referal ID). In subsequent queries the value(s) retrieved from the first query, e.g., ValueX, are used to retrieve additional related data.

Listing 4.3: Example of a sequence of GET-queries in the canonical representation with a possible implicit relation

```
{GET,[entityX],[attributeA,attributeB],[]}
{GET,[entityY],[attributeM,attributeN,attributeP],
    [(attributeM,ValueX)]}
```

A single occurrence of this sequence has no value, but when this sequence is frequent, it indicates that query results are possibly combined by the application. The only way to be absolutely certain is to compare the returned values as a result of the first query with ValueX in the next query. This would however increase the

overhead of the abstraction layer (O_A) immensely. In order to keep O_A as low as possible, this comparison is not performed, which might lead to several false positives, but it is expected that the relative frequency of these false positives will be significantly lower than actual relations, allowing them to be filtered out by the optimization procedure described in Section 4.2.3.

Another possibility is a sequence of queries which all refer to a same ID in the selection array. Listing 4.4 shows an example of such a sequence of queries applied to the e-commerce example where information is required from both the product catalog and the product inventory.

Listing 4.4: Example of a sequence of GET-queries in the canonical representation with a possible implicit relation applied to the e-commerce example

```
{GET,[productCatalog],[name,weight,category],[(
    productID,ValueX)]}
{GET,[productInventory],[stock,supplier],[(productID
    ,ValueX)]}
```

Based on these examples of implicit relations, the task of detecting implicit relations can be translated into discovering recurring sequences of queries. The problem of detecting the frequent occurrences of sequences (in this case entities) is defined as discovering all subsequences appearing frequently in a set of sequences. First proposed by Agrawal [8] and Srikant [9], it is often referred to as sequential pattern mining or frequent sequence mining. In this chapter, suffix trees are used to detect frequent entity sequences in order to optimize the data schema for queries using these entities in a polyglot persistent environment. Section 4.3 details the implementation of these suffix trees.

4.2.5 Data Schema Optimization Architecture

Figure 4.5 depicts the architecture for the proposed data schema optimization for polyglot persistent environments. The solid arrow considers a workflow for queries from users and applications, transforms them into the canonical representation, and then to the specific technology in which the data is stored. The contributions mentioned in Sections 4.2.2 - 4.2.4 are clearly marked with the numbers in black circles:

1. The canonical model is extended with a new layer that represents the different technologies in the polyglot persistent environment to map Entities onto specific storage technologies (detailed in Section 4.2.2).

2. A schema optimization module that uses the relations from the canonical model to optimize the storage environment (described in Section 4.2.3).

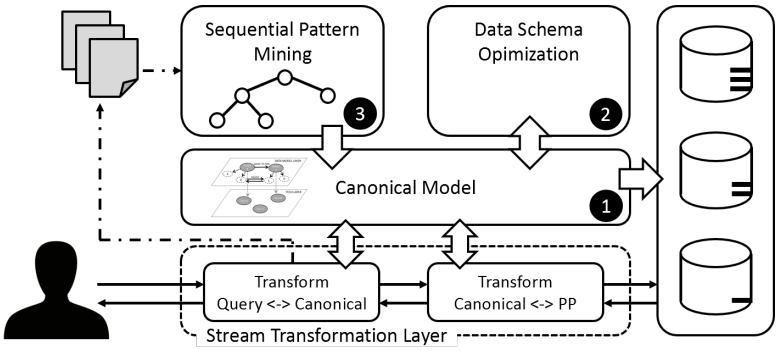

Figure 4.5: Architecture overview of the data schema optimization approach for polyglot persistent environments.

3. A sequential pattern mining component to detect potential relationships between entities in the canonical model based on incoming queries from users and applications (detailed in Section 4.2.4).

With these new components, queries from the users and applications are translated to their canonical representation in the stream transformation layer. While the queries still continue to retrieve data from the storage environment, a copy of the canonical queries is stored as well. This copy is then mined for sequential patterns of queries to deduce potential relations between entities in the canonical model. The results are stored in the canonical model and used in the schema optimization module to improve the schema by duplicating the data of the most frequent relationships. This module relays this information back to the canonical model, which in turn updates the polyglot persistent environment.

The next section focuses on the detection of relations in queries sent by applications through sequential pattern mining with suffix trees.

4.3 Abstraction Layer Implementation

4.3.1 Sequential pattern mining

A subdomain of sequential pattern mining is the domain of string mining. String mining typically works with very long sequences expressed in a limited alphabet. A well-known example of the string mining problem is the analysis of DNA sequences in bio-informatics [10]. The mining approach involves storing the sequence in an efficient data structure, which can be used for different operations, such as retrieving patterns and their occurrence frequency. One such data structure is the suffix tree [11]: a compressed tree that stores all the suffixes of a text in such a way that they can be efficiently searched.

A suffix tree is often used and most efficient in a context where the strings are built from a limited alphabet. For example, in DNA analysis patterns are searched in strings that are built out of the alphabet 'A', 'G', 'C' and 'T'. In the suffix tree used in this chapter one query, as for example in Listing 4.1, is considered to be a letter of our alphabet. Since our query is represented in a defined language in our canonical model, the alphabet is limited for a certain storage environment, but can still be significantly large. It is therefore interesting to limit the alphabet by eliminating variations that have no influence on the meaning of the query.

Listing 4.5: Example of a two sequences of GET-queries with different projections but similar indication for a relation

```
{GET,[productCatalog],[name,weight,category],
    [(productID,ValueX)]}
{GET,[productInventory],[stock,supplier],
    [(productID,ValueX)]}

{GET,[productCatalog],[name,description,category],
    [(productID,ValueY)]}
{GET,[productInventory],[warehouse],
    [(productID,ValueY)]}
```

For example, in Listing 4.5 two sequences are detailed that indicate a similar potential relationship between the *productCatalog* and the *productInventory* entities but will be detected as two different sequences as part of the suffix tree. While the projection is important to limit the results for the application, it currently serves no interest of the sequential pattern mining approach. Moreover, it negatively impacts the efforts as indicated relations can be potentially missed since the frequency of the entire relation is spread out across different sequences.

Three main algorithms have been defined in previous work for building a suffix tree: a baseline longest-to-shortest suffix approach [11], an offline algorithm detailed by McCreight [12], and a storage-optimized live algorithm identified by Ukkonen [13]. The baseline approach has a $O(n^2)$ complexity while both the McCreight's and Ukkonen's approaches are $O(n)$. The difference between Ukkonen and McCreight is that Ukkonen's algorithm is online, while McCreight requires the entire string from the beginning.

Before a decision is made on which algorithm should be used, it is important to consider the context in which it will be applied. A live application will continuously query its data sources at varying rates. It is also expected that the way an application queries its data stores could change over time because the nature of the application changes or the usage by its users evolves. Therefore, it is important that once an initial suffix tree is built for an application, it is extended, continuously or at regular intervals, with the most recent queries as to match the

application's current use of the data stores. However, if the initial suffix tree is only extended it can become a very complex data structure where new patterns might not be detected because of their low relative frequency compared to all the irrelevant historical patterns. The suffix thus needs to be both extended with the most recent queries and cleaned of the oldest queries. This should be translated to a suffix tree built on a sliding window that moves through the list of queries.

Based on this sliding window requirement, the Ukkonen algorithm is chosen for its online approach. The suffix tree built with Ukkonen's algorithm can be easily and efficiently extended. On the other hand, the indexed approach allows to easily delete branches of the tree that contain the old patterns. Section 4.4.2 details results on building the suffix tree and retrieving the query patterns as a function of the sliding window sizes.

4.3.2 Relation selection heuristic

Since solving an ILP problem can be time-consuming, an additional heuristic algorithm is useful to calculate a solution within in a reasonable time frame. Many methods exist to implement heuristics for ILP problems. A possible fast approach for the ILP defined in Section 4.2.3 is to list all detected sequences from the suffix tree in descending frequency order. Starting with the sequence that most frequently occurs, a view is created for that relation in the polyglot environment until the maximum number of explicit replication, ER_{max}, has been reached. When frequencies are widely separated, it is expected that this heuristic can reach a near-optimal solution. This can be explained because of the weight of the frequency factor in the minimization function in the ILP. However, when frequencies are close to each other, this heuristic might not at all reach a near-optimal solution since it does not take into account the access cost of a specific implementation.

Combining all the building blocks from the previous sections, the following process is built: a window of a specific size is set and filled up with canonical GET queries. Based on this window of GET queries a suffix tree is built and the recurring patterns are identified. The heuristic decides which relations are optimized in the data model and this optimization is executed. The following section evaluates the detection of relations based on a suffix tree and the execution time of the entire optimization process.

4.4 Evaluation

4.4.1 Experimental setup

The Ukkonen algorithm for building a suffix tree is implemented in Python 3.6.2. along with the methods to extend an existing tree and delete the oldest branches without impacting the characteristics of the suffix tree. All results in Section 4.4.2

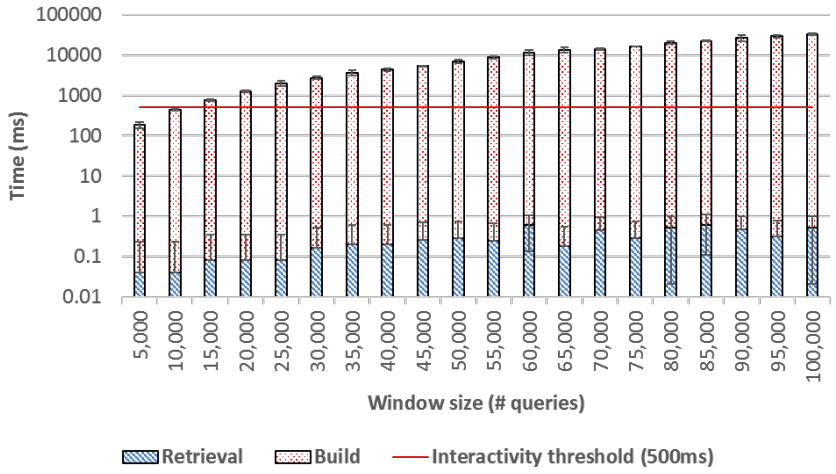

Figure 4.6: Total time to build a suffix tree and retrieve the full set of patterns as a function of the sliding window size.

are from execution on Intel i5-4200U CPU @ 1.60GHz with 8GB of RAM and show the average and standard deviation for 25 iterations. The GET-queries used in the evaluation are derived from 'select-from-(join)-where'-queries that indicate both explicit (e.g., JOIN-query) and implicit relations.

4.4.2 Results

4.4.2.1 Correctness

The implementation of Ukkonen's algorithm was validated by building a suffix tree from a randomly generated list of queries injected with specific pre-known patterns. The absolute frequencies retrieved from the suffix tree were then compared to these pre-known patterns and should to exactly represent all the possible patterns present in the list of queries. This was achieved for varying window sizes of randomly generated queries.

4.4.2.2 Window size

Figure 4.6 depicts the time it takes to build a suffix tree and retrieve the patterns from that tree for varying window sizes. Important to note is that the y-axis is in a logarithmic scale because there is a significant difference between the build and the pattern-retrieval time. For example, for a window size of $50,000$ queries it takes 6.95 seconds on average to build the suffix tree, while retrieving all the frequent sequential query patterns only takes around 0.28ms (0.00028 seconds) on

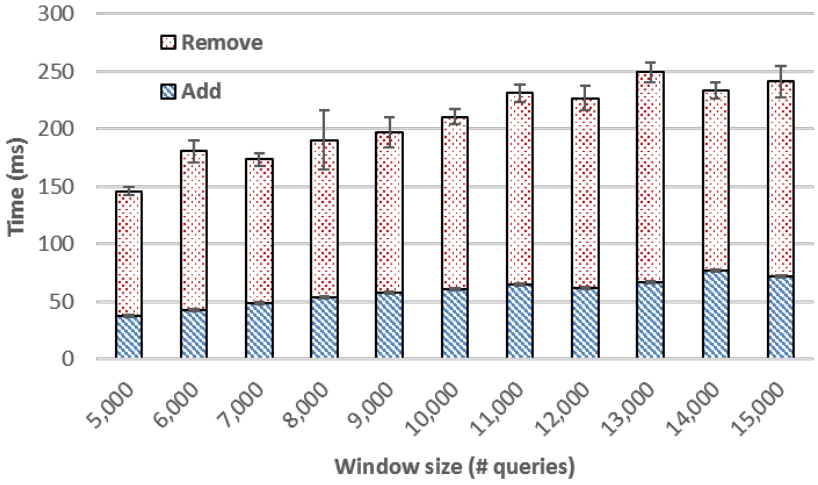

Figure 4.7: Time to remove and add queries to an existing suffix tree as a result of a window slide of 1000 queries.

average. As expected, the build time shows an overall linear trend in line with the $O(n)$ complexity of Ukkonen's algorithm [13].

The graph also shows an interactivity threshold at 500 ms [14]. The only window sizes that achieve a time below that interactivity threshold for both build and retrieval time are $5,000$ and $10,000$ with a combined time of 181.57ms and 432.91ms respectively. This means that for a window size up until $10,000$ query patterns can be retrieved from a newly-built suffix tree within a real-time setting.

4.4.2.3 Suffix tree expansion

As mentioned in Section 4.3.1 the benefit of working with Ukkonen's algorithm is its online approach and this is clearly reflected in the flexibility of the suffix tree. The algorithm can be used to extend the tree with new strings and because of the characteristics of the tree it is easy to identify and delete the oldest branches of the tree. Through these additional methods the sliding window is reflected in the suffix tree.

Figure 4.7 shows the execution time in milliseconds for both deleting old branches of and adding new queries to a suffix tree with a varying window size. The results are based on a situation where the windows slide $1,000$ queries. This means that the $1,000$-oldest branches of the suffix tree are deleted and $1,000$ new queries are added to the suffix tree. Adding the queries to the suffix tree takes less time than the removal of the oldest branches. The Ukkonen algorithm is an online algorithm that is highly optimized to add new input to the suffix tree, but it does

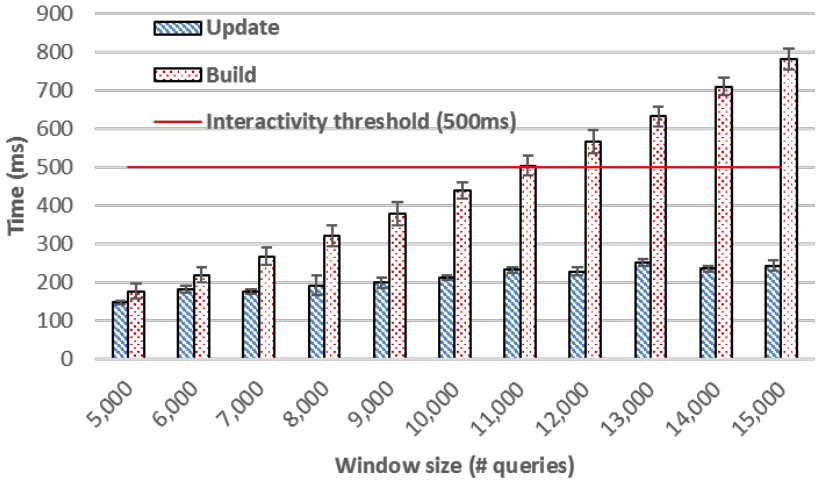

Figure 4.8: Comparison of the execution time between adjusting the suffix tree and building a suffix tree from scratch.

not provide any insight into the removal of branches in the tree, which explains the difference in speed. However, the total time for executing a sliding window is still far below the interactivity threshold of 500 ms.

Figure 4.8 compares the results of adjusting a suffix tree to building it from scratch with Ukkonen's algorithm. While for smaller window sizes the difference is minimal, for larger window sizes it is significant. Non-interactive window sizes, i.e., larger than $11,000$ in Figure 4.8, become interactive in the context of an adjusting suffix tree based on a sliding window. Larger window sizes can thus be used in an interactive setting only after building it in a non-interactive way or after gradually increasing the window size whenever a window slide is enacted until the desired size has been reached.

4.5 Discussion

Equation 4.1 defines that an abstraction layer for a polyglot persistent environment is beneficial when the overhead introduced by the abstraction layer (O_A) does not nullify the gains made by moving from a single storage solution into polyglot persistence (Δx). The gains are calculated by subtracting the average query time in polyglot persistence (x_{Ap}) from the average query time in a single storage environment (x_{As}). There are two ways of positively influencing this equation: lower the overhead of the abstraction layer ($\searrow O_A$) or increase the gains made ($\nearrow \Delta x$). Previous work by the authors has already actively reduced the overhead of the

abstraction through the stream transformation layer [3]. The work described in this chapter focuses on increasing the gains that are made by moving into polyglot persistence and, since x_{As} is considered to be a constant outside of the influence of this approach. This means the goal is to lower the average querying time in polyglot storage even further ($\searrow x_{Ap}$).

Again, going back to Table 4.1 showing the latency for a JOIN query in MySQL for $5,000,000$ records and the same query for MongoDB where the relation between the Entities used in the JOIN query is implemented with duplication in the collection. These results show that choosing an optimal technology and optimizing the schema based on relations can yield gains upwards of 120s for this specific schema. The contributions of this chapter allow to reach such improvements even when relations are not explicitly defined within a data schema.

The time in which these improvements can be implemented are clearly indicated by the results in Section 4.4.2. The entire process to update the canonical model with relations consists of first building/adjusting the suffix tree and, secondly, selecting the relations to be implemented. Building the suffix tree from scratch can be done in real-time up to a size of $10,000$ queries. If a larger window size is desired, the suffix tree can initially be built with a window size of $10,000$ and over time increased to the desired size (e.g., increase in steps of $1,000$ queries taking 67 ms each on average). When the sliding window is moved, the suffix tree can be updated efficiently in less than 250 ms for window sizes below $15,000$ queries (cfr. Figure 4.7). All the frequent query patterns can be retrieved within 1 ms (cfr. Figure 4.6). Now only a choice needs to be made as to which relations will be implemented. The current baseline heuristic defined in Section 4.3.2 sorts the patterns based on frequencies and selects the top ER_{max} queries to implement. This gives the following total time if a $5,000$ query window size is considered:

$$\begin{array}{rrr} \text{Build} + & \text{Retrieve} + & \text{Select} \\ 181.61ms + & 0.04ms + & 2.48ms \end{array}$$

The choice for a window size of $5,000$ queries for this example was made as this can be executed the fastest, but as previously mentioned the window size can be extended over time. Selecting the relations to be implemented with our baseline heuristic takes 2.48 ms as this only encompasses sorting a list and retrieving the first ER_{max} elements. Sorting a list of $5,000$ elements, which is far more than the number of relations that could be potentially detected, and retrieving any number of sequential elements is thus negligible.

Finally, important to note is that the evaluation in Section 4.4.2 only took one application or user sending queries to the data stores into account. If multiple applications would be sending queries, fewer or incorrect relations would be detected as actual string sequences would be potentially interlaced with queries from other applications or users. A simple solution would be to create suffix trees for each

distinct application and user. A more advanced solution would be to allow for a limited gap between queries and still consider them to be a sequence in the suffix tree. This solution would however increase the complexity of building the suffix tree. Further optimization for supporting multiple applications and users is part of future work.

4.6 Related Work

Polyglot persistence and its benefits and disadvantages for applications have been heavily researched [4, 15, 16]. Even before the actual term was coined, solutions were introduced to reduce the complexity of accessing heterogeneous data sources [17]. Both current and legacy solutions often introduce a single general model encompassing the different data models of the heterogeneous data stores [15]. This is an effective way to shield the applications from the complexity of a polyglot persistent environment, but they do often introduce a new type of query language to which the application needs to adapt. For example, Sellami and Defude [15] propose an approach with a virtual data store where SQL-like queries are sent in a JSON format and then translated to API's communicating with the various data store technologies. While new applications could easily benefit from this approach, legacy applications would require code changes to adopt the new query language. Currently, research on legacy applications has mostly focused on the modernization of the application itself, but not the data [18, 19]. The approach described in this chapter also introduces a canonical model, but, together with the work of the authors concerning a transformation algorithm [3], does not require any application code changes. It enables data stores of legacy applications to be migrated and transformed into a polyglot persistent environment, while the data schema is optimized based on usage.

The use of a generalized data model shields applications from the complexity of a heterogeneous data store environment, but also introduces an overhead as queries need to be translated towards the different data store technologies. It is important to note that this overhead needs to remain smaller than the actual query performance gain that was created by moving into a polyglot persistent solution. In order to achieve maximum gain, two additional optimizations can be implemented: data schema optimization and query optimization.

Data schema optimization can be traced back to E.F. Codd and his article on the relational database model [20]. He describes a normalization process to reduce data redundancy, which can be regarded as a form of schema optimization. A lot of research since then has sought to find different solutions for achieving highly optimized schemas in different contexts [6, 21]. One of these contexts is based on the actual use of the data store, i.e., optimizing the storage specifically for the application using it. Soon after the paper by E.F. Codd, Mitoma et al. proposed

an optimization based on relative frequency of relational queries [6]. Relational queries were optimized as they usually incur the highest access cost. With the introduction of NoSQL, schema optimization became much more complicated as data models became less stringent or were removed completely in these new technologies. However, similar to RDBMS, tools were introduced to automatically create highly optimized data schemas for NoSQL data stores. For example, Mior et al. [22] built a prototype of the NoSQL Schema Evaluator (NoSE), a recommendation system for NoSQL data schemas, more specifically, the Cassandra column data store. While data schema optimization has made the transition into NoSQL, it has yet to do so for polyglot persistence. The solution in this chapter applies the concept of data schema optimization onto a canonical model equipped with the concept of polyglot persistence. It acknowledges the existence of relations between data, even in NoSQL technologies, and detects them through pattern mining in (near-)sequential queries. Discovered relations are represented in the canonical model which then allows an algorithm to optimize the data schema as described above.

Compared to schema optimization, query optimization is less invasive as the internals of both the application and data store remain untouched. Query optimization can be achieved by reformulating the received queries before passing them on to the data store [15, 23]. While the chapter has less explicitly focused on query optimization in favor of schema optimization, it is encompassed within the proposed architecture. The stream transformation layer transforms queries from one technology into another. During this transformation several steps are taken to optimize the query for the specific technologies. This is not in the scope of this chapter and is deferred to future work.

4.7 Conclusions and Future Work

The new paradigm introduced by the NoSQL domain sees applications store data and information in polyglot persistence. This method where different data formats are stored in different database technologies promises a better query performance, but also introduces a higher complexity for the application. An application can be shielded from this complexity through an abstraction layer, but this impacts the performance gain as an overhead is introduced. Therefore, it becomes imperative that data schemas of the data stores are optimized for the application's use. This chapter introduces an approach to data schema optimization based on data relations. Relations are first discovered through sequential pattern mining and then an relation selection algorithm can decide which relations are implemented based on the frequency of the occurrence and replication limitations. This information is used to update the canonical model that represents the polyglot persistent environment. An implementation is provided for sequential pattern mining on incoming

queries through a suffix tree. Results show that building the suffix tree and retrieving the frequent query patterns can be done in real-time for up to $10,000$ queries. When applying a sliding window approach, updating an existing suffix tree with $1,000$ queries and deleting the $1,000$ oldest branches can be done even more efficiently. The resulting optimizations allow query latency improvements of up to two orders of magnitude for a realistic scenario.

Future work will look into advanced heuristics for the schema optimization module, query optimization during the stream transformation and applying this approach to an environment with more than one application. Finally, work also remains to aggregate the different results in an automated way.

References

[1] M. A. Mohamed, O. G. Altrafi, and M. O. Ismail. *Relational vs. nosql databases: A survey*. International Journal of Computer and Information Technology, 3(03):598–601, 2014.

[2] A. Haseeb and G. Pattun. *A review on NoSQL: Applications and challenges.* International Journal of Advanced Research in Computer Science, 8(1), 2017.

[3] T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck. *Data transformation as a means towards dynamic data storage and polyglot persistence*. International Journal of Network Management, 27(4):e1976–n/a, 2017. e1976 nem.1976. Available from: http://dx.doi.org/10.1002/nem.1976, doi:10.1002/nem.1976.

[4] K. Srivastava and N. Shekokar. *A Polyglot Persistence approach for E-Commerce business model*. In Information Science (ICIS), International Conference on, pages 7–11. IEEE, 2016.

[5] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1 edition, August 2012.

[6] M. F. Mitoma and K. B. Irani. *Automatic Data Base Schema Design and Optimization*. In Proceedings of the 1st International Conference on Very Large Data Bases, VLDB '75, pages 286–321, New York, NY, USA, 1975. ACM. Available from: http://doi.acm.org/10.1145/1282480.1282503, doi:10.1145/1282480.1282503.

[7] M. Komorowski. *A History of Storage Cost*, March 2014. http://www.mkomo.com/cost-per-gigabyte-update (Last Visited January 9, 2018).

[8] R. Agrawal and R. Srikant. *Mining sequential patterns*. In Data Engineering, 1995. Proceedings of the Eleventh International Conference on, pages 3–14. IEEE, 1995.

[9] R. Srikant and R. Agrawal. *Mining sequential patterns: Generalizations and performance improvements*. Advances in Database TechnologyEDBT'96, pages 1–17, 1996.

[10] P. P. Panigrahi and T. R. Singh. *Data Mining, Big Data, Data Analytics: Big Data Analytics in Bioinformatics*. In Library and Information Services for Bioinformatics Education and Research, pages 91–111. IGI Global, 2017.

[11] P. Weiner. *Linear pattern matching algorithms*. In Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, pages 1–11. IEEE, 1973.

[12] E. M. McCreight. *A space-economical suffix tree construction algorithm*. Journal of the ACM (JACM), 23(2):262–272, 1976.

[13] E. Ukkonen. *On-line construction of suffix trees*. Algorithmica, 14(3):249–260, 1995.

[14] Z. Liu and J. Heer. *The effects of interactive latency on exploratory visual analysis*. IEEE transactions on visualization and computer graphics, 20(12):2122–2131, 2014.

[15] R. Sellami and B. Defude. *Complex Queries Optimization And Evaluation Over Relational And NoSQL Data Stores In Cloud Environments*. IEEE Transactions on Big Data, 2017.

[16] F. R. Oliveira and L. del Val Cura. *Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications*. In Proceedings of the 20th International Database Engineering & Applications Symposium, pages 230–235. ACM, 2016.

[17] A. Rajaraman, A. Y. Levy, and J. O. Joann. *Querying heterogeneous information sources using source descriptions*. In Proceedings of the 22nd International Conference on Very Large Databases, VLDB-96, Bombay, India, 1996.

[18] T. C. Fanelli, S. C. Simons, and S. Banerjee. *A Systematic Framework for Modernizing Legacy Application Systems*. In Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, volume 1, pages 678–682. IEEE, 2016.

[19] P. V. Beserra, A. Camara, R. Ximenes, A. B. Albuquerque, and N. C. Mendonça. *Cloudstep: A step-by-step decision process to support legacy application migration to the cloud*. In Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the, pages 7–16. IEEE, 2012.

[20] E. F. Codd. *A relational model of data for large shared data banks*. Communications of the ACM, 13(6):377–387, 1970.

[21] T. A. Halpin and H. A. Proper. *Database schema transformation and optimization*. In M. P. Papazoglou, editor, OOER '95: Object-Oriented and Entity-Relationship Modeling: 14th International Conference Gold Coast,

Australia, December 13–15, 1995 Proceedings, pages 191–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. Available from: http://dx.doi.org/10.1007/BFb0020532, doi:10.1007/BFb0020532.

[22] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. *NoSE: Schema design for NoSQL applications*. In Data Engineering (ICDE), 2016 IEEE 32nd International Conference on, pages 181–192. IEEE, 2016.

[23] M. Jarke and J. Koch. *Query Optimization in Database Systems*. ACM Comput. Surv., 16(2):111–152, June 1984. Available from: http://doi.acm.org/10.1145/356924.356928, doi:10.1145/356924.356928.

5

City of Things: Smart Cities beyond Open Data

T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, P. Leroux, and F. De Turck.

Submitted to IEEE Communications Magazine, September 2017.

⋆ ⋆ ⋆

This chapter describes real-life use cases as a part of the City-of-Things project in which the work of Chapters 2, 3, and 4 is applied. City-of-Things is a smart city testbed in Antwerp (Belgium) that allows stakeholders not only access to sensor data but to an automated analytics environment as well. Sensor data from gateways spread out across the city is gathered on a central platform and can be used in user-created applications. The transformation algorithm from Chapter 3 can be used to facilitate access to this sensor data, while the schema optimization approach detailed in Chapter 4 is used for the storage environment in the back-end to optimize the data delivery to the different users.

5.1 Introduction

In recent years increasingly more connected devices have become part of the internet, birthing what is now referred to as the Internet-of-Things (IoT). In IoT live device and sensor information is used to create novel applications and services in

different domains. Well-known applications of IoT can be found in home automation, eCare, and Industry 4.0.

The latest developments in IoT have seen applications on a much larger scale in so-called smart cities. Smart cities strive to improve the quality of urban life through use of new technologies and innovative IT solutions [1]. While the term has been in use since the early 90s, the actual creation of smart cities has only boomed more recently because of the meteoric rise of IoT devices. Although it is arguable that true smart cities already exist, many cities around the world have taken important steps towards such a smart city environment by deploying devices and sensors all over the urban area. These cities are used as testbeds to facilitate the development of applications and services towards a truly smart city. An example of such a smart city testbed can be found in the Spanish city of Santander [2]. It currently houses over 3.000 IoT devices in the city with future plans extending the number of devices and sensors to 12.000.

With that amount of devices deployed in only a single city, it is clear that large amounts of data from these sensors become available for processing. These large amounts of data can be defined as *big data*, meaning that they have become so large that they are no longer able to be processed through conventional means. In order to be defined as big data, a data set needs to be large (volume), it needs to be growing at a fast rate (velocity), and different types of data, potentially from different sources, need to be part of a data set (variety). Therefore, data generated by smart cities is definitely big data as a lot of data is generated by the sensors, it constantly continues growing, and it is generated by a variety of sensors. Moreover, this data comes on top of the more traditional data that is kept on a city level such as event data, road constructions, and emergency plans.

A lot of the data provided by the smart city testbeds is part of an open data program, i.e., data is openly accessible for everyone. This follows a trend where governments have made many data sets publicly available through *e-gov portals* [1]. These include, but are not limited to, data on population growth, location of educational institutions, and statistics about the justice system. The reasoning behind opening up access to this data is to facilitate the creation of new applications and services by not only large companies, but SMEs and even citizens as well. However, as established in the previous paragraph, the data in smart cities and smart city testbeds can be considered as big data, which requires significant infrastructure and resources to process. Aside from wealthy institutions and companies, one does not generally have access to these kind of resources. This means the open data provided by the smart city testbed could be left untapped because of technical barriers [3]. An example of the impact of this complexity can be found in companies such as VITO [2], a European independent research and technology

[1] http://data.gov.be/en
[2] https://vito.be/en

organisation tackling large societal challenges in sustainable chemistry, energy, health, materials management and land use. Their solutions in land use constantly receive satellite images which are then used for analysis purposes. In order to grant their customers access to their large data set they have built virtual instances that have been pre-installed and pre-configured with all their available tools because otherwise it would be too complex to access the data. This approach clearly compromises on the flexibility of the users working with the tools. The proposed approach in this chapter lowers the complexity significantly without giving in on the flexibility of use.

This chapter introduces a smart city environment where not only open data is provided to stakeholders but an analytics sandbox as well. The City of Things (CoT) project is a smart city testbed [4] in Antwerp, Belgium, and a first proof-of-concept for this specific approach. Built on the Tengu platform [5], CoT eliminates the threshold for all stakeholders to access and effectively process open data provided by a smart city which in turn can lead to new applications and services for the smart city ecosystem. By eliminating technical barriers the project paves the way for open information, beyond open data.

5.2 Open Big Data

Big data and the Internet-of-Things have inherently been connected as IoT frameworks often generate a steady stream of data from various sources. This definition of IoT data sets clearly covers the commonly agreed upon definition of the 3 Vs of a big data set: the data set is a of a certain *volume*, and it grows at a significant *velocity*, while the data originates from a *variety* of sources. With all the complexity in the multi-billion dollar big data market it is no wonder that most of the issues in building IoT frameworks are similar to those of a general big data framework [3, 6]. We list the most important issues below.

1. **Access to infrastructure** For big data sets the traditional processing and storage solutions no longer suffice, but require parallel software running in clusters of tens, hundreds or even thousands of servers instead [7]. While large companies and institutions have the funds required to host such an infrastructure, this is not the case for individuals and/or smaller companies. Infrastructure-as-a-Service (IaaS) providers such as Google, Amazon, and Microsoft have already partly bridged this gap by providing infrastructure on a pay-per-use basis. However, as the data sets continue to grow, these costs might still prove too big for some stakeholders.

2. **Complex technology decisions** In order to support the new paradigm of cluster-based processing, new technologies were invented for the analysis

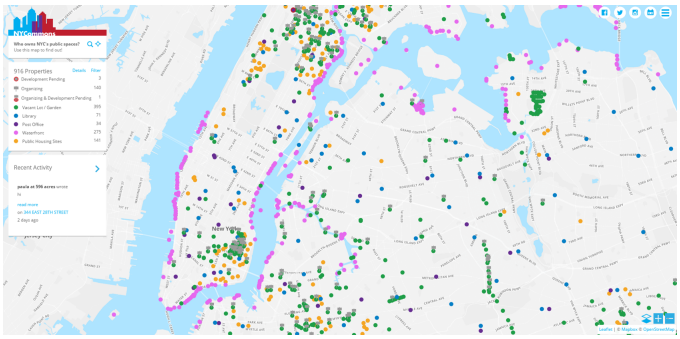

Figure 5.1: Online advocacy tool NYCommons showing an interactive map built by 596 acres.

and storage of these sizes of data sets. Analysis frameworks have been divided in two categories: batch analysis where a dataset is processed as a whole and streaming analysis where data is analyzed on message-by-message basis in real-time. Well known open-source analysis frameworks are Apache Hadoop, based on the MapReduce computational framework originally developed by Google [8], Apache Spark, Apache Storm, and Apache Nifi. Where analysis frameworks have been divided in two categories, the storage solutions are dispersed over the *Not-only SQL* (NoSQL) domain. NoSQL data stores offer a flexible or entirely schema-less data model, contrary to classical relational database management systems. Some important technologies in the NoSQL domain are MongoDB, Cassandra, Elastic Search, and Accumulo. Because of this plethora of technologies, issues arise where stakeholders need to make choices on which technology best fits their solution. Without up-to-date know-how or experience this becomes a complicated decision.

3. **Manual big data operations** Finally, when computational and storage infrastructure is available and the technologies are chosen, there is one final issue: deploying the chosen technologies, algorithms, and code on the infrastructure. In a cluster-based environment the installation and configuration of technologies is not a trivial task as it requires a thorough level of know-how and a lot of manual work.

These issues impede the creation of IoT frameworks to process the data produced by sensors. Despite the fact that in smart city environments this sensor data is made available as open data, the yield is extremely limited as the data can only used by a select group of companies or institutions that have enough knowledge, know-how and resources to overcome the above listed issues. In this chapter we tackle these issues to facilitate the creation of new applications and services on the

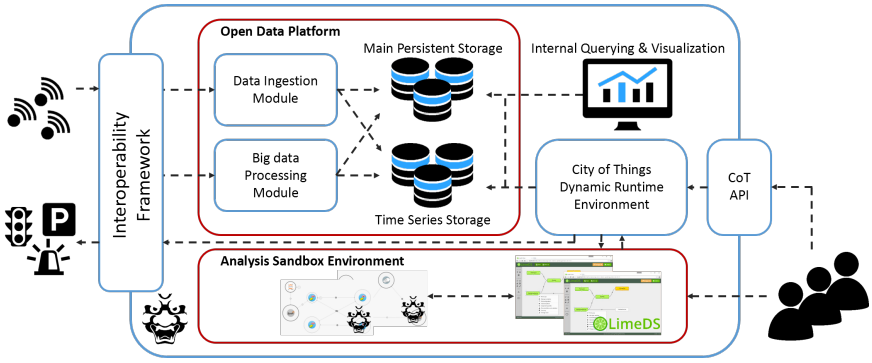

Figure 5.2: The open information architecture in the CoT project.

open data sets in a smart city environment, more specifically the CoT.

5.3 Beyond Open Data

Open data allows communities to set up great projects, such as 596 acres [3]. The project started when the founder discovered a spreadsheet marking all vacant land in Brooklyn, NY. Based on the data a map, as shown in Figure 5.1, was made to bring citizens in contact with these public pieces of land, eventually spawning a community advocating the need for public access to public land in New York City. While this is a great example of how open data can involve citizens in the city planning, the data that started the community was fairly manageable, i.e., a spreadsheet. In a smart city the amount of data originating from the sensors and devices deployed in the city would be less so.

CoT is such a large-scale testing ground for smart city experiments in the city of Antwerp, Belgium. Instead of carrying out experiments within research environments, the CoT smart city testbed offers researchers a city-wide sandbox to take the first important steps towards a truly smart city. It is different from other smart city testbeds as it not only provides access to open data from sensors in the urban area, but to actuators as well. For example, smart boilers that can be steered to optimize energy consumption and smooth out spikes in the energy grid. Currently over 100 sensors are deployed in the city with a thousand more planned in the near future.

The project aims for three important goals:

1. **Testing new network technologies** By rolling out the Internet-of-Things across the entire city, CoT provides an ideal and realistic testing environment for new network technologies.

[3] http://596acres.org

2. **Big data mining** The project provides an open data platform for monitoring life in Antwerp in real time. It aims to turn these data streams into valuable information to be used by new applications.

3. **Citizen engagement** CoT leverages interactive user research allowing citizens to give feedback on applications, but also be able to co-create new applications from scratch.

In order to achieve these goals it is clear that the CoT smart city testbed needs to create an environment in which researchers and users alike can really use the open data sets. We refer to this situation as access to open information. This is derived from a distinction made in big data analysis where the analysis process turns raw data into information. Only with open information can we be assured that the data will be applied in real use cases. The following section explains how the CoT architecture removes the barriers to reach this goal.

5.4 City of Things architecture

Figure 5.2 illustrates the architecture of the CoT smart city testbed. It can be divided in two main parts which will be discussed further in the following subsections: the open data platform and the analysis sandbox environment.

5.4.1 Open data platform

On the leftmost side of Figure 5.2 the interoperability framework is detailed that receives data from the different types of sensors and devices deployed in the city. All this data is forwarded to a data ingestion module or a processing module. The data ingestion module handles pre-processing of the data and then routes data to persistent storage. Specifically the data is formatted and routed to the best fitting storage technology in order to make responses to API requests as easy as possible. The processing module on the other hand provides advanced analysis modules for the incoming data after which the processed data can also be routed to persistent storage. Examples of these advanced analysis modules are trend analysis and prediction algorithms.

There are two different types of storage in the platform: main persistent storage, which contains events, meta-data, and context, and a specific storage solution for time series. Both storage types are used for two different endpoints. The first endpoint is the internal querying and visualization tool. It is used to monitor the environments and provide insights for city services on potentially restricted data. The second endpoint is the CoT dynamic runtime environment. This environment is the access point for all stakeholders. They communicate with this environment through an API that stipulates the commands to access the available open data.

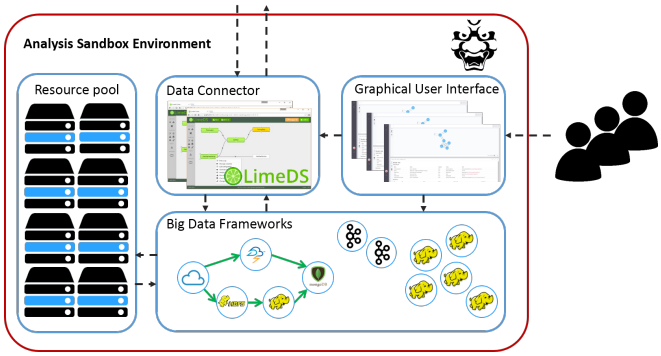

Figure 5.3: Detail of the analysis sandbox environment in the open information architecture of CoT.

The CoT dynamic runtime environment can retrieve data for stakeholders out of the open data sets, or closed sets they have access to, but also engage with the actuators in the urban environment. Additionally, the runtime environment can also be used by researchers to deploy experiments on devices in the smart city testbed in order to test new sensors or new sensor functionality.

The API that relays the commands to the digital runtime environment has four entry points: sources, types, locations, and labels. The first entry point, sources, holds all the data on the available sensors, their details and the actual events of these sensors. The types-API grants access to data on the different types of sensors and which of the sources are of which type. Based on locational data and source type the locations-API can provide the events that originated at a certain location. The final entry point, labels, is a simple grouping for all types/sources that are part of a certain experiment or use case. If it resolves in a type, the types-API can be used further on, and similarly if a source is returned, the sources-API is used. A conscious decision was also made to only allow polling on the API, contrary to the subscriber-based streaming support in the smart city testbed of Santander. The benefit of the polling-only approach is that the API becomes scalable because the unilateral urls to the different entry points allow for better caching strategies. Moreover, even though the API only allows polling, this apparent restriction will be mitigated by the analysis sandbox environment.

5.4.2 Analysis sandbox environment

Figure 5.3 shows a more detailed view of the analysis sandbox environment. This entire environment is managed by the Tengu platform. Tengu is a Platform-as-a-Service developed at Ghent University to orchestrate the setup of big data frameworks [5]. This means that Tengu automatically installs and configures custom big

data environments. Currently, Tengu manages the entire data backend of the CoT project, including the open data platform detailed in Section 5.4.1. For the sandbox environment Tengu provides the necessary flexibility to quickly set up custom big data frameworks, for example for the development of proof-of-concepts.

Stakeholders connect to the analysis sandbox environment through a graphical user interface from which they are able to build their custom big data solution. They have a wide selection of open source technologies for both analysis and storage which can be chained together. Components can also be added that retrieve data from the open data sets from the smart city environment, allowing for a smooth integration between the analysis sandbox and the open data platform.

As soon as their custom big data framework is designed, resources are allocated by the Tengu platform. These resources can originate from public cloud providers such as Google, Amazon, and Microsoft, or can be part of a private infrastructure. On these resources the technologies defined by the creating stakeholder are installed and configured correctly. The data feeding from the open data platform is realised through an advanced data connector, supported by LimeDS [9]. Starting as an OSGi abstraction layer, LimeDS grew to a visual toolset to quickly wire data-driven services together. LimeDS supports the creation of new data flows and changes to existing ones at runtime which allows Tengu to configure custom flows for the analysis sandbox. LimeDS communicates with the CoT dynamic runtime environment to retrieve data from the open data platform or to send commands to the actuators in the smart city.

The addition of this analysis sandbox environment to the smart city testbed of CoT enables it to overcome the challenges listed in Section 5.2. Infrastructure is made available by the IT department of the city, University of Antwerp and Ghent University. Tengu uses this infrastructure as a resource pool for the custom created big data frameworks which is automatically allocated when needed. As for the choice of technologies, Tengu provides predefined bundles for specific architectures that guide users in choosing relevant technologies for their custom solutions. Finally, the big data operations are completely automated by Tengu so no know-how or manual interaction are required during the setup process.

5.5 Use cases

The architecture detailled in Figure 5.2 and Figure 5.3 is able to overcome the issues detailed in Section 5.2 and achieving the goals of the CoT project. However, in order to make the contributions of the open information architecture more tangible, we present three use cases in the following subsections.

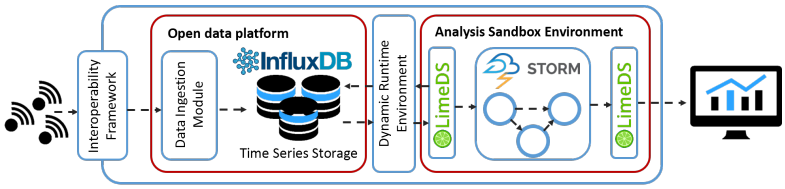

Figure 5.4: Subscription service use case for testing new sensors where data is processed by a custom Apache Storm topology and sent to an external visualization tool.

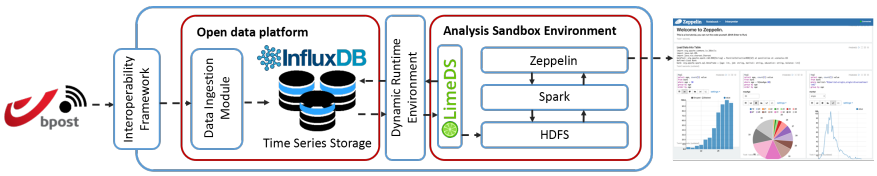

Figure 5.5: Air quality use case setup in the CoT project where data is stored in HDFS, processed by Apache Spark, and visualized by a Zeppelin dashboard.

5.5.1 REstore case

The standard API to access the open data platform follows strict unilateral URLs in order to support caching for scalability. However, this means that some applications will sometimes need to form several requests and combine the returned data events themselves in order to get the information they want. These applications can also opt to offload this complexity to the analysis sandbox. In the sandbox LimeDS can perform several requests and combine the returning data events. Additionally, it creates a new custom API that is accessible for the applications that need to retrieve these combined results. An example of a custom API built with the analytics sandbox is for REstore [4]. The custom API contains prepared data for the highly specialized REstore analytics platform towards smart power control.

5.5.2 New sensor case

As mentioned in Section 5.4.1 the analysis sandbox environment is able to mitigate the polling-only restriction on the open data platform API. This use case shows a subscription-like service set up with the streaming technologies available in the sandbox, which can be relevant for researchers and companies testing new sensors in the smart city environment. A streaming solution allows them to quickly follow up on early data events from the sensors and intervene, if necessary.

Figure 5.4 illustrates such a solution designed in the sandbox. Data events from the new sensors are sent to the interoperability framework that routes the data

[4] https://www.restore.eu

events through the data ingestion module into InfluxDB, a time series data store. Meanwhile, after the design of our streaming solution was completed, LimeDS requested the runtime environment to be kept up to date on data events of the new sensors. Therefore, as soon as data events arrive in InfluxDB, LimeDS also sends this data event to Storm, a streaming technology that performs pre-processing on the data events before sending them back to LimeDS. Finally, LimeDS connects to an external data sink and sends the data over. In this case the external data sink is a custom visualization tool from the company or researchers that are performing the sensor tests.

5.5.3 BPost case

As part of the CoT project, BPost, Belgium's leading postal operator, had air quality sensors installed on their delivery vans in the Antwerp area. At regular intervals the air quality sensors report on their current values together with their location. With these data points available in the open data platform it is interesting to create a trend analysis on the evolution of air quality in the city. Figure 5.5 details the complete setup of this specific use case. Data from the air quality sensors is sent to the CoT platform through the interoperability framework. Via the data ingestion module the measuring events are stored in InfluxDB. The process up until now is similar to the previous use case.

In order to calculate the evolution of air quality in a specific location for a certain time frame, a big data framework for batch analysis has been instantiated with Spark. On top of Spark Zeppelin has been integrated which allows to visualize the results. When executing, LimeDS will request the relevant data events in the given timeframe from the runtime environment. These data events are then transferred to the Hadoop Distributed FileSystem (HDFS) where they become available for the algorithms executed in Spark. Once Spark is finished the results are visualized in Zeppelin.

Aside from these air quality calculations, the movement of the BPost delivery vans can also be tracked in real-time based on the location data. Figure 5.6 shows an interactive map that was built for this use case. It shows real-time information on the location of two different BPost vans in Antwerp during their delivery rounds. The map also marks the data points where air quality data was measured and sent to the data backend of CoT.

5.6 Next steps

While the proposed open information architecture for the smart city testbed in CoT enables access to open data, several steps can still be taken to further lower the threshold to unlock open information.

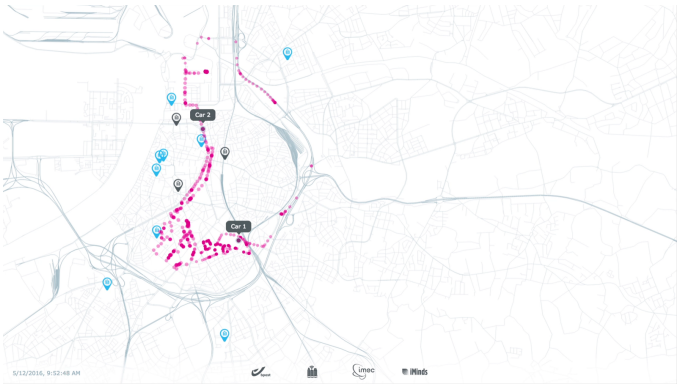

Figure 5.6: Visual representation of two BPost delivery vans moving through Antwerp, Belgium, with marked measuring points at certain intervals on their route (pink dots).

A first important aspect that has not been mentioned so far is the structure of the data accessible through the CoT platform. Currently, the data is delivered as is to the external APIs and the analysis sandbox. However, application developers might want to make use of specific data store technologies with characteristics matching the requirements of their use cases. Given the dispersion of storage technologies in the NoSQL domain, it is currently left to the responsibility of the developer to transform the data into the preferred format. Research has been performed towards the automated transformation of both schema and data between data store technologies. The algorithm proposed in [10] detects the format of the data from the original data store and transforms it in such a way that it can benefit from the characteristics of the target data store technology, allowing the data to be presented to application builders in any supported technology of their choice.

A second extension to the analysis sandbox functionality allows CoT to even offer more than just open information. Where big data analysis allows to generate information from raw data, additional tools for pattern detection and visualization can extract and present knowledge from this information. For such tools to deliver insights from large data sets in near real-time, query pre-processing and caching components are being studied to avoid large latencies in visualizing the information in a dashboard.

5.7 Conclusion

In recent years the Internet-of-Things has seen applications in different domains, among which the smart cities. The goal of IoT in smart cities is to create new applications and services based on real time data retrieved from sensors and de-

vices placed all around the urban area. While this data is often openly accessible to everyone to engage local citizens, several barriers prevent open data to be truly accessible by everyone. We identified the most important challenges: access to infrastructure, complex technology decisions and manual big data operations. The chapter presents the open information architecture of the CoT project, a smart city testbed environment in which stakeholders are able to access open data as well as a sandbox analysis environment. The sandbox is driven by Tengu, an automated orchestration service for designing big data frameworks. This sandbox environment lowers the threshold for accessing, analyzing, and processing the open data which in turn allows the involvement of citizens and SMEs in the design of new applications and services. We referred to this situation as access to open information. Several use cases were presented showing tangible examples of the applicability in the smart city environment. Finally, some next steps were identified towards a further ease-of-use of open data.

Acknowledgment

The work presented in this article was partly funded by the City of Things project (Antwerp, Belgium) by imec.

References

[1] A. Cocchia. *Smart and digital city: A systematic literature review*. In Smart city, pages 13–43. Springer, 2014.

[2] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, and E. Theodoridis. *SmartSantander: IoT experimentation over a smart city testbed*. Computer Networks, 61:217–238, 2014.

[3] A. Zuiderwijk and M. Janssen. *Barriers and development directions for the publication and usage of open data: A socio-technical view*. In Open government, pages 115–135. Springer, 2014.

[4] S. Latre, P. Leroux, T. Coenen, B. Braem, P. Ballon, and P. Demeester. *City of things: An integrated and multi-technology testbed for IoT smart city experiments*. In Smart Cities Conference (ISC2), 2016 IEEE International, pages 1–8. IEEE, 2016.

[5] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester. *Tengu: An Experimentation Platform for Big Data Applications.* In ICDCS Workshops, pages 42–47. IEEE, 2015.

[6] A. Alharthi, V. Krotov, and M. Bowman. *Addressing barriers to big data*. Business Horizons, 2017.

[7] A. Jacobs. *The Pathologies of Big Data*. Commun. ACM, 52(8):36–44, August 2009. Available from: http://doi.acm.org/10.1145/1536616.1536632, doi:10.1145/1536616.1536632.

[8] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[9] S. Verstichel, W. Kerckhove, T. Dupont, B. Volckaert, F. Ongenae, F. De Turck, and P. Demeester. *LimeDS and the TraPIST project: a case study*. In 7e International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management (IC3K 2015), volume 2, pages 501–508, 2015.

[10] T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck. *Data transformation as a means towards dynamic data storage and polyglot persistence*. International Journal of Network Management, 27(4):e1976–n/a, 2017. e1976 nem.1976. Available from: http://dx.doi.org/10.1002/nem.1976, doi:10.1002/nem.1976.

6
Conclusions and Perspectives

"Beliefs do not change facts. Facts, if one is rational, should change beliefs."

–Ricky Gervais (1984 - now)

The big data paradigm has had an enormous impact on the way applications interact with data not in the least by the emergence of NoSQL. Polyglot persistence is considered to be one of the most important consequences of the NoSQL domain where applications are no longer bound to a single data store technology, optimizing data access. However, this comes with the drawback of a higher management complexity for applications, especially for legacy applications that need to transform existing data from a single storage solution. In this dissertation 4 research questions were defined:

1. **How can the complexity of a polyglot persistent environment be mitigated for applications?**
2. **How can data be transformed between data store technologies in an automated and extensible way?**
3. **Can the impact of the transformation on the application changes be eliminated?**
4. **Can the polyglot persistent data model be optimized based on how data is retrieved by an application?**

The following contributions detailed in this dissertation provide a response to the above questions:

1. **A technology-independent framework of a true Lambda architecture with no (temporary) information loss or redundancy.**
2. **Definition and design of a canonical model that acts as a generic representation of a data schema, independent of underlying technologies.**
3. **An extensible transformation algorithm between data storage technologies supporting continuous transformation.**
4. **A framework for the detection of potential relations between data through sequential pattern mining on live queries.**

In the following sections these contributions are summarized.

6.1 Lambda Architecture

The Lambda architecture is a hybrid approach to data processing that combines batch and stream analysis frameworks, conceptualized by Nathan Marz. However, as data is processed in both the on- and offline layer, any implementation of the architecture is susceptible to potential (temporary) information loss and/or redundancy. In Chapter 2 a technology-independent framework for the Lambda architecture is presented, proven to eliminate information loss and redundancy at any given time. As soon as data enters the system, it is tagged which allows a centralized controller to track grouped data throughout the Lambda framework. An implementation is provided with Apache Hadoop and Apache Storm as the off- and online layers respectively. Results show that the implementation functions correctly without any (temporary) state of information loss or redundancy. Moreover, when redundancy or loss is manually introduced into the environment, the Lambda implementation is able to recover within a couple of batch execution runs.

This contribution of the dissertation does not directly answer any of the posed research questions but acts as a great use case of how big data and NoSQL leverage polyglot persistence for applications as results of the different layers are stored in disparate technologies. Moreover, the Lambda architecture proves to be an extremely suitable framework for the transformation described in Chapters 3 and 4 and therefore indirectly contributes to the answer of Research Question 2.

Over the past years the Lambda architecture has been under fire because of its high maintenance. Especially the necessity of maintaining two separate code bases for the on- and offline layer has provoked heavy criticism. As a reaction to this criticism the Kappa architecture was introduced where the offline batch

layer is completely removed and all data is processed as a stream. However, this does not mean that all Lambda architecture use cases can now be transferred to the Kappa architecture. While the Kappa architecture effectively simplifies code base maintenance, it loses the capability of accessing all historical data as a whole. Some processing challenges can use this access to highly optimize batch algorithm performance. Choosing between the Lambda and Kappa architecture therefore becomes a trade-off between algorithm performance and code base complexity.

6.2 Canonical model

In Chapter 3 a canonical model is introduced representing a technology-independent data schema. It is built up around Entities that represent subjects, which in turn are a combination of Attributes. Using this centralized model as a part of the transformation algorithm allows the algorithm to be easily extended with support for new data stores. Additionally, Chapter 4 extends the canonical model with a technology layer that maps Entities onto data stores. Together with the stream transformation this effectively forms an abstraction layer of the polyglot persistent environment. The canonical model itself provides a clear overview of how the data model is built up in the heterogeneous storage, while the continuous transformation allows queries to query their polyglot persistent data as if it was stored in one technology.

This dissertation also provided a proof-of-concept implementation of the canonical model as direct graph in a graph data store, Neo4j, with support for transformations from and to MySQL, Cassandra, and MongoDB. This implementation of the canonical model also stores limited additional information on attributes, such as the primary key or identifier, and even foreign keys from MySQL. In a full implementation the goal would be to store all metadata, representing functionality, from the original data store to match it as closely as possible. The graph representation

6.3 Transformation algorithm

Chapter 3 defines an extensible transformation algorithm between two data store technologies (SQL and NoSQL). The algorithm is able to derive the data schema from the original data store and through the canonical model both the schema and data are transformed into a new technology, making sure that the loss of specific characteristics from the original data store is minimized. Using the Lambda architecture from Chapter 2, the entire transformation can be executed in the background of a live application, even providing a seamless changeover and a continuous transformation in the online layer. This allows an application to still query in the original query language with a total overhead of less than 100 ms in the proof-

of-concept implementation. In other words, the transformation procedure requires no changes to the application's code and introduces only a limited impact on query latency.

In a production environment it can be argued that the amount of layers between an application and the actual data store(s) should be limited. Considering a one-to-one transformation between data store technologies, the continuous transformation only serves as an intermediate between the old and new querying language. The continuous transformation could then be regarded as a temporary state until the application is rewritten to support the new query language. This allows the application to benefit from the full expressiveness of the new query language without the intermediate layer, further improving data access latency. As for bringing a legacy application into polyglot persistence, i.e., a one-to-many transformation, the continuous transformation is more than just an intermediate between querying languages since it also acts as a shield for the complexity of the heterogeneous data stores together with the canonical model.

As mentioned in Section 6.2, this dissertation provides a proof-of-concept implementation of the transformation algorithm and support in the canonical model for MySQL, Cassandra, and MongoDB and a basic working feature set. The batch transformation allows for the creation of new Entities and the insertion and updating of data. In the continuous transformation retrieval of data was also considered with support for basic queries, such as the 'select-from-(join)-where'-queries in MySQL and equivalents in CQL and MongoDB. The goal is not to provide a full implementation of all features of many storage technologies, but rather prove that the chosen approach with the canonical model allows for extensibility through supporting new data store technologies, as well as expanding the feature sets of already supported technologies.

6.4 Data schema optimization

The final contribution of this dissertation aims to use information from the application to tailor and optimize the canonical model and underlying data schema(s) correspondingly. Specifically, this dissertation looks into potential relations between data inside a data model based on live queries by the application in an attempt to optimize or eliminate complex queries (e.g., JOIN queries). Chapter 4 defines a sequential pattern mining approach with suffix trees that is able to retrieve frequent query patterns used by the applications or users. Based on these patterns, relations between Entities can be defined in the canonical model, which in turn can be used to optimize the data schema of the polyglot storage environment.

The sequential pattern mining is applied to technology-independent GET-queries in the canonical model, allowing relations to be defined across data store technologies. The GET-queries currently support multiple entities per query (e.g., JOIN

Figure 6.1: Overview of the building blocks of the Zeta architecture as documented by Jim Scott.

queries), projection of attributes, and selection of the attributes based on equality with values or other attributes. These queries are then sequentially put into a suffix tree, built with Ukkonen's algorithm, to determine frequent patterns. The decision of which frequent patterns are implemented as relations is derived from an ILP. In the implementation a very simplistic heuristic was used which ranked frequent patterns in descending order of occurrence. The information is then used to optimize the polyglot persistent data schema which can then be used by the continuous transformation layer for the recurring complex queries.

6.5 Future Work

This dissertation has made several contributions to the domain of big data and storage in polyglot persistent environments. In the following sections further challenges are identified for the research community.

6.5.1 Zeta architecture

The Lambda architecture in Chapter 2 is only the first of several hybrid frameworks that have been released during the time of this PhD. A more recent framework is the Zeta architecture. The Zeta architecture is a new enterprise architecture that consists of 6 building blocks around a global resource management system. The building blocks are depicted in Figure 6.1. Similar challenges can be identified for the Zeta architecture concerning the synchronization of different components that function within the architecture, such as the real-time data storage, the distributed file system, and the execution engine. It would be interesting to provide a similar implementation approach for the Zeta architecture managing the synchronization of its building blocks.

6.5.2 Dynamic Data Storage

Chapter 3 shows that the correct choice of a data store technology is of prime importance for an application. However, since new technologies become available on a weekly basis, it is not inconceivable to think that one of these new technologies might be more beneficial for an existing application. Moreover, as the (non-)functional requirements of applications might change over time because the number of users changes, or new functionality is supported in the application itself, the current supported data store might no longer suffice. At such a time it would be interesting to autonomously transform the data store, or data stores, of an application in the newly most appropriate storage technology. A (self-learning) decision-making algorithm would need to be designed that can identify such change-over points in an application's lifetime. At first this could be done for a single storage environment, but this work could also be extended into polyglot persistence.

6.5.3 Data Schema Optimization

The schema optimization in Chapter 4 optimizes the storage environment based on the detected relations in the data schema, but other approaches exist to increase the read performance. Automated indexing would add indexes to entities eliminating costly full entity scans to retrieve data. Another approach can precompute the result of frequent single queries allowing results to be returned much faster. However, indexing is only useful in when queries request a small subset of an Entity and query precomputing yields the highest gains when used with aggregate functions (e.g., COUNT, SUM, and AVG). Therefore, an optimization algorithm would need to decide, based on used queries and their results, which approaches could be beneficial for the data schema.

A

Live Datastore Transformation for optimizing Big Data applications in Cloud Environments

T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck.

Published in Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM), May 2015.

⋆ ⋆ ⋆

This appendix describes research on the transformation algorithm that precedes the work detailed in Chapter 3. While the general approach in this appendix is similar to the final version, certain components, such as the canonical model, have been extended and optimized. This appendix is added to the dissertation for completeness sake as Chapter 3 refers to specific sections in this work. It contains a framework for the live transformation of the schema and data of datastores, using a previous version of the canonical data model. A transformation is given between MySQL and Cassandra as a proof-of-concept with no support for continuous transformation and limited SQL/CQL features. The correctness of the transformation is shown and performance results, in terms of transformation times and overhead, are provided.

A.1 Introduction

Vendor lock-in and interoperability issues are still considered to be top inhibitors to cloud adoption, according to a survey by North Bridge among 855 respondents [1]. The choice between cloud providers is in most cases difficult as there are several large players such as Google, Microsoft, and Amazon, but even more smaller players. Comparing these public cloud providers is a tedious task and in order to help future customers decide which cloud provider is best suited for them, tools have been created for the automated comparison of providers based on different requirements [2, 3]. For example, Ruiz-Alvarez and Humphrey have an automated approach of selecting the best storage service for a given dataset of a particular application [3]. Once such a choice is made, the migration to the cloud is a complex process. Firstly, because of the size of current data sets, traditional processing and storage solutions no longer suffice. Working with these big data sets requires parellel software running in clusters of tens, hundreds or even thousands of servers [4]. Secondly, this process usually involves changes to the application, extensive (re)configuration, and/or downtime. But as applications tend to evolve with frequent updates and feature requests on the one hand, and increasing user numbers on the other, their requirements change and scalability issues arise. This leads to a situation where the original, optimal choice of datastore is no longer optimal, i.e., the performance of the applications suffers from this choice. This might call for another migration, even potentially to another provider, again a costly operation.

In this appendix, we propose a new framework for live datastore transformation as part of a new Platform-as-a-Service Tengu, previously known as Kameleo [5]. The proposed framework aims to migrate and transform the schema and data of any datastore without any necessary changes to or downtime of the application. It introduces the concept of dynamic storage which allows the stored data to be stored in the optimal format for the application, transforming the format when necessary, i.e., when certain requirements are no longer met (e.g., query time exceeds a certain threshold). This appendix shows the extensible approach for transforming datastores live and the architecture to support it. A proof-of-concept implementation is detailed showing the transformation between MySQL, a relational database, and Cassandra, a NoSQL column-oriented datastore. The authors want to emphasize that although datastore is a term often used within the NoSQL domain, while RDBMS prefers the term database, this appendix uses datastore as a general term for both.

The remainder of this appendix is structured as follows: the architecture of the framework is described in Section A.2, while Section A.3 details the transformation principles and the corresponding workflow. The algorithm for the transformation is stipulated in Section A.4. In Section A.5, the implementation details of

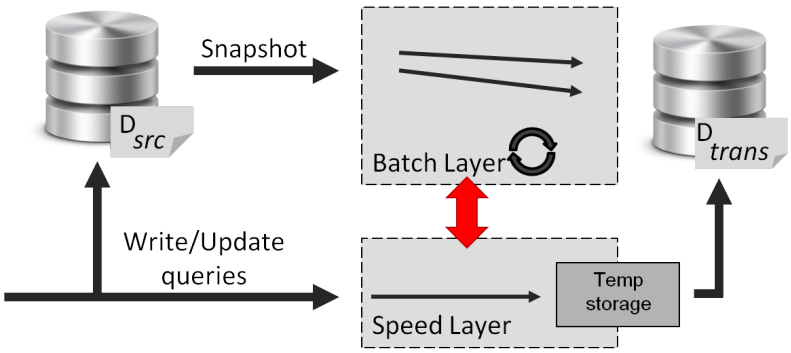

Figure A.1: General overview of the architecture with a batch layer and parallel speed layer.

the framework are provided. Section A.6 details the experimental setup, whereas results can be found in Section A.7. The discussion of the results in regard to future work can be found in Section A.7.3. Section A.8 gives an overview of related work in the field of migration and transformation of datastores. Finally, the main conclusions are presented in Section A.9.

A.2 Architecture overview

When applying a transformation on a datastore, it is important that the live application it supports, encounters no or minimal impact on it's operations. Secondly, the faster a transformation can be completed, the better, as the data is highly susceptible to redundancy and loss in this state. It can be reasonably assumed queries will continue to arrive while the transformation is in progress, considering a live application. Reading information from the datastore during the transformation is straightforward as these queries can be handled by the original or source datastore (D_{src}), but queries inserting new or modifying existing data also need to be transformed, otherwise the transformed datastore (D_{trans}) will not contain the latest data and/or reflect the latest changes to its data and structure. A simple solution would be to store these queries and transform them as soon as the first transformation is finished. However, during this second transformation new queries would possibly still arrive as well, yielding an almost infinite loop. Introducing a real time transformation for these queries, parallel to the batch transformation, solves this issue.

For the sake of completeness, we mention that the Tengu platform already provides a batch and speed layer architecture as a service, the Lambda architecture [5, 6]. It is a specific approach for Big Data analysis leveraging the computing power of batch processing with the responsiveness of a real-time computation system.

Figure A.1 shows a general overview of the proposed architecture. The batch layer uses a snapshot to transform the structure and data present in D_{src} at that time, while the speed layer transforms queries that add new data or transform existing data or structure. The latter transformations are stored in sequence until the batch layer is finished, after which the queries are executed on the newly created D_{trans}. It is important to note that all queries arriving after the snapshot are still being executed on D_{src} as well, since it is still being used for reads. Once the batch layer is finished and while the stored queries from the speed layer are executing on D_{trans}, a changeover process will be started. This stops all queries from being sent to D_{src} and completes the changeover to D_{trans}.

A.3 Transformation and workflow

A.3.1 Approach

Two main approaches can be identified when looking at the actual transformation of a datastore: direct transformation and transformation through a centralized data model. The first approach is fairly straightforward as one datastore is directly mapped onto another. Unique properties of a certain datastore can be mapped onto specific traits of the other entirely. However, for each new supported data model, this approach would require a new implementation for transforming the new data model into each of the already supported models. Using a centralized data model would solve this issue by first transforming the structure and data of each datastore to the data model, after which it is transformed into the new datastore. Supporting new datastores would then only require a transformation towards and from the abstract or canonical model. While this solution does support the extensibility of additional datastores being added, it also has several drawbacks. Firstly, the solution requires an extra transformation for every conversion between datastores introducing additional overhead. Secondly, while transforming to the centralized data model, it is not possible to assume anything about the unique characteristics of D_{trans} as the destination datastore is not yet known at that point.

Within the centralized data model, two possibilities exist: an abstract and a canonical model. An abstract model can represent the most common characteristics shared by several datastores, while the canonical model aims to support every specific characteristic of each supported datastore. Although the abstract data model allows a general representation of the datastore's structure and data, not all unique characteristics of the datastores are supported and any related advantages are also lost. With this in mind, the approach with a canonical model is preferred. The complexity in developing such a solution will be mostly contained in the first stage. Once the canonical model is in place, adding support for new datastores is significantly easier. Even if this approach performs worse time-wise, compared to

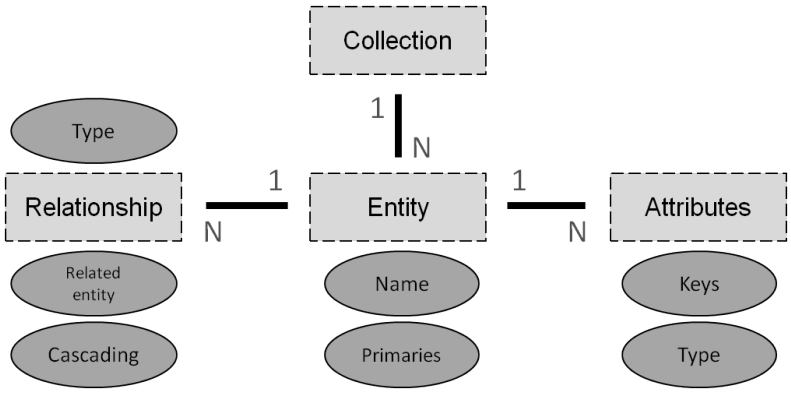

Figure A.2: Canonical model for the structure of a dataset.

a direct transformation, the architecture proposed in Section A.2 still allows for the application to operate with minimal impact. That is, during the transformation, D_{src} is still the main datastore, i.e., it still processes all the queries from the application, while the speed layer transforms any queries that update or insert data in the datastore.

Figure A.2 represents a diagram of a proposed canonical model, based on the Entity-Relationship (ER) model [7], for the structure of a dataset. The central element in this canonical model is the Entity. It represents a subject and is built up by different Attributes. It can be compared to a table in SQL, but, as demonstrated in Section A.5, not every table in SQL can be mapped onto an Entity. An Entity also keeps information about its primary keys and Attributes. Relations between Entities can also be represented with a specific type, such as many-to-one, many-to-many and one-to-one. Additional information about relationships, such as cascading, can be stored here too. Finally, a Collection combines several Entities, much like the keyspace combines column families in Cassandra. While the tuples (i.e., the actual representation of the data in the datastore) are not mentioned in Figure A.2, a tuple can be regarded as a combination of singular pieces of information, related to attributes as part of an entity (e.g., a row in a SQL table).

A.3.2 Workflow

This section summarizes the typical workflow of a transformation by the framework. The transformation process can be described in four steps:

1. **Initiate transformation:** the transformation is initiated, based on monitoring data or by request. A snapshot is taken from D_{src} and passed on to the batch layer. Until the handover, the final step, D_{src} acts as the main datastore for the application(s), i.e., all queries are still passed on to this

datastore. However, all queries that insert or update data in the datastore are also forwarded to the speed layer as soon as the snapshot is initiated.

2. **Transform structure:** before the data can be transformed, the batch layer transforms the structure or schema of D_{src}. The speed layer is only collecting queries, but not yet transforming them, as information is needed about the transformed schema of the datastore.

3. **Transform data:** based on the transformed shema of D_{src}, a new datastore, D_{trans}, is set up. Both batch and speed layer start transforming the data from the snapshot and queries respectively.

4. **Handover:** as soon as the data from the snapshot is transformed and put into D_{trans}, the handover is initiated. All queries are then redirected to D_{trans} with respect to any queries still in queue at the speed layer.

At this point, the application still queries in the language of D_{src} which leads to the following possible scenarios:

- The application maintains the original language and every query is translated by the speed layer. The application thus remains dependant on the proposed architecture with a minimal overhead introduced by the continuous transformation.

- The application was prepared for this transformation and changes its querying language to that of D_{trans}.

- The application communicates to the datastore through an abstract data layer, such as Hibernate ORM/OGM or PlayORM

It is clear that in order to eliminate the need for the application to change, the first option, continuous transformation of the queries, is required. Although we mention the different possibilities here, the further evaluation of these scenarios is outside the scope of this appendix and part of Chapter 2.

A.4 Transformation algorithm

As discussed in Section A.3, based on the canonical model approach, the transformation is divided into two parts: the first part transforms D_{src} into a canonical model and in a second phase from the canonical model into D_{trans}. To clarify the entire process, the transformation is drawn up for two specific datastores. In the context of companies migrating to the cloud and issues like vendor lock-in, an interesting use case is that of a company with a classic RDBMS wanting to migrate to a NoSQL datastore. MySQL, one of the most popular open-source RDBMS

solutions, and Cassandra, a popular NoSQL column store, are selected for the proof-of-concept as D_{src} and D_{trans}, respectively. It is important to note that all concepts used in MySQL are part of the ANSI SQL standard and can therefore be applied to any ANSI SQL standard supporting implementation.

A.4.1 SQL to canonical

The following schema shows how the different datastructures from SQL are mapped onto the canonical data model:

$$\begin{aligned} \text{Database} &\Rightarrow \text{Collection} \\ \text{Table} &\Rightarrow \text{Entity} \\ \text{Column} &\Rightarrow \text{Attribute} \\ \text{Foreign keys} &\Rightarrow \text{Relationships} \end{aligned}$$

The first three structures are trivial: a database is a collection of tables and thus entities. The tables have columns, which are represented by the attributes of entities. Relationships between tables in SQL and between entities in our canonical model are however more complicated. In SQL the relationships are defined through foreign keys, primary keys, and table use. Three types of relationships exist: one-to-one, many-to-one and many-to-many. For each type of relationship the use of foreign keys are detailed below:

- **One-to-One:** a relationship where a record of a table is connected to at most one record of the other table. In MySQL this is usually defined be two tables having the same primary key. In one of the tables, this primary key is also the foreign key referring to the primary key of the other table. For example, a table ”Customer” has a one-to-one relationship with the table ”Address”. The primary key of ”Address” is also its foreign key and refers to the primary key of ”Customer”. Both tables thus have the same primary key. Another possibility is to have a foreign key in both tables referring to the other table’s primary key. One-to-one relationships can also be used for the inheritance between tables, but this specific kind of relationship was not considered at this point.

- **Many-to-One:** a relationship where one record of a table can be connected to multiple records of another table, while the latter are only used in one relation at most. To express this relationship, a foreign key is used in the records on the ”many”-side. An example is the relationship between a table ”Order” and a table ”Customer” where one customer can have many orders, but an order is only related to one customer. In the ”Order” table therefore a foreign key is held, referring to the primary key of ”Customer”.

- **Many-to-Many:** a relationship where records from both tables can be in multiple relations between each other. In MySQL it is not possible to express this relationship with only foreign keys. A new table is therefore introduced with only two foreign keys mapping records of the two tables, sometimes identified by a single primary key if needed. This map-table has a many-to-one relationship with each of the other two tables. For example, the relation between a table ”Orders” and a table ”Products”. Orders can contain several products, but products can also be in more than one order.

When a foreign key in a table is recognized during the transformation to the canonical model, without any additional information the only correct assumption that can be made is that there exists some kind of relationship with this other table. Therefore, in a first step this general relationship will be translated into the canonical model and, after all tables have been mapped onto entities, the specific types of relationships can be defined in the canonical model as follows:

- **A one-to-one relationship** is thus detected if the local attribute in the entity is also the primary key of that entity or if the related entity also has a relationship referencing the entity.

- **The many-to-many relationship** is more complicated because of an additional table (i.e., entity) introduced by MySQL. This map entity is only used as an aid to represent the many-to-many relationship between two ”true” entities and therefore it will be referred to as a ”false” entity. As mentioned before, it is reasonable to assume this false entity only has two attributes, representing the many-to-one relationships with the two true entities, with the exception of a possible extra attribute serving as an id. The false entity is now marked for deletion, but not effectively removed as data stored in the original table still needs to be transformed afterwards. Finally, a many-to-many relationship is added to both true entities.

- **Many-to-one relationships** are the relationships that remain and do not satisfy the previous conditions.

The entire datastore schema has now been transformed into the proposed canonical model.

A.4.2 Canonical to Cassandra

The following schema shows how the canonical data structures are mapped on Cassandra:

$$\begin{aligned} \text{Collection} &\Rightarrow \text{Keyspace} \\ \text{Entity} &\Rightarrow \text{Column family} \\ &\Rightarrow \text{Composite column} \\ &\Rightarrow \text{Super column} \\ \text{Attribute} &\Rightarrow \text{Column} \\ \text{Relationships} &\Rightarrow ... \end{aligned}$$

The analogy between collection and keyspace on the one hand, and attribute and column on the other, is straightforward. For the entities and their relationships this is less so. An entity can be any of the following data structures: a column family, a composite column and a super column. Although super columns are no longer favoured as a result of their bad performance, they are only mentioned here for completeness. While relationships are not enforced in Cassandra, they can be represented when present in the canonical model in order to :

- All (true) entities are temporarily considered to be column families
- **One-to-one relationships** are eliminated through inclusion of one entity in the other as a composite column. Which entity is included in the other is decided as follows:
  1. If one of the entities still has other relationships, or more relationships compared to the other, this entity includes the other.
  2. If both entities have no or the same amount of relationships, the entity with most attributes includes the one with less attributes.
  3. If both entities have the same amount of attributes, one of the entities is randomly chosen to be included in the other.
- **Many-to-one relationships** are represented by adding an additional column family to the keyspace. This so called ”index” column family maps the ”one”-side column family on the ”many”-side column family. For example, the column family ”Order” has a many-to-one relationship with the column family ”Customer”. It is easy to determine the customer related to a certain order as this is saved in the column family. A harder query would be to get all the orders for a certain customer. As Cassandra strives towards fast lookup times, joining column families is not an option and therefore an index column family is added allowing these kind of fast lookups. The

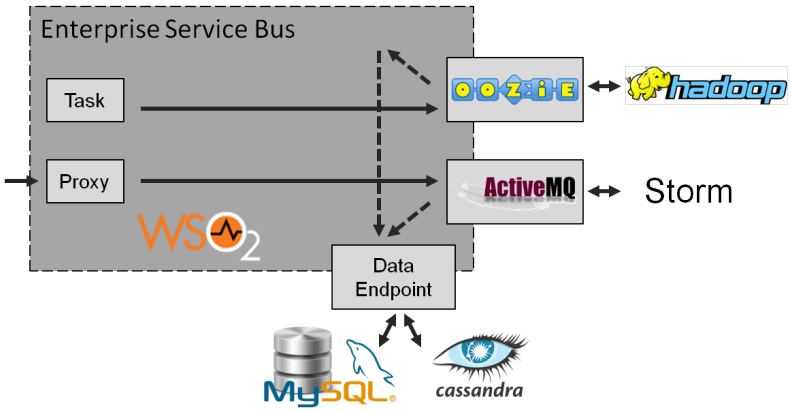

Figure A.3: Instantiation of the framework with all the implemented technologies.

first consequence of this approach is that more writes are needed to add a record to the ”many” side column family (e.g., ”Order”), but Cassandra is optimized to handle these concurrent writes [8]. Secondly, the datastore is denormalized and data is redundantly stored, which is important to remember when querying or updating the store.

- **Many-to-many relationships** can be similarly addressed as the many-to-one relationships, but from both sides. Therefore two index column families are created to again ensure a fast lookup time. For example, the column families ”Order” and ”Products” with a many-to-many relationship. It is important to have easy access to all the products that are related to an order, but also to all the orders where a specific product is included.

The entire datastore schema has been transformed into a Cassandra datastore and now the data can be transformed from D_{src} to D_{trans} based on the created canonical model. This is done in a similar fashion by mapping the data from MySQL onto the canonical model and then transforming it from the canonical model to Cassandra. The speed layer also performs this transformation in parallel to the batch layer for all the INSERT or UPDATE queries that have been received during the transformation of the structure.

A.5 Implementation details

A.5.1 Technology choice and motivation

The proposed architecture in Section A.2 requires an intelligent controller-like component responsible for the communication between the batch/speed layer and

handling input/output for those respective layers. Several possibilities were considered, such as a Message Broker (MB), an Enterprise Service Bus (ESB) and a Complex Event Processor (CEP).

A CEP takes actions based on certain events that occur in the system, while a MB allows for the asynchronous communication between applications. The ESB also allows for the communication between applications, but takes a routing approach based on a bus architecture. The CEP may not be as suitable for the proposed architecture as we would have limited control over the messages sent and received. While a MB may suffice for the simple exchange of information between several applications, an ESB allows for more control on the routing, mediation and transformation of the processed messages.

Based on these considerations, the ESB was chosen as central component. The most-used open source ESBs are UltraESB, WSO2 ESB, Mule ESB and Talend ESB. All have an active community with sufficient documentation provided for new users. Performance testing of these open source ESBs shows that on average both the WSO2 ESB and UltraESB have the best performance compared to Mule and Talend [9–11]. However, the UltraESB is less mature than the WSO2 ESB, having less iterations, and therefore the WSO2 ESB was chosen for the implementation.

A choice also needs to be made regarding the batch layer technologies. In order to achieve such a transformation of a big data set, powerful computing frameworks are needed. One of the best-known batch frameworks is MapReduce [12], originally developed by Google, but made popular by its open-source implementation Apache Hadoop. Another increasingly popular batch framework is Spark [13]. Spark is proven to execute certain programs up to 100 times faster than Hadoop in memory or 10 times on disk. MapReduce is also not the only approach to Big Data analysis. Solutions like the HPCC Systems platform and PowerGraph leverage other programming models to achieve this. However, considering the proposed transformation, there is a distinct similarity with the MapReduce model. The transformation to the canonical model can be considered as a map task, while the conversion from the canonical model can be seen as a reduce task. Based on these findings the batch layer in this proof-of-concept was implemented in Hadoop MapReduce. A scalable workflow management system, Oozie [14], was also installed on top of Hadoop. Oozie also provides a REST API which can be used by the ESB, allowing it to send commands to the Hadoop cluster.

For the speed layer, the most notable candidates are Storm and S4 [15]. Storm, recently introduced in the Apache Incubator project, provides a continuously running topology made up of singular nodes, called bolts, thus creating custom analysis streams. S4 was released by Yahoo in 2010 and also became an Apache Incubator project in 2011. It consists of processing elements, interconnected by streams and bundled in apps. These apps are then deployed and run on nodes. This and

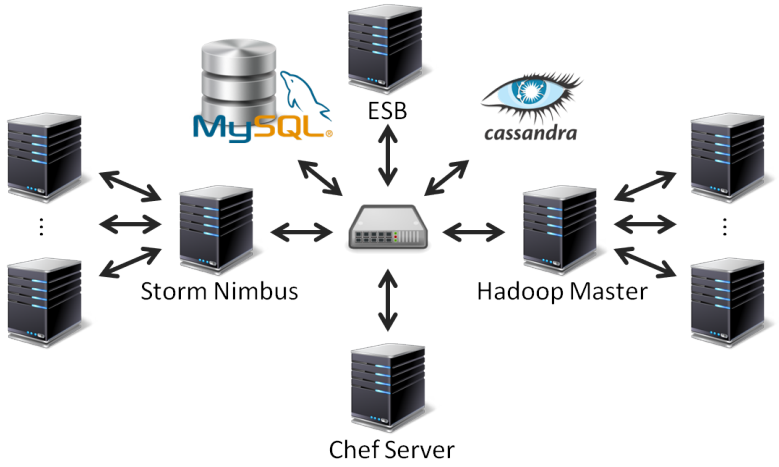

Figure A.4: Setup of the implementation on the iLab.t Virtual Wall.

a publish/subscribe system of messages makes the framework modular in such a way that apps can interconnect and be assembled in larger systems. Based on the ability of Storm to guarantee processing of the queries and the active community surrounding the project, it was chosen for the proof-of-concept implementation. Both Hadoop and Storm also use Java, which means code is reusable accross both layers. The connection between the ESB and Storm is handled by Java Message Service (JMS) and ActiveMQ [16]. An overview of all the chosen technologies can be found in Figure A.3.

A.6 Experimental setup

The implemented instantiation of the architecture was deployed on the Virtual Wall. The iLab.t Virtual Wall facility [1] is a generic test environment for advanced network, distributed software and service evaluation, and supports scalability research. The Virtual Wall contains 100 nodes with Dual CPU (Quad core) with 12GB of RAM and 1x160GB disk.

Figure A.4 details the deployment of our implementation on the Virtual Wall. The setup of every node is done through a combination of the JFed software [2] and the configuration management tool Chef [17]. Once the experiment is deployed, scripts are started on all nodes for the installation of Chef. One node is installed with a Chef server, while all other nodes are installed as a Chef client. Through cookbooks and recipes on the Chef server, the clients are then put into their roles of ESB, Hadoop master, Hadoop slaves, Storm nimbus, and Storm slaves. Oozie

[1] http://ilabt.iminds.be/
[2] http://jfed.iminds.be/

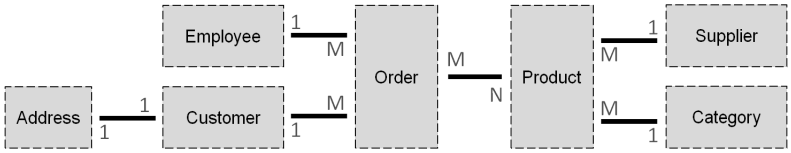

Figure A.5: Structure of the proof-of-concept datastore.

is installed on the Hadoop master, while ActiveMQ is installed on the ESB node. Although a choice was made to use Hadoop and Storm for the batch and speed layer respectively, the ESB can support different technologies and a new test environment can be deployed swiftly using JFed, and Chef and its cookbooks.

The structure of the proof-of-concept datastore needs to show the transformation can handle the different types of relationships between entities. With this in mind the structure in Figure A.5 is created to serve as a proof-of-concept datastore. It shows a datastore with information concerning a company selling products, provided by a supplier and part of a category. Orders map these products to customers and are handled by an employee. The address of a customer is saved in a separate table.

The original dataset contains 50 tuples in each of the following tables: ”Address”, ”Category”, and ”Customer”. Tables ”Employee” and ”Product” both contain 100 tuples, while tables ”Order” and ”Supplier” contain 200 and 10 tuples, respectively. The table that maps the many-to-many relationship between ”Order” and ”Product” saves 300 tuples. This original setup is then extended linearly to match datasizes of 5, 10, 15, and 20 Gigabytes (GB), where 1 GB of tuples equals 11.5 million tuples. A 20 GB file is therefore equivalent to over 230 million tuples.

A.7 Results

A.7.1 Batch layer

The batch layer is responsible for the transformation of the structure of D_{src} to D_{trans} and the data contained in the snapshot. Table A.1 details the average execution times for the transformation of the structure of D_{src} in Hadoop. The first job contains the map and reduce function responsible for the initial transformation from MySQL to the canonical model and the optimization step (cfr. Section A.5). In the second job, the reduce step transforms the canonical representation into Cassandra. Note that the execution times of map and reduce do not add up to the total of each job because of overhead introduced by Hadoop for sorting and routing the data. The execution time of both jobs is also not dependent on the size of the dataset as the structure remains the same.

Figure A.6 shows the average execution times for the transformation of the

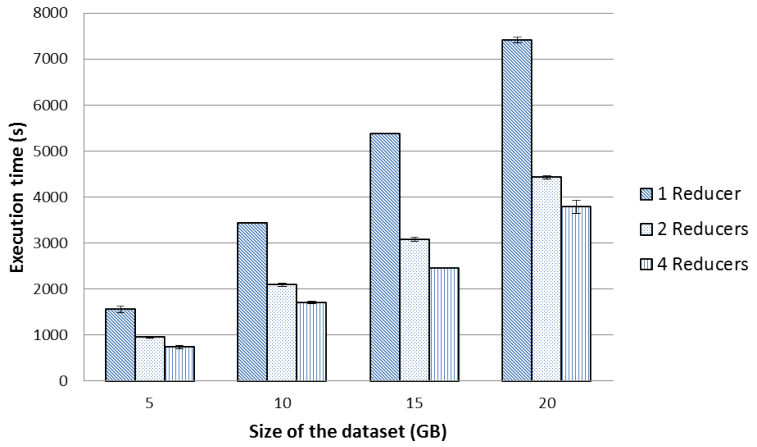

Figure A.6: Average execution times and standard deviation for the transformation of the data of the datastore in Hadoop for increasing dataset sizes.

data in Hadoop for increasing dataset sizes. As expected, increasing the dataset size also increases the time needed to transform the data. This increase follows a linear trend. In Hadoop it is also possible to configure the number of parallel reduce tasks in a job. Increasing the number of parallel reducers decreases the execution time, however doubling the number of reducers from 1 to 2 does not halve the execution time. The time gain is even less when increasing the reducers from 2 to 4. This is the result of a Hadoop overhead as it needs to route and copy the data to the correct nodes of the cluster. This becomes more complicated when more reducers are used, and thus the time gain is lowered. The number of map tasks can not be configured directly as Hadoop divides large files in smaller chunks automatically to provide them to a mapper and thus largely decides this autonomously. The number of parallel map tasks in these tests is 4.

Table A.1: Average execution times for the transformation of the structure of the datastore in Hadoop.

	Job 1			Job 2		
	Map	Reduce	Total	Map	Reduce	Total
Avg (s)	1.1818	7.0909	14.1818	1.3636	8.2727	15.2727
Std dev	0.6030	0.3015	0.7508	0.5045	0.4671	0.4671

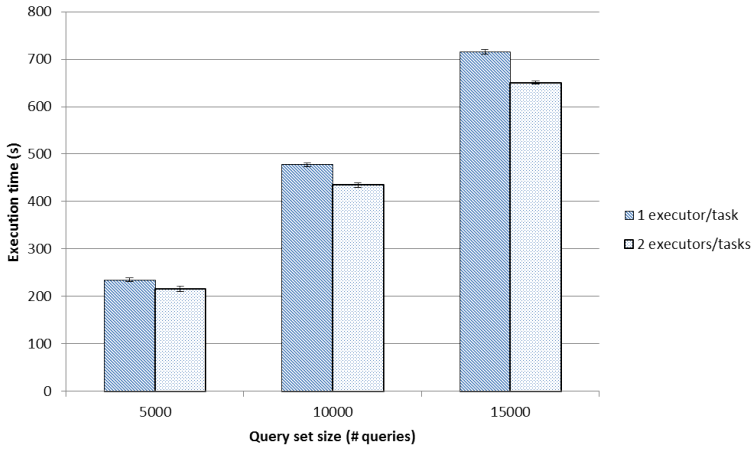

Figure A.7: Average execution times and standard deviation for the transformation of an entire query set in Storm for different query set sizes.

A.7.2 Speed layer

The speed layer, implemented in Storm, is responsible for transforming INSERT and UPDATE queries for D_{src} into queries for D_{trans}. While we mention INSERT and UPDATE queries as specific query types in ANSI SQL, it is clear that this translates to any query type that inserts new or updates data in any datastore. This process is identical to the data transformation in the batch layer, but with the map and reduce functions accommodated in bolts. Figure A.7 depicts the total time for the Storm topology to process the entire set of queries for different query set sizes. In a first stage, every bolt in the topology is limited to 1 executor/task, i.e., no parallel execution of the bolts. A linear increase of the query set yields a linear increase in execution time. This linear trend is confirmed when observing the average overhead for one query passing through the entire Storm topology in Table A.2. The average processing time of a query remains constant at around 52ms for increasing query set sizes.

In a second test the number of executors and tasks per bolt was doubled, allowing for parallelism in the bolts. The total execution time in Figure A.7 set shows a small gain for all query set sizes. As with Hadoop, Storm also accounts for some overhead for routing the queries through the topology. In Storm, it is also possible to highly tune the parallelism of every bolt in the topology independently. This is necessary because of the varying tasks each bolt has to perform. Execution times may vary between bolts and might lead to bottlenecks in the topology. Simply doubling the capacity of each bolt may therefore not halve the execution time. The results in Figure A.7 can thus be seen as an upper limit for the execution times.

Table A.2: Average processing times per query in the Storm topology.

		1 executor/task	2 executors/tasks
5000	Avg (ms)	52.170	51.817
	Std dev	0.140	0.093
10000	Avg (ms)	51.997	51.705
	Std dev	0.119	0.078
15000	Avg (ms)	51.979	51.670
	Std dev	0.137	0.059

The average processing time per query remains unchanged (cfr. Table A.2).

A.7.3 Discussion

While increasing the number of reducers in the batch layer yields a decreasing execution time until a certain point, a direct approach will almost always be faster as there is no additional transformation to and from a canonical model. However, this approach was chosen for the needed extensibility of the platform in the heterogeneous storage environment. Additionally, at this point in the transformation workflow, after the batch layer transformed the schema of D_{src}, the speed layer is running in parallel to catch up D_{trans} to the most recent state of D_{src}. The possible negative influence of this approach on the live application(s) is also limited as D_{src} is still the main datastore for all reads and writes during these transformations and the only additional stress on D_{src} will have been the moment where a snapshot of the datastore was taken.

After the transformation, and handover, the speed layer may also be responsible for the continuous transformation of queries, including search and range queries. Although this scenario is outside of the scope of this appendix, it is part of the next step towards a system where application changes are no longer required. The results, shown in Figure A.7, bode well in this regard with a limited overhead of around 52ms per query.

A.8 Related work

This section discusses related work in the field of migration and transformation of datasets.

The Extract Transform Load (ETL) principle is a process, frequently used in data warehousing, where data is extracted from an outside source, transformed according to several rules into a predefined format and loaded into an operational datastore or data warehouse. The proposed framework can be considered as a type of ETL framework: the structure and data are extracted from a source datastore,

they are transformed and loaded into a new datastore. While ETL processes focus on data alone and are often part of a long term solution [18, 19], the proposed framework transforms both structure and data and loads it into a newly created datastore instead of an already operational store. However, as the process shows a large resemblance with any data transformation, this section follows the different steps in the process, i.e., extract, transform and load.

Before any transformation can be performed, the schema and data of D_{src} needs to be extracted and migrated to the framework. The migration of big data sets has already been researched thoroughly. The obstacles posed by migration have been approached in numerous ways, such as using high-performance networks [20], having a workload-aware strategy [21], or through a cost-minimizing approach [22]. Considering the cloud context, an additional obstacle arises as many applications have to meet strict service-level agreements (SLA), therefore the downtime of the applications needs to be limited or eliminated entirely. Performing a data migration with no downtime of the application is called a *live* migration. With the growing popularity of the cloud, extensive research has been done in this sub-domain of data migration [21, 23, 24]. The Albatross technique for shared storage, developed by Das et al. [23], uses an iterative technique where a snapshot of the source datastore at the destination tries to catch up by iteratively copying the changes. Elmore et al. propose a technique for shared nothing datastores where pages of the store are pulled on-demand by the destination [24]. Both proposals do however assume several characteristics of their respective datastores. As this framework aims to support any datastore and therefore to be easily extensible, assuming anything about possible datastores would limit the flexibility of the framework significantly. While they perform no or limited changes to a datastore's structure or data, they also provide some interesting insights into the "live" aspect for the proposed framework.

A domain directly related to transformation of schemas and data, is schema matching and mapping [25]. Schema matching is the task of finding semantic correspondences between elements of two datastore schemas, while schema mapping aims to find a query or set of queries to map a source datastore into a destination datastore. Challenges in the automation of this process have been largely related to the heterogeneity, e.g., different technologies or semantics. The advent of NoSQL datastores has not eased these issues due to their own heterogeneity and flexible, or even schemaless, datamodels. Ontologies have provided a solution to the semantic heterogeneity in the form of ontology matching [26], but many other challenges exist. While this entire domain provides many solutions for transforming data, it differs from the problem this appendix solves in the sense that both the data schemas are known in advance in schema matching and mapping. In this framework the schema of D_{trans} is not known in advance, but is derived from the information in D_{src}.

A.9 Conclusion

This appendix proposes an approach for the live transformation of datastores through a canonical model. A new framework is introduced, based on the concept of the Lambda architecture with a parallel batch and speed layer. The framework is implemented as an Enterprise Service Bus with Hadoop and Storm as services for the batch and speed layer, respectively. This prototype showcases an implementation of the transformation between a MySQL database and a Cassandra NoSQL store. Results show a linear trend in execution times for increasing dataset sizes in the batch layer. Hadoop overhead also limits the time gain when increasing the number of parallel reducers. In the speed layer, Storm allows for a quick catch-up by D_{trans} after the batch layer has finished it's schema transformation and before the changeover from D_{src}. These results are also promising for a situation where a continuous transformation is necessary to avoid the application changes. While the impact of this continuous transformation needs to be evaluated further, these initial results prove that the speed layer is able to limit the introduced overhead.

Other future work will focus on the integration of this new framework into the Tengu platform for dynamic changeovers between data models. Here, monitoring data will be used to identify turning points in the performance of applications using specific datastores in order to autonomously decide when a transformation is needed.

Acknowledgment

This work has partly been funded by the IWT TWIRL project (110580) and the iMinds $(DMS)^2$ project (120442).

References

[1] M. J. Skok. *The 3rd Annual Future of Cloud Computing*. Technical report, North Bridge and GigaOM, 2013. http://www.northbridge.com/2013-future-cloud-computing-survey-reveals-business-driving-cloud-adoption-everything-service-era-it.

[2] A. Li, X. Yang, S. Kandula, and M. Zhang. *CloudCmp: comparing public cloud providers*. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pages 1–14. ACM, 2010.

[3] A. Ruiz-Alvarez and M. Humphrey. *An Automated Approach to Cloud Storage Service Selection*. In Proceedings of the 2Nd International Workshop on Scientific Cloud Computing, ScienceCloud '11, pages 39–48, New York, NY, USA, 2011. ACM. Available from: http://doi.acm.org/10.1145/1996109.1996117, doi:10.1145/1996109.1996117.

[4] A. Jacobs. *The Pathologies of Big Data*. Commun. ACM, 52(8):36–44, August 2009. Available from: http://doi.acm.org/10.1145/1536616.1536632, doi:10.1145/1536616.1536632.

[5] T. Vanhove, J. Vandensteen, G. Van Seghbroeck, T. Wauters, and F. De Turck. *Kameleo: Design of a new Platform-as-a-Service for Flexible Data Management*. In Proceedings of the 2014 IEEE/IFIP Network Operations and Management Symposium (NOMS 2014), 2014.

[6] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[7] P. P.-S. Chen. *The Entity-relationship Model - Toward a Unified View of Data*. ACM Trans. Database Syst., 1(1):9–36, March 1976. Available from: http://doi.acm.org/10.1145/320434.320440, doi:10.1145/320434.320440.

[8] P. McFadin. *The Data Model is dead, long live the Data Model*. DataStax Webinar, May 2013.

[9] D. Abeyruwan. *ESB Performance Round 6.5*. Technical report, WSO2, January 2013. http://wso2.com/library/articles/2013/01/esb-performance-65/.

[10] A. C. Perera and R. Linton. *ESB Performance Round 7*. Technical report, AdroitLogic, October 2013. http://esbperformance.org/display/comparison/ESB+Performance.

[11] S. Anfar. *ESB Performance Round 7.5*. Technical report, WSO2, February 2014. http://wso2.com/library/articles/2014/02/esb-performance-round-7.5/.

[12] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[14] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. *Oozie: towards a scalable workflow management system for hadoop*. In Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, page 4. ACM, 2012.

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. *S4: Distributed Stream Computing Platform*. In Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pages 170–177, Dec 2010. doi:10.1109/ICDMW.2010.172.

[16] B. Snyder, D. Bosnanac, and R. Davies. *ActiveMQ in action*. Manning, 2011.

[17] M. Marschall. *Chef Infrastructure Automation Cookbook*. Packt Publishing, 2013.

[18] S. Henry, S. Hoon, M. Hwang, D. Lee, and M. D. DeVore. *Engineering trade study: extract, transform, load tools for data migration*. In Systems and Information Engineering Design Symposium, 2005 IEEE, pages 1–8. IEEE, 2005.

[19] H. Agrawal, G. Chafle, S. Goyal, S. Mittal, and S. Mukherjea. *An enhanced extract-transform-load system for migrating data in Telecom billing*. In IEEE 24th International Conference on Data Engineering (ICDE 2008), pages 1277–1286. IEEE, 2008.

[20] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. M. Ruwart. *A Technique for Moving Large Data Sets over High-performance Long Distance Networks*. In Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/MSST.2011.5937236.

[21] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai. *Workload-aware Live Storage Migration for Clouds*. SIGPLAN Not., 46(7):133–144, March 2011. Available from: http://doi.acm.org/10.1145/2007477.1952700, doi:10.1145/2007477.1952700.

[22] L. Zhang, C. Wu, Z. Li, C. Guo, M. Chen, and F. Lau. *Moving Big Data to The Cloud: An Online Cost-Minimizing Approach*. Selected Areas in Communications, IEEE Journal on, 31(12):2710–2721, December 2013. doi:10.1109/JSAC.2013.131211.

[23] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. *Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration*. Proc. VLDB Endow., 4(8):494–505, May 2011. Available from: http://dl.acm.org/citation.cfm?id=2002974.2002977.

[24] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. *Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms*. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM. Available from: http://doi.acm.org/10.1145/1989323.1989356, doi:10.1145/1989323.1989356.

[25] E. Rahm and P. A. Bernstein. *A survey of approaches to automatic schema matching*. the VLDB Journal, 10(4):334–350, 2001.

[26] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

B

Tengu: an Experimentation Platform for Big data Applications

T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester.

Published in Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), June 2015.

⋆ ⋆ ⋆

The work detailed in this dissertation is evaluated in big data environments with a plethora of technologies for both data processing and storage. Setting up such environments is a manual task that requires a lot of time and knowledge on technologies to be set up. Moreover, the entire task is complicated by the clustered nature of many of the used technologies. In this appendix the Tengu platform is presented. It allows researchers to quickly set up custom big data environments without any manual interaction. This drastically reduces the experimenting time compared to a manual approach. For the work in this dissertation, the Tengu platform is used to set up the Lambda architecture implementation detailed in Chapter 2 as well as the transformation used in Chapter 3 and 4. Additionally, the Tengu platform has proven its use in other dissertations, publications, and different projects with Ghent University and industry partners.

B.1 Introduction

Big data applications require scalable and fault-tolerant frameworks to handle the high volumes of data and guarantee high processing speeds. A plethora of technologies have been invented to support efficient analysis, processing and storage of big data sets for many different scenarios [1–4]. Nevertheless, these technologies are often cluster-based in order to meet the scalability and fault-tolerance requirements of the applications. Testing these applications therefore requires large experimentation facilities. These setups are expensive in resource cost, but the clustered setup of the majority of these technologies complicates and increases the configuration time.

We propose Tengu, a new experimentation platform deployed on the Virtual Wall. The iLab.t Virtual Wall facility[1], based on the Emulab[2]/GENI platform, is a generic test environment for advanced network, distributed software and service evaluation, and supports scalability research. Tengu provides an automatic setup and deployment of big data analysis frameworks (e.g., Hadoop and Storm), SQL data bases (e.g., MySQL), data stores (e.g., Cassandra and ElasticSearch) and other cloud technologies, such as OpenStack. Furthermore, the platform offers several unique features: the Lambda architecture [5], which combines batch and stream big data analysis frameworks, and live data store transformations [6].

Many ground-breaking novel applications and services cover multiple innovation areas. Therefore, the need for these solutions to be tested on cross-domain experimentation facilities with both novel infrastructure technologies and newly emerging service platforms is rising. The Fed4FIRE project[3] aims at federating otherwise isolated experimentation facilities from the FIRE[4] community in order to foster synergies between research areas [7]. As the Virtual Wall facility is part of this federation, the Tengu platform is interconnected with other testbeds offering wired, wireless and sensor networks, SDN and OpenFlow technologies, cloud computing and smart city services, which ensures that researchers can perform experiments across the boundaries of the big data area.

Fed4FIRE offers various forms of federation. While testbeds can be merely *associated* with the federation (i.e., listed on the website with links to contact information, documentation and tutorials), the primary options are to be integrated through *advanced* or *light* federation.

- Light: access to the testbeds services is realized by exposing a Web-based API. This option does not allow full control over the individual testbed resources, but ensures unified access to experimenters.

[1] http://ilabt.iminds.be
[2] https://www.emulab.net
[3] http://www.fed4fire.eu
[4] http://www.ict-fire.eu

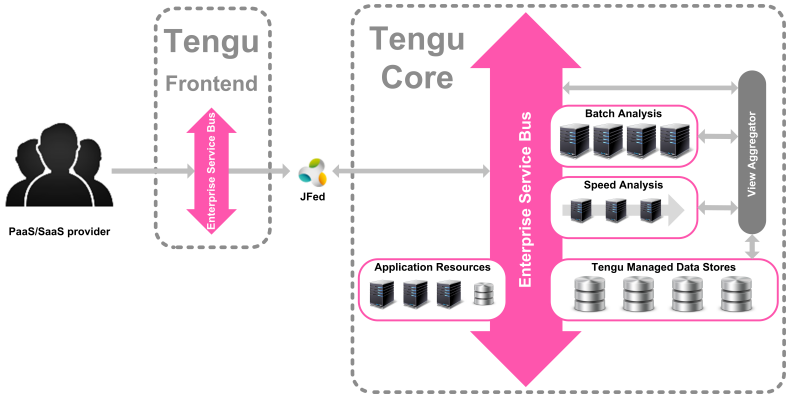

Figure B.1: General overview of the Tengu architecture

- Advanced: the testbed is fully integrated in the federation so that experimenters can interact with their experiment during all stages of the experiments life cycle (resource selection, instantiation, control, monitoring, etc.). This option requires the implementation of the Federation Aggregate Manager (AM) API on top of the testbed.

Tengu is federated through the *light* option. Its RESTful API for example allows for the instantiation of the platform through simple HTTP POST commands. Tengu in its turn relies on the same client tools offered by Fed4FIRE. The POST command to instantiate the platform creates an RSpec (a description of the requested resources), which is then deployed using the jFed client tool. As a consequence, the same tools can be used to for example visualize the setup through the jFed GUI [5] and interact with the underlying resources, if required.

The remainder of this appendix is organized as follows: Section B.2 illustrates the architecture of the Tengu platform, consisting of the core platform and its front end. The service platform, containing the RESTful API, is detailed in Section B.3. Section B.4 lays out a use case of the Tengu platform related to social network monitoring, followed by a description of the demo in Section B.5. Finally, related work is discussed in Section B.6, while Section B.7 concludes this appendix and offers several insights towards future work.

B.2 Architecture

The overall Tengu architecture has two distinct stages as illustrated in Figure B.1. The first stage consists of the front end which translates RESTful method calls

[5] http://jfed.iminds.be/

from the experimenters into an RSpec of a new Tengu core instantiation. Depending on the types of nodes defined in the RSpec, it can then be deployed by jFed on several testbeds of the Fed4FIRE federation. Once the deployment is finished, the second stage starts. The configuration management software makes sure all different components are correctly configured and interconnected, resulting in a new full-featured Tengu core setup.

In the following subsections we will dive deeper into the different stages and the involved architectural components.

B.2.1 Tengu core setup

The core setup of Tengu can again be divided into three main parts: a computational unit, data stores managed by Tengu and the application specific resource pool. An Enterprise Service Bus manages the communication between these different parts. It acts as a middleware software shielding the different components from each others specific implementation and routes data and messages between them.

B.2.1.1 Lambda architecture

The computational unit that is provided to all users of the Tengu platform is based on the concept of the Lambda architecture [5]. This concept, coined by Nathan Marz, combines two current approaches to big data analysis: batch and real-time or stream data processing. Batch data analysis frameworks analyze entire big data sets and create a view on this data set based on the implemented algorithms. However, specific types of applications, processing and analyzing data from sensor networks, social media, and network monitoring applications, generate data streams, causing results provided by a batch analysis to be always out of sync with the real-life application as new data is continuously created during the batch analysis run. This cultivated the need for (near) real-time or stream processing [3]. These stream processing technologies provide incremental updates to their results whenever new data is received, but in doing so they lack a general overview of the entire data set.

The Lambda architecture, depicted in Figure B.2, thus is a specific hybrid approach for big data analysis leveraging the computing power of batch processing in a batch layer with the responsiveness of real-time computing system in a speed layer. The batch layer provides a batch view on the entire data set, while new data is instantly analyzed by the speed layer, yielding a speed view on the most recent data. Additionally, new data is also added to the data set so it can be analyzed by the batch layer in a subsequent run. Once this run is completed, the batch view is updated while any redundant information is removed from the speed view.

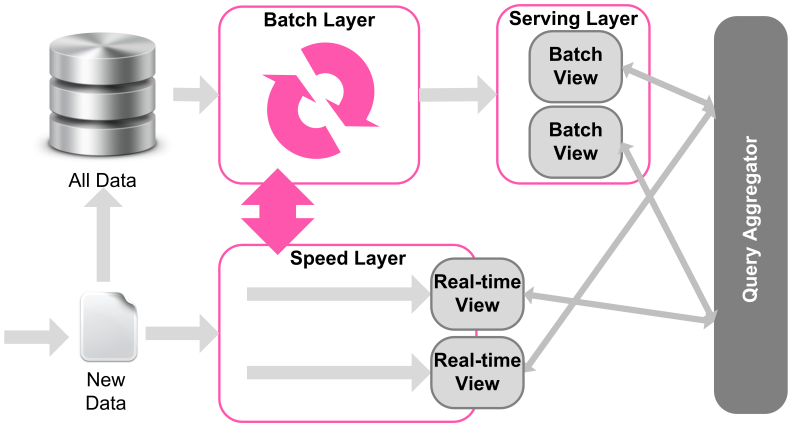

Figure B.2: Conceptual overview of the Lambda architecture

Querying the views of an applications big data set will therefore always include the aggregation of information in both the batch and speed view.

This entire computational unit is provided as a service to the applications on the Tengu platform. However, it is important to note that while the entire Lambda architecture can be provided, both batch and speed layer can also be used separate of each other. In this context the ESB manages the coordination between both layers, routing messages to the correct layer and data stores.

B.2.1.2 Tengu managed data stores

In the big data domain, technologies for storing data are designed aimed to scale horizontally, providing read/write operations distributed over many servers. This yields a new category of storage systems called NoSQL data stores [4]. As many different data stores exist today, each with their own (dis)advantages in specific scenarios, the Tengu platform offers many different data store solutions. This allows applications to use the optimal data store or even multiple data stores for their data.

Long-term experimental applications tend to evolve with frequent updates and changing user numbers, rendering the once optimal data store no longer optimal. Hence Tengu provides additional features for data stores managed by the platform, such as a live transformation between two data stores [6]. This transformation is executed using the same Lambda architecture provided to the applications. A snapshot (schema and data) of the original data store is transformed in the batch layer, while the speed layer transforms any new queries that arrive after the snapshot is taken. Once the batch layer is finished, a new data store is set up with the transformed schema and data, then updated with the queries that were transformed by

the speed layer, after which a turnover is initiated to use the new transformed data store. During this entire process, the application still queries the original data store as to eliminate any downtime.

Normally this process would also require some form of change in the application code, using the new query language. However, the ESB shields the application from its data store and through the continuous transformation of queries in the speed layer, an application can still query in the language of the original data store, even though it has been transformed [6].

B.2.1.3 Application specific resource pool

This resource pool contains several servers for the deployment of the applications on the Tengu platform. For example, while an application takes advantage of the computational unit of the Tengu platform, it can still manage its own data store outside of the Tengu environment. Additionally, applications might require specific resources such as a Tomcat server. These can also be provided in this pool. To better utilize the available infrastructure, the application specific resource pool is set up as a private cloud.

B.2.2 Tengu front end

The Tengu front end consists of a RESTful API component and jFed. The RESTful API component transforms incoming user requests for platform instantiations to an RSpec that can be deployed by jFed on testbeds in the Fed4FIRE federation. A user in this context is an experimenter who either wants to experiment with Tengu as a Platform-as-a-Service for big data applications or who wants to use Tengu to experiment with an application for big data analysis. The RESTful API has POST methods to create and deploy new Tengu core instances and GET methods to receive important information about a particular instance. It is discussed in more detail in the next section.

Deployment of the Tengu core instances is handled by jFed, as is retrieving the state information of the deployment (both general state as for example the used nodes). Using jFed has many advantages over using the underlying Fed4FIRE APIs (Aggregate Manager, User and Slice API). jFed validates the provided RSpec, not only in formatting errors, but for example also the used slice identifiers. The tool also combines many individual API calls into a single operation, e.g., requesting user information, access control checks, allocating resources and state handling. The only drawback of working with jFed, is that it does not come with an easy RESTful interface (or any other remote interface). To make the interaction with the Tengu RESTful API easier a RESTful wrapper was created around the different jFed commands.

B.3 Service Platform

B.3.1 RESTful API

RESTful API 1. *POST /tengu/core*

This API call allows to asynchronously create and deploy a new Tengu core instance. It has three mandatory query parameters: `testbed`, `snodes` and `hnodes`. The `testbed` parameter is to specify on which Fed4FIRE testbed the Tengu core instance has to be deployed. The `snodes` and `hnodes` parameters define the size of the Storm cluster and Hadoop cluster, respectively. The response of this POST includes a unique identifier, more specifically a UUID. This identifier can be used to retrieve the information about the Tengu core instance that is being deployed.

Listing B.1: The message format of a response to a POST call

```
<ten:tengu ... >
 <ten:platform>
  <ten:id>{uuid}</ten:id>
  <lnk:link method="get" href="/tengu/{uuid}" />
 </ten:platform>
</ten:tengu>
```

Listing B.1 shows response's message format. Notice the usage of XHTML links, this to easily show how an experimenter can proceed next.

RESTful API 2. *GET /tengu/{uuid}*

To retrieve information about an individual Tengu core instance, a user can perform this RESTful method call. The only thing that has to be provided, is the unique identifier (`uuid`) of the instance. The content of the response depends on the current state of the instance. At the moment we have three states: unknown (the deployment is not finished yet), ready (deployment is done and all components are correctly configured) and failed. If the Tengu core instance is fully deployed, the response also includes links to the important Tengu components (e.g., the Hadoop Job history, the HDFS namenode, the Storm web UI, the OpenStack horizon web front end, etc.)

Listing B.2: The message format of the response for a GET call

```
<ten:tengu ... >
   <ten:platform>
   <ten:id>{uuid}</ten:id>
   <ten:id>{UNKNOWN|READY|FAILED}</ten:id>
   <lnk:link method="..." rel="..." href="..." /> *
   </ten:platform>
</ten:tengu>
```

Table B.1: Special RESTful API calls to set up a specific cluster. All calls have the same set of query parameters: nodes, testbed. The calls will deploy a cluster of size {nodes} on the specified {testbed}

RESTful API calls	Description
`POST /tengu/hadoop`	deploy a Hadoop cluster
`POST /tengu/storm`	deploy a Storm cluster
`POST /tengu/openstack`	deploy an OpenStack cluster

The Tengu core setup already includes a wide variety of technologies (e.g., Hadoop, Storm and OpenStack). By using the already available RSpec generation process, deployment with jFed and configuration with Chef, it is very straightforward to also provide separate API calls to set up these individual environments. This allows the user to deploy for example only an OpenStack environment or a ready to use Hadoop cluster. The extra API calls are presented in Table B.1 together with the necessary query parameters. The response to such a call is the same as shown in Listing B.2. Information requests for these specialized environments also use the GET request (RESTful API 2), the responses will of course show a different set of links, depending on the deployed environment.

B.3.2 RSpec generation and deployment

The RESTful API is implemented as part of an Enterprise Service Bus (ESB), more particularly the WSO2 ESB [6]. The main reason for choosing this software component is not only its straightforward manner to define RESTful APIs, but its routing capabilities. It is easy to configure the ESB so it, provided certain parameters, execute a particular workflow. During this workflow it is also possible to change the incoming and outgoing messages. It is exactly this process that is used to construct the RSpec.

The first step in the RSpec creation process, starts with a templated version of the RSpec. The parameters provided with the POST RESTful call are integrated in this template. For the variable clusters (currently Hadoop, Storm, OpenStack), the template has included a placeholder. Using XSLT transformations this placeholder is changed into correct RSpec node information.

B.3.3 Deployment scripts

The RSpec refers to a script that will install a Chef server and workstation on a separate node. Chef is a configuration management software automating the build, deployment and management of the Tengu infrastructure. All technologies in the Tengu platform are defined through cookbooks and recipes, which are basically

[6] http://wso2.com/products/enterprise-service-bus

idempotent step-by-step installation scripts. Multiple executions of these scripts will therefore never change the outcome; in most cases a running framework or service.

Based on the nodes defined in the RSpec, the script deploys the cookbooks for the corresponding technologies and executes them on the correct nodes. For example, if a master node is requested in the setup, together with one or several Hadoop slave nodes (hnode), a Hadoop cluster will be deployed on these nodes. If these nodes are not present, the Hadoop cookbook will not be deployed. This modular approach to building and deploying the Tengu platform, together with Chef, allows for a flexible setup of the platform, tailored to the requirements of the experimenter.

Current available technologies include Hadoop and Storm for batch and speed layer respectively. Three data stores are already supported: MySQL, Cassandra and ElasticSearch. Other supported technologies include Tomcat, Zookeeper and Kafka. Nonetheless, the chosen approach with Chef and the ESB allows an easy integration of new technologies for the batch/speed layer and data stores.

B.3.4 Application deployment

Applications for the Tengu platform typically exist of several combinations of batch jobs and speed jobs, more specifically in the current setup of the Tengu platform, these are Hadoop MapReduce jobs and Storm topologies. For the application's UI, Tengu currently provides an Apache Tomcat Server and the application's specific meta data can be stored in a separate data store. Exactly which components are required is application specific, but to ease the setup of the actual application components, Tengu uses Chef to assist the experimenter. Consequently, the platform can be easily extended with all software components already provided through Chef's Supermarket [7].

The actual resources that are used by Chef to deploy the necessary application specific software components are part of the OpenStack private cloud set up in Tengu. This means that Chef will create a new virtual machine on the OpenStack cluster and also deploys the correct cookbooks and recipes on this virtual machine. The method of using a virtual environment for application specific resources opens up a lot of possibilities towards multi-tenancy, resource management and optimization, resource isolation, and automation.

B.4 Use case

The AMiCA (Automatic Monitoring for Cyberspace Applications) project aims to mine relevant social media (blogs, chat rooms, and social networking sites) and

[7] https://supermarket.chef.io/

collect, analyse, and integrate large amounts of information using text and image analysis. The ultimate goal is to trace harmful content, contact, or conduct in an automatic way. Essentially, a cross-media mining approach is taken that allows to detect risks ”on-the-fly”. When critical situations are detected (e.g., a very violent communication), alerts can be issued to moderators of the social networking sites.

The AMiCA project leverages the Lambda architecture using the speed layer to get near real-time feedback on developing situations on the Social Network Site (SNS), while the batch layer provides specific views on the entire history of the site. In this use case an example chat conversation between two generic users of a generic SNS is used.

The big data set of the SNS contains the entire history of the relationship of these two users, including their chat conversations. During the execution of the batch layer, the speed layer provides an analysis of the most recent incoming chat messages since the batch layer started its execution. It is clear that the speed layer does not have access to any other messages of the conversation other than the message provided at that specific moment in time. This limits the analytical power of the speed layer, but it can still provide some valuable feedback in terms of language used, picture or video/audio analysis [8, 9]. The information retrieved from this limited analysis is used in two ways. Firstly, querying the results requires an intelligent aggregation of the information in both the batch and speed view. For example, a single aggressive comment might not mean anything if the relationship has not shown any or limited signs of aggression in the past. However, when this fits into a relationship of repeated aggression, additional steps need to be taken (e.g., blocking the account of the aggressor). Secondly, in extreme cases (e.g., very violent language or graphics) the information immediately triggers an alert for the moderators of the SNS.

The other partners in this project provide components for text, image or video analysis and do not need a thorough knowledge of the setup and deployment of big data analysis frameworks. The Tengu service platform allows them to set up these frameworks for their short and long term experimenting needs.

B.5 Demo

The demo of the Tengu platform illustrates the ease of use for the setup of the platform and a comparison of the use case application running on a Tengu setup with varying cluster sizes. Figure B.3 shows the JFed GUI which allows for an easy interaction with all the resources of the Tengu platform once it has been instantiated. JFed is also responsible for the authentication and of experimenters on the different testbeds in the Fed4FIRE federation. The Tengu platform deployed in Figure B.3 utilizes resources from the Virtual Wall, depicted in Figure B.4.

In the demo Tengu setups with different cluster sizes of both batch and speed

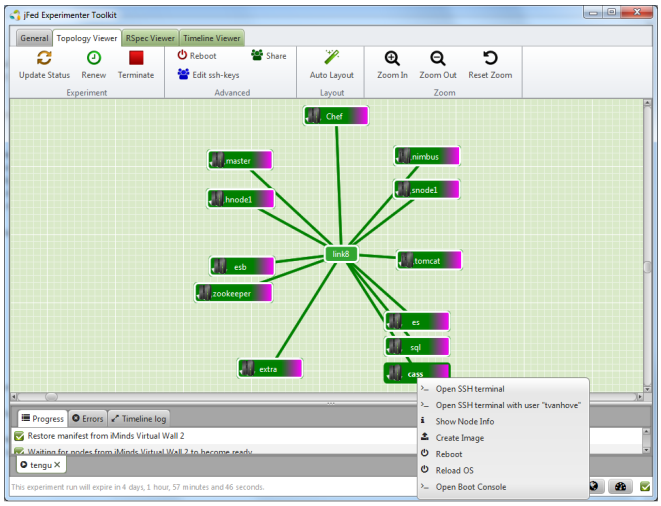

Figure B.3: Screenshot of the JFed GUI showing an instantiation of Tengu

layer are deployed on the Virtual Wall. With experiments running on these different setups, several key points of interest are highlighted, such as number of messages processed per minute in the batch and speed layer, and number of queries handled per minute. A clear seperation is made between queries on batch views and speed views. The demo also clearly shows the possibility of experiment repeatability and reproducibility even though in between tests the nodes of the Virtual Wall are released for other experiments.

The next step is to dynamically increase the cluster size of any depending technology. While some preliminary tests have already been conducted dynamically increasing the size of the Hadoop cluster, this functionality has yet to be integrated into the Tengu platform and is therefore part of future work. This requires a monitoring framework within the platform, which could also provide statistics about application performance to the experimenters. Additionally, these performance statistics could also contain valuable information regarding the views and/or data stores: if a certain view/data store is no longer performing well relative to certain performance constraints, a transformed as described in Section B.2.1.2 could be automatically initiated.

B.6 Related work

This Section discusses related work, similar setups and their main differences with the Tengu platform.

Many big data analysis frameworks are already offered as a service by the large

Figure B.4: The iLab.t Virtual Wall facility

cloud providers such as Amazon [8], Google [9] and Microsoft [10]. These frameworks are tightly integrated within their platform environment giving customers access to many other services as well for storage, networking, and elastic scaling among others. They do however often require application developers to adopt the in-house technologies, creating a vendor lock-in. Once embedded in the ecosystem, moving to another provider would require a large investment of time and resources. Tengu eliminates this vendor lock-in by using open source technologies that are already available in both research and industry. Moreover, the ESB middleware, shielding an application from its data store(s), and the live data store transformation limit application changes altogether.

HPPC Systems [10] offers a massive open source parallel computing platform that is also built around the principles of the Lambda architecture. They have a dedicated batch and serving layer, called Thor and Roxie respectively. Their speed layer is built up from several components from Thor and Roxie combined with an Apache Kafka consumer plugin. They furthermore allow for incremental updates of views through a concept of superfiles and superkeys. While HPCC Systems provides an interesting approach to big data analysis through the Lambda architecture, the Tengu platform aims to bundle existing technologies, integrated in research and industry, while every the specific combination of technologies is tailored to every application. The WSO2 ESB is the core of the Tengu platform and a necessary part of every setup, but in doing so technologies for batch/speed

[8] http://aws.amazon.com/elasticmapreduce/
[9] https://cloud.google.com/appengine/docs/python/dataprocessing/
[10] http://azure.microsoft.com/en-us/documentation/services/hdinsight/

layer, data stores or other cloud technologies are not fixed. This also preserves the platform-agnostic idea of the Lambda architecture.

With the large amount of SQL and NoSQL data stores, persistence frameworks are trying to eliminate the complexity of these different technologies by creating an abstract layer on top of the data stores. Examples like Hibernate ORM/OGM [11], PlayORM [12] and Kundera [13] have already found their way into many projects. Through a unified querying language and schema all supported data stores can be queried. The application is shielded from the complexity of the different data stores but in most cases the querying and schema language are newly introduced languages by the developers of the persistence framework. Tengu again introduces no new querying language for the communication with the data stores as many developers are already familiar with one or more data stores. The application can use the querying language and schema representation it is most familiar with, the continuous transformation in the speed layer will transform these queries into the querying language of the actual data store. This also allows for experimenters to easily try out new data stores and evaluate what this could mean for their applications.

B.7 Conclusion and Future work

This appendix presented the Tengu platform, a new experimentation platform part of the Fed4FIRE federation. It allows for the automatic setup and deployment of different big data analysis frameworks, SQL/NoSQL data stores and other cloud technologies.

As mentioned in Section B.4, the Tengu platform still requires a service that can autonomously decide to rescale the clusters of the different technologies (analysis frameworks and data stores). This requires a monitoring system tracking the performance of all the clusters. The monitoring information could also prove valuable for deciding when to transform between data stores: when the query response time in a certain data store no longer meets the requirements, a transformation to a more appropriate data store. Finally, this monitoring information could be relayed to the users directly as well, providing them with detailed information about the performance of their entire application.

While the RESTful API currently returns all important links to the different technologies, in the future it would be interesting to include an automated application deployment. Users would then be able to pass a bundle of their entire application to the service, which would then be deployed on the various parts of the Tengu core platform.

[11] http://hibernate.org/
[12] http://buffalosw.com/wiki/playorm-documentation/
[13] https://github.com/impetus-opensource/Kundera

Acknowledgment

This work was partly carried out with the support of the Fed4FIRE project (”Federation for FIRE”), an integrated project funded by the European Commission through the 7th ICT Framework Programme (318389), and the AMiCA (Automatic Monitoring for Cyberspace Applications) project, funded by IWT (Institute for the Promotion of Innovation through Science and Technology in Flanders) (120007).

References

[1] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, January 2008. Available from: http://doi.acm.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[3] J. Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 2010.

[4] R. Cattell. *Scalable SQL and NoSQL Data Stores*. SIGMOD Rec., 39(4):12–27, May 2011. Available from: http://doi.acm.org/10.1145/1978915.1978919, doi:10.1145/1978915.1978919.

[5] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[6] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester. *Tengu: An Experimentation Platform for Big Data Applications.* In ICDCS Workshops, pages 42–47. IEEE, 2015.

[7] T. Wauters, B. Vermeulen, W. Vandenberghe, P. Demeester, S. Taylor, L. Baron, M. Smirnov, Y. Al-Hazmi, A. Willner, M. Sawyer, et al. *Federation of Internet experimentation facilities: architecture and implementation*. In European Conference on Networks and Communications (EuCNC 2014), pages 1–5, 2014.

[8] B. Desmet and V. Hoste. *Recognising Suicidal Messages in Dutch Social Media*. In Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14), Reykjavik, Iceland, may 2014. European Language Resources Association (ELRA).

[9] B. Verhoeven, J. Soler Company, and W. Daelemans. *Evaluating Content-Independent Features for Personality Recognition*. In Proceedings of the 2014 ACM Multi Media on Workshop on Computational Personality Recognition, WCPR '14, pages 7–10, New York, NY, USA, 2014. ACM. Available from: http://doi.acm.org/10.1145/2659522.2659527, doi:10.1145/2659522.2659527.

[10] A. M. Middleton. *Introduction to HPCC (High-Performance Computing Cluster)*. White Paper, May 2011.