

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/361334530>

The Big Data Textbook – teaching large-scale databases in universities

Book · June 2023

CITATIONS

0

READS

19,699

1 author:



Ghislain Fourny

ETH Zurich

33 PUBLICATIONS 149 CITATIONS

SEE PROFILE

The Big Data Textbook

Teaching large-scale databases in universities

by Ghislain Fourny

January 15, 2024

The Big Data Textbook by Ghislain Fourny

Copyright ©2016-2024 Ghislain Fourny.

Some pictures used in this book are under copyright and were purchased from 123RF.com as a license.

RAM, CPU: ©radub85 / 123RF.com

Desktop computer: ©blueringmedia / 123RF.com

Dice: ©merznatalia / 123RF.com

Tree: ©xjbxjhxm / 123RF.com

Yes/no ticks: ©simo988 / 123RF.com

Rack tower: ©baibaz / 123RF.com

Data center: ©boscorelli / 123RF.com

This is a university textbook meant for teaching. Like for any other university textbooks, this means in particular that the ideas in this book are the result of the work and efforts of thousands of researchers and scientists across the world and over multiple centuries. This textbook is an attempt to reflect the body of knowledge as of 2023. It is recommended that lecturers using this textbook in their courses also list relevant scientific publications in their material, which is what we do at ETH Zurich. You may share this book with others, but it is only allowed to do so by sharing the following link (not by sharing the PDF file itself):

<https://ghislainfourny.github.io/big-data-textbook/>

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

DISCLAIMER: Although the author and publisher have made every effort to ensure that the information in this book was correct at press time, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Revision history for the First Edition (Paperback)

2023-06-07 First Release

ISBN 979-8396714199

Contents

Contents	5
1 Introduction	15
1.1 Scale	15
1.2 A short history of databases	16
1.2.1 Prehistory of databases	16
1.2.2 Modern databases	17
1.3 The three Vs of Big Data	17
1.3.1 Volume	17
1.3.2 Variety	18
1.3.3 Velocity	19
1.4 Applications	21
1.5 Scope of this book	21
1.6 Learning objectives	23
1.7 Literature and recommended readings	23
2 Lessons learned from the past	25
2.1 Data independence	25
2.2 Formal prerequisites	26
2.2.1 Sets and relations	26
2.2.2 Sets commonly used	27
2.3 Relational database management systems	28
2.3.1 Main concepts	28
2.3.2 The formalism behind the relational model	29
Relational integrity	32
Domain integrity	33
Atomic integrity	34
The relational model and beyond	35
2.3.3 Relational algebra	36
2.3.4 Selection	38
2.3.5 Projection	39
2.3.6 Grouping	39
2.3.7 Renaming	40
2.3.8 Extended projections	41

2.3.9	Cartesian products	41
2.3.10	Joins	42
2.3.11	Combining operators	43
2.4	Normal forms	43
2.5	The SQL language	46
2.6	Languages	48
2.7	Transactions	49
2.8	Scaling up and out	50
2.9	Learning objectives	51
2.10	Literature and recommended readings	51
3	Cloud storage	53
3.1	Storing data	53
3.2	The technology stack	55
3.3	Databases vs. Data lakes	57
3.4	From your laptop to a data center	57
3.4.1	Local file systems	57
3.4.2	More users, more files	58
3.4.3	Scale up vs. scale out	58
3.4.4	Data centers	59
3.5	Object stores	60
3.5.1	Amazon S3	60
3.5.2	Azure Blob Storage	60
3.6	Guarantees and service level	61
3.6.1	Service Level Agreements	61
3.6.2	The CAP theorem	62
3.7	REST APIs	63
3.8	Object stores in practice	65
3.8.1	Static website hosting	65
3.8.2	Dataset storage	65
3.9	Key-value stores	66
3.9.1	Key-value store API	66
3.9.2	Design principles	67
3.9.3	A ring to rule them all	67
3.9.4	Preference lists	72
3.9.5	An improvement: tokens and virtual nodes	73
3.9.6	Partition tolerance and vector clocks	74
3.10	Learning objectives	77
3.11	Literature and recommended readings	78
4	Distributed file systems	79
4.1	Main requirements of a distributed file system	79
4.2	The model behind HDFS	81
4.2.1	File hierarchy	81
4.2.2	Blocks	81

4.2.3	The size of the blocks	83
4.3	Physical architecture	83
4.3.1	The responsibilities of the NameNode	86
4.3.2	The responsibilities of the DataNode	87
4.3.3	File system functionality	89
4.4	Replication strategy	97
4.5	Fault tolerance and availability	99
4.6	Using HDFS	104
4.7	Paths and URIs	105
4.8	Logging and importing data	106
4.9	Learning objectives	107
4.10	Literature and recommended readings	108
5	Syntax	109
5.1	Why syntax	109
5.1.1	CSV	110
5.1.2	Data denormalization	110
5.2	Semi-structured data and well-formedness	114
5.3	JSON	116
5.3.1	Strings	116
5.3.2	Numbers	117
5.3.3	Booleans	118
5.3.4	Null	119
5.3.5	Arrays	119
5.3.6	Objects	120
5.4	XML	121
5.4.1	Elements	121
5.4.2	Attributes	123
5.4.3	Text	125
5.4.4	Comments	125
5.4.5	Text declaration	126
5.4.6	Escaping special characters	127
5.4.7	Namespaces in XML	128
Namespace URIs	128	
An entire XML document in a namespace	129	
QNames	130	
Attributes and namespaces	133	
5.4.8	Datasets in XML	134
5.5	XML vs. JSON, or how to troll internet forums	136
5.6	Learning objectives	137
5.7	Literature and recommended readings	138
6	Wide column stores	139
6.1	A sweet spot between object storage and relational database systems	139

6.2	History	140
6.3	Logical data model	141
6.3.1	Rationale	141
6.3.2	Tables and row IDs	143
6.3.3	Column families	144
6.3.4	Column qualifiers	144
6.3.5	Versioning	145
6.3.6	A multidimensional key-value store	145
6.4	Logical queries	146
6.4.1	Get	146
6.4.2	Put	146
6.4.3	Scan	147
6.4.4	Delete	148
6.5	Physical architecture	149
6.5.1	Partitioning	149
6.5.2	Network topology	149
6.5.3	Physical storage	153
6.5.4	Log-structured merge trees	159
6.6	Additional design aspects	166
6.6.1	Bootstrapping lookups	166
6.6.2	Caching	167
6.6.3	Bloom filters	168
6.6.4	Data locality and short-circuiting	168
6.7	Learning objectives	171
6.8	Literature and recommended readings	172
7	Data models and validation	173
7.1	The JSON Information Set	174
7.2	The XML Information Set	176
7.2.1	Document information item	178
7.2.2	Element information items	178
7.2.3	Attributes information items	179
7.2.4	Character information items	179
7.2.5	The entire tree	180
7.3	Validation	180
7.4	Item types	183
7.4.1	Atomic types	183
	Strings	184
	Numbers: integers	185
	Numbers: decimals	185
	Numbers: floating-point	185
	Booleans	186
	Dates and times	186
	Durations	187
	Binary data	188

	Null	189
7.4.2	Structured types	190
	Lists	190
	Records	190
	Maps	190
	Sets	190
	XML elements and attributes	190
	Type names	191
7.5	Sequence types	191
	7.5.1 Cardinality	191
	7.5.2 Collections vs. nested lists	192
7.6	JSON validation	193
	7.6.1 Validating flat objects	193
	7.6.2 Requiring the presence of a key	194
	7.6.3 Open and closed object types	195
	7.6.4 Nested structures	196
	7.6.5 Primary key constraints, allowing for null, default values	198
	7.6.6 Accepting any values	200
	7.6.7 Type unions	201
	7.6.8 Type conjunction, exclusive or, negation	201
7.7	XML validation	202
	7.7.1 Simple types	202
	7.7.2 Builtin types	203
	7.7.3 Complex types	204
	7.7.4 Attribute declarations	206
	7.7.5 Anonymous types	207
	7.7.6 Miscellaneous	207
7.8	Data frames	208
	7.8.1 Heterogeneous, nested datasets	208
	7.8.2 Dataframe visuals	212
7.9	Data formats	218
7.10	Learning objectives	220
7.11	Literature and recommended readings	221
8	Massive Parallel Processing	223
8.1	Counting cats	223
	8.1.1 The Input	223
	8.1.2 The Map	224
	8.1.3 The Shuffle	224
	8.1.4 The Reduce	225
	8.1.5 The Output	225
8.2	Patterns of large-scale query processing	226
	8.2.1 Textual input	226
	8.2.2 Other input formats	228

8.2.3	Shards	228
8.2.4	Querying pattern	229
8.3	MapReduce model	233
8.4	MapReduce architecture	239
8.5	MapReduce input and output formats	241
8.5.1	Impedance mismatch	241
8.5.2	Mapping files to pairs	241
8.6	A few examples	243
8.6.1	Counting words	243
8.6.2	Selecting	244
8.6.3	Projecting	245
8.6.4	MapReduce and the relational algebra	246
8.7	Combine functions and optimization	247
8.8	MapReduce programming API	249
8.8.1	Mapper classes	249
8.8.2	Reducer classes	250
8.8.3	Running the job	250
8.9	Using correct terminology	251
8.9.1	A warning	251
8.9.2	Functions	251
8.9.3	Tasks	252
8.9.4	Slots	254
8.9.5	Phases	255
8.10	Impedance mismatch: blocks vs. splits	257
8.11	Learning objectives	259
8.12	Literature and recommended readings	260
9	Resource management	261
9.1	Limitations of MapReduce in its first version	261
9.2	YARN	262
9.2.1	General architecture	262
9.2.2	Resource management	264
9.2.3	Job lifecycle management and fault tolerance	265
9.2.4	Scheduling	265
9.3	Scheduling strategies	266
9.3.1	FIFO scheduling	266
9.3.2	Capacity scheduling	266
9.3.3	Fair scheduling	267
9.4	Learning objectives	270
10	Generic Dataflow Management	271
10.1	A more general dataflow model	271
10.2	Resilient distributed datasets	272
10.3	The RDD lifecycle	273
10.3.1	Creation	273

10.3.2 Transformation	273
10.3.3 Action	274
10.3.4 Lazy evaluation	275
10.3.5 A simple example	275
10.4 Transformations	276
10.4.1 Unary transformations	277
10.4.2 Binary transformations	279
10.4.3 Pair transformations	281
10.5 Actions	285
10.5.1 Gathering output locally	285
10.5.2 Writing to sharded datasets	289
10.5.3 Actions on Pair RDDs	289
10.6 Physical architecture	290
10.6.1 Narrow-dependency transformations	290
10.6.2 Chains of narrow-dependency transformations	294
10.6.3 Physical parameters	296
10.6.4 Shuffling	296
10.6.5 Optimizations	298
Pinning RDDs	298
Pre-partitioning	298
10.6.6 Summary	299
10.7 DataFrames in Spark	300
10.7.1 Data independence	300
10.7.2 A specific kind of RDD	301
10.7.3 Performance impact	302
10.7.4 Input formats	302
10.7.5 DataFrame transformations	304
10.7.6 DataFrame column types	304
10.7.7 The Spark SQL dialect	306
10.8 Learning objectives	311
10.9 Literature and recommended readings	312
11 Document stores	313
11.1 Relational databases	313
11.2 Challenges	314
11.2.1 Schema on read	314
11.2.2 Making trees fit in tables	314
11.3 Document stores	318
11.4 Implementations	319
11.5 Physical storage	320
11.6 Querying paradigm (CRUD)	320
11.6.1 Populating a collection	321
11.6.2 Querying a collection	322
Scan a collection	322
Selection	322

Projection	324
Counting	324
Sorting	325
Duplicate elimination	326
11.6.3 Querying for heterogeneity	326
Absent fields	326
Filtering for values across types	326
11.6.4 Querying for nestedness	327
Values in nested objects	328
Values in nested arrays	329
11.6.5 Deleting objects from a collection	329
11.6.6 Updating objects in a collection	330
11.6.7 Complex pipelines	330
11.6.8 Limitations of a document store querying API	331
11.7 Architecture	331
11.7.1 Sharding collections	331
11.7.2 Replica sets	331
11.7.3 Write concerns	332
11.8 Indices	332
11.8.1 Motivation	332
11.8.2 Hash indices	334
11.8.3 Tree indices	336
11.8.4 Secondary indices	339
11.8.5 When are indices useful	340
11.9 Learning objectives	342
11.10 Literature and recommended readings	342
12 Querying denormalized data	343
12.1 Motivation	343
12.1.1 Where we are	343
12.1.2 Denormalized data	347
12.1.3 Features of a query language	350
Declarative	350
Functional	350
Set-based	350
12.1.4 Query languages for denormalized data	350
12.1.5 JSONiq as a data calculator	351
12.2 The JSONiq Data Model	353
12.3 Navigation	354
12.3.1 Object lookups (dot syntax)	358
12.3.2 Array unboxing (empty square bracket syntax)	359
12.3.3 Parallel navigation	359
12.3.4 Filtering with predicates (simple square bracket syntax)	360
12.3.5 Array lookup (double square bracket syntax)	361

12.3.6 A common pitfall: Array lookup vs. Sequence predicates	361
12.4 Schema discovery	361
12.4.1 Collections	362
12.4.2 Getting all top-level keys	363
12.4.3 Getting unique values associated with a key	364
12.4.4 Aggregations	364
12.5 Construction	365
12.5.1 Construction of atomic values	365
12.5.2 Construction of objects and arrays	366
12.5.3 Construction of sequences	366
12.6 Scalar expressions	367
12.6.1 Arithmetic	367
12.6.2 String manipulation	368
12.6.3 Value comparison	369
12.6.4 Logic	370
12.6.5 General comparison	370
12.7 Composability	371
12.7.1 Data flow	372
12.8 Binding variables with cascades of let clauses	373
12.9 FLWOR expressions	375
12.9.1 Simple dataset	375
12.9.2 For clauses	376
12.9.3 Let clauses	379
12.9.4 Where clauses	385
12.9.5 Order by clauses	386
12.9.6 Group by clauses	388
12.9.7 Tuple stream visualization	391
12.10 Types	392
12.10.1 Variable types	392
12.10.2 Type expressions	393
12.10.3 Types in user-defined functions	394
12.10.4 Validating against a schema	394
12.11 Architecture of a query engine	395
12.11.1 Static phase	395
12.11.2 Dynamic phase	398
Materialization	399
Streaming	400
Parallel execution (with RDDs)	400
Parallel execution (with DataFrames)	403
Parallel execution (with Native SQL)	404
12.12 Learning objectives	405
12.13 Literature and recommended readings	406
13 Graph databases	407

13.1 Why graphs	407
13.2 Kinds of graph databases	408
13.3 Graph data models	408
13.3.1 Labeled property graphs	408
13.3.2 Triple stores	414
13.4 Querying graph data	414
13.5 Learning objectives	420
13.6 Literature and recommended readings	421
14 Acknowledgements	423
15 Latest updates	425

Chapter 1

Introduction

1.1 Scale

Humankind noticed very early the Sun, the Moon and the stars in the sky. It also noticed that some celestial bodies, which it called planets, were moving on strange paths. But it is only relatively recent in our history that we understood that the planets are orbiting around the Sun, just like the Earth, and that planets have satellites too (Phobos and Deimos for Mars, etc).

The Solar system alone forces us to experience larger scales: meters for us humans, kilometers when we drive cars, Megameters when we consider the size of our Planet, Gigameters when we consider the distance to the Sun. Jupiter brings us to a Terameter, and the entire Solar system fits in a Petameter.

Even more recently, we discovered that the spiral-shaped nebulas in the sky are actually galaxies just like this large line of stars in the sky we called Milky Way. It pushed us even farther: not just Exameters, but even Zettameters just for one galaxy and about 400 Yottameters for the entirety of what we are able to see.

I am still not sure today what is the most amazing: is it that the visible universe is large beyond anything we can imagine with its 400,000,000,000,000,000,000,000 meters in size? Or is it that this printed number, on this page, is so small and that our standardized prefixes are more than enough to express this size?

Our experience with the growth of data, somehow, is different. We grew accustomed to bytes, kilobytes, Gigabytes, Terabytes in our everyday life without realizing that, with this analogy with astronomy, computers went to Jupiter in just a few decades of existence.

Astrophysicists going to such large scales, however, quickly noticed that, to understand the Universe at such large scales, they also need to understand it at much smaller scales.

Data Science is not really different from Physics, in this and many other respects. Data at the scale of Exabytes can only be understood and tamed if we also model it at the tiniest scales.

Another aspect of Data Science that makes it akin to Physics is that, unlike Mathematics and Computer Science, Data Science is about studying what the world *actually is*, as opposed as *how it could have been*, a phenomenon also called contingency in modal logic theory. Many processes in Data Science (e.g., hypothesis testing) are much closer to physics and the discovery of theories than one would think.

This book provides an introduction to the beautiful world of Big Data as I currently teach it in the lectures *Big Data* and *Big Data for Engineers* at ETH Zürich.

1.2 A short history of databases

1.2.1 Prehistory of databases

Let us now quickly sketch a short history of databases.

Data storage, in fact, predates us: all living beings have at the core of their cells DNA, which is nothing else than data storage based on bits with 4 possible values (A, T, G, C) rather than 2.

But the first data storage that was in the control of humans was simply their brains. Humans recollect events, and transfer this knowledge from generation to generation by simply speaking, telling and singing stories to the younger generations. This, of course, is prone to distortions in the information, loss of information, and the introduction of errors in the information.

The first revolution happened thousands of years ago with the invention of writing. Writing was first made on clay tablets. Once dry, information written thereon can survive for at least thousands of years. This is why we have much more knowledge about humankind after the mastery of writing: we could decode them and have access to pristine information from these older times.

But what is truly amazing about clay tablets is that they were not only used for writing texts: we found clay tablets with relational tables (!), the earliest known example being Plimpton 322, which is 3,800+ years old. This alone illustrates how natural relational tables are to human beings, and was an omen for what was to come centuries later.

The second revolution was the invention of the printing press. Before it, making duplicates of documents was very costly and, above all, required manual work, copy by copy. This also slowed down the spread of knowledge and information. With the printing press, it became straightforward to make large numbers of copies of the same text, which led in particular to the Golden age of the written press.

The third revolution was the invention of modern, silicon-based computers. Computers accelerated the processing of data.

1.2.2 Modern databases

The birth of database management systems in our modern understanding is often placed in 1970. This is the year where a seeding paper by Edgar Codd was published, introducing the concept of data independence. In the early days of computers, people were managing data on storage devices themselves, which was very demanding in resources. Edgar Codd suggested that a usable database management system should hide all the physical complexity from the user and expose instead a simple, clean model. He further suggested that this model should be based on tables, which gave birth to the relational model and the relational algebra.

More recently though, the explosion in the quantity of the data we produce brought this model to its limits a few decades later, with the emergence of modern systems such as key-value stores, wide column stores, document stores and graph databases.

1.3 The three Vs of Big Data

The recent evolution in the domain of large-scale data processing over the past two decades is often explained in terms of the three Vs: Volume, Variety, Velocity.

1.3.1 Volume

First, the volume of data stored worldwide is increasing exponentially. The total amount of data stored digitally worldwide is estimated to be getting close to 100 ZB as of 2021 (zettabytes). This is a 1 followed by 23 zeros. There are many reasons for this: we have automated the collection of data (sensors, logs, etc), and we also have all the necessary space to store it without needing to delete it. Many companies actually keep all their data just in case they might need it later, although this is likely to become less commonplace in the future with new regulations such as the European GDPR.

Prefixes have been standardized for large powers of ten, all the way to 10^{24} . They should be known by heart:

kilo (k)	1,000 (3 zeros)
Mega (M)	1,000,000 (6 zeros)
Giga (G)	1,000,000,000 (9 zeros)
Tera (T)	1,000,000,000,000 (12 zeros)
Peta (P)	1,000,000,000,000,000 (15 zeros)
Exa (E)	1,000,000,000,000,000,000 (18 zeros)
Zetta (Z)	1,000,000,000,000,000,000,000 (21 zeros)
Yotta (Y)	1,000,000,000,000,000,000,000,000 (24 zeros)
Ronna (R)	1,000,000,000,000,000,000,000,000,000 (27 zeros)
Quetta (Q)	1,000,000,000,000,000,000,000,000,000,000 (30 zeros)

They can be used in particular for bytes (B), i.e., $1,000,000 \text{ B} = 1,000 \text{ kB} = 1 \text{ MB}$, but also in conjunction with any other units in the international system.

Computer Scientists have often used prefixes to express powers of 2 rather than powers of 10, using the coincidence that 2^{10} is very close to 10^3 . For example, 1 kB actually meaning 1,024 B and not 1,000 B, although this was an abuse of notation. In order to make it official, prefixes have been defined for this purpose, even though many people nowadays continue to confuse the two series. It is, of course, useless to remember the exact values of the powers of 2, but one needs only understand that each prefix corresponds to a 2^{10n} .

kibi (ki)	1,024 (2^{10})
Mebi (Mi)	1,048,576 (2^{20})
Gibi (Gi)	1,073,741,824 (2^{30})
Tebi (Ti)	1,099,511,627,776 (2^{40})
Pebi (Pi)	1,125,899,906,842,624 (2^{50})
Exbi (Ei)	1,152,921,504,606,846,976 (2^{60})
Zebi (Zi)	1,180,591,620,717,411,303,424 (2^{70})
Yobi (Yi)	1,208,925,819,614,629,174,706,176 (2^{80})

1.3.2 Variety

Second, new shapes of data have emerged. While Edgar Codd suggested to focus on data organized in tables because it is the most intuitive for human beings, more recent systems involve data models relying on four more shapes:

- trees: trees correspond to denormalized data often found in formats such as XML and JSON, but also more recently such as Parquet, Avro, etc.
- unstructured: a lot of data is simply accumulated in a raw form such as text, pictures (pixels), audio, video, etc.
- cubes: in the 1990s, data cubes became very popular in the field of business analytics, with the primary use case of analyzing sales

data and producing reports for the top management and strategic decision making.

- graphs: there are database systems (such as neo4j, Oracle PGX, etc) that expose data as a graph with nodes and vertices. Graph shapes are especially useful when the use case is focused on the efficient traversal of data (equivalent to joins in tables, which are rather slow and expensive in comparison to other operations).

The relevance of these new shapes is increasing, and it is important to keep in mind that it is desirable to keep a clean, logical and abstract view of the data for all of them, not only tables. It is commonly said for example that tree-like data undesirably exposes physical layouts to the user, but this statement is inaccurate and this misconception is due to the fact that the principle of data independence has not yet found its way in all database management systems with a focus on tree shapes (e.g., document stores), which unlock immense power to normalize and denormalize data at will.

1.3.3 Velocity

Third, a distortion has appeared between how much data we can store in a given volume, how fast we can read it and with which latency. This distortion carried so many challenges that this drove the emergence of many data processing technologies that we know today (MapReduce, Apache Spark, etc.).

Today, data is generated automatically: by sensors, but also by logging how people use websites or applications: it has simply become a byproduct of human activity.

In order to understand what happened in the past 20 years, we need to look at three factors:

- Capacity: how much data can we store per unit of volume?
- Throughput: how many bytes can we read per unit of time?
- Latency: how much time do we need to wait until the bytes start arriving?

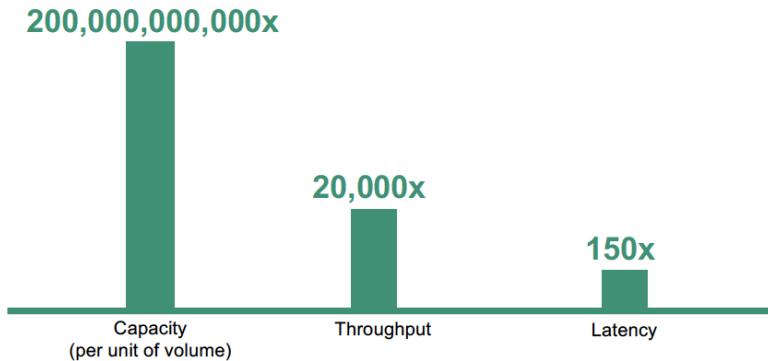
The earliest commercially available hard drive dates back to 1956 and was the IBM RAMAC 350. It had a capacity of 5 MB, a throughput of 12.5 kB/s and a latency of 600 ms. Its dimensions were 1.7m by 1.5m by 70cm, which would be enormous for today's standards.

Nowadays, we can hold a thousand times this much information in our fingers with a USB stick. But more precisely, let us look at the largest available hard drive in 2021, which is as far as we are aware the Western Digital Ultrastar DC HC650. It has a capacity of 20 TB, a

throughput of 250 MB/s and a latency of 4.16 ms. Its dimensions are 10.1cm by 14.7cm by 2.6cm.

Now, if we compare how these quantities have evolved:

- Capacity per unit of volume has increased by a factor of 200,000,000,000.
- Throughput has increased by a factor of 20,000.
- Latency has decreased by a factor of a mere 150.



Let us make an analogy: take a book with 600,000 words, which an average human can read with, say, 1,000 words a minute. The book can be read in 10 hours.

Let us now imagine that, in two centuries from now (which is more than the 70 years considered for hard drives), the size of books is hypothetically multiplied by the same factor of 200,000,000,000, making it 120,000,000,000,000,000 words. And that the speed at which we read it increased by the same factor of 20,000, making it 20,000,000 words per minute. Now, it will take 11,400 years to finish the book.

Yes. We Are In Big Trouble.

However, a key insight is: imagine we spread the read to 20 million people over a social network: then, if everybody takes one page and reads it in parallel, we are back to 10 hours. This is called parallelization, and this is the first technique used by modern data processing platforms.

The second technique addresses the growing discrepancy between throughput and latency: batch processing. Rather than processing records one by one, batches are created and the processing is done batch by batch. Also this is used by modern data processing platforms, where batches are commonly known as tasks.

Which leads us to my attempt to define Big Data: *Big Data is a portfolio of technologies that were designed to store, manage and analyze data that is too large to fit on a single machine while accommodat-*

ing for the issue of growing discrepancy between capacity, throughput and latency.

1.4 Applications

There are numerous applications to Big Data. One of the paramount consumers is the field of High-Energy Physics: at CERN, 50 PB of data is produced every year. There are a billion collisions happening every second. CERN has to-date about 15,000 servers with in total 230,000 cores. In fact, the quantity of raw data generated is so enormous that most of it is filtered out right away and only a tiny part is actually stored on persistent storage. Then, more passes and quality filters will lead to the publication of smaller and more manageable, curated datasets.

And this might surprise you: such vast quantities of data are often archived, in 2021, on tape. This might seem outdated, but this remains the cheapest storage technology available for long-term archiving, if you are fine with waiting for a few hours until you can get your hands on the data.

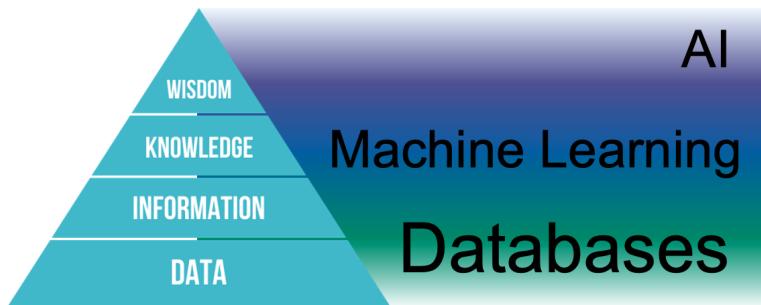
Another use case for large amounts of data is the attempt by the Sloan Digital Sky Survey (SDSS) to catalogue all visible objects in the sky. Phase IV has finished last year, and this dataset is the most detailed 3D map of the universe ever made. 200 GB of data was generated every night.

Another storage support gaining in importance is DNA: a major breakthrough to edit it was made with the CRISPR-Cas9 technique. Also, everybody on the planet now knows about the messenger RNA technique (which is, in fact, a few decades old but had not been tested as such large scales before), which resembles Computer Science and assembly code development more than biology¹.

1.5 Scope of this book

The scope of data science is very fast: from data we extract information, which we turn into knowledge and finally wisdom.

¹see <https://berthub.eu/articles/posts/reverse-engineering-source-code-of-the-biontech-pfizer-vaccine/> for more detailed explanations



Data storage and management is only the first stage of this pyramid, and is completed with Machine Learning and Artificial Intelligence. In this course, we focus on data storage, management and processing at large scales but keeping in mind that the separation from Machine Learning is becoming more blurry every day and in several aspects.

	Concepts	Technologies
Storage	Object storage	S3, Azure Blob Storage
	Distributed file systems	HDFS
	Syntax	XML, JSON
Models	Wide column stores	HBase
	Data models and schemas	XML/JSON Schema
	Cubes	OLAP
	Graphs	neo4j, Cypher
Processing	2-step distributed query processing	Hadoop MapReduce
	Resource management	YARN
	DAG-based distributed query processing	Spark
Management	Document storage	MongoDB
	Query languages	JSONiq

We now start our journey by brushing up on relational database management systems and SQL.

1.6 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Are you capable of sketching the history of databases (ancient and modern) to a colleague in a few minutes?
- b. Do you know who Edgar Codd is?
- c. Can you explain what Data Independence is, and why it is important?
- d. Do you know the rough conceptual difference between data, information, and knowledge?
- e. Can you cite the five fundamental shapes of data, and how structured data can be characterized?
- f. Can you briefly explain what a data model is?
- g. Do you know the standard prefixes of the International System of Units (when the exponent in base 10 is a positive multiple of 3)?
- h. Can you list the main four technologies commonly referred to as NoSQL?
- i. Do you know the three Vs?
- j. Can you briefly define capacity, throughput and latency? Do you know their typical units?
- k. Can you explain why and how the evolution of capacity, throughput and latency over the last few decades has influenced the design of modern database systems?
- l. Can you name a few big players in the industry that accumulate and analyze massive amounts of data?
- m. Do you know the difference between bit and byte?
- n. Can you name a few concrete examples that illustrate the various orders of magnitude of amounts of data?

1.7 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Codd, EF (1970). *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM.

Chapter 2

Lessons learned from the past

2.1 Data independence

In spite of the many revolutions that happened in the past two decades, it is very important not to lose sight of the fundamental aspects of data management. The fundamental socle on which modern data management is based is *data independence*, following the seeding paper by Edgar Codd in 1970.

Data independence means that the logical view on the data is cleanly separated, decoupled, from its physical storage. In its original version, it was suggested that the most natural logical view on data is in form of a table. Everybody understands tables and, as we say, they are thousands of years old.

A relational database management system (RDBMS) exposes this logical model, together with logical building blocks for manipulating it, on top of a physical layer. In the 1970s, this physical layer was an individual computer, and the data was stored on the hard drive, in a format that is irrelevant to the user of the RDBMS. Then, the RDBMS would get updates, for example, every year, and the format on the disk might evolve. But the user does not have to change anything in the way it interacts with the system: their queries will continue to work. However, as products get updates and machines get more powerful, the user might notice that the queries get automagically faster. Of course, new features can be added on the logical layer, but this is always with long-term thinking.

Furthermore, if these features get standardized, the user can even reuse their queries with different vendors. Vendors generally have an interest to cooperate with each other and standardize their models and query languages for interoperability across different systems.

A database management system stack can be viewed as a four-layer stack:

- A logical query language with which the user can query data;
- A logical model for the data;
- A physical compute layer that processes the query on an instance of the model;
- A physical storage layer where the data is physically stored.

What happened in the past two decades is that the physical layer was changed: the same logical model and query language can now run on clusters with thousands of machines, rather than a single machine. Such large-scale clusters of machines will be the focus of our lecture. What is important to remember in this chapter is that the look and feel of querying data should remain the same to the end user, whether we are talking about GBs of data on a single machine or of PBs of data on a large cluster.

Of course, we will see that there are other data shapes out there than tables: there are trees, there are cubes, there are graphs, and there is unstructured data. But data shapes should be seen as a logical concept: data independence continues to apply no matter what data shape is supported by the database management system.

This is the reason why we will now take the time to look into the lessons learned from 50 years of relational database management systems, and insist that they are still very relevant today even for non-relational database management systems.

2.2 Formal prerequisites

Before we dive into the relational model and algebra, let us take a brief detour through some mathematics formalism and notations that are typically taught at the Bachelor's level.

2.2.1 Sets and relations

set $x \in S$ denotes that a set S contains the element x . $x \notin S$ denotes that it does not. As it only relies on the notion of containment, a set is thus unordered, and has no duplicates.

inclusion A set A is included in another set B if $\forall a \in A, a \in B$. This is denoted $A \subseteq B$.

Cartesian product $A \times B$ denotes the cartesian product of the set A with the set B , i.e., it is a set that contains all pairs made of one element of A and one element of B . \times is associative and one can compute cartesian products over any number of sets: $A \times B \dots \times Z$

relation A relation R on a family of sets $(A_i)_{i=1..n}$ is a subset of their cartesian product: $R \subseteq A_1 \times \dots \times A_n$. A relation is thus a list of tuples. The set of all relations on $(A_i)_{i=1..n}$ is the powerset of the cartesian product: $\mathcal{P}(A_1 \times \dots \times A_n)$.

partial function A partial function p between two sets A and B is a relation that does not associate any element of A to more than an element of B :

$$\forall (x_A, x_B), (y_A, y_B) \in p, x_A = y_A \Rightarrow x_B = y_B$$

Rather than writing $(a, b) \in p$ (with the semantics of p being a relation), we write $p(a) = b$ or sometimes $p_a = b$ or $p : a \mapsto b$.

The set of all partial functions from A to B is denoted $A \rightarrowtail B$. A is called the domain of p , B its codomain.

The subset of A with the elements that do get associated to an element of B is denoted $support(p)$. It is the inverse image of B .

$$support(p) = \{a \in A \mid \exists b \in B, p(a) = b\} = p^{-1}(B)$$

function A function f between two sets A (the domain) and B (the codomain) is a relation that associates every element of A to exactly one element of B :

$$\forall a \in A, \exists! (x_A, x_B) \in f, x_A = a$$

A function is nothing else than a partial function of which the support is the entire domain A .

The set of all functions from A to B is denoted $A \rightarrow B$ or sometimes B^A .

Rather than writing $(a, b) \in f$ (with the semantics of f being a relation), we write $f(a) = b$ or sometimes $f_a = b$ or $f : a \mapsto b$.

Functions can be injective, surjective, bijective, but we do not need these concepts here.

2.2.2 Sets commonly used

There are a few standard mathematical sets such as \mathbb{N} (natural integers), \mathbb{Z} (relative integers), \mathbb{D} (decimals), \mathbb{Q} (rational numbers), \mathbb{R} (real numbers).

We also use the set of the two booleans, \mathbb{B} and denote the set of all strings \mathbb{S} . \mathbb{S} is a monoid when augmented with a concatenation operation.

Finally, we use the notation \mathbb{V} to denote all values, which includes the union of all sets above, but also many other possible values such as dates, times, timestamps, URLs, sequences of bits, but also not excluding data structures such as sets, lists, trees, maps and so on. Ideally, it would be the set of 'everything', but this is a concept that leads to contradictions and kept a lot of mathematicians busy in the 19th century. However, considering that the storage capacity of a single machine, but also even of the visible universe is finite, it is reasonable to consider that \mathbb{V} contains anything that can be stored on persistent storage or in memory as a sequence of 0s and 1s and according to a convention to interpret these bits.

\mathbb{V} is thus, as far as we are concerned a finite set, and we also consider \mathbb{S}, \mathbb{N} , etc, to be limited to values that fit on a machine.

We also use the set $\mathbb{A} \subset \mathbb{V}$ that contains all atomic values. Atomic values are values that are not structured, i.e., this excludes objects, arrays, lists, sets, trees, bags, etc. but only includes strings, integers, Booleans, dates and so on.

2.3 Relational database management systems

2.3.1 Main concepts

Relational database management systems (RDBMS) are based on a tabular data format. The core of the relational model is thus the concept of table.

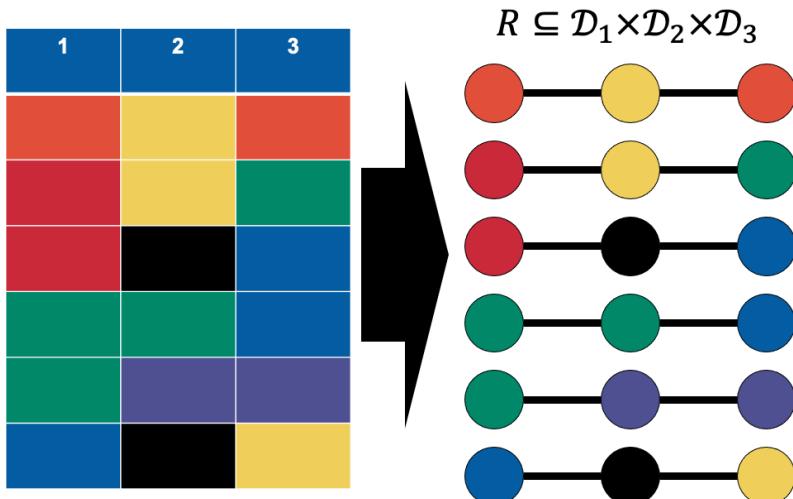
On a high level, a table is quite simple and the first class citizens in this model are:

- The *table* itself, which can be seen as a collection of records. For example, in an employee table, each record would be a person, in a products table, each record would be a product, etc. Thus, in some products with models that generalize tables, the term *collection* is used instead.
- The *attribute*, which is a property that records can have. For example, if a record is an employee, then an attribute can be their last name, their city of residence, etc. Other terminologies commonly found as synonyms of *attribute* are: *column*, *field*, *property*, *key*.
- The *row*, which is a record in a collection. A row associates properties with the values applicable for the record it represents. Other terminologies commonly found as synonyms of *row* are: *record*, *entity*, *document*, *item*, and even the classy *business object*, which seems quite popular with people who have an MBA.

- The *primary key*, which is a particular attribute or set of attributes that uniquely identify a record in its table. For example, the social security number (AHV in Switzerland) can identify a person, or a code (HG, CAB) can identify a building at ETH.

2.3.2 The formalism behind the relational model

Tables can be defined formally, that is, as purely mathematical objects. Most database textbooks introduce tables as mathematical relations over the domains associated with each attribute. A mathematical relation over several domains is defined as a subset of the Cartesian product of the domains. Each element in a mathematical relation, that is, a record in the table, is thus a tuple made of values picked in each domain.



For example, let us consider a table with four attributes:

- Name, the domain of which is a string (let us denote this set \mathbb{S});
- First name, the domain of which is a string (\mathbb{S});
- Physicist, the domain of which is \mathbb{B} , the set of Booleans (true and false);
- Year, the domain of which is \mathbb{N} , the set of natural integers.

A relational table is thus a subset of $\mathbb{S} \times \mathbb{S} \times \mathbb{B} \times \mathbb{N}$. An example of tuple that could belong to this subset (and thus to the table) is

(Einstein, Albert, true, 1905), where each value is implicitly associated with an attribute like so:

- Name: Einstein
- First name: Albert
- Physicist: true
- Year: 1905

And a relational table is simply a set of such tuples. In fact, this is where the names *relational* as well as *relation* come from.

Expressed formally, given a family of attributes $(A_i)_{1 \leq i \leq n}$ and their associated domains

$$(Domain(A_i))_{1 \leq i \leq n}$$

A table T is a relation over these domains, i.e.

$$T \subseteq Domain(A_1) \times Domain(A_2) \times \dots \times Domain(A_n)$$

However, there are several issues with the above definition:

- The values in a mathematical tuple are indexed by natural integers, whereas in the logical relational model, they are indexed by strings: the attributes they are associated with. For example, Albert is associated with “First name” and not with the integer 2, its position.
- Mathematical tuples are ordered, whereas in a relational table, the order of the columns is irrelevant.
- Mathematical relations can only be sets, which makes it very difficult to easily relax and generalize to alternative forms of the relational model based on lists or bags.

These issues make the bridge between the math and the logical relational model a bit blurry, which also raises a lot of questions on smaller details that can distract from the content actually being taught and usable in practice.

A collection of records is a set of partial functions from \mathbb{S} to \mathbb{V} . We denote the set of collections \mathcal{C} .

$$\mathcal{C} = \mathcal{P}(\mathbb{S} \rightarrow \mathbb{V})$$

Each record is thus modelled as a partial function mapping strings to values, e.g. with our example, if we call our record f:

- $f(\text{"Name"}) = \text{"Einstein"}$
- $f(\text{"First name"}) = \text{"Albert"}$
- $f(\text{"Physicist"}) = \text{true}$
- $f(\text{"Year"}) = 1905$
- f is undefined for any other string.

In practice and in the context of databases, we prefer to use subscript notations for the function calls, and the quotes of the attributes are omitted if they are simple (e.g., no spaces):

- $f_{\text{Name}} = \text{"Einstein"}$
- $f_{\text{"First name}} = \text{"Albert"}$
- $f_{\text{Physicist}} = \text{true}$
- $f_{\text{Year}} = 1905$
- f is undefined for any other string.

We can also represent the partial function extensively:

Name \rightarrowtail Einstein

First name \rightarrowtail Albert

Physicist \rightarrowtail true

Year \rightarrowtail 1905

S \qquad **V**

However, most data scientists feel more comfortable with a visual:

S	Name	First name	Physicist	Year
V	Einstein	Albert	true	1905

The definition with partial functions reflects more accurately the fact that attributes are unordered, and avoids “hand-waving” explanations with ordered tuples indexed by integers.

The concept of collection does not fully reflect what a table is. For a table, we need to throw in three additional constraints: relational integrity, domain integrity and atomic integrity.

Relational integrity

A collection T fulfils relational integrity if all its records have identical support:

$$\forall t, u \in T, \text{support}(t) = \text{support}(u)$$

If a collection, in particular a table, has relational integrity, then this common support is a property of the table and contains the attributes of the table T : Attributes_T .

The extension of the table, sometimes denoted Extension_T when used together with Attributes_T , is its actual content, which is T itself. We use T or Extension_T interchangeably depending on the context.

This collection does not respect relational integrity:

	Name	First name	Physicist	Year
Country	Einstein	Albert	true	1905
		Alan	false	1936
A	Gödel	Kurt		1931

This one does:

	Name	First name	Physicist	Year
✓	Name	First name	Physicist	Year
Einstein	Albert	true	1905	
✓	Name	First name	Physicist	Year
Turing	Alan	false	1936	
✓	Name	First name	Physicist	Year
Gödel	Kurt	false	1931	

in which case we typically "merge" the attributes in the display:

Name	First name	Physicist	Year
Einstein	Albert	true	1905
Turing	Alan	false	1936
Gödel	Kurt	false	1931

Domain integrity

A collection T fulfils domain integrity if the values associated with each attribute are restricted to a domain. We define these domains with a function D mapping strings (the attributes) to domains (unused attributes are just associated with empty domains, i.e. this does not need to be a partial function):

$$D \in \mathcal{P}(\mathbb{V})^{\mathbb{S}}$$

A collection T fulfils the domain integrity constraint specified by D if, for each row, the values are in the specified domains:

$$\forall t \in T, \forall a \in \text{support}(t), t.a \in D(a)$$

Typically, the domains $(D(a))_a$ will be standard sets such as integers, strings, booleans, dates and so on.

Concretely, the domain mapping D is called a *schema*, i.e., the names of the columns, and the domain of each column (string, integer, date, boolean, and so on).

This collection respects domain integrity:

Name	First name	Physicist	Year
String	String	Boolean	Integer
Einstein	Albert	true	1905
Turing	Alan	false	1936
Gödel	Kurt	false	1931

This one does not:

Name	First name	Physicist	Year
String	String	Boolean	Integer
Einstein	Albert	true	1905
Turing	Alan	0	1936
Gödel	Kurt	false	thirty-one

Note that the definition of domain integrity still allows records with missing values. When combining domain integrity with relational integrity, we typically only consider the case when the support of the schema (attributes associated with non-empty domains) matches exactly the support common to all records; otherwise, this is still sound, however the extra domains in the schema are unused and thus useless.

Atomic integrity

A collection T fulfils the atomic integrity constraint if the values used in it are only atomic values, i.e.

$$T \subseteq \mathbb{S} \rightarrow \mathbb{A}$$

This means that the collection does not contain any nested collections or sets or lists or anything that has a structure of its own: it is “flat.”

This collection respects atomic integrity:

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Set theory	Pfäffikon	SZ	8808
25-000-000	Felix Bloch	123-4567-89L	Set theory	Pfäffikon	ZH	8330

This one does not:

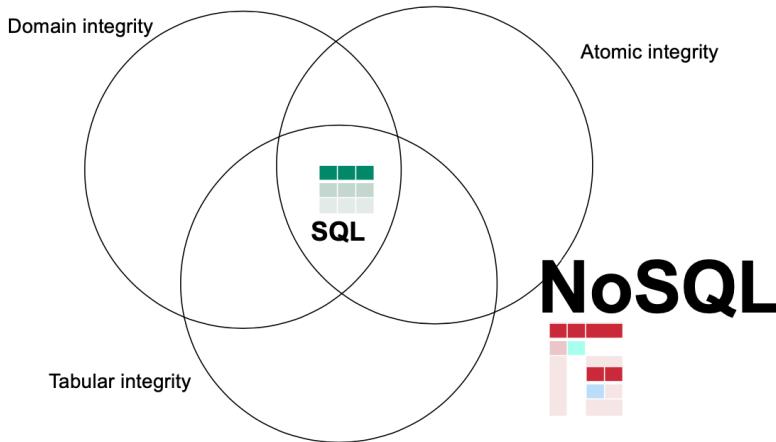
Legi	Name	Lecture		City	State	PLZ
32-000-000	Alan Turing	Lecture ID xxx-xxxx-xxX	Lecture Name Cryptography	Bletchley Park	UK	MK3 6EB
		263-3010-00L	Big Data			
62-000-000	Georg Cantor	Lecture ID 263-3010-00L	Lecture Name Big Data	Pfäffikon	SZ	8808
		123-4567-89L	Set theory			
25-000-000	Felix Bloch	Lecture ID 123-4567-89L	Lecture Name Set theory	Pfäffikon	ZH	8330

The relational model and beyond

A relational table is defined as a collection that fulfills all three fundamental integrity constraints: all records have the same support, there

is a schema assigning a domain to all attributes and the records respect the schema, and the values are all atomic.

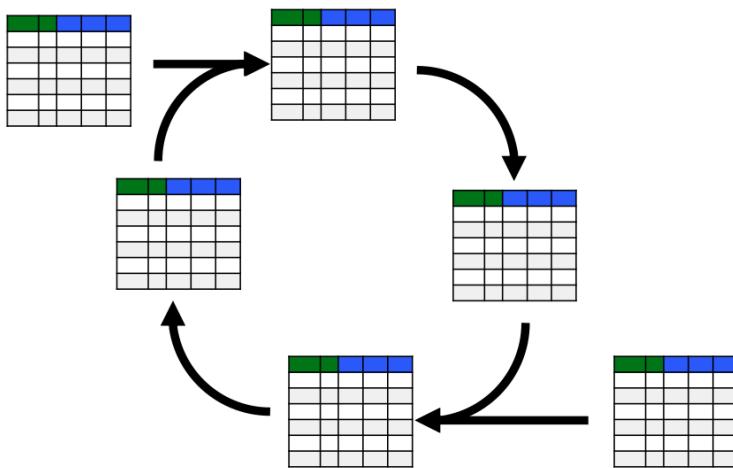
Being able to express these three integrity constraints separately from the core definition of a collection has a high pedagogical value, because it raises the awareness of these constraints for tables in the relational world, but also will allow us, in this course, to selectively relax one, two or all three constraints as we see fit. When these constraints are relaxed, we enter the beautiful world of NoSQL databases, and we will see that this comes down to manipulating collections of trees, not tables.



Also, as the definition of collections is explicitly based on a *set* of partial functions, it can very easily be extended to bag semantics, or even list (ordered rows) semantics based on the needs of the relational algebra: all we need to do is substitute “set” for “bag” or “list” in the definition.

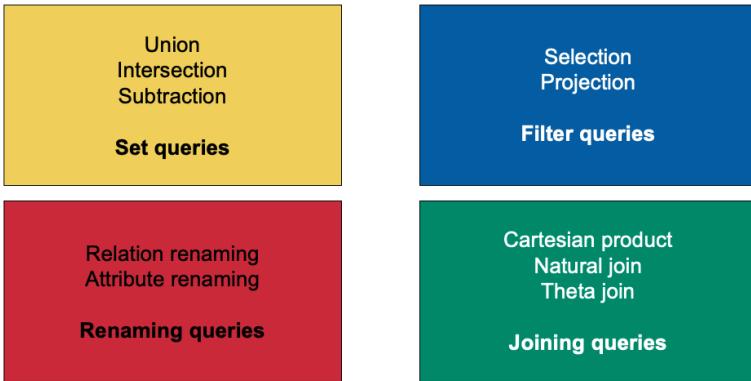
2.3.3 Relational algebra

Once relational tables have been formally defined, it is possible to manipulate them. Mathematically, the framework that allows for this is called the relational algebra. Like numbers can be manipulated with addition, multiplication, relational tables can be manipulated with many different operators.



These operators can be classified in four broad categories:

- Set queries act on relational tables as sets (as previously described): one can take the union or the intersection of two sets, or subtract a set from another. These operators directly and naturally translate to relational tables.
- Filter queries. These operators take a portion of a table: some or all columns, some or all rows, etc. They are known as projection and selection. There also exists a fancy operator called the “extended projection” that can be used to add more computed columns.
- Renaming queries. These operators can rename columns.
- Joining queries. These operators can take the Cartesian product of two tables, potentially filtering to match values from both sides. The latter is called a join.



Let us give a few examples with the most common operators.

2.3.4 Selection

A selection takes a subset of the records belonging to the table. It takes a parameter, which is a predicate on the attributes. This predicate is evaluated for each record: if it is true, the record is kept, if it is false, it does not appear in the selection.

The notation used is the σ letter. For example,

$$S = \sigma_{B \leq 2}(R)$$

corresponds visually to:

R		
A	B	C
string	integer	boolean
foo	1	true
bar	2	false
foo	3	false
foobar	4	true

S		
A	B	C
string	integer	boolean
foo	1	true
bar	2	false

Predicates commonly involve arithmetics (addition (+), subtraction (-), multiplication (\times), division (/), integer division (e.g., n), modulo (%)), comparison ($=$, \neq , $<$, $>$, \leq , \geq), logic (\wedge (and), \vee (or), \neg (not)), constant literals (hard-coded strings like “foo”, numbers like 3.14, etc)

and attributes (the names of the column of which to take the value)). Pay attention to precedence, or use parentheses to avoid ambiguities. Predicate syntax can be extended at will, but of course it is important to carefully document any extension so other people can understand it.

2.3.5 Projection

A projection keeps all records, but removes columns. It takes as a parameter the names of the attributes to keep in the projection.

The notation used is the π letter. For example,

$$S = \pi_{A,C}(R)$$

corresponds visually to:



R		
A	B	C
string	integer	boolean
foo	1	true
bar	2	false
foo	3	false
foobar	4	true

S	
A	C
string	boolean
foo	true
bar	false
foo	false
foobar	true

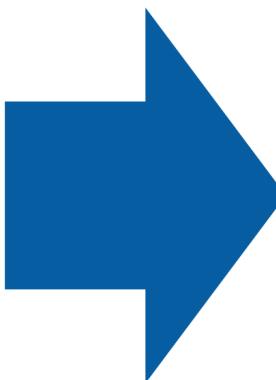
2.3.6 Grouping

A grouping, also called aggregation, merges records by grouping on some attributes, and aggregating on all others.

The notation used is the γ letter. For example,

$$S = \gamma_{G, \text{SUM}(A) \rightarrow A}(R)$$

groups by G and aggregate the values in column A (within the same group) with a sum, and corresponds visually to:



R	
G	A
string	integer
foo	19
bar	28
bar	265
foo	4
foobar	54
foo	46
bar	245
foobar	3456
bar	139

S	
G	A
string	integer
foo	69
bar	677
foobar	3510

It is easy to get things wrong with grouping. A general recommendation is to make sure to carefully classify each attribute as either a grouping attribute (it appears on its own in the γ subscript, like G above) or a non-grouping attribute (it must then either be left out, or appear in an aggregation function, with an arrow pointing to the new name of the aggregated column, $SUM(A) \rightarrow A$ in our example). The new column may have a different name than the original one. Do not use an attribute both as grouping and non grouping. Grouping with no grouping key is possible and results in an aggregation of the entire table to a single value (for example, to take a max or a sum of all values in an attribute).

The most common aggregation functions are COUNT, SUM, MAX, MIN, AVERAGE but there are many others (ask a statistician!). Please note that, in the relational algebra, aggregation functions must aggregate to a single value, otherwise it leads to nested structures that break atomic integrity (you may have guessed that we will, of course, be doing this sort of thing in the course, but this is then NoSQL).

2.3.7 Renaming

Renaming a column changes the name of a column and is denoted with ρ , i.e.,

$$S = \rho_{A \rightarrow D}(R)$$

2.3.8 Extended projections

Extended projections are less common in the formalism of the relational algebra, however they are used very often in practice, so that it is worth being aware of its existence.

An extended projection can compute values with syntax similar to that used in selection predicates, and then assign the result to a new column denoted with an arrow, like the rename operator.

$$S = \pi_{A+C \times 2 \rightarrow D}(R)$$

It is possible to mix several such computations, also with pure projections:

$$S = \pi_{B,A+C \times 2 \rightarrow D,2 \times B \rightarrow E}(R)$$

You need to be careful not to create several attributes with the same name.

2.3.9 Cartesian products

A Cartesian product combines each tuple from the left relational table with each tuple from the right relational table.

The notation used is the \times symbol. For example,

$$T = R \times S$$

is the Cartesian products of R with S and corresponds visually to:

R			T				
A	B	C	A	B	C	D	E
string	integer	boolean	string	integer	boolean	string	integer
foo	1	true	foo	1	true	foo	1
bar	2	false	foo	1	true	bar	2
			foo	1	true	foo	3
			bar	2	false	foo	1
			bar	2	false	bar	2
			bar	2	false	foo	3

S	
D	E
string	integer
foo	1
bar	2
foo	3

Cartesian products should be handled with care: imagine the size of the resulting table if the table on the left has a billion records and that on the right has a million records (tables with millions or billions of records are quite common in the real world).

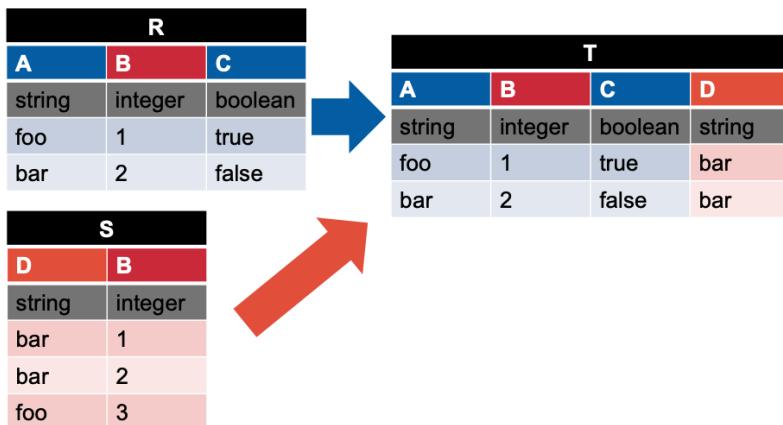
2.3.10 Joins

Often, we do not use Cartesian products but joins, which are a “filtered” Cartesian product in which we only combine directly related tuples and omit all other non-matching pairs.

The notation used is the \bowtie symbol. For example,

$$T = R \bowtie S$$

joins R and S but only keeps records that coincide on the attributes common to both sides (here, B must match):



The joins as above, called natural joins, are very common in relational databases and are less expensive than Cartesian products, but remain more expensive than simpler operators such as projection and selection.

There are other kinds of joins:

- Theta joins explicitly specify the joining criterion instead of matching the common attributes. The criterion is then supplied as a predicate in the subscript of \bowtie with the same syntax as σ , e.g. $T = R \bowtie_{A=D} S$. Note that, in this case, common attributes require care and, if expressed with mathematical formulas, renames should be used to keep the formulas consistent, e.g. $T = R \bowtie_{B=E} \rho_{B \rightarrow E} S$.
- Outer joins keep records with no match on the other size, keeping the other attributes absent. Note that this breaks relational integrity in its strictest sense, as these joined records will not have the full support of the resulting relational table. Relational

integrity can also be preserved by picking a default value in the domains of the attributes that are missing.

- Semi-outer joins are like (full) outer joins but only keep unmatched records on the left, or only on the right.

2.3.11 Combining operators

Relational algebra operators can be combined at will and nested in more complex formulas. It is important to make sure you can write and read formulas involving relational algebra operators, explain what a formula does in English terms, or design one when given instructions in English.

2.4 Normal forms

When creating databases, the schemas, that is, the columns and their domains, should be designed with care. If some rules are not respected, things can go wrong as the data can easily become inconsistent because it is duplicated:

- deletion anomalies occur in poorly designed databases when deleting a record makes the database inconsistent. For example, deleting the only order of a specific phone in an online shop might completely delete the phone data.
- insertion anomalies occur in poorly designed databases when inserting a new record makes the database inconsistent. For example, inserting a new order might cause duplicate and inconsistent data to appear on the product.
- update anomalies occur in poorly designed databases when updating a record makes the database inconsistent. For example, updating data on a phone in an order might make it inconsistent with the data in other orders.

In order to avoid such anomalies, best practice dictates to follow so-called normal forms. Normal forms are typically taught in Bachelor-level database lectures in Computer Science curricula in many universities (in particular in German-speaking countries). A complete theory of normal forms would demand much more space, so that we only quickly give a small survey to develop an intuition.

The first normal form was already covered earlier: it is in fact atomic integrity.

This table does not respect the first normal form: there are tables nested in it.

Legi	Name	Lecture		City	State	PLZ
		Lecture ID	Lecture Name			
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
		263-3010-00L	Big Data			
62-000-000	Georg Cantor	Lecture ID	Lecture Name	Pfäffikon	SZ	8808
		263-3010-00L	Big Data			
		123-4567-89L	Analytical Engines			
25-000-000	Ada Lovelace	Lecture ID	Lecture Name	Pfäffikon	ZH	8330
		123-4567-89L	Analytical Engines			

This table does respect the first normal form: all values are atomic.

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Analytical Engines	Pfäffikon	SZ	8808
25-000-000	Ada Lovelace	123-4567-89L	Analytical Engines	Pfäffikon	ZH	8330

The second normal form takes it to the next level: it requires that each column in a record contains information on the *entire* record.

This table is not in second normal, because the column “Name” refers to a student (identified with their Legi), but the whole record semantically corresponds to an attendance of a student to a lecture; we say that the Legi and Lecture ID, together, are a primary key of the record, but “Name” does not functionally depend on the full primary key, only on half of it:

Legi	Name	Lecture ID	Lecture Name	City	State	PLZ
32-000-000	Alan Turing	xxx-xxxx-xxX	Cryptography	Bletchley Park	UK	MK3 6EB
32-000-000	Alan Turing	263-3010-00L	Big Data	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	263-3010-00L	Big Data	Pfäffikon	SZ	8808
62-000-000	Georg Cantor	123-4567-89L	Analytical Engines	Pfäffikon	SZ	8808
25-000-000	Ada Lovelace	123-4567-89L	Analytical Engines	Pfäffikon	ZH	8330



These three tables are in second normal form, in fact, they are how the previous table can be fixed: by cleanly separating information on the attendance from information specific to the lecture or specific to the student in separate tables:

Legi	Name	City	State	PLZ	Legi	Lecture ID
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB	32-000-000	xxx-xxxx-xxX
62-000-000	Georg Cantor	Pfäffikon	SZ	8808	32-000-000	263-3010-00L
25-000-000	Ada Lovelace	Pfäffikon	ZH	8330	62-000-000	263-3010-00L
					62-000-000	123-4567-89L
					25-000-000	123-4567-89L
Lecture ID		Lecture Name				
xxx-xxxx-xxX		Cryptography				
263-3010-00L		Big Data				
123-4567-89L		Analytical Engines				

If you paid attention, you will have noticed that the former table can be obtained by joining the latter three tables. Normalizing, in fact, can be seen as the opposite of joining (which is also called denormalizing as we will see in this course).

The third normal form additionally forbids functional dependencies on anything else than the primary key, for example the following table is not in third normal form because “PLZ” functionally depends on “City” + “State”, but this is not a primary key (column “Legi” is).



Legi	Name	City	State	PLZ
32-000-000	Alan Turing	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Pfäffikon	SZ	8808
25-000-000	Ada Lovelace	Pfäffikon	ZH	8330

This can be fixed, again, by separating the data into two tables, with the data on “City”+“State” moved out to another table, like so:

Legi	Name	City	State	City	State	PLZ
32-000-000	Alan Turing	Bletchley Park	UK	Bletchley Park	UK	MK3 6EB
62-000-000	Georg Cantor	Pfäffikon	SZ	Pfäffikon	SZ	8808
25-000-000	Ada Lovelace	Pfäffikon	ZH	Pfäffikon	ZH	8330

If this sounds complicated, the following will probably come as good news: in Big Data, we often have to throw away normal forms and denormalize our data for better scalability and performance. More in the next chapters.

2.5 The SQL language

SQL was a follow up on the original idea of data independence by Edgar Codd. It was contributed by Don Chamberlin and Raymond Boyce (who gave his name to a normal form, too: the Boyce-Codd normal form).

SQL was originally named SEQUEL, for Structured English QUERy Language. The first commercial relational database that implemented it was called System R and was born in IBM Almaden (San Jose). Their first customer was Pratt & Whitney. However, due to a trademark issue, the name SEQUEL had to be changed to SQL. This explains why many people, including yours truly, pronounce SQL see-kwel while others pronounced es-kew-el.

In 1977, a company named Software Development Laboratories was created with another implementation. It was renamed in 1979 to Relational Software and then in 1982 as Oracle, who is one of the major RDBMS players today alongside IBM and Microsoft.

SQL is a declarative language, which means that the user specifies what they want, and not how to compute it: it is up to the underlying system to figure out how to best execute the query.

It is also a set-based language, in the sense that it manipulates sets of records at a time, rather than single values as is common in other languages.

It is also, to some limited extent, a functional language in the sense that it contains expressions that can nest in each other (nested queries). We will see, however, that the extent to which it can be done is limited and that new-generation querying languages that go beyond data in normal form are even more functional.

It is very important that you learn SQL as this is a fundamental building block in the area of databases. It is very common nowadays for many to use Python or R with a DataFrame API such as pandas. Even worse, it is also common to use spreadsheets to store and query sensitive data (with disastrous consequences regularly being reported in the news and causing damage to the reputation of any institution who gets it wrong – and it is easy to get things wrong on the consistency of data stored in a spreadsheet). However in many cases, when dealing with structured data, SQL is the language of choice (and can also be nested in host languages such as Python).

We will give an example of a query that has many of the relational operators:

```
SELECT c.century AS cent,
       COUNT(c.name) AS num_captains,
       SUM(s.id) AS ships
  FROM captains c, ships s
 WHERE c.id = s.captain
 GROUP BY century
 HAVING COUNT(c.name) >= 3
 ORDER BY century DESC
 LIMIT 3 OFFSET 2
```

The `FROM` clause selects from which tables to read the data, in this case two tables, “captains” and “ships”. Implicitly the Cartesian product is computed.

The `WHERE` clause performs a selection: it only keeps the records for which the captain (from the captains table) is the captain of the ship (from the ships table). If you pay attention, you will recognize that this filter together with the Cartesian product is actually a theta join. Any reasonable SQL implementation will be smart enough to detect this and evaluate this query efficiently (joins can be computed in linear time rather than quadratic!).

The `GROUP BY` clause performs an aggregation, with century as a grouping key. Aggregations on the captain name (`COUNT`) and the ship id (`SUM`) are done in the `SELECT` clause.

The `SELECT` clause is also where projections are made: it lists the columns to include in the results. Renames are also made in this clause with `AS`.

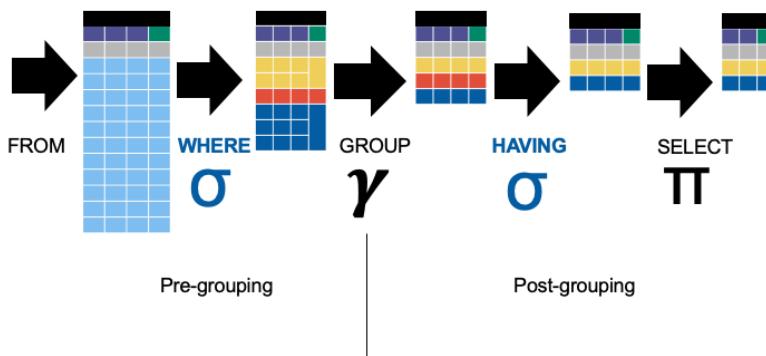
The `HAVING` clause is in fact like the `WHERE` clause, but performs a selection after, rather than before, the grouping.

The `ORDER BY` clause reorders the output rows according to the specified keys.

The `LIMIT` and `OFFSET` clauses allow pagination of the output: `OFFSET` specifies how many records to skip, and `LIMIT` specifies how many records to output after the skipped ones.

All clauses are optional except for `SELECT` and `FROM`.

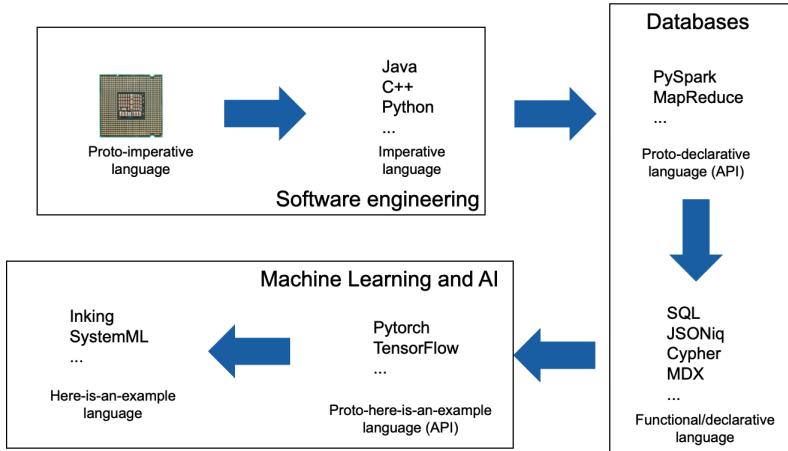
Internally a SQL query will be converted to a query plan that is closely related to the relational algebra, for example like so:



2.6 Languages

In the bigger picture, there are six main categories of languages: assembly code, later (fortunately) superseded by higher-level imperative pro-

gramming languages (C, C++, Java, ...). Functional and declarative query languages (SQL, JSONiq, SPARQL...), often co-habiting with lower-level APIs (DataFrame APIs such as in Apache Spark, ...). And Machine Learning frameworks (tensorflow, pytorch, keras...), which are slowly (and hopefully increasingly) supplemented with higher-level declarative languages for ML:



In this course, we are interested in relational, functional query languages as well as lower-level APIs such as DataFrames.

2.7 Transactions

Relational databases also offer convenient guarantees that prevent things from going wrong. There are four main properties (often called ACID) that the good old systems provide:

- Atomicity: either an update (called a transaction if it consists of several updates) is applied to the database completely, or not at all;
- Consistency: before and after the transactions, the data is in a consistent state (e.g., some values sum to another value, another value is positive, etc);
- Isolation; the system “feels like” the user is the only one using the system, where in fact maybe thousands of people are using it as well concurrently;
- Durability; any data written to the database is durably stored and will not be lost (e.g., if there is an electricity shortage or a disk crash).

We will see that many systems who scale beyond one machine have to make compromises on these guarantees, even though a lot of progress has recently been made.

2.8 Scaling up and out

So how are the limits of traditional relational database management systems reached? In many ways:

- there can be lots of rows: beyond a million records, a system on one machine can start showing signs of weakness; even though more recent systems manage to push it a bit higher on a single machine (e.g., close to a billion);
- there can be lots of columns: beyond 255 columns, a system on one machine can start showing signs of weakness or even not support it at all;
- there can be lots of nesting: many systems do not support nested data or, if they do, do so only in a limited fashion and it becomes quickly cumbersome;

This concludes our brush-up, as in fact most of what follows will be directly related to data that has lots of rows, lots of columns, or lots of nesting:

Lots of rows	Object Storage
Lots of rows	Distributed File Systems
Lots of nesting	Syntax
Lots of rows/columns	Column storage
Lots of nesting	Data Models
Lots of rows	Massive Parallel Processing
Lots of nesting	Document Stores
Lots of nesting	Querying

In the next chapter, we will start with storage systems that massively scale on the number of records that can be stored.

2.9 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain why it is important to take into consideration whether a use case is read-intensive, or write-intensive, or in-between?
- b. Can you explain why normal forms are important?
- c. Can you describe the first normal form in simple terms?
- d. Can you describe in simple terms (that is, without knowing the details of them) how higher normal forms (like Boyce-Codd) are related to joins?
- e. Can you explain why it is common, for large amounts of data, to drop several levels of normal form, and denormalize data instead?
- f. Can you give simple examples of denormalized data?
- g. Can you explain what a declarative language is?
- h. Can you explain what a functional language is?
- i. Can you explain why it matters to design query languages that are declarative and functional?
- j. Can you describe the major relational algebra operators: select, project, aggregate, sort, Cartesian product, join?
- k. Do you know the names of the basic components of the tabular shape at an abstract level (table, row, column, primary key) as well as the names of the most common corresponding counterparts in the NoSQL world?
- l. Do you know what each letter stands for in ACID?
- m. Do you know the basic SQL constructs: SELECT FROM WHERE, GROUP BY, HAVING, JOIN, ORDER BY, LIMIT, OFFSET, as well as nested queries?
- n. Can you write SQL queries?

2.10 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Garcia-Molina, H, Ullman, J, Widom, J (2008). *Database Systems: The Complete Book*. Pearson, 2nd edition. Chapters 1, 2, 3, and 6.

Lecture recordings. *Information Systems for Engineers*. ETH Zurich.

<https://www.youtube.com/c/GhislainFournysLectures>

Chapter 3

Cloud storage

In Chapter 1, we mentioned the Sloan Digital Sky Survey dataset, which is the result of several years of work. It is the most detailed 3D map of the universe ever made.

If you browse through the dataset, you will see that it has 273 TB of data, organized as 176,000,000 files in 680,000 directories. This data contains 260 millions of stars, 208 millions of galaxies, in total 1,231,000,000 objects on 4 spectra.

This is much more than any single disk in 2021 can store, which means that a solution with a single machine cannot possibly work.

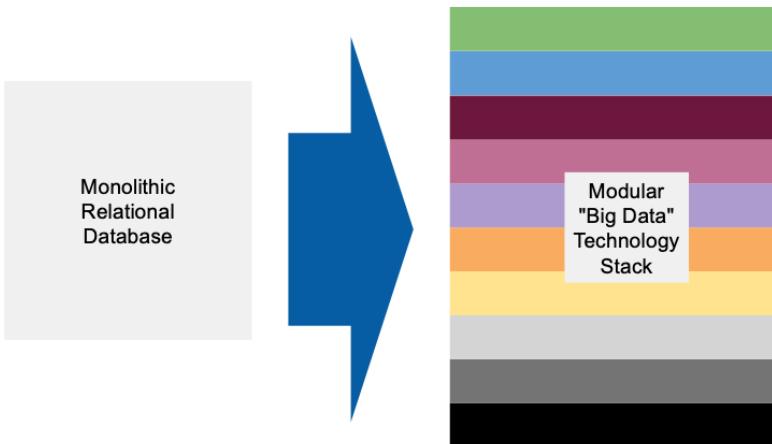
So, how do we deal with 273 TB of data organized as 176,000,000 files in 680,000 directories?

This is the question we answer in this chapter.

3.1 Storing data

In the previous chapter, we had a quick survey of relational databases. Traditionally, a relational database management system fits on a single machine. But Petabytes of data do not fit on a machine.

As a consequence, in this course, we will have to rebuild the entire technology stack, bottom to top, with those same concepts and insights that we got in the past decades, but on clusters of machines rather than on a single machine.



Indeed, the good news is that 99% of what we learn in the past 50+ years in the field of SQL and relational database management systems can be directly reused.

The relational algebra: selection, projection, grouping, sorting, joining, ... still makes sense on clusters.

Language features: SQL, declarative languages, functional languages, optimizations, query plans, indices... still make sense on clusters.

The table layout: tables, rows, columns, primary keys... still makes sense on clusters.

But not everything. We will have to let go, partially or fully, of consistency constraints. In the Big Data world, relational integrity may or may not hold. Domain integrity may or may not hold. Atomic integrity (also known as the first normal form) may or may not hold. And all normal forms may or may not hold.

Indeed, in what is called the NoSQL universe, these concepts are replaced with:

- Nested data: data that is not in first normal form;
- Heterogeneous data: data that does not fulfil domain integrity (it may not even have a schema!) and also not relational integrity.

Collectively, this alternate design is referred to as *data denormalization*. At large scales, it becomes desirable to sometimes denormalize, rather than normalize data, in order to get better performance.

The same goes for the ACID paradigm: Atomicity, Consistency, Isolation, Durability are very challenging to obtain on large systems (even though there is a lot of research going in this direction). As a consequence, this concept is replaced with thinking in terms of the so-called CAP theorem:

- (Atomic) Consistency
- Availability
- Partition tolerance

and in particular of a weaker condition of eventual consistency instead of full atomic consistency. But we will shortly come back to this. Before this, let us have a look again at our technology stack.

3.2 The technology stack

The technology stack we are going to rebuild goes from a level very close to the hardware (storage) and all the way to interaction with the end user (query language or even voice assistant).



Examples of storage systems involve the local file system on your laptop, network file systems (NFS), distributed file systems (HDFS from Hadoop, GFS from Google), as well as hosted and fully managed cloud storage (S3, Azure Blob Storage...).

Encodings are used to convert data (for textual syntax) to bits, as this is what computers understand. In the early days, ASCII was used,

later on ISO-8859-1 for latin characters but the recommendation today is to stick with Unicode, for example UTF-8, for full compatibility also across (human) languages. There are also binary encodings such as BSON (which is a binary format corresponding roughly to JSON, used in MongoDB).

Syntax is used when data is stored as (human-readable) text. This includes among others raw text (for natural language processing), CSV (for tables), XML and JSON (for trees), RDF/XML and Turtle (for graphs) and XBRL (for trees). Data can also be stored in binary form (protocol buffers, BSON, Parquet...), in which case it cannot be read by a human in a text editor.

Models are a higher-level representation of the data not necessarily tied to a particular syntax or binary format. For tables, it is the relational model. For trees, there are several (the XML infoset, the JSONiq data model...) as well as for graphs (RDF triples and labeled property graphs). For cubes (often “abusively” referred to as OLAP), the most common model uses dimensional coordinates with dimensions organized as hierarchies.

Validation involves languages to add constraints to data: XML Schema, JSON schema, JSound (which are optional for trees), relational schemas (in relational database systems), XBRL taxonomies (for cubes), etc.

Processing frameworks come in different forms and are mostly (at least on the theoretical level) orthogonal to data shapes: the older, two-stage forms (MapReduce) and newer generations that handle generic Directed Acyclic Graphs of dataflows (Tez, Apache Spark, Apache Flink, Ray...). There are also lower level frameworks at the level of the virtual machine, accessed via a command-line shell (Amazon EC2...).

Indices accelerate data lookups. They are mostly orthogonal to data shapes: hash indices, B+-trees, geographical or spacial indices, etc.

Data stores are the generic product providing all the functionality to end users: a relational database management system (RDBMS), document stores (e.g., MongoDB, CouchBase, ElasticSearch), higher-level wrappers on data processing (Hive, HBase, Databricks, Cassandra...) or more general integrated products (Snowflake, MarkLogic...).

Higher-level languages (to be contrasted with data processing APIs like Pandas or Spark’s DataFrames) are supported by some but not all data stores. SQL is the undisputed choice for tables. For trees, there are 20+ competing languages among which JSONiq, XQuery, N1QL, PartiQL... for graphs there is mostly Cypher and SPARQL. REST APIs (e.g. with GraphQL) also provide a (lower-level) access to a data store but are meant to be consumed by a host language, whereas query languages can be used on their own.

And finally, user interfaces can hide even the query language from the end user: spreadsheet software, Business Intelligence tools (Tableau,

QLikview, Access...), and even voice assistants (Siri, Alexa...).

But for this chapter, we are focusing on the storage level and in particular, with cloud storage services that require no installation.

3.3 Databases vs. Data lakes

It is obvious, but needs to be said: data needs to be stored somewhere. There are two main paradigms for storing and retrieving data.

On the one hand, data can be *imported* into the database (this is called ETL, for Extract-Transform-Load. ETL is often used as a verb). This causes more work for the user, as they need to import all their data before they are able to query it. Relational database management systems require an ETL of the data as well as some NoSQL products (MongoDB, HBase...). The reason is that the data is internally stored as a proprietary format that is optimized to make queries faster. This includes in particular building indices on the data.

On the other hand, data can also just be stored on some file system (whether local or distributed or in the cloud) and queried in place (*in situ*) by a data processing engine either with an API (Pandas, Apache Spark...) or query language (RumbleDB...). This paradigm is called the *data lake* paradigm and gained a lot of popularity in the past two decades. It is slower, however users can start querying their data without the effort of ETLing. Data lakes are best used in a context in which a full scan of the data is needed (e.g. with MapReduce or Apache Spark).

3.4 From your laptop to a data center

3.4.1 Local file systems

Typically, on a single machine, the data is stored on the local file system. This is possible if it is no more than a few Terabytes, depending on the disk capacity. There are also newer generations of disks called SSDs (Solid State Disks) that are more expensive and usually a bit smaller, but there exist SSDs of 1 TB nowadays.

When a file is stored on the local filesystem, it consists of both content and metadata. The content is made of the bits of the file itself (e.g., the text encoded to bits in UTF-8 if it is a JSON or XML file). The content is also sliced in blocks that are roughly 4 kB of size (this varies with the file system, but the order of magnitude does not change) and a bit is never read individually: files are read block by block. This is to optimize the balance between throughput and latency (more on this in the next chapter).

The metadata can be seen as a small relational table whose attributes are: the name of the file, access rights, the owner, the group of the owner, the last modification time, the creation time, the size, etc.

Furthermore, files are organized in a hierarchy of files and directories familiar to most people.

3.4.2 More users, more files

How does this scale to more users? A disk can be made accessible via the network (LAN for Local Area Network) and shared with other people. There exists also larger networks (WAN for Wide Area Networks), however it is difficult to scale concurrent access and problems can arise already when two people work on a file at the same time (e.g., for sharing an Excel file, a special mode must be enabled to avoid issues, a feature that text files surely do not have).

The other scalability problem is that a local file system can easily support 1,000 files or even 1,000,000 files, but can hardly make it to billions of files.

In order to make this scale, the approach for large scales, namely here, object storage, is to:

- throw away the hierarchy: there are no directories;
- make the metadata flexible: attributes can differ from file to file (no schema);
- use a very simple, if not trivial, data model: a flat list of files (called objects) identified with an identifier (ID); blocks are not exposed to the user, i.e., an object is a blackbox;
- use a large number of cheap machines rather than some “supercomputer”;

3.4.3 Scale up vs. scale out

When a system is too slow on a single machine, there are several ways out.

First, one can buy a bigger machine: more memory, more or faster CPU cores, a larger disk, etc. This is called *scaling up*.

Second, one can buy more, similar machines and share the work across them. This is called *scaling out*.

Scaling up has its limits: while increasing memory from 8 GB to 16 or 32 GB works, 64 GB or 128 GB quickly becomes more expensive, 6 TB is probably beyond what a private individual or small company wants to have, and 1 PB of working memory would require billions if not trillions of dollars of investment to hire a team of dozens of researchers in a Research and Development unit to maybe achieve this in 20 years:

in other words the costs of scaling up grow up exponentially, if not hyperbolically (vertical asymptote), making it impractical.

Thus, scaling out is the better approach, and it is also the one taken in Big Data systems: this is why, in a cluster within a data center, there are thousands or even tens of thousands of servers, of all which have performance numbers similar, or slightly better, in order of magnitude, to your personal laptop.

There is also a third way: optimizing your code. In fact, you should always first try to improve your code before scaling out. The vast majority of data processing use cases fit on a single machine, and you can save a lot of money as well as get a faster system by squeezing the data on a machine (possibly compressed) and writing very efficient code.

Scaling out in the context of data processing is typically worth it in just two cases: either because the data does not fit on your laptop, e.b., you are looking at 50 TB of data, or because your laptop cannot read the data fast enough, i.e., your bottleneck is the disk I/O.

3.4.4 Data centers

So in a cluster, in a data center, there are thousands to tens of thousands of machines. Why not more? Well first, it is difficult to know because companies do not always fully communicate on their numbers. However, it is difficult to go beyond this because of pragmatic reasons having to do with power and cooling. A data center consumes as much electricity as an entire airport, and it becomes almost impossible to handle 100,000+ machines together. In fact, the trend at places like CERN is to have less servers, but more cores per server.

A server is also called a node in the context of a data center. Servers typically have between 1 and 64 cores and the number keeps going up.

The working memory available on a node ranges from 16 GB to 6 TB as of today.

The local storage available (SSD or HDD) ranges from 1 to 20 TB.

The network bandwidth goes from 1 to 100 Gb/s but higher speeds are also possible especially in the context of High-Performance Computing (HPC). Bandwidth is the highest within the same cluster and can be slower across clusters or data centers, even though there also exist high-speed connections across data centers.

Nodes are typically flat, rectangular boxes that are piled up in what is called a rack, which looks like a tower. A cluster is just a room filled with racks put next to each other.

Each module in a rack can be a server, or pure storage (many disks), or a network switch, etc. The height of a module is standardized and measured in so-called rack units (RU). Each module has typically between 1 and 4 RU.

Data centers are typically concentrated with a few players, called cloud providers. The big three are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. Most companies on the planet rent resources from these players, typically not directly physically but via the creation and “deletion” of virtual machines and/or of high-level services. Object storage is one of them.

3.5 Object stores

Now that we have the hardware in place and have out our disposal thousands of cheap commodity servers, we can come back to the problem at hand: how do we store plenty of data?

Most cloud providers offer a solution for cloud storage, which is typically called object storage. These solutions are typically as-a-service, meaning that users can directly go ahead and use them rather than have to install anything.

3.5.1 Amazon S3

Amazon’s object storage system is called Simple Storage Service, abbreviated S3. From a logical perspective, S3 is extremely simple: objects are organized in buckets. Buckets are identified with a bucket ID, and each object within a bucket is identified with an Object ID.

Thus, any object in S3, worldwide, is uniquely identified with a Bucket ID and an Object ID. “Object”, in fact, is only a synonym for “file” and can thus be a text file, a picture, a video, a dataset (CSV, JSON...) etc. But in the context of a object storage, we say “object” rather than “file”. The main reason is that files are organized in hierarchies, but object stores have no hierarchy: just a flat list of objects organized in buckets.

An object can be at most 5 TB (to remember this: it is typically something that fits on a single disk). However, it is only possible to upload an object in a single chunk if it is less than 5 GB – otherwise, the upload must be done in several blocks. We will cover blocks in the next chapter and here focus our attention on objects seen as black boxes.

By default, users can have up to 100 buckets but can ask for more on request.

3.5.2 Azure Blob Storage

Azure Blob Storage is another object storage technology in the Azure cloud.

It is similar in spirit to Amazon S3, in that it is a fully managed cloud storage service. However, it differs from S3 in several ways:

- Its architecture is publicly documented in scientific papers, making it easier to understand how it works.
- Objects are identified with three, rather than two IDs: An Account, a Container (called partition in scientific papers) and a Blob.
- Azure Blob Storage exposes more details to the user. In particular, it exposes that objects are divided in several blocks more prominently than Amazon S3 (in S3, only advanced users would actually know and access blocks within an object, while the front-end hides them).
- it differentiates between Block Blobs, Append Blobs (for logging) and Page Blobs (for storing and accessing the memory of virtual machines).
- the maximum sizes are different and go from 195 GB for an Append Blob to 190.7 TB for a Block Blob.

Block storage will be covered in more details in the next chapter on HDFS.

On the physical level, Azure Blob Storage is organized in so-called storage stamps located in various data centers worldwide. Each storage stamp consists of 10 to 20 racks, with each rack containing around 18 storage nodes (the disks + servers). In all, a storage stamp can store up to ca. 30 PB of data. However, a storage stamp will not be filled more than 80% of its total capacity in order to avoid being full: if a storage stamp reaches capacity, then some data is going to be reallocated to another storage stamp in the background. And if there are not enough storage stamps, well new racks will need to be purchased and installed at the locations that make the most sense.

3.6 Guarantees and service level

3.6.1 Service Level Agreements

Most cloud services come with a cost, but also a promise. This promise is a contract between the provider and the user called a Service Level Agreement, abbreviated as SLA.

For example, S3 promises that it loses, each year, less than one object in 100 billion. This is formulated as a durability of 99.99999999%.

Another promise is availability: S3 is available 99.99% of the time, meaning that it will be down less than 1 hour per year.

SLAs very often contain numbers with a lot of 9s. 99% of availability means at most 4 days a year of downtime, 99.9% less than 10 hours,

99.999% six minutes, 99.9999% 32 seconds and 99.99999% less than 4 seconds.

Response times are also expressed in terms of the 99.9% percentile, for example one can promise that 99.9% of the requests will be served in less than 300ms. However, S3 does not make promises in terms of latency: it varies depending from where you request an object, and usually objects are large enough so that the bottleneck is in the time to download the object over the network, not in the latency. Rather, the promise is made in terms of throughput, which in this context is the number of objects read or written per second: typically 5,500 reads/s and 3,500 writes/s.

3.6.2 The CAP theorem

Early relational database management system relied on ACID properties (see Chapter 2). In order to scale out, many distributed systems have to make a compromise on the transactional guarantees that they offer. This is best explained with the so-called CAP theorem.

The CAP theorem is basically an impossibility triangle: a system cannot guarantee at the same time:

- (atomic) Consistency: at any point in time, the same request to any server returns the same result, in other words, all nodes see the same data;
- Availability: the system is available for requests at all times (SLA with very high availability);
- Partition tolerance: the system continues to function even if the network linking its machines is occasionally partitioned.

The CAP theorem is, in fact, not formally proven and it would be more appropriate to call it a conjecture. An intuitive way of explaining it is as follows.

When an update is made via some server, this update must propagate to other nodes to the extent that a specific replication factor for the data is met.

If there is a partition of the network leading to two disconnected sub-networks and the propagation of some data did not have a chance to complete, then there are two possible design decisions:

1. either the system is temporarily put to a halt (synchronous propagation) and thus becomes unavailable until the network partition is resolved and the propagation can be completed. In this case the system is atomically consistent but not available (CP).

2. or the servers continue to serve requests, but the data served by the two disconnected sub-networks will be different. In this case the system is available but not atomically consistent (AP). Such systems propagate updates asynchronously and are also often called *eventually consistent* in the sense that, if one hypothetically would no longer receive updates, there exists a time at which the system will be consistent. In practice of course, as updates keep arriving, the system rarely becomes fully consistent – a common misconception about eventual consistency.

A third possibility is for the system not to be partition tolerant: in this case, the system is available and consistent (AC) but only for as long as no network partition occurs. If a network partition occurs, this is unknown territory and no further guarantees exist.

Beware of systems that claim to be available, atomically consistent and partition-tolerant at the same time: this is often just marketing. In practice, the companies who claim such properties (ab)use the fact that a partition of their network is very rare. But when a partition happens, they have in fact to make a choice between A or C, even though users will barely notice.

3.7 REST APIs

Data stores, be it object stores, key-value stores, document stores, wide column stores, distributed file systems, etc expose their functionality via an API. A vast majority of them offers a so-called REST API.

REST means REpresentational State Transfer. In fact, it is really just “HTTP done right” in the sense that it closely uses HTTP functionality: methods and resources.

The benefit of offering a REST API is tremendous: it makes an integration with any host language very easy, because HTTP clients exist in almost any host language (Java, Python, PHP, R...) and it is straightforward to implement a “wrapper API” in the host language that forwards all requests through the REST API to the data store. If you design a new system in the future, it is almost a no-brainer decision to support a REST API.

Other possibilities are a native driver, but this needs to be done for each host language. MongoDB is an example of document store that works with drivers.

Another protocol is the XML-based SOAP protocol.

The HTTP protocol, which REST builds on, was invented back in 1989, thirty years ago, by Sir Tim Berners-Lee, who is also the creator of the Web.

A client and server communicate with the HTTP protocol interact in terms of *methods* applied to *resources*.

A resource can be anything: a document, a PDF, a person, a calendar entry, or even a physical object such as a smart plug (“Web of things”). A resource is referred to with what is called a URI. URI stands for Uniform Resource Identifier¹. A URI looks like so:

`http://www.example.com/api/collection/foobar?id=foobar#head`
where

- “http” is the scheme;
- “//www.example.com“ is the authority.;
- “/api/collection/foobar” is the path;
- “?id=foobar” is the query;
- “#head” is the fragment.

There exist other schemes than http, for example mailto, ftp, hdfs, file, and so on. However, http is the most popular schemes even for resources that are “offline”, i.e., not actually reachable via HTTP.

A client can act on resources by invoking methods, with an optional body. The most important methods are:

- GET (without a body): this method returns a representation of the resource in some format (text, XML, JSON...). GET should have no side effects (beyond logging, of course).
- PUT: this method creates or updates a resource from a representation of a newer version of it, in some format (text, XML, JSON...). PUT has the side effect that a subsequent GET asking for the same format should return the same representation. PUT is idempotent, in that calling PUT with the same resource and body is identical to calling it just once.
- DELETE (without a body): this method deletes a resource. DELETE has the side effect that a subsequent GET asking for a representation of the resource should fail with a not-found (404) error.
- POST: this method is a blank-check, in that it acts on a resource in any way the data store seems fit; the behavior, of course, should be publicly documented. A typical use of POST is to create new resources but letting the REST server pick a resource URI for this new resource.

¹In fact, we should normally call them IRI (Internationalized Resource Identifier) for the most recent version, however most people continue to use the term URI. URIs used to be divided in URLs (locators) and URNs (names), however this distinction is now obsolete.

An example of URI for S3 is “`http://bucket.s3.amazonaws.com`” for a bucket and “`http://bucket.s3.amazonaws.com/object-name`” for an object.

3.8 Object stores in practice

Although object stores are based on a flat key-value model, most object storage services emulate a file hierarchy by allowing slash (/) characters in keys, and interpreting these slashes as virtual paths. As far as the storage layer is concerned, slashes are a character like any other. But on the logical level, a logical hierarchy enables more use cases for object stores.

3.8.1 Static website hosting

Object stores such as S3 or Azure Blob Storage can be used for static website hosting in a very straightforward way.

Static website hosting means that there is no dynamically generated content, as would be the case for example with a PHP or Java EE server (Tomcat, etc). In a static website, the HTML pages, pictures, CSS stylesheets, (client-side) JavaScript scripts, videos are delivered *as is*.

It is both very convenient and relatively cheap to drop HTML pages, pictures, CSS stylesheets and JavaScript scripts in an object storage service, and use this as a simple website backend. Most cloud providers provide a very simple way of accessing these objects via HTTP URLs.

It is common to place a content delivery network (CDN) service on top of the storage bucket of a website in order to accelerate and cache these files at multiple places on the planet. This is particularly useful if very high traffic is expected over short periods of time, for example for delivering a file that many people will rush to download at the same time. For the user, content delivered with a CDN feels instantaneous.

3.8.2 Dataset storage

Object stores are popular as data lakes, in the sense that datasets can be stored as objects in any desired format (CSV, JSON, ...).

As we will see later, a dataset often consists of several objects rather than just one. Hence, even though the size of an object is typically limited (e.g., 5 TB on S3), the size of a dataset is not. This way of splitting a dataset into multiple chunks is called sharding and will play a fundamental role throughout this course, as we will see when we study MapReduce and Spark.

3.9 Key-value stores

To what extent can we consider object stores such as Amazon S3 or Azure Blob Storage as a database system? While they are able to store large objects well into the TB range, which typically does not fit in a relational database system, their latency (100 milliseconds or more) is higher than typical database systems (between 1 and 9 milliseconds).

In fact, there exists a paradigm that has a similar key-value model (based on a get/put/delete API) with a low latency. The price to pay is that objects will be smaller (the order of magnitude being up to a few hundreds of kilobytes, 400 kB for Dynamo). Also, no metadata can be stored along objects, as is otherwise possible with S3 or Azure Blob Storage.

Key-value stores have known a moment of hype and glory for a while (the buzz word is “eventual consistency”), to the point that they were used beyond their design goals. One of the typical use cases for a key-value store is storing shopping carts for a very large online shop, another is for storing likes and comments on social media.

A key-value store differs from a typical relational database in three aspects:

- Its API is considerably simpler than that of a relational database (which comes with query languages)
- It does not ensure atomic consistency; instead, it guarantees eventual consistency, which we covered earlier in this Chapter.
- A key-value store scales out well, in that it is very fast also at large scales.

3.9.1 Key-value store API

A key-value store, on the high level, is based on the key-value model; it can be seen as some sort of a gigantic associative array (also logically known as “map” or “dict”) that is constantly modified and accessed.

A key-value store, on the high level typically has three access methods, which should not come as a surprise:

`get(key)` retrieves the value associated with a key.

`put(key, value)` associates a new value with the key.

`delete(key)` removes any association of a value with the key.

In practice, the API is slightly more complex because of conflict resolution. In the case of Dynamo, for example, `get(key)` not only returns value(s), but also a context indicating potential conflicts, which for now can be seen as a black box that we will get back to. Likewise, `put` has a third parameter, `context` (which is typically forwarded “as is” from a previous `get` call), and `put(key, context, value)` associates the key with a new value while resolving any conflict.

3.9.2 Design principles

We will focus on a specific key-value technology for our explanations: Amazon Dynamo. It is itself based (with some modifications) on the Chord protocol, which is a Distributed Hash Table.

Distributed Hash Tables are generally highly scalable, robust against failure and self organizing. However, they do not provide any support for range queries, guarantees on data integrity (which is pushed to the user), and do not deal with any security issues.

On the physical level, a distributed hash table is made of nodes (the machines we have in a data center, piled up in racks) that work following a few design principles. The first design principle is *incremental stability*. This means that new nodes can join the system at any time, and nodes can leave the system at any time, sometimes gracefully, sometimes in a sudden crash.

The second principle is symmetry: no node is particular in any way

The third principle is decentralization: there is no “central node” that orchestrates the others. One may wonder what the difference with symmetry is? A symmetric system could still have protocols that elect a leader than then takes up an orchestration role. But this is not the case in a distributed hash table.

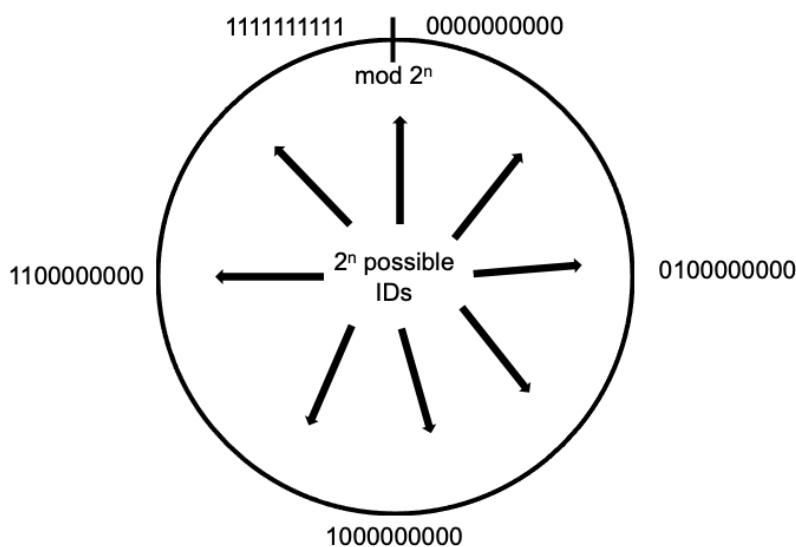
The fourth principle is heterogeneity: the nodes may have different CPU power, amounts of memory, etc.

3.9.3 A ring to rule them all

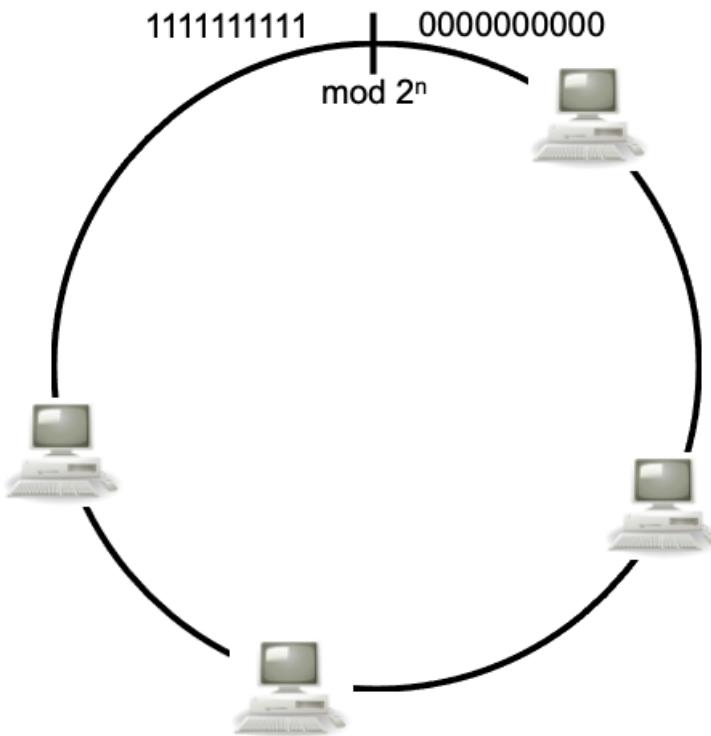
There are several ways that nodes can be connected together. Two main paradigms are peer-to-peer (every node speaks to every other node) and another one is centralized (there is a Coordinator Node that has a special role, and controls the Worker Nodes). In the case of distributed hash tables though, we have a peer to peer network, which is a natural thing to do since we require symmetry and decentralization.

A central aspect of the design of a distributed hash table, and part in particular of the Chord protocol, is that every logical key is hashed to bits that we will call IDs. In the case of Dynamo, the hash is made of 128 bits (7 bytes).

Mathematically, all possible IDs form, quite naturally, a ring structure (consider $\mathbb{Z}/2^{128}\mathbb{Z}$), although in this context we will not need any ring properties in a mathematical sense, but we only need to think of this ring as a circle.

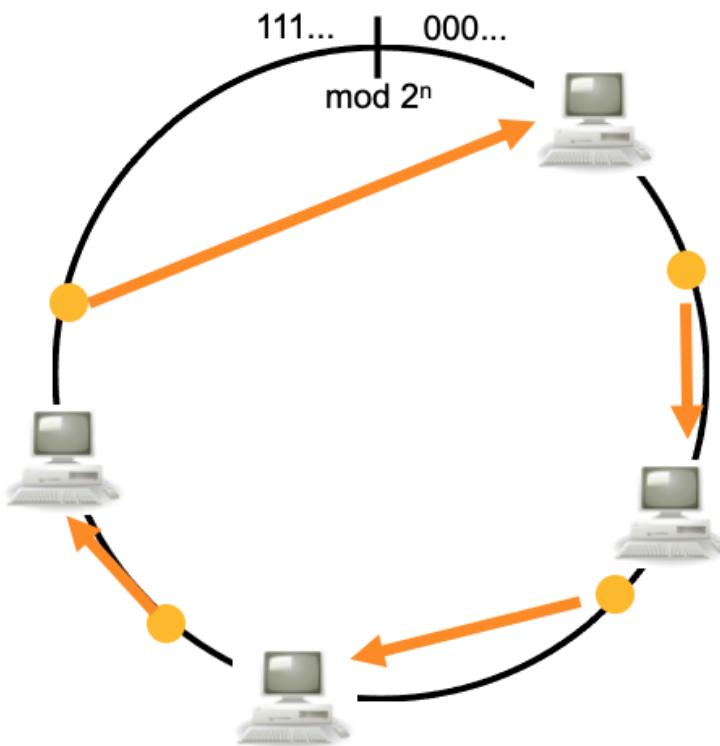


The next conceptual step of the design is that all nodes (remember, this is what we call the machines in the data center) get a random position on that ring, i.e., a random 128-bit number.

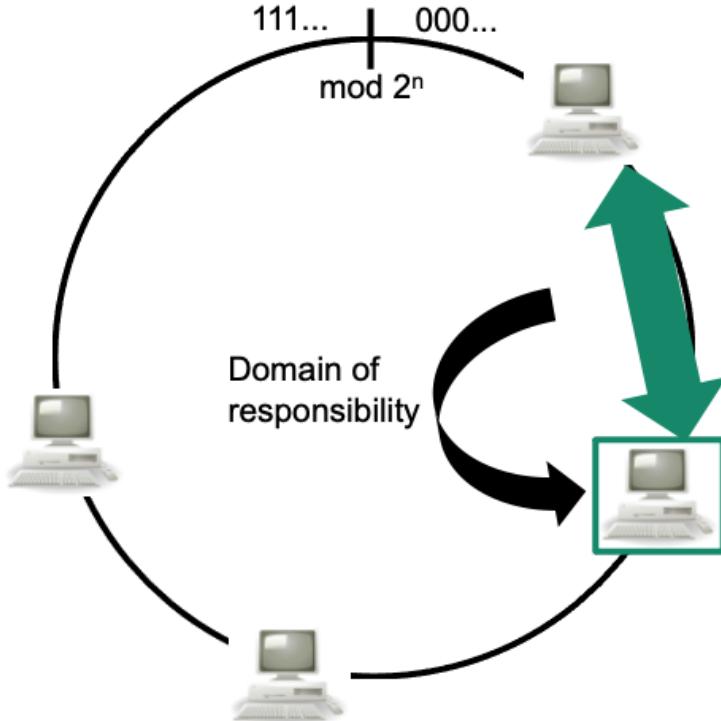


Which node stores what key is then decided based on the ring topology. Given a key k , its 128-bit hash is taken and it corresponds to a position on the ring. From that position on, one follows the ring clockwise, until one reaches one of the nodes (remember, they were logically assigned a position on the ring); this node is responsible for storing key k .

Here are a few examples:



The domain of responsibility of a single node n is thus the ring interval separating it from the next node counterclockwise: all keys whose hash lands in that interval are stored on, and the responsibility of, node n .



Now, what happens when nodes come and go? Well, when a new node n joins the cluster, it gets assigned a position on the ring. This position is in the domain of responsibility of some existing node m . This ring interval is now going to be split: everything between m and n remains within the responsibility of m , but the other half of the domain of responsibility is newly the responsibility of n . The corresponding data needs to be transferred from m to n .

What about when a node leaves? Let us start with the case of a graceful exit, i.e., the node (call it n) informs the rest of the cluster before leaving. Then, it will transfer all its interval of responsibility to the next node clockwise (call it m) on the ring: two intervals merge into a single one. Here too, data needs to be transferred from n to m .

Now, let us look into an abrupt exit: the node crashes with no opportunity to inform the cluster or prepare its exit.

And we have a problem. Because the data for that node is gone. And this problem is going to be solved with a fundamental principle

that will follow us through the course: data replication. We slightly change the design: data within an interval will not be stored only the node that follow the interval clockwise, but on the next N nodes that follow the interval clockwise. Equivalently explained, a node is responsible not only for the interval that follows it counterclockwise, but for the next N intervals that follow it counterclockwise.

3.9.4 Preference lists

The next step of the design is: given a key (and its 128-bit hash), how do we *physically* find out which node(s) hold the associated value? The explanations we gave so far are logical and based on the ring; but this needs to also work in practice, with the software installed on all nodes. And the challenge is that the system is decentralized, so there is nothing like a central place to look at. The client who wants to access a key needs to know which one of the (say) 1,000 machines it needs to connect to with the hope of getting an answer.

In the chord protocol, a technology called a *finger table* is used. Each node knows the next node clockwise, and the second node, and the 4th node, and the 8th node, etc: going through the powers of two. As you probably guessed, this means that a binary search is used to find the desired node within logarithmic time, hopping from node to node with increasing precision and until one finds The One(s).

Dynamo changes this design to so-called “preference lists”: each node knows, for every key (or key range), which node(s) are responsible (and hold a copy) of it. This is done by associating every key (key range) with a list of nodes, by decreasing priority (going down the ring clockwise). The details of this protocol are left out, but in many systems, sharing the same view between nodes is done either through some peer-to-peer synchronization protocol with all nodes keeping a local copy, or through access to a shared network drive or database (much smaller than the key-value store itself).

The size of every list must be at least N , where N is the parameter discussed earlier (the minimum number of nodes holding a copy of the value for each key). Note that it can get (temporarily) larger than N if there are network partitions or nodes leaving and taking over. The first node in the list (for each key range) is called the coordinator.

Two more parameters come into play: R and W , with

$$R + W > N$$

Upon reading a value, R is the minimum number of nodes (among the N nodes responsible for the key) from which a copy of the value must be obtained and compared. Upon writing a new value, W is the minimum number of nodes (among the N nodes responsible for the

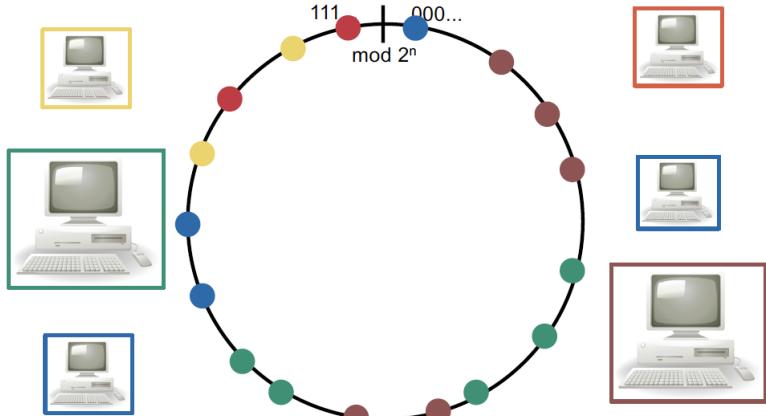
key) that must have acknowledged receipt and confirmed storage of the value before the put or delete transaction succeeds.

An initial request (put, delete or get) is typically dispatched, thanks to a load balancer, to a random node. This random node then looks at the key in its copy of the preference lists, and redirects the request to the coordinator node (first in the preference list). If this fails, an attempt is made with the next node, etc. Then, the coordinator contacts all other ($N-1$) nodes in the list, waiting for R (or W) distinct answers from the other nodes, and then the answer to the request is sent back to the client.

3.9.5 An improvement: tokens and virtual nodes

The design described so far has two issues: first, we might be out of luck with the (logical) distribution of the nodes on the ring, with some nodes responsible for large intervals and some others for smaller ones. Second, the nodes may not all have the same hardware resources: some may have more memory or less CPU, etc.

These two issues are addressed by virtualizing the nodes: instead of directly mapping each node to one position on the ring, a larger quantity of (virtual) nodes, called tokens, are positioned on the ring.



For example, there might be 100 nodes in the cluster, and we will position 1000 tokens on the ring. Then, the tokens are assigned to the nodes, in numbers that match their resources: a large node may get, say, 20 tokens, while a small one may get 3. Then the system works as above, each physical node handling the responsibilities assigned with all of its assigned tokens.

Adding or removing a node is straightforward: all tokens corresponding to a node leaving the network are redistributed to other nodes (not necessarily the same one), and a node joining the network gets as-

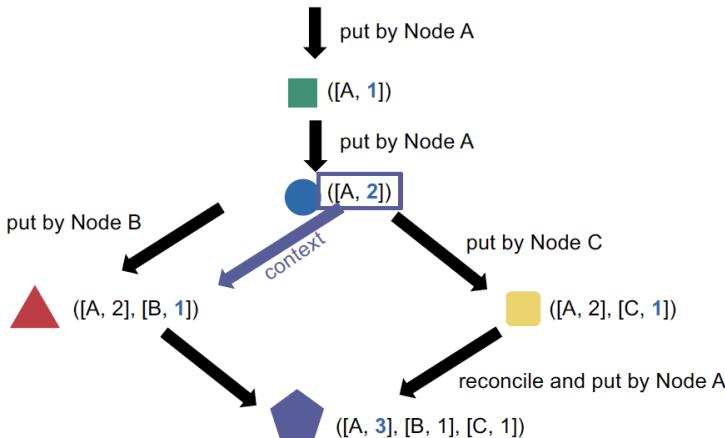
signed tokens taken over from other (not necessarily the same one) nodes.

3.9.6 Partition tolerance and vector clocks

We now analyze how a distributed hash table behaves in the presence of network partitions. As we saw previously, when there is a network partition a system will either crash (CA), become temporarily unavailable (CP), or accept inconsistencies, downgrading the guarantee to eventual consistency (AP). Distributed hash tables, including Dynamo, are typically AP.

A fundamental conceptual tool in AP systems is the use of vector clocks. In a system with no partitions, versions of the data can be seen as linear, and could be identified with integers (in fact, some code repository systems do exactly this).

But in case there are network partitions, the versions may fork, forming a DAG (Directed Acyclic Graph) structure rather than a line. Vector clocks are a way to annotate the versions when they follow a DAG structure.



A vector clock can logically be seen as a map from nodes (machines) to integers, i.e., the version number is incremented per machine rather than globally. A version number increases for a machine when this machine processes the corresponding entry (key-value) and updates it. For machines that have not yet touched a given entry, the corresponding number (implicitly 0) can be left out from the vector clock: it gets added the first time the machine updates the entry.

On the picture above, node A creates the entry and associates it with a vector clock

```
{ "A" : 1 }
```

Then node A (which is probably at the top of the preference list) updates the entry again and it increases:

```
{ "A" : 2 }
```

Then for some reason, the entry was processed, separately and without knowing about each other (network partition!) by nodes B and C. Node B updates the value and associates it with vector clock

```
{ "A" : 2, "B" : 1 }
```

while node C, who does not know about node B, does the same and associates it with vector clock

```
{ "A" : 2, "C" : 1 }
```

We now have two conflicting versions, which will be returned to the user upon getting the value for that key (assuming it sees both node B and C, i.e., the network partition is over), together with the context, which "merges" the conflicting vector clocks with the max integers:

```
{ "A" : 2, "B" : 1, "C" : 1 }
```

This context is passed (via `put()`) to node A for another update to resolve the conflict, leading to the new vector clock:

```
{ "A" : 3, "B" : 1, "C" : 1 }
```

Note that vector clocks can be compared to each other with a partial order relation \leq . A partial order is a relation that is antisymmetric (if the order holds in both directions, then the vector clocks must be equal), reflexive (the order always holds if both sides are the same vector clock), and transitive (if the order holds between vector clocks A and B as well as B and C, then it must hold between vector clocks A and C), but it might not be total in the sense that vector clocks might not be comparable (the order is undefined).

More specifically, a vector clock is smaller or equal to another if for each machine, the two associated integers are also smaller or equal (in the same direction). For example.

$$\{ "A" : 1 \} \leq \{ "A" : 1 \}$$

$$\{ "A" : 1 \} \leq \{ "A" : 2 \}$$

$$\{ \text{"A" : 1} \} \leq \{ \text{"A" : 2, "B" : 1} \}$$

$$\{ \text{"A" : 1} \} \leq \{ \text{"A" : 2, "C" : 1} \}$$

$$\{ \text{"A" : 1} \} \leq \{ \text{"A" : 3, "B" : 1, "C" : 1} \}$$

$$\{ \text{"A" : 2} \} \leq \{ \text{"A" : 2} \}$$

$$\{ \text{"A" : 2} \} \leq \{ \text{"A" : 2, "B" : 1} \}$$

$$\{ \text{"A" : 2} \} \leq \{ \text{"A" : 2, "C" : 1} \}$$

$$\{ \text{"A" : 2} \} \leq \{ \text{"A" : 3, "B" : 1, "C" : 1} \}$$

$$\{ \text{"A" : 2, "B" : 1} \} \leq \{ \text{"A" : 3, "B" : 1, "C" : 1} \}$$

$$\{ \text{"A" : 2, "C" : 1} \} \leq \{ \text{"A" : 3, "B" : 1, "C" : 1} \}$$

A partial order, unlike a total order, may not have a *maximum* (greater than all others), but it may have several *maximal elements* instead (not smaller than any other). When there is a unique maximal element (which is then the maximum because there is a finite number of vector clocks), there is no conflict and only one value is associated with the key throughout the DHT; and conflicts can be recognized because there are several maximal elements (typically one for each former partition), at least until the conflict is resolved.

3.10 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you describe the limitations of traditional (local) file systems?
- b. Can you explain what object storage is?
- c. Can you explain what the benefits of object storage are?
- d. Can you contrast or relate object storage with block storage, a file system, and the key-value model?
- e. Can you explain the three different ways, on the physical level, to deal with more data (scale up, scale out, write better code)?
- f. Can you explain why scaling out is less expensive than scaling up?
- g. Can you explain what aspects of the design of object storage enable scaling out and why?
- h. Do you know what a data centre is made of (racks, server nodes, storage nodes, etc.)?
- i. Do you know the difference between storage, memory, CPU and network and how the three are paramount in a cluster?
- j. Do you know rough, typical numbers (per-node storage capacity, memory, number of cores, etc.)?
- k. Do you know how storage and memory technologies (HDD, SSD and RAM) compare in terms of capacity and throughput?
- l. Can you sketch the (key-value) data model behind object storage? Can you spot instances of the key-value model in several other technologies seen in the lecture?
- m. Do you know the difference between data and metadata?
- n. Do you know the order of magnitude that can be achieved in terms of number of objects, and object size?
- o. Can you name a few big players for cloud-based object storage (vendors, consumers)?
- p. Can you describe the features of S3 and Azure Blob Storage on a high level? Do you know what a bucket and object are? What block blob storage, append blob storage and page blob storage are and how they work?

- q. Can you describe what the most important SLA (Service Level Agreement) parameters mean (e.g., latency, availability, durability) as well as their typical range?
- r. Can you describe a typical use case for object storage?
- s. Can you explain what a REST API is, what resources and methods are? Do you know the most important status codes (200, 404, 500, ...)? Do you know the most important methods? Do you know what a URI is? (Note that the difference between URI, IRI, URN, URL has become irrelevant lately and for the purpose of the course)
- t. Do you know what each letter stands for in CAP?
- u. Can you explain why, for large amounts of data, CAP becomes relevant over ACID?

3.11 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Calder, B, et al. (2011). *Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency*. SOSP '11: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.

Fielding, R.T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Chapter 5.

Chapter 4

Distributed file systems

There is Big Data and Big Data.

In the previous chapter, we saw that cloud storage services provide a global service that is able to store billions, trillions of objects. We also saw that the size of these objects is limited, for example in Amazon S3, an object cannot exceed 5 TB. In other words, they can store huge amounts of large files.

What about large amounts of huge files? Is it possible to store individual objects bigger than a single machine? The answer is yes. It requires a shift in paradigm that:

- brings back the file hierarchy natively;
- supports block-based storage natively.

4.1 Main requirements of a distributed file system

Data goes through a lifecycle: there is the raw data that is directly collected from the real world (e.g., sensors, logs, etc). Later in the pipeline, this data gets processed, analyzed and turned into derived datasets. These datasets need to be written back to a large-scale storage backend. In turn, they might be processed again leading to more derived datasets, etc.

Such a storage backend must first be resilient to failure. While a single hard drive might occasionally fail, as some of us probably already experienced with their laptop, the nodes in a cluster with 1,000 machines and more is *guaranteed* to fail. Not once, not twice, all the time. Thus, the storage technology must be capable of

- monitoring itself;
- detecting failures;

- automatically recovering;
- being, in the end and as a whole, fault tolerant.

The access mode is also an important feature of the storage backend. On a laptop hard drive, random access is paramount: any part of the disk can be read and written at any time, and in any order. Of course, accessing data randomly and back and forth will be slower than reading a file stored contiguously on disk, or writing a file as a stream and in one go. But laptop hard drives are *designed* to be practically usable for random access.

For distributed data storage though, and for the use case at hand where we read a large dataset, analyze it, and write back the output as a new dataset, random access is not needed. Rather, we need to be able to:

- efficiently scan a large file (in its entirety) – for data analysis
- append efficiently new data at the end of an existing large file – particularly for logging and sensors

Furthermore, this must be supported even with hundreds of concurrent users reading and writing to and from the same file system.

Going back to our capacity-throughput-latency view of storage, a distributed file system is designed so that, in cruise mode, its bottleneck will be the data flow (throughput), not the latency. This aspect of the design is directly consistent with a full-scan pattern, rather than with a random access pattern, the latter being strongly latency-bound (in the sense that most of the time is spent waiting for data to arrive).

We saw that capacity increased much faster than throughput, and that this can be solved with parallelism.

We saw that throughput increased much faster than latency decreased, and that this can be solved with batch processing.

Distributed file systems support both parallelism and batch processing natively, forming the core part of the ideal storage system accessed by MapReduce or Apache Spark.

The origins of such a system come back to the design of GoogleFS, the Google File System. Later on, an open source version of it was released as part of the Hadoop project, initiated by Doug Cutting at Yahoo, and called HDFS, for Hadoop Distributed File System. The Hadoop projects further includes other open source releases of MapReduce (with the same name as Google's original MapReduce) and HBase (corresponding to Google's BigTable).

Between 2006 and 2016, in just ten years, the cluster size running HDFS successfully went from 188 nodes to almost 42,000 nodes (and by now probably more), storing hundreds of Petabytes of data.

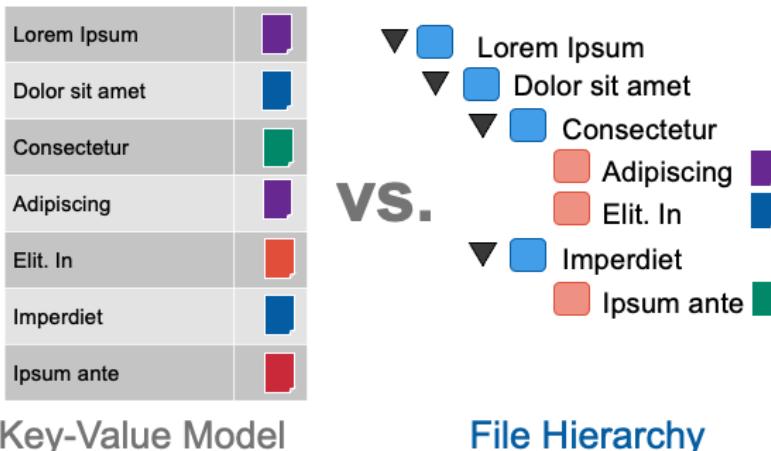
4.2 The model behind HDFS

In almost every chapter of this course, we will carefully introduce the logical level separately from its physical implementation. Thus, we start by describing HDFS on the logical level.

4.2.1 File hierarchy

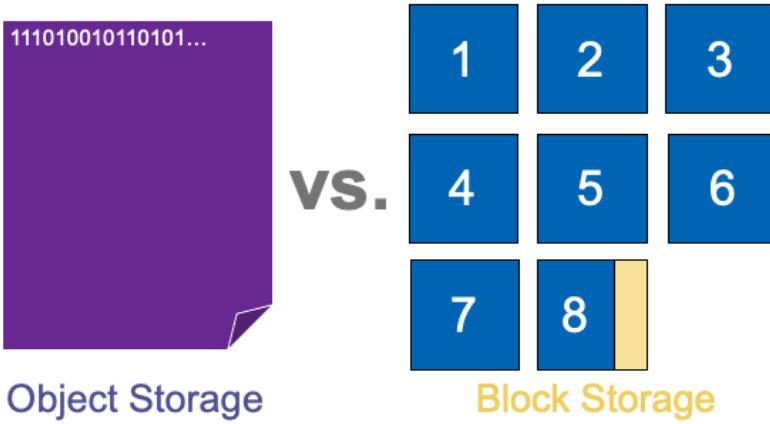
First, in HDFS, the word “file” is used instead of “object”, although they correspond to S3 objects logically.

HDFS does not follow a key-value model: instead, an HDFS cluster organizes its files as a hierarchy, called the file namespace. Files are thus organized in directories, similar to a local file system.



4.2.2 Blocks

Unlike in S3, HDFS files are furthermore not stored as monolithic black-boxes, but HDFS exposes them as lists of blocks – also similar to a local file system.

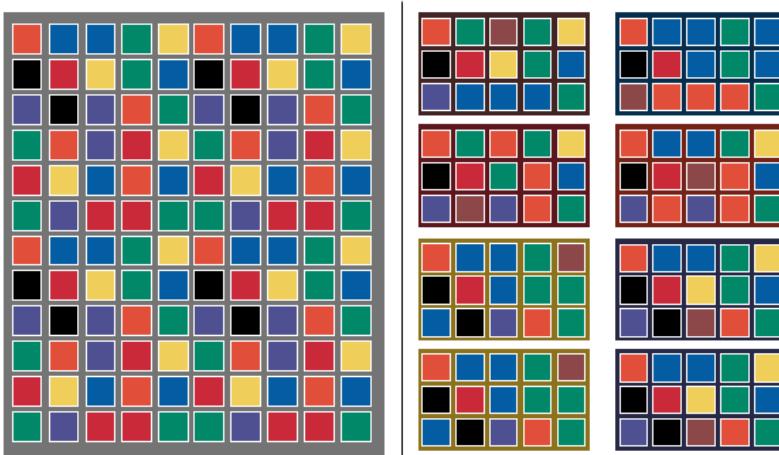


In Google's original file system, GFS, blocks are called chunks. In fact, throughout the course, many words will be used, in each technology, to describe partitioning data in this way: blocks, chunks, splits, shards, partitions, and so on. It is more important to understand that they are almost interchangeable, while it is secondary to learn "by heart" which technology uses which word.

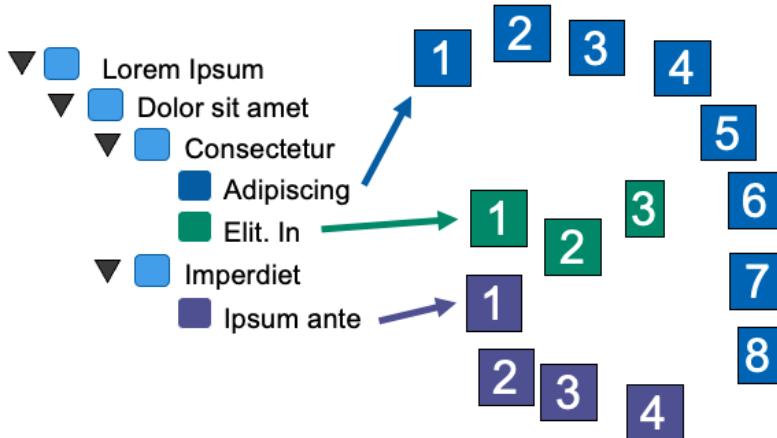
Why does HDFS use blocks?

First, because a PB-sized file surely does not fit on a single machine as of 2022: files have to be split in some way. Second, it is a level of abstraction simple enough that it can be exposed on the logical level.

Third, blocks can easily be spread at will across many machines, a bit like a large jar of cookies can easily be split and shared among several people.



Putting it all together, the following picture summarizes best the logical model of an HDFS cluster:



4.2.3 The size of the blocks

As for the block size: HDFS blocks are typically 64 MB or 128 MB large, and are thus considerably larger than blocks on a local hard drive (around 4 kB). This is because of the prerequisites of HDFS: first, it is not optimized for random access, and second, blocks will be shipped over the network.

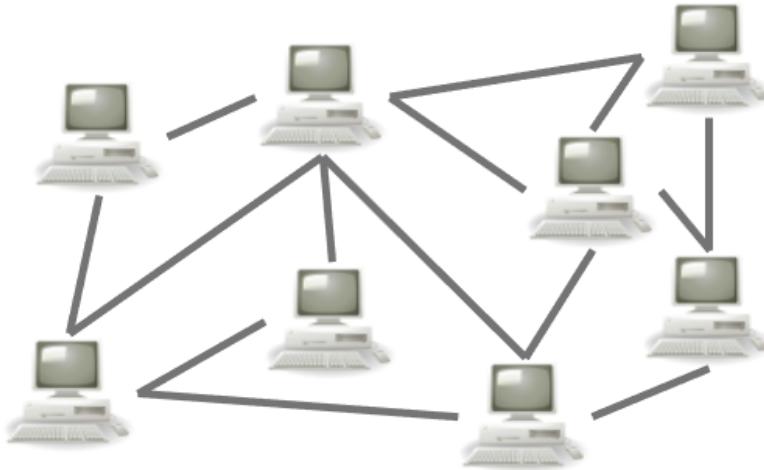
Thus, blocks of 128 MB are large enough that time is not lost in latency waiting for a block to arrive, i.e., access to the HDFS cluster will be throughput-bound during a full-scan analysis with MapReduce or Spark. And at the same time, they are small enough for a large file to be conveniently spread over many machines, allowing for parallel access, and also small enough for a block to be sent again without too much overhead in case of a network issue.

4.3 Physical architecture

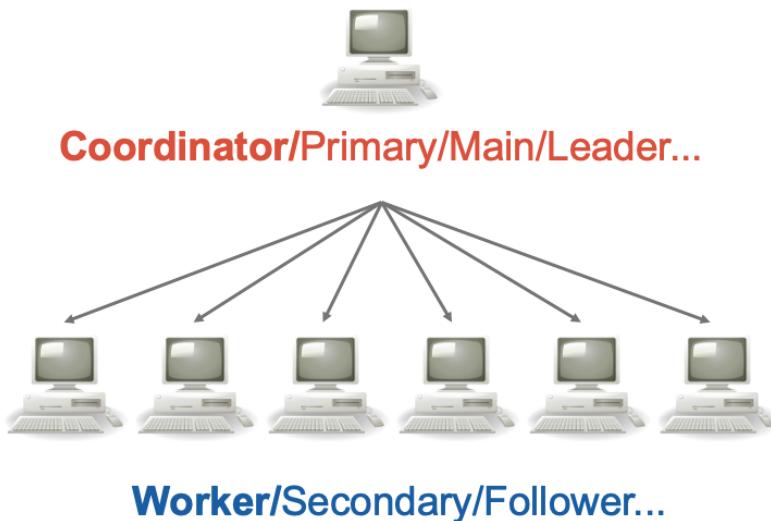
HDFS is designed to run on a cluster of machines. The number of machines can range from just a handful of them to thousands of machines. How are they connected?

One way to connect machines is called a peer-to-peer, decentralized network. In a peer-to-peer network, each machine talks to any

other machine. This architecture, for example, is used in the Bitcoin blockchain and was also popular in the 1990s with the Napster network.



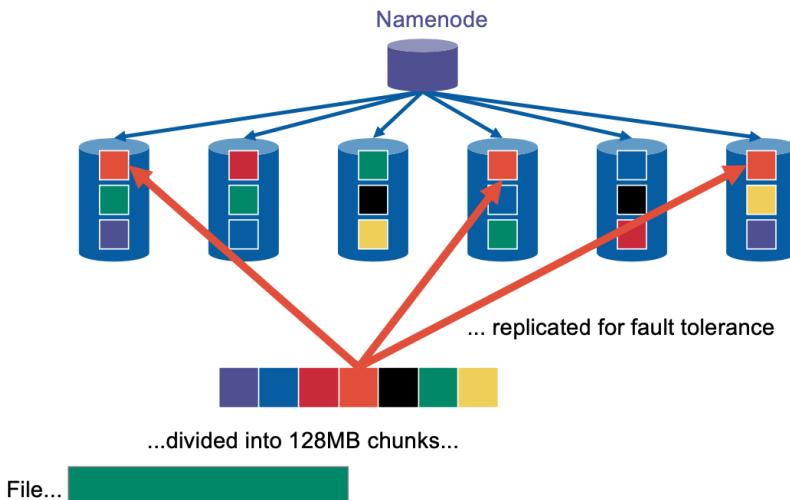
By contrast, HDFS is on the contrary implemented on a fully centralized architecture, in which one node is special and all others are interchangeable and connected to it.



The terminology used to describe the nodes in a centralized network architecture varies in the literature. In the case of HDFS, the central node is called the NameNode and the other nodes are called the DataNodes. In fact, more precisely, the NameNode and DataNodes are processes running on these nodes, and the CamelCase notation often used to write their names down corresponds to the Java classes implementing these processes. By metonymy, we also use these names to describes the nodes themselves.

Now that we have the NameNode and DataNodes in place, how is the logical HDFS model implemented on top of this architecture?

As we saw before, every file is divided into chunks called blocks. All blocks have a size of exactly 128 MB, except the last one which is usually smaller (indeed, what are the odds that the file size would be exactly a multiple of 128 MB?).

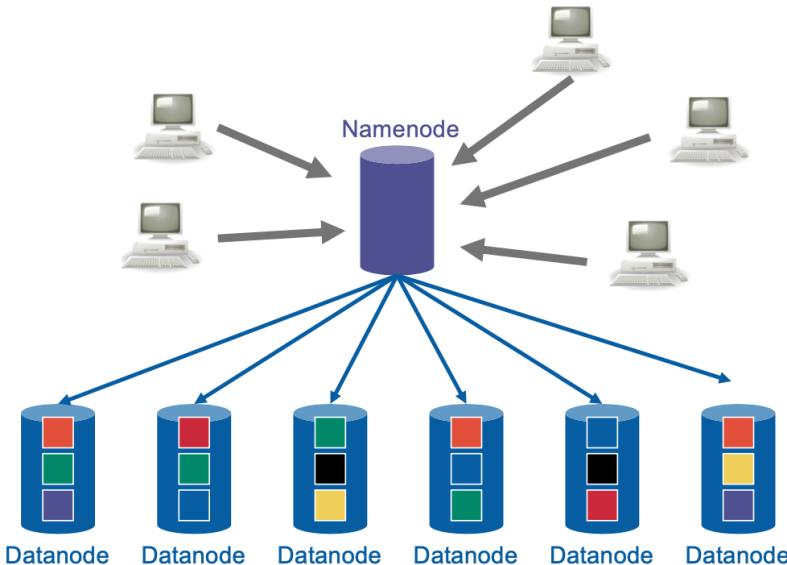


Each one of the blocks is then replicated and stored on several DataNodes. How many times? This is a parameter called the *replication factor*. By default, it is 3, but this can be changed by the user.

There is no such thing as a primary replica: all replicas are on an equal footing. In other words, by default, three replicas of each block are stored by default. Overall, in a production instance, there is a very high number of block replicas and they are spread more or less evenly across the nodes, to have a certain balance.

An HDFS cluster is then accessed concurrently by many clients. Typically, the cluster is owned by one company or organization and centrally managed by a specialized team (e.g., site reliability engineers).

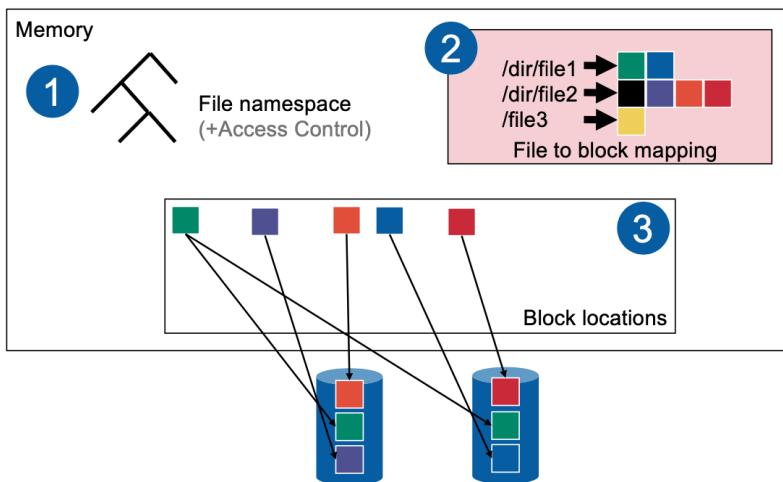
The clients correspond to users within this organization who read from and write to the HDFS cluster.



4.3.1 The responsibilities of the NameNode

The NameNode is responsible for the system-wide activity of the HDFS cluster. It stores in particular three things:

- the file namespace, that is, the hierarchy of directory names and file names, as well as any access control (ACL) information similar to Unix-based systems.
- a mapping from each file to the list of its blocks. Each block, in this list, is represented with a 64-bit identifier; the content of the blocks is not on the NameNode.
- a mapping from each block, represented with its 64-bit identifier, to the locations of its replicas, that is, the list of the DataNodes that store a copy of this block.



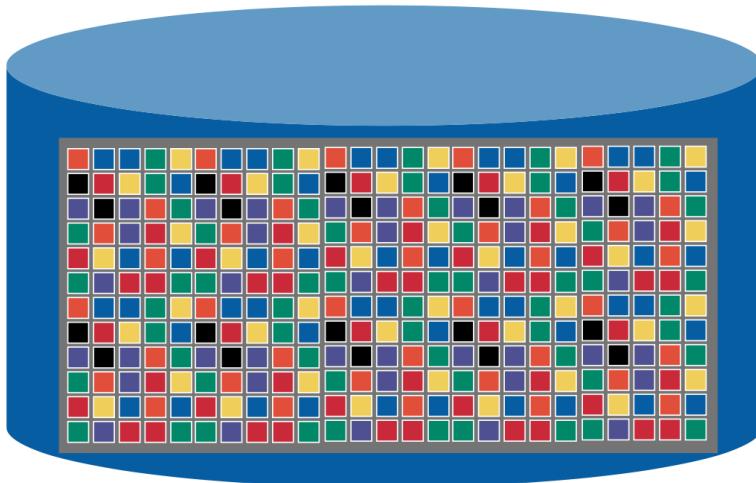
The NameNode updates this information whenever a client connects to it in order to update files and directories, as well as with the regular reports it receives from the DataNodes.

Clients connect to the NameNode via the Client Protocol. Clients can perform metadata operations such as creating or deleting a directory, but also ask to delete a file, read a file or write a new file. In the latter case, the NameNode will send back to the client block identifiers (for reading them), or lists of DataNode locations (for reading and writing them).

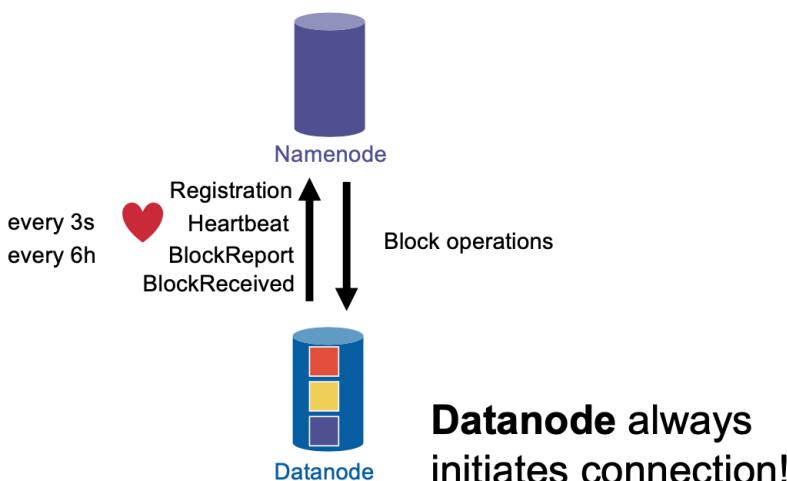
4.3.2 The responsibilities of the DataNode

The DataNodes store the blocks themselves. These blocks are stored on their local disks¹. Each block is stored as a 128 MB local, physical file on the DataNode. If the block is the last block of an HDFS file and thus less than 128 MB, then the physical file has exactly the size of the block: there is no waste of space. This is different from the physical blocks (4 kB or so) of a local hard drive, which have a constant size.

¹In a cloud installation, these local disks can also be virtually attached, but this is the same from the perspective of the DataNode process.



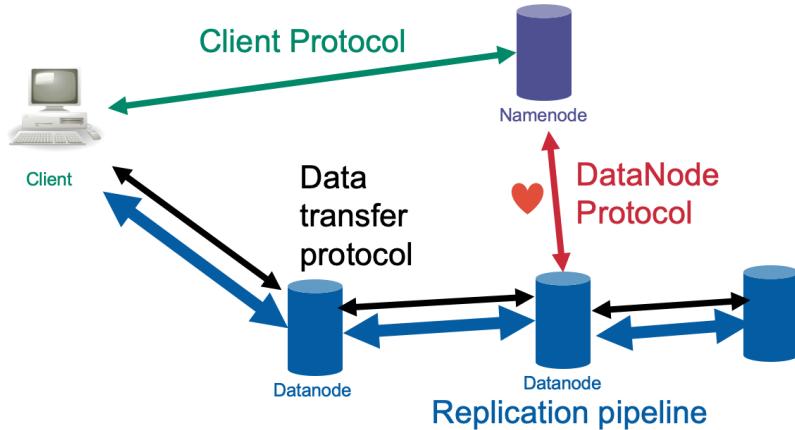
DataNodes send regular heartbeats to the NameNode. The frequency of these heartbeats is configurable and is by default a few seconds (e.g., 3s, but this value may change across releases). This is a way to let the NameNode know that everything is alright. The heartbeat may also contain a notification to the NameNode that a new block was received and successfully stored, and that the DataNode can thus be used as a location for this block. Finally, the DataNode also sends, every couple of hours (e.g., 6h, but this value may change across releases), a full report including all the blocks that it contains.



If there is an issue with the local disk or the node, then the DataNode can report the block as corrupted to the NameNode, which will then ensure, asynchronously, its replication to somewhere else. “Asynchronously,” as opposed to “synchronously,” means that this is done in the background at some later time and the DataNode does not idly wait for this to happen. In the meantime, the block is simply marked as underreplicated.

A NameNode never initiates a connection to a DataNode. If the NameNode needs anything from a DataNode, for example, if it needs to request a DataNode to download an underreplicated block from another DataNode and store a new replica of it, then the NameNode will wait until the next heartbeat, and answer to it with this request.

Finally, DataNodes are also capable of communicating with each other by forming replication pipelines. A pipeline happens whenever a new HDFS file is created. The client does not send a copy of the block to all the destination DataNodes, but only to the first one. This first DataNode is then responsible for creating the pipeline and propagating the block to its counterparts.



When a replication pipeline is ongoing and a new block is being written to the cluster, the content of the block is not sent in one single 128 MB packet. Rather, it is sent in smaller packets (e.g., 64 kB) in a streaming fashion via a network protocol. That way, if some packets are missing, the client can send them again. The client receives the acknowledgements in a streaming fashion as well.

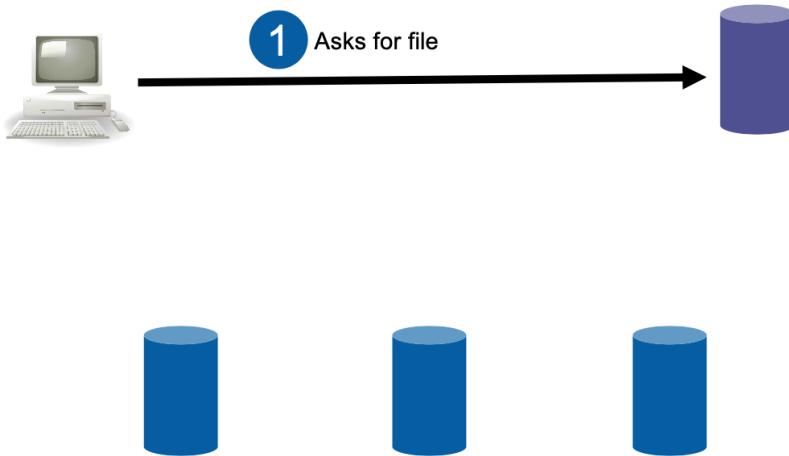
4.3.3 File system functionality

HDFS exposes to the client, via the NameNode, an API that allows typical operations available on a file system: creating a directory, deleting

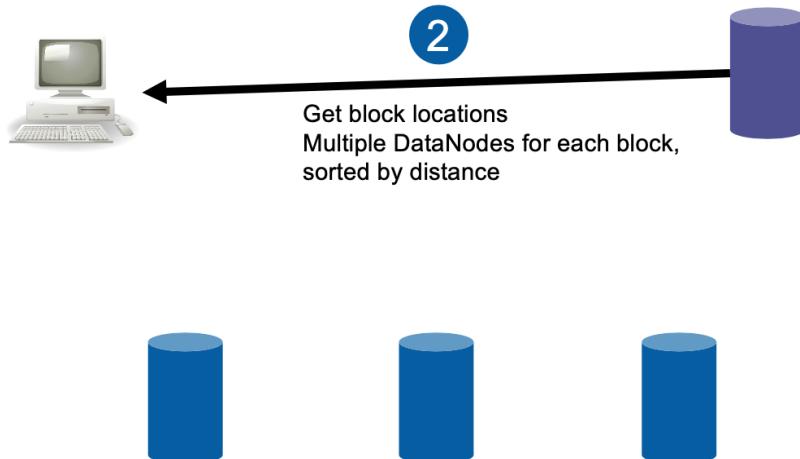
a directory, listing the contents of a directory, creating a file, appending to a file, reading a file, deleting a file.

Many of these involve the NameNode only. However, reading, writing and deleting a file involves communication across the cluster, which we now detail.

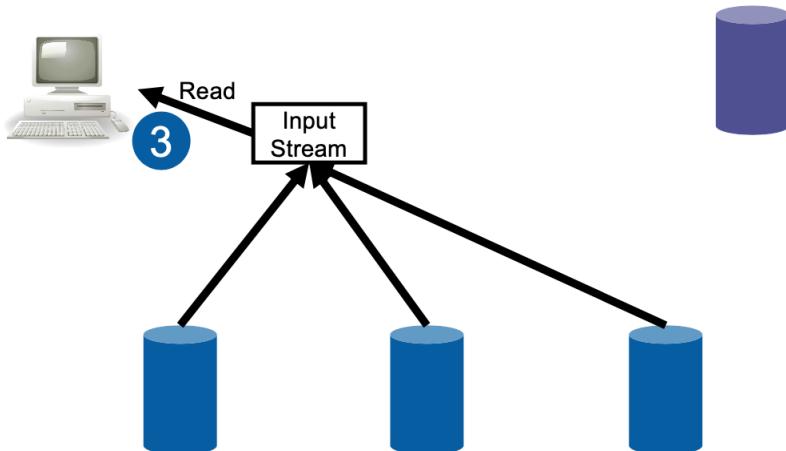
Let us start with reading a file. The client first connects to the NameNode to initiate the read and request info on the file.



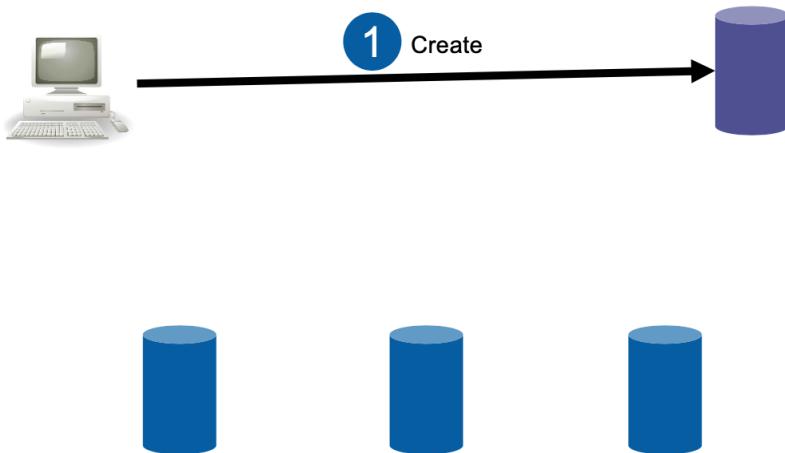
Then, the NameNode responds with the list of all blocks, as well as a list of DataNodes that contains a replica of each block. The DataNodes are furthermore sorted by increasing distance of the client (which is typically itself one of the nodes in the cluster, we will come back to this).



The client then connects to the DataNodes in order to download a copy of the blocks. It starts with the first (closest) DataNode in the list provided by the NameNode for each blocks, and will go down the list if a DataNode cannot be reached or is not available. In the simple case that the client wants to stream its way through an HDFS file, bit by bit, it will download each block in turn. However, this functionality is typically nicely encapsulated in additional, client-side Java libraries that expose this as a "single big input stream" with the `InputStream` classes familiar to Java programmers. Switching between the DataNodes is hidden and the user just sees a stream of bits flowing through. Note that with streaming, it is possible to process files larger than the working memory, because older blocks can be thrown away from the memory of the client once processed. This is something we will come back to when we look into the architecture of querying engines. Alternatively, the client can also download a multi-block HDFS file and store it as a single big file on its local drive as long as it fits.

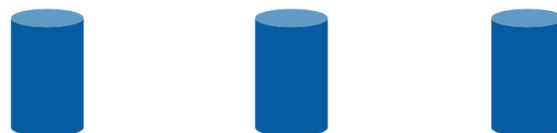


Let us now write a new file to HDFS. This can be either a simple upload of a large local file, or it can be from a stream of bits created on the fly by a program. As for reading, the client first connects to the NameNode formulating its intent to create a new file.

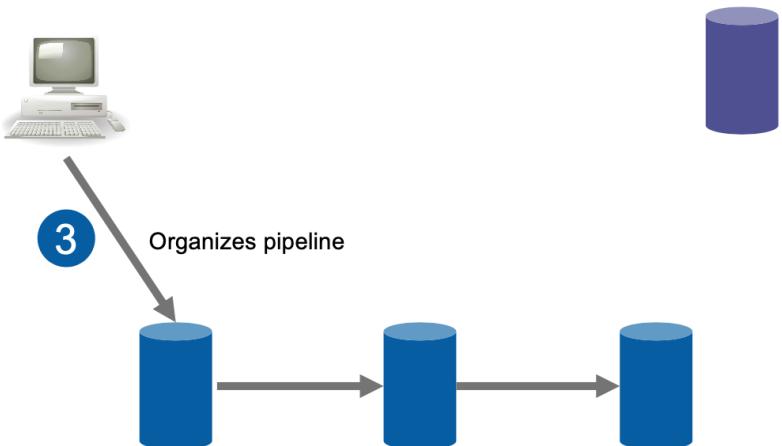


The NameNode then responds with a list of DataNodes to which the content of the first block should be sent. Note that, at that point,

the file is not yet guaranteed to be available for read for other clients², and it is locked in such a way that nobody else can write to it at the same time.

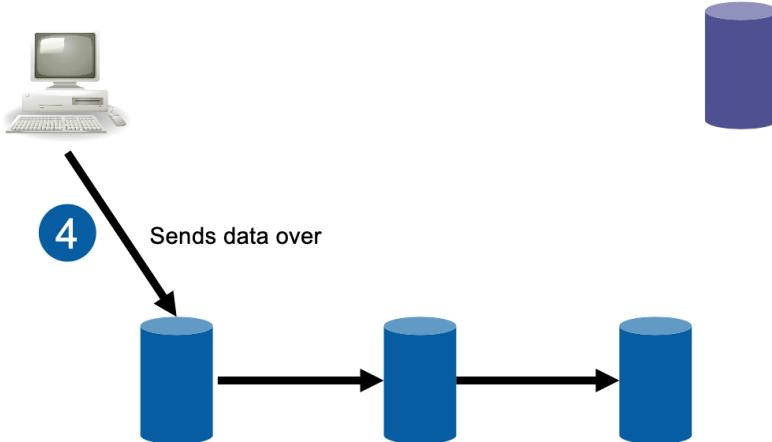


The client then connects to the first DataNode and instructs it to organize a pipeline with the other DataNodes provided by the NameNode for this block.

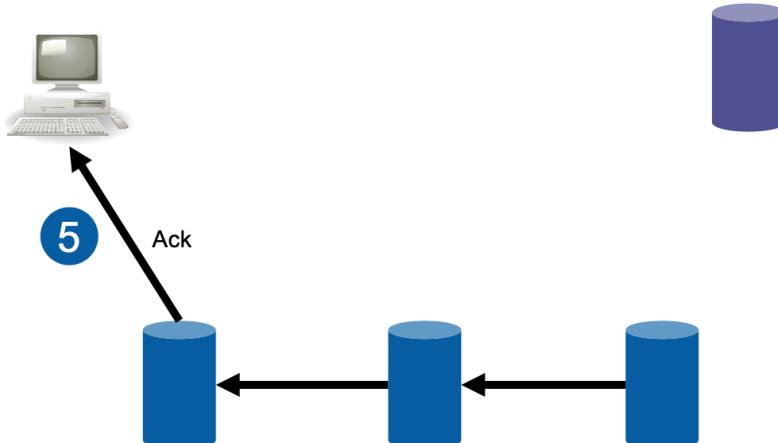


²Although it is possible that other clients *might* see the first blocks of a file being written before the write is completed.

The client then starts sending through the content of that block, as we explained earlier. The content will be pipelined all the way to the other DataNodes.



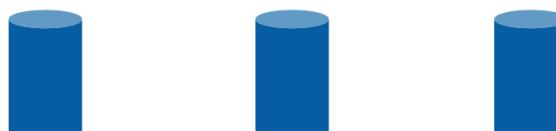
The client receives regular acknowledgements from the first DataNode that the bits have been received.



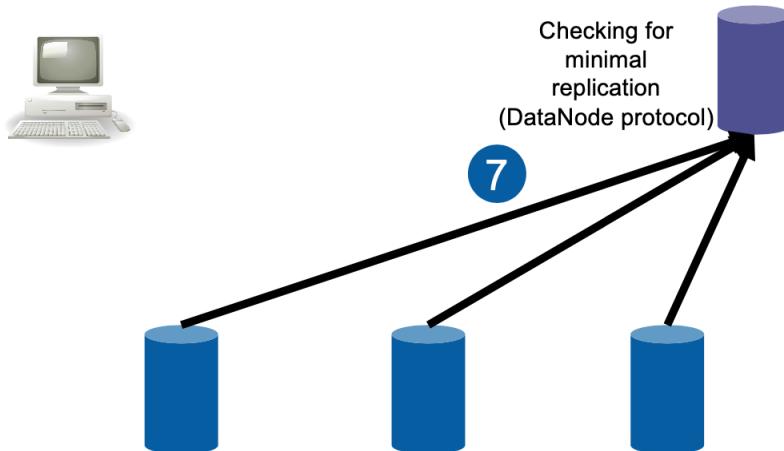
When all bits have been acknowledged, the client connects to the NameNode in order to move over to the second block. Then, the same steps (2, 3, 4, 5) as before are repeated for each block.



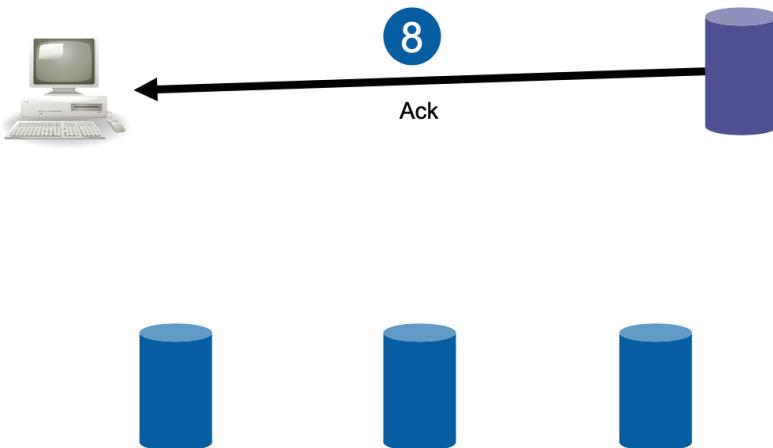
Once the last block has been written, the client informs the NameNode that the file is complete, and asks to release the lock.



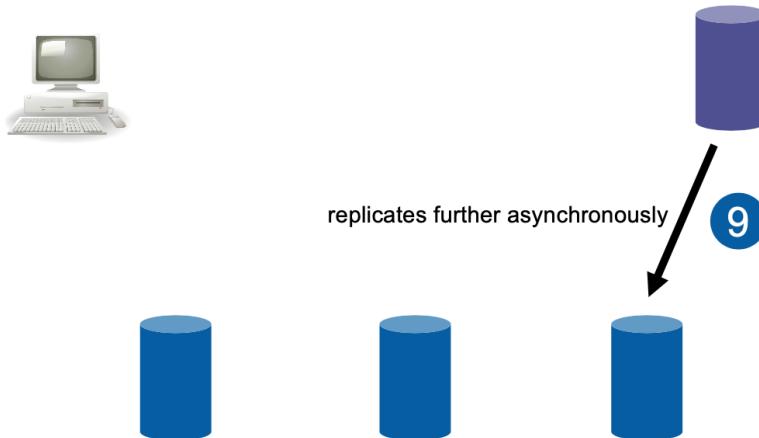
The NameNode then checks for minimal replication through the DataNode protocol.



... and gives its final acknowledgement to the client.



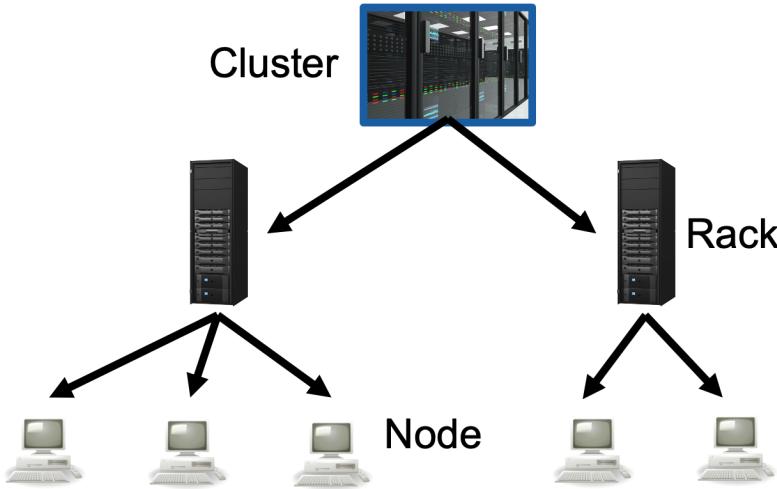
From now on, and separately (this is called “asynchronous”), the NameNode will continue to monitor the replicas and trigger copies from DataNode to DataNode whenever necessary, that is, when a block is underreplicated.



4.4 Replication strategy

By default, three replicas of each block are stored, although this can be changed by users for every file. Note that there is no such thing as a “main replica”, there are just several replicas.

The placement strategy of replicas is based on knowledge of the physical setup of the cluster. As you may recall from the previous chapter, servers, called nodes, are organized in racks, and several racks are placed in the same room or data center. This gives a natural tree topology.



With this topology in mind, one can define a notion of distance between two nodes: two nodes in the same rack have a distance of 2 (one edge from the first node to the rack, and one from the rack to the other node). Two nodes in different racks have a distance of 4 (one edge from the first node to the first rack, one from the first rack to the data center, one from the data center to the other rack, one from the other rack to the other node). It is this distance that can then be used by the NameNode to sort DataNodes by increasing distance from the client.

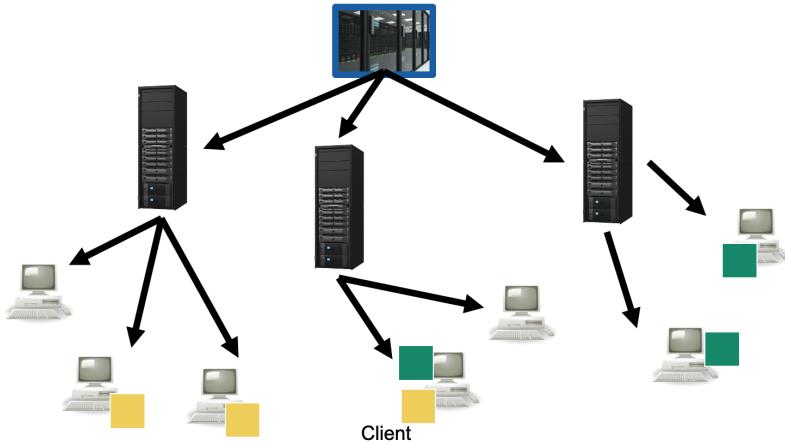
Now, the strategy for placing replicas of a new blocks is as follows. First, it is important to understand that, in practice, the client that is writing the file is a process running on one of the nodes of the HDFS cluster. This will become obvious when we study massive parallel computing (e.g., MapReduce and Spark), where reading and writing is done in a distributed fashion over the same cluster. But even without parallel computing, when users create clusters in public clouds such as AWS or Azure or Google Cloud, they connect to the machines using SSH (a safe protocol for remotely accessing a machine). Thus, anything they will do will originate from the machine they are connected to, which is in the cluster.

Having this in mind, the first replica of the block, by default, gets written to the same machine that the client is running on – keep in mind that this machine is typically a DataNode (again, this will become obvious when we look into MapReduce and Spark).

The second replica is written on a DataNode sitting in a different rack than the client, that we call B. The third replica is written to another DataNode on the same rack B. And further replicas are written

mostly at random, but respecting two simple rules for resilience: at most one replica per node, and at most two replicas per rack.

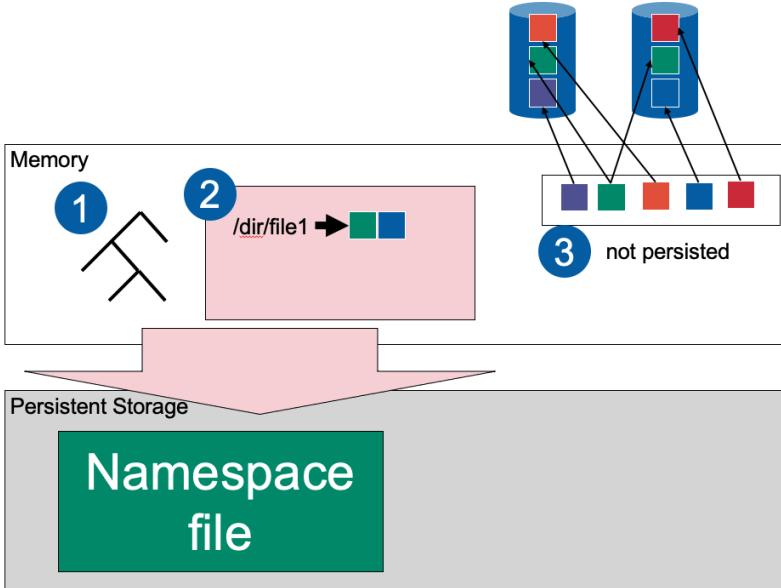
Below is an example with the three replicas of two different blocks, written by the same client.



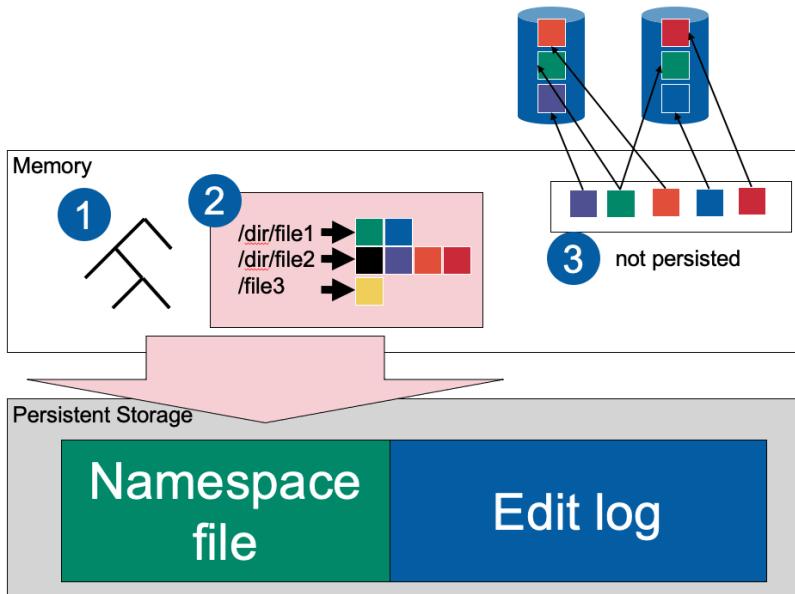
4.5 Fault tolerance and availability

HDFS has a single point of failure: the NameNode. If the metadata stored on it is lost, then all the data on the cluster is lost, because it is not possible to reassemble the blocks into files any more.

For this reason, the metadata is backed up. More precisely, the file namespace containing the directory and file hierarchy as well as the mapping from files to block IDs is backed up to a so-called snapshot. Note that the mapping of block IDs to DataNodes does not require a backup, as it can be recovered from the periodic block reports.

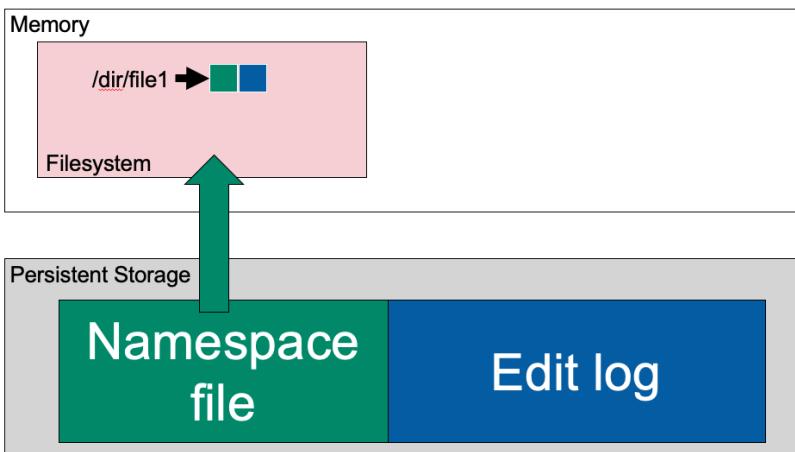


Since the HDFS system is constantly updated, it would not be viable to do a backup upon each update. It would also not be viable to do backups less often, as this could lead to data loss. Thus, what is done is that updates to the file system arriving after the snapshot has been made are instead stored in a journal, called edit log, that lists the updates sorted by time of arrival.

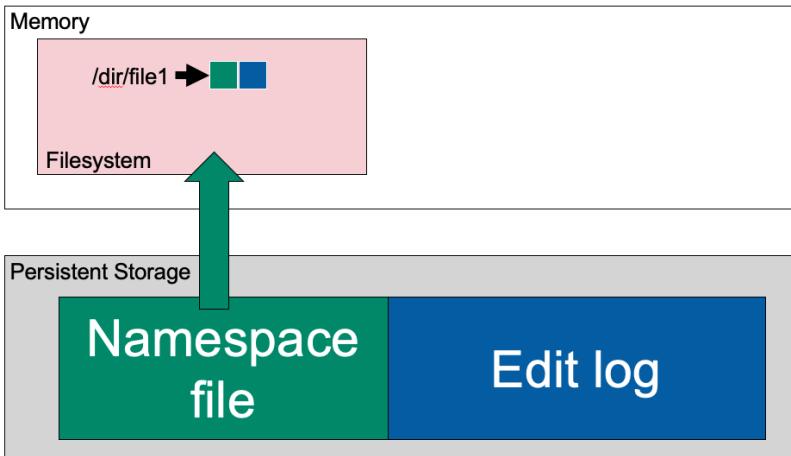


The snapshot and edit log are stored either locally or on a network-attached drive (not HDFS itself). For more resilience, they can also be copied over to more backup locations.

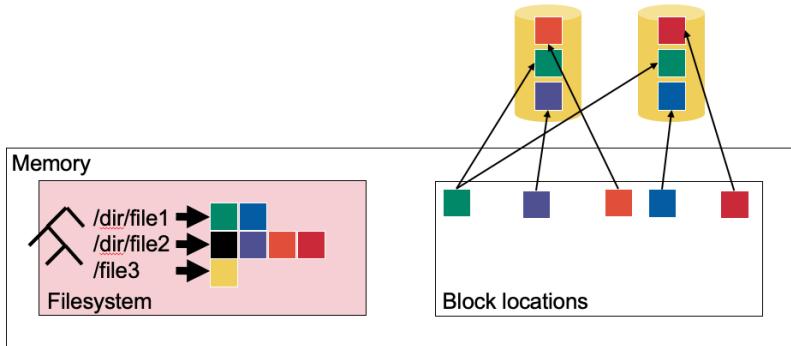
If the NameNode crashes, it can be restarted, the snapshot can be loaded back into memory to get the file namespace and the mapping of the files to block IDs.



Then the edit log can be replayed in order to apply the latest changes.



And the NameNode can wait for (or trigger) block reports to rebuild the mapping from block IDs to the DataNodes that have a replica of them.

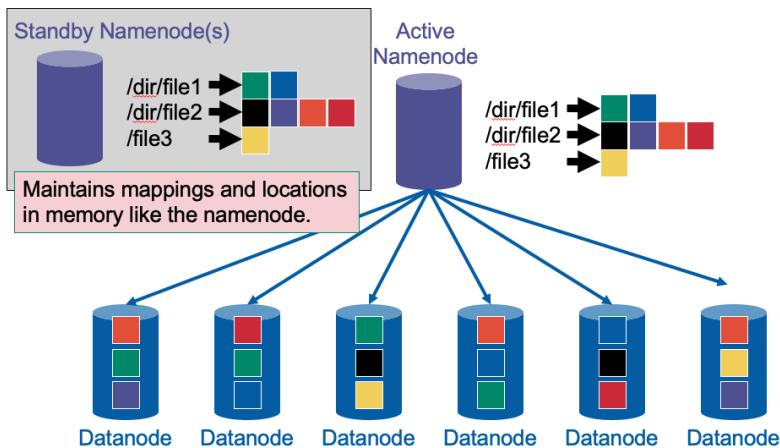


However, as more and more updates are applied, the edit log grows. This can easily lead to a long delay of 30+ minutes to restart the NameNode after a crash. More strategies had to be put in place. This was done incrementally in each HDFS release, and there are many different kinds of additional NameNodes that came into place and succeeded to one another: the Checkpoint NameNode, the Backup NameNode, the Secondary NameNode, the Standby NameNode, etc. We will not go into the details of each one of the version, but summarize the most important take aways.

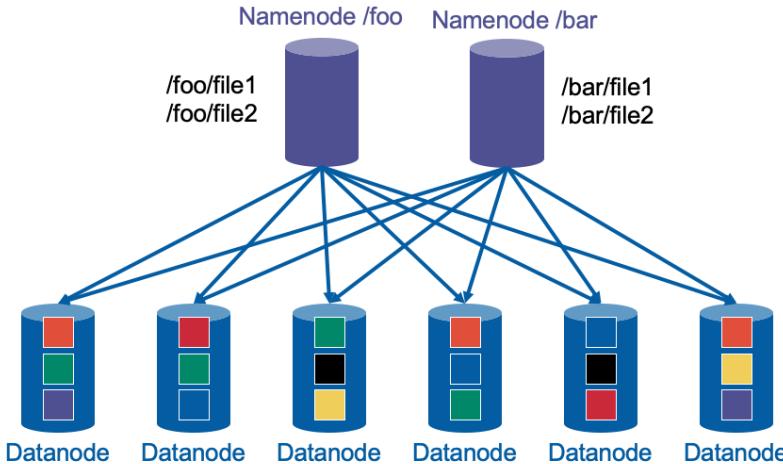
First, the edit log is periodically merged back into a new, larger snapshot and reset to an empty edit log. This is called a checkpoint. This can be done with a “phantom NameNode” (our terminology) that keeps the exact same structures in its memory as the real NameNode, and performs checkpoints periodically.

Second, it is possible to configure a phantom NameNode to be able to instantly take over from the NameNode in case of a crash. This considerably shortens the amount of time to recover.

As of the time of writing, these are called Standby NameNodes, but we do warn that the HDFS architecture will continue to evolve.



Another change to the architecture was made later: Federated NameNodes. In this change, several NameNodes can run at the same time, and each NameNode is responsible for specific (non overlapping) directories within the hierarchy. This spreads the management workload over multiple nodes.



4.6 Using HDFS

One of the ways to access HDFS is via the command line (a shell). The HDFS command line interface is mostly POSIX compliant, which means that the names of the commands are quite similar to those found on Linux or macos.

HDFS command start with

```
hdfs dfs <put command here>
```

Although we recommend to use the hadoop command instead, because it can also connect to other file systems (S3, your local drive, etc).

```
hadoop fs <put command here>
```

You can list the contents of the current directory with:

```
hadoop fs -ls
```

You can print to screen the contents of a file with:

```
hadoop fs -cat /user/hadoop/dir/file.json
```

You can delete a file with:

```
hadoop fs -rm /user/hadoop/dir/file.json
```

You can create a directory with:

```
hadoop fs -mkdir /user/hadoop/dir/file.json
```

You can upload files from your local computer to HDFS with:

```
hadoop fs -copyFromLocal localfile1 localfile2  
          /user/hadoop/targetdirectory
```

You can upload a file from HDFS to your local computer with:

```
hadoop fs -copyToLocal /user/hadoop/file.json  
          localfile.json
```

4.7 Paths and URIs

Although in the previous examples we used relative paths, generally, paths are URIs, as seen in the previous chapter. The scheme determines the file system.

This is what an HDFS URI looks like:

```
hdfs://www.example.com:8021/user/hadoop/file.json
```

This is what an S3 URI looks like (there are also the s3a and s3n schemes):

```
s3://my-bucket/directory/file.json
```

In Azure blob storage:

```
wasb://mycontainer@myaccount.blob.core.windows.net  
          /directory/file.json
```

And on the local file system:

```
file:///home/user/file.json
```

Typically, a default file system can be defined in a configuration file called core-site.xml:

```
<properties> <property>  
<name>fs.defaultFS</name>  
<value>hdfs://hdfs.mycluster.example.com:8020</value>  
<description>NameNode hostname</description>  
</property> </properties>
```

Then, the scheme is not needed to access files on this system and an absolute path starting with a slash can be used instead:

```
/user/hadoop/file.json
```

Note that other file systems can still be accessed (if set up properly) by using a URI scheme.

As for relative paths, they are resolved against the working directory, which should be unsurprising to anybody familiar with the command line interface even outside the context of HDFS.

Please mind that the current working directory might be on a different file system than the default file system; then you have situations in which a relative path will be local, while an absolute path will be on cloud, or distributed storage. This can cause unexpected issues. A common mistake is for users to use a relative path within a massive parallel computing framework (MapReduce, Apache Spark) when the working directory is local, which causes the data to be read, or written to the local disk rather than the HDFS cluster. If the job succeeds but the output is nowhere to be seen, it is likely it was just spit all over the local disks of the cluster. You can play treasure hunt and go after each machine to download the results, or you might realize that it will be easier to just rerun your job with the correct paths.

4.8 Logging and importing data

Two tools are worth mentioning: Apache Flume lets you collect, aggregate and move log data to HDFS. Apache Sqoop lets you import data from a relational database management system to HDFS.

This completes the chapter on distributed file systems. The most adventurous readers are encouraged to create a small HDFS cluster (this only takes a few clicks with Amazon EMR or Azure HDInsights), upload a few files, and log onto the machines with SSH to try to locate the physical block replicas as local files on the DataNodes.

4.9 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain the difference between block storage and object storage?
- b. Can you explain the difference between the (logical) key-value model and a file system?
- c. Can you contrast block storage and object storage in terms of maximum number of objects/files?
- d. Can you contrast block storage and object storage in terms of the maximum size of objects/files?
- e. Do you know the order of magnitude of a block size for a local filesystem and for a distributed file system? Can you explain the rationale behind them with respect to latency and throughput?
- f. Do you know where HDFS shines and why?
- g. Do you know that HDFS files are updated by appending atomically and why?
- h. Do you know how HDFS performs in terms of throughput and latency?
- i. Do you know the main benefits of HDFS (commodity hardware, etc.)?
- j. Can you contrast centralized architectures to decentralized (peer-to-peer) architectures?
- k. Can you explain the HDFS architecture, what a NameNode is and what a DataNode is, how blocks are replicated?
- l. Can you sketch how the various components communicate with each other (client, NameNode, DataNode)?
- m. Can you point to the single points of failure of HDFS and explain how they can be addressed?
- n. Can you explain how the NameNode stores the file system namespace, in memory and on disk? In particular, can you explain how the namespace file and the edit log work together at startup time and how they get modified once the system is up and running?

- o. Can you explain what a Standby NameNode is? (Note: it has many predecessors that only have historical relevance in the development of HDFS: Backup NameNode, Secondary NameNode, Checkpoint NameNode, etc, but this is not important for the course)
- p. Are you able to use the HDFS shell (creating directories, reading, uploading and downloading files, etc.)?

4.10 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Shvachko, K. et al. (2010). *The Hadoop Distributed File System*. 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)

Ghemawat, S. et al. (2003). *The Google File System*. Proceedings of the nineteenth ACM symposium on Operating systems principles. Vol. 37. pp 29-43.

White, T. (2015). *Hadoop: The Definitive Guide*. 4th edition. O'Reilly Media, Inc. Chapters 1 and 3.

The HDFS architecture page, the HDFS shell commands page, and the WebHDFS REST API page.

hadoop.apache.org/docs/current

Chapter 5

Syntax

5.1 Why syntax

Now that we have a storage layer in place, be it as a public cloud service like S3 or Azure blob storage, or as one's own HDFS cluster, we can start working our way up the Big Data technology stack by looking at what files we actually store.

We already mentioned that the public cloud is very popular for storing pictures, videos (for streaming and video on demand) as well as for static website hosting. But the public cloud is also great for storing datasets. Storing datasets directly in the public cloud or on a distributed file system is also known as a data lake.

What makes a data lake different from the way a classical database management system works is that the latter stores all its data in a proprietary format, hidden from the user. Any data that enters the system must be imported, more precisely, ETL¹. ETLing brings a lot of bells and whistles such as faster queries through efficient storage formats, indices, and so on. But ETLing also takes time and efforts and it might not be worth it in all cases. In-situ querying (querying in place) increased in importance in the last decades.

In a data lake, the syntax of the data is exposed to the user. A prominent example is that of CSV files that are known beyond the computer science community. CSV is syntax for tabular data.

¹ETL is for Extract, Transform, Load

5.1.1 CSV

```
ID,Last name,First name
1,Einstein,Albert
2,Gödel,Kurt
```



ID	Last name	First name
1	Einstein	Albert
2	Gödel	Kurt

CSV is a textual format, in the sense that it can be opened in a text editor. This is in contrast to binary formats that are more opaque.

Each record (a table row) corresponds to one line of text in a CSV file. Having a record per line of text is a common pattern not unique to CSV, as we will see later in this course. This is what makes it possible to scale up data processing to billions of records.

What appears on each line of text is specific to CSV. CSV means comma-separated values.

The main challenge with CSV files is that, in spite of a standard (RFC 4180), in practice there are many different dialects and variations, which limits interoperability. For example, another character can be used instead of the comma (tabulation, semi-colons, etc). Also, when a comma (or the special character used in its stead) needs to actually appear in a value, it needs to be escaped. There are many ways to do so; one of them is to double-quote the cell, which implies in turn that quotes within quotes must be escaped. There are many different conventions for doing so.

```
ID,Last name,First name,Theory
1,Einstein,Albert,"General, Special Relativity"
2,Gödel,Kurt,"""Incompleteness"" Theorem"
```

5.1.2 Data denormalization

We saw in a previous chapter that it is desirable to store data in so-called normal forms in a relational database management system. As you may recall, data in the first normal form cannot nest, and data in higher normal forms are split across multiple tables that get joined at query time. As a rule of thumb, normalizing data means joining it back at query time.

In the context of data lake and large-scale data processing, it is often desirable to go exactly the opposite way. This is called data denormalization. This means that not only several tables can be merged

into just one (with functional dependencies that would otherwise have been considered “undesirable”), it also means that we can nest data: tables in tables in tables.

While this is likely to come as a shock to any student who spent hours learning normal forms in a Bachelor’s level database lecture (widespread in European curricula, in particular in German-speaking countries), it has to be said that data denormalization should be done with knowledge of normal forms, because one needs to have a deep understanding of what one is doing and why one is doing it.

Data denormalization makes a lot of sense in read-intensive scenarios in which not having to join brings a significant performance improvement. In read-intensive scenarios, we love anything that is linear, which corresponds to a full scan of the dataset. This is as opposed to point queries more commonly found in traditional databases.

Thanks to the way that we defined tables in chapter 2, data denormalization is straightforward to explain. A table is a collection of tuples.

We required identical support (relational integrity), flat rows (atomic integrity, which is also the first normal form), and homogeneous data types within a column (domain integrity). Denormalization simply means that we drop all three constraints (or two, or just one).

Let us dive into this.

A tuple, mathematically, can be formalized as a partial function mapping strings to values:

product \mapsto Phone

price \mapsto 800

customer \mapsto John

quantity \mapsto 1

S V

As it turns out, a tuple can also be represented in a purely textual fashion (we will see this is called JSON – we will learn JSON in details later in this chapter).

```
{
  "product" : "Phone",
  "price" : 800,
  "customer" : "John",
  "quantity" : 1
}
```

The difference with CSV is that, in JSON, the attributes appear in every tuple, while in CSV they do not appear except in the header line. JSON is appropriate for data denormalization because including the attributes in every tuple allows us to drop the identical support requirement.

If we now look at a table (which checks all three integrity boxes), we can re-express it in a JSON-based textual format like so:

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	integer
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2



```
{ "product" : "Phone", "price" : 800, "customer" : "John", "quantity" : 1 }
{ "product" : "Phone", "price" : 800, "customer" : "Peter", "quantity" : 2 }
{ "product" : "Phone", "price" : 800, "customer" : "Mary", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "John", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "Mary", "quantity" : 1 }
...

```

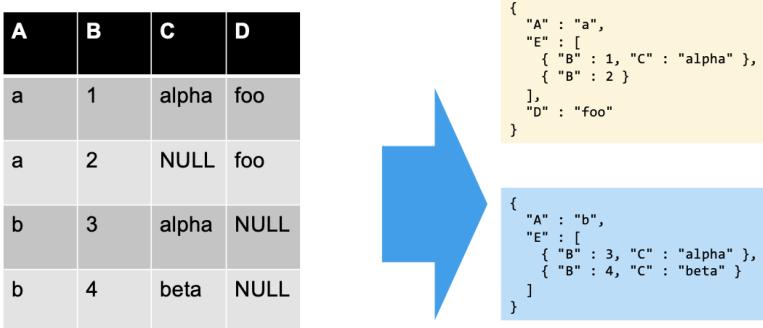
Now, if we are to drop relational integrity and allow for nestedness, the table could look like so:

sales											
product	orders										
varchar(30)	text										
Phone	<table border="1"> <thead> <tr> <th>customer</th><th>quantity</th></tr> </thead> <tbody> <tr> <td>text</td><td>integer</td></tr> <tr> <td>John</td><td>1</td></tr> <tr> <td>Peter</td><td>2</td></tr> <tr> <td>Mary</td><td>1</td></tr> </tbody> </table>	customer	quantity	text	integer	John	1	Peter	2	Mary	1
customer	quantity										
text	integer										
John	1										
Peter	2										
Mary	1										
<table border="1"> <thead> <tr> <th>customer</th><th>quantity</th></tr> </thead> <tbody> <tr> <td>text</td><td>integer</td></tr> <tr> <td>John</td><td>3</td></tr> <tr> <td>Mary</td><td>1</td></tr> </tbody> </table>	customer	quantity	text	integer	John	3	Mary	1			
customer	quantity										
text	integer										
John	3										
Mary	1										
<table border="1"> <thead> <tr> <th>customer</th><th>quantity</th></tr> </thead> <tbody> <tr> <td>text</td><td>date</td></tr> <tr> <td>Mary</td><td>1</td></tr> </tbody> </table>	customer	quantity	text	date	Mary	1					
customer	quantity										
text	date										
Mary	1										

CSV would not be powerful enough to express such data. But JSON is able to. For example, the first tuple of the table above, expressed in JSON, looks like so:

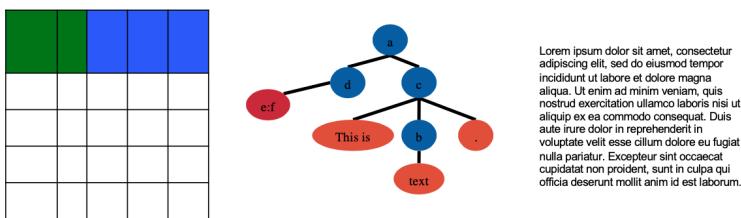
```
{
  "product": "Phone",
  "orders": [
    { "customer": "John", "quantity": 1 },
    { "customer": "Peter", "quantity": 2 },
    { "customer": "Mary", "quantity": 2 }
  ]
}
```

Concretely, data denormalization means that we abandon the paradigm of homogeneous collections of flat items (tables) and instead consider heterogeneous collections of nested items;



5.2 Semi-structured data and well-formedness

The generic name for denormalized data (in the same of heterogeneous and nested) is “semi-structured data”. Textual formats such as XML and JSON have the advantage that they can both be processed by computers, and can also be read, written and edited by humans. As we will see in Chapter 12, though, binary formats exist, too, although they are typically less denormalized.



Another very important and characterizing aspect of XML and JSON is that they are standards: XML is a W3C standard. W3C, also known as the World Wide Web consortium, is the same body that also standardizes HTML, HTTP, etc. JSON is now an ECMA standard, which is the same body that also standardizes JavaScript. In fact, the JS in JSON comes from JavaScript, because its look was inspired by JavaScript.

This is what an XML document looks like:

```

<?xml version="1.0"?>
<country code="CH">
    <name>Switzerland</name>
    <population>8014000</population>
    <currency code="CHF">Swiss Franc</currency>
    <cities>
        <city>Zurich</city>
        <city>Geneva</city>
        <city>Bern <!-- the Federal City --></city>
    </cities>
    <description>
        We produce <b>very</b> good chocolate.
    </description>
</country>

```

This is what a JSON document looks like:

```

{
    "code": "CH",
    "name": "Switzerland",
    "population": 8014000,
    "currency": {
        "name": "Swiss Franc",
        "code": "CHF"
    },
    "confederation": true,
    "president": "Ueli Maurer",
    "capital": null,
    "cities": [ "Zurich", "Geneva", "Bern" ],
    "description": "We produce very good chocolate."
}

```

In the rest of the chapter, we will explore both XML and JSON in details. It is commonly believed that XML is losing in popularity and JSON is “the new cool stuff”, however this is not fully accurate; while on the research side, the publications on XML have become less widespread, in companies, XML is very popular due to its very mature ecosystem supported by several other W3C standards. A few examples are that the mandatory financial reports of US public companies must

be filed in XML, and in Switzerland, electronic tax statements are also stored in XML. What is important is to understand that none of them is better than the other; this is highly use-case dependent and in some cases XML will be a better fit (this is typically the case in the publishing industry), while in other cases JSON will be a better fit.

Whichever syntax is used, they have in common the concept of well-formedness. For any input text document, one can ask whether it is well-formed XML, or whether it is well-formed JSON. This is a yes-no question: given a specific chosen syntax, either a document is well-formed, or it is not well-formed. In theoretical computer science, XML and JSON would be called languages. Formally, languages are sets of strings (which is the fancy word for “text”): all the strings that belong to this language. A string is said to be well-formed if it belongs to the language. Concretely, when a document is well-formed XML, it means that it can be successfully opened by an editor as XML with no errors, and plenty of bells and whistles kick in such as fancy colours to facilitate the reading by humans, automatic indentation, etc. A well-formed XML document can then be further processed (we will see later how). For well-formed JSON, this works exactly in the same way.

On the other hand, a non-well-formed document cannot be used and cannot benefit from all the XML and JSON tools until it is fixed and edited into a well-formed document. This is all or nothing.

Since JSON and XML are standards, there is a very large number of tools and libraries that support them out of the box, both for reading and for writing, many being free and open source. Thus, producing well-formed XML and JSON documents is very easy with these tools. It is probably the existence of all these tools that pushes people to use these syntaxes rather than inventing a new one, and having to redesign all the tooling.

5.3 JSON

Let us now dive into the details of the JSON syntax. JSON stands for JavaScript Object Notation because the way it looks like originates from JavaScript syntax, however it is now living its own life completely independently of JavaScript².

JSON is made of exactly six building blocks: strings, numbers, Booleans, null, objects, and arrays. Let us go through them.

5.3.1 Strings

Strings are simply text. In JSON, strings always appear in double quotes. This is a well-formed JSON string:

²which is not a query language!

```
"This is a string"
```

Obviously, strings could contain quotes and in order not to confuse them with the surrounding quotes, they need to be differentiated. This is called escaping and, in JSON, escaping is done with backslash characters (\). This should be quite familiar to many people with programming experience:

```
"The word \"quoted\" is quoted."
```

There are several other escape sequences in JSON, the most popular ones being:

\\\	\
\n	new line
\r	carriage return
\t	tabulation
\u followed by four hexadecimal digits	any character

The last one, in fact, allows the insertion of *any* character via its Unicode code point. Unicode is a standard that assigns a numeric code (called a code point) to each character in order to catalogue them across all languages of the world, even including emojis. The catalogue evolves with regular meetings of the working group. If you know the Unicode code point of a special character, then it is straightforward to escape it, e.g., for the letter Π:

```
"\u03c0"
```

The code point must be indicated in base 16 (digits 0 to 9, plus letters from A to F). Code points can easily be looked up with a search engine by typing a description of what you are looking for, even though more complex strings will typically be created automatically.

5.3.2 Numbers

JSON generally supports numbers, without explicitly naming any types nor making any distinction between numbers apart from how they appear in syntax. The way a number appears in syntax is called a *lexical representation*, or a *literal*. These two words, in fact, also generally apply to many other types as we will see in subsequent chapters.

Generally, a number is made of digits, possibly including a decimal period (which must be a dot) and optionally followed by the letter e (in either case) and a power of ten (scientific notation). Both the number and the optional power of ten can also have an optional sign.

These are a few examples of well-formed JSON number literals:

```
0
1234
12.34
-132.54
12.3E45
12.3e-45
-12.3e-45
```

JSON places a few restrictions: a leading + is not allowed. Also, a leading 0 is not allowed except if the integer part is exactly 0 (in which case it is even mandatory, i.e., .23 is not a well-formed JSON number literal):

```
0.23
```

JSON numbers are unquoted. Otherwise, they would be recognized as strings by the parser and not as numbers.

A warning: the same (mathematical) number might have several literals to represent it.

```
2
20e-1
2.0
```

It is important, as we will see later, to have in mind that the literal, which is the syntactic representation, is not the same as the actual, logical number. The above three literals have in common their “two-ness”.

5.3.3 Booleans

There are two Booleans, true and false, and each one is associated with exactly one possible literal, which are, well, true and false.

```
true
false
```

In spite of the fact that there is only exactly one literal for each Boolean, it is also important to distinguish the literal *true*, which is the sequence of letters t, r, u and e appearing in JSON syntax, from the actual concept of “true-ness,” which is an abstract mathematical concept. While in this chapter we focus on syntax and literals, when later we actually query data, we will work on the abstract concepts, not on the underlying syntax.

Boolean literals are unquoted. Otherwise, they would be recognized as strings by the parser and not as Booleans.

5.3.4 Null

There is a special value, null, which corresponds to the (unique) literal.

```
null
```

The concept of “null-ness” can be subject to debate: some like to see this as an unknown or hidden value, others as equivalent to an absent value, etc. In this course, on the logical level, we will consider that an absent value is not the same thing as a null value.

Null literals are unquoted. Otherwise, they would be recognized as strings by the parser and not as nulls.

5.3.5 Arrays

Arrays are simply lists of values. The concept of list is abstract and mathematical, i.e., lists are considered an abstract data type and correspond to finite mathematical sequences.

The concept of array is the syntactic counterpart of a list, i.e., an array is a physical representation of an abstract list.

The members of an array can be any JSON value: string, number, Boolean, null, array, or (we will get to it quite shortly) object. They are listed within square brackets, and are separated by commas.

```
[ 1, 2, 3 ]  
[ ]  
[ null, "foo", 12.3, false, [ 1, 3 ] ]
```

It can also be convenient to let arrays “breathe” with extra spaces, which are irrelevant when parsing JSON (except if they are *inside* a string literal!). In fact, there are plenty of libraries out there that can nicely do this, which is known as “pretty-printing.”

```
[  
  1,  
  2,  
  3  
]  
[ ]  
[  
  null,  
  "foo",  
  12.3,  
  false,  
  [  
    1,
```

```

    3
]
]
```

5.3.6 Objects

Objects are simply maps from strings to values. The concept of map is abstract and mathematical, i.e., maps are considered an abstract data type and correspond to mathematical partial functions with a string domain and the range of all values.

The concept of object is the syntactic counterpart of a map, i.e., an object is a physical representation of an abstract map that explicitly lists all string-value pairs (this is called an *extensional* definition of a function, as opposed to the way functions are typically defined in mathematics (e.g., $f(x) = x + 1$, which is, by contrast, intensional)).

The keys of an object must be strings. This excludes any other kind of value: it cannot be an integer, it cannot be an object. This also implies that keys must be quoted. While some JSON parsers are lenient and will accept unquoted keys, it is very important to never create any JSON documents with unquoted keys for full compatibility with all parsers.

The values associated with them can be any JSON value: string, number, Boolean, null, array, or object. The pairs are listed within curly brackets, and are separated by commas. Within a pair, the value is separated from the key with a colon character.

```

{ "foo" : 1 }
{
{ "foo" : "foo", "bar" : [ 1, 2 ],
  "foobar" : [ { "foo" : null }, { "foo" : true } ]
}
```

It can also be convenient to let objects “breathe” with extra spaces, as was already explained for arrays.

```
{
  "foo" : "foo",
  "bar" : [
    1,
    2
  ],
  "foobar" : [
    {
      "foo" : null,
      "bar" : 2
    },
    ],
```

```
{
    "foo" : true,
    "bar" : 3
}
]
```

The JSON standard recommends for keys to be unique within an object. Many parsers and products will reject duplicate keys, because they rely on the semantics of a map abstract data type. If one downloads a dataset that has duplicate keys and the engine one intends to use to process it does not allow them, then this will require extra work. In particular, one needs to find a JSON library that accepts duplicate keys, and use it to fix the dataset by disambiguating the keys to make it parseable with any engine. It is very important to never create any JSON documents with duplicate keys for full compatibility with all parsers, to avoid creating this extra workload for the consumers.

5.4 XML

XML stands for eXtensible Markup Language. It resembles HTML, except that it allows for any tags and that it is stricter in what it allows.

XML is considerably more complex than JSON but, fortunately, most datasets only use a subset of what XML can do. In our course, we will stick to the most common features of XML.

XML's most important building blocks are elements, attributes, text and comments.

5.4.1 Elements

XML is a markup language, which means that content is “tagged”. Tagging is done with XML elements.

An XML element consists of an opening tag, and a closing tag. What is “tagged” is everything inbetween the opening tag and the closing tag.

This is an example with an opening tag, some content (which can be recursively anything, as we will see), and then a closing tag. Tags consist of a name surrounded with angle brackets `< ... >`, and the closing tag has an additional slash in front of the name.

```
<person>(any content here)</person>
```

If there is no content at all, the lazy of us will appreciate a convenient shortcut to denote the empty element with a single tag. Mind that the slash is at the end:

```
<person/>
```

is equivalent to:

```
<person></person>
```

Elements nest arbitrarily:

```
<person><first>(some content)</first><student/>
<last>(some other content)</last></person>
```

Like JSON, it is possible to use indentation and new lines to pretty-print the document for ease of read by a human:

```
<person>
  <first>(some content)</first>
  <student/>
  <last>(some other content)</last>
</person>
```

Unlike JSON keys, element names can repeat at will. In fact, it is even a common pattern to repeat an element many times under another element in plural form, like so:

```
<persons>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
  <person>
    <first>(some content)</first>
    <last>(some other content)</last>
  </person>
</persons>
```

Some care needs to be put in “well-parenthesizing” tags, for example, this is incorrect and not well-formed XML:

```
<foo></bar></foo></bar>
```

because the inner elements must close before the outer elements.

Elements cannot appear within opening or closing tags, they must appear between tags. This is not well-formed XML:

```
<foo <bar/>></foo>
```

At the top-level, a well-formed XML document must have exactly one element. Not zero, not two, exactly one. This is not well-formed XML:

```
<person>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
<person>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
<person>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

5.4.2 Attributes

Attributes appear in any opening elements tag and are basically key-value pairs. In the following examples, we added two attributes with the keys “birth” and “death.”

```
<person birth="1879" death="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Values can be either double-quoted or single-quoted. This is also well-formed XML:

```
<person birth='1879' death='1955'>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

As well as this:

```
<person birth="1879" death='1955'>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

The key is never quoted, and it is not allowed to have unquoted values³. This is not well-formed XML:

```
<person birth=1879 "death"=1955>
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Within the same opening tag, there cannot be duplicate keys. This is not well-formed XML:

```
<person birth="1879" birth="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

Attributes can also appear in an empty element tag:

```
<person birth="1879" death="1955"/>
```

Attributes can never appear in a closing tag. This is not well-formed XML:

```
<person>
    <first>(some content)</first>
    <last>(some other content)</last>
</person birth="1879" death="1955">
```

Elements cannot nest within attribute values. This is not well-formed XML:

```
<person birth="<date>1879</date>" death="1955">
    <first>(some content)</first>
    <last>(some other content)</last>
</person>
```

It is not allowed to create attributes that start with XML or xml, or any case combination (XmL, etc). This is because this is reserved for another use (namespaces, as we will see shortly).

³This is unlike HTML, in which values can be without quotes

5.4.3 Text

Text, in XML syntax, is simply freely appearing in elements and without any quotes (attribute values are not text!). For example, we can have text inside the first and last elements like so:

```
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

Text cannot appear on its own at the top level. This is not well-formed XML:

```
Albert <person/> Einstein
```

Within an element, text can freely alternate with other elements. This is called mixed content and is unique to XML, like so:

```
<person>
    <style>His Royal Highness</style>
    The <title>Duke of <location>Cambridge</location></title>
</person>
```

This feature of XML makes it very popular in the publishing industry, where it is very convenient to have books, papers, etc, with the text tagged with extra information.

5.4.4 Comments

Comments in XML look like so:

```
<!-- This is a comment -->
```

but, as we saw, a single comment alone is not well-formed XML (remember: we need exactly one top-level element). This would be well-formed XML with a comment:

```
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
    <!-- He is still famous today -->
</person>
```

Comments can also appear at the top-level though, but under the condition that there is exactly one top-level element.

```
<!-- He is still famous today -->
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
<!-- He is -->
<!-- He totally is -->
```

The reason why comments specifically look like this is historical: XML was derived as a simplified subset of an older markup language called SGML. Many of the strange-looking symbols of XML are in fact coming from SGML.

5.4.5 Text declaration

XML documents can be identified as such with an optional text declaration containing a version number and an encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

The version is either 1.0 or 1.1, but there is no need to understand the difference for this course. It is rather subtle and mostly due to more permissive behaviour with international characters in 1.1. Most XML documents out there are version 1.0.

The encoding is a physical detail and gives information on how the document is stored as bits on the disk. This is also advanced and out of the scope of this course. If in doubt, UTF-8 is the recommended standard as of 2023.

Another tag that might appear right below, or instead of, the text declaration is the doctype declaration. It must then repeat the name of the top-level element, like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE person>
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

or like so:

```
<!DOCTYPE person>
<person birth="1879" death="1955">
    <first>Albert</first>
    <last>Einstein</last>
</person>
```

This might be familiar to HTML aficionados:

```
<!DOCTYPE html>
<html>
    ...
</html>
```

Doctype declarations exist also for historical reasons and are part of the DTD validation mechanism, which is out of scope for this course because it was superseded with more modern mechanisms such as XML Schema, which we will look at in Chapter 7.

All in all, the only reason why we showed what text declarations look like and what doctype declarations look like is so you are not surprised when you see them.

5.4.6 Escaping special characters

As you might have guessed, if characters such as < are used in the text, or characters such as " or ' are used in attribute values, it will cause problems. This is not well-formed XML:

```
<equation name="the "basic" comparison">
    1 < 2
</equation>
```

Remember that in JSON, it is possible to escape sequences with a backslash character. In XML, this is done with an ampersand (&) character.

There are exactly five possible escape sequences pre-defined in XML:

Escape sequence	Corresponding character
<	<
>	>
"	"
'	'
&	&

For example, the above document can be turned into a well-formed XML document like so:

```
<equation name="the "basic" comparison">
    1 &lt; 2
</equation>
```

Escape sequences can be used anywhere in text, and in attribute values. At other places (element names, attribute names, inside comments), they will not be recognized or will lead to well-formedness errors.

But there are a few places where they are mandatory:

- In text, & and < MUST be escaped. The other characters may, but need not, be escaped.
- In double-quoted attribute values, ", & and < MUST be escaped. The other characters may, but need not, be escaped.
- In single-quoted attribute values, ', & and < MUST be escaped. The other characters may, but need not, be escaped.

5.4.7 Namespaces in XML

When a lot of data is created in the XML format, scaling issues start appearing because people use the same element and attribute names for different purposes. For example, an element named “client” can be used in customer relationship datasets, or in computer network datasets.

Namespaces are an extension of XML that allows users to group their elements and attributes in packages, similar to Python modules, Java packages or C++ namespaces. This is a very natural thing to do.

Namespace URIs

A namespace is identified with a URI. We already studied URIs in the context of REST in Chapter 3. A point of confusion is that XML namespaces often start with *http://*, but are not meant to be entered as an address into a browser! It might sometimes work, but this will only be because the owner of the namespace was kind enough to put a page (often with documentation on the namespace) on the Web at the same URI.

Here are just a few examples of namespaces:

Namespace	use
http://www.w3.org/1999/xhtml	HTML
http://www.w3.org/1998/Math/MathML	MathML (formulas)
http://www.music-encoding.org/ns/mei	Music sheets

If you think about it, this is not so different from Java: Java just uses a different convention with reversed domains and dots instead of slashes. For example the Music package would probably be called org.music-encoding.ns.mei in Java.

An entire XML document in a namespace

Let us start with something easy. It is possible to put all elements of an XML document in a namespace, here <http://www.example.com/persons>, like so:

```
<persons xmlns="http://www.example.com/persons">
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
</persons>
```

In the above document, the elements person, first and last all live in the namespace <http://www.example.com/persons>. This is because of what looks like an xmlns attribute, associated with the namespace <http://www.example.com/persons> as the value. But in fact, xmlns is not an attribute, it is really a namespace declaration. If you remember, we saw that attributes starting with xml are forbidden, and this is because this is reserved for namespace declarations.

The document below is different, because it does not have this declaration. So, the elements person, first and last do not live in any namespace, we say that the namespace is absent for these elements:

```
<persons>
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
    <person>
        <first>(some content)</first>
        <last>(some other content)</last>
    </person>
</persons>
```

Here is another example, this time a MathML document, in the corresponding namespace:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <apply>
    <eq/>
    <ci>x</ci>
    <apply>
      <root/>
      <cn>2</cn>
    </apply>
  </apply>
</math>
```

QNames

What about documents that use multiple namespaces? This is done by associating namespaces with prefixes, which act as shorthands for a namespace. Then, we can use the prefix shorthand in every element that we want to have in this namespace.

This is the same MathML document as previously seen, except that now we explicitly associate the MathML namespace with prefix m. This is done by using xmlns:m instead of just xmlns, and by adding m: in front of every element that we want to have in this namespace, like so:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
  <m:apply>
    <m:eq/>
    <m:ci>x</m:ci>
    <m:apply>
      <m:root/>
      <m:cn>2</m:cn>
    </m:apply>
  </m:apply>
</m:math>
```

What is important to understand is that, semantically, this is the *same* document as the one we saw without the prefix: all elements are, in both cases, in the MathML namespace. The part of the element name that appears on the right of the colon sign (or the entire element name, if it does not have a prefix) is called the local name of the element.

Every element has a local name. An element may have a prefix (on the left of the colon sign, say *foo*), in which case the corresponding namespace is looked up with the appropriate *xmlns:foo* declaration with the same prefix. An element may also not have a prefix, in which case the prefix is said to be absent. In this case the corresponding namespace

(called the default namespace) is looked up with the appropriate *xmlns* declaration. If there is no such declaration, then the namespace is said to be absent, too.

So, given any element, it is possible to find its local name, its (possibly absent) prefix, and its (possibly absent) namespace. The triplet (namespace, prefix, localname) is called a QName (for “qualified name”).

For the purpose of the comparisons of two QNames (and thus of documents), the prefix is ignored: only the local name and the namespace are compared. The following document, which uses yet another prefix, is again the same document as the previous two MathML documents:

```
<foo:math xmlns:foo="http://www.w3.org/1998/Math/MathML">
  <foo:apply>
    <foo:eq/>
    <foo:ci>x</foo:ci>
    <foo:apply>
      <foo:root/>
        <foo:cn>2</foo:cn>
    </foo:apply>
  </foo:apply>
</foo:math>
```

This document, however, is different, because its elements are in no namespace at all:

```
<math>
  <apply>
    <eq/>
    <ci>x</ci>
    <apply>
      <root/>
        <cn>2</cn>
    </apply>
  </apply>
</math>
```

With the QName machinery, it is possible to have as many namespaces and prefixes as one wants, in this example four of them:

```
<?xml version "1.0"?>
<a:bar
  xmlns:a="http://example.com/a"
  xmlns:b="http://example.com/b"
  xmlns:c="http://example.com/c"
  xmlns:d="http://example.com/d">
```

```

<b:foo/>
<c:bar>
  <d:foo/>
  <a:foobar/>
</c:bar>
</a:bar>

```

Namespaces are quite flexible: xmlns declarations can be put anywhere and it can quickly become messy and out of control. Thus, we highly recommend to stick to several rules that are common practice if you want to keep your sanity:

- only put xmlns and xmlns:prefix declarations in the top-level element. Nowhere else.
- make sure the prefix-namespace mapping is bijective. Do not use twice the same prefix with the same namespace. Do not bind two namespaces with the same prefix (which would in fact be an error).
- do not mix the default namespace (xmlns declaration) with namespaces associated with prefixes in the same document (xmlns:prefix declarations). It is either or: either you have a single namespace for all elements in the entire document that is the default namespace, or you have one or several namespaces that are all associated with a (non-absent) prefix. Mixing the two types of declarations quickly leads to confusion for people looking at the document.

As a counterexample showing bad practice, this is what should be avoided:

```

<foo:bar xmlns:foo="http://example.com/foo">
  <foo:foo/>
  <bar:foobar xmlns:bar="http://example.com/bar">
    <bar:foo/>
    <foo:foo/>
    <foo/>
    <foo/>
  </bar:foobar>
  <foo xmlns="http://example.com/foo"/>
  <foo:bar/>
  <foo:bar xmlns:foo="http://example.com/bar"/>
  <foo:foo/>
</foo:bar>

```

Attributes and namespaces

Attributes can also live in namespaces, that is, attribute names are generally QNames. However, there are two very important aspects to consider.

First, unprefixed attributes are not sensitive to default namespaces: unlike elements, the namespace of an unprefixed attribute is always absent even if there is a default namespace. The attribute attr in this example is in no namespace, although all (unprefixed) elements live in the `http://example.com/foo` namespace:

```
<?xml version "1.0"?>
<bar xmlns="http://example.com/foo">
    <foo/>
    <foobar attr="value">
        <foo/>
        <foo/>
        <foo/>
        <foo/>
    </foobar>
    <foo/>
    <bar/>
    <bar/>
    <foo/>
</bar>
```

Second, it is possible for two attributes to collide if they have the same local name, and different prefixes but associated with the same namespace (but again, we told you: do not do that!). The following document is thus not well-formed.

```
<?xml version "1.0"?>
<bar
    xmlns:foo="http://example.com/foo"
    xmlns:bar="http://example.com/foo">
    <foo/>
    <foobar foo:attr="value" bar:attr="value">
        <foo/>
        <foo/>
        <foo/>
        <foo/>
    </foobar>
    <foo/>
    <bar/>
    <bar/>
    <foo/>
</bar>
```

5.4.8 Datasets in XML

Let us now look deeper at how to express tabular data in XML, in order to demonstrate that XML subsumes tabular data.

The tuple marked in red here:

sales			
product	price	customer	quantity
varchar(30)	numeric	text	integer
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

can be stored in XML as:

```
<sale>
  <product>Phone</product>
  <price>800</price>
  <customer>John</customer>
  <quantity>1</quantity>
</sale>
```

The tuples marked in red here:

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

can be stored in XML using the plural-singular convention as:

```

<sales>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>John</customer>
    <quantity>1</quantity>
  </sale>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>Peter</customer>
    <quantity>2</quantity>
  </sale>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <customer>Mary</customer>
    <quantity>1</quantity>
  </sale>
  <sale>
    <product>Laptop</product>
    <price>200</price>
    <customer>John</customer>
    <quantity>3</quantity>
  </sale>
</sales>

```

Finally this nested tuple:

sales													
product	price	orders											
varchar(30)	numeric	text											
Phone	800	<table border="1"> <thead> <tr> <th>customer</th><th>quantity</th></tr> </thead> <tbody> <tr> <td>text</td><td>date</td></tr> <tr> <td>John</td><td>1</td></tr> <tr> <td>Peter</td><td>2</td></tr> <tr> <td>Mary</td><td>1</td></tr> </tbody> </table>	customer	quantity	text	date	John	1	Peter	2	Mary	1	
customer	quantity												
text	date												
John	1												
Peter	2												
Mary	1												

can be stored in XML using nested elements as:

```
<?xml version "1.0"?>
```

```
<!DOCTYPE sales>
<sales>
  <sale>
    <product>Phone</product>
    <price>800</price>
    <orders>
      <order>
        <customer>John</customer>
        <quantity>1</quantity>
      </order>
      <order>
        <customer>Peter</customer>
        <quantity>2</quantity>
      </order>
      <order>
        <customer>Mary</customer>
        <quantity>1</quantity>
      </order>
    </orders>
  </sale>
</sales>
```

5.5 XML vs. JSON, or how to troll internet forums

Whether XML or JSON is better is the topic of an intense debate, a bit like vi vs. emacs or mac vs. PC. The reason is simple: neither is. It depends on the use case. Objectively, one can nevertheless say that XML is quite suitable and popular for the publishing industry and “text-oriented”, tagged data because of its unique mixed content feature. Data itself can be stored indifferently in XML or in JSON, as we saw. The XML ecosystem is also more mature and enterprise-ready because it is older, however JSON has gained so much popularity in the recent decade that it is expected that the JSON ecosystem will catch up.

5.6 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you describe why syntax is relevant to data management and to Big Data, for all data shapes?
- b. Can you give examples of syntax for trees? For tables? (The two data shapes extensively covered in the lecture)
- c. Can you explain what well-formedness is with respect to syntax?
- d. Can you list the JSON basic building blocks (object, array, string, number, Boolean, null) and do you know what they look like?
- e. Can you tell whether a given JSON document is well-formed, also with your own eyes?
- f. Can you list the XML basic building blocks (document, element, attribute, text) and do you know what they look like?
- g. Can you tell whether a given XML document is well-formed, with some specialized software (oXygen) as well as with your own eyes?
- h. Do you know the five fundamental pre-defined entities and their syntax?
- i. Do you know when some characters must be escaped (e.g., < in text)?
- j. Can you explain how XML namespaces work and how they are bound to prefixes (special attributes starting with xmlns)?
- k. Can you tell whether two XML documents are equivalent according to namespace semantics?
- l. Can you tell whether an XML document is well-formed in terms of namespaces (Note: only in the simple case that we covered, where all namespace-prefix bindings are on the root element)?
- m. Can you give, and give examples of, the most common patterns with which XML namespaces are used in practice? Can you still make sense of XML namespaces when documents are produced that do not follow these patterns?

5.7 Literature and recommended readings

The following is a list of recommended material for further reading and study.

JSON website

<https://www.json.org/>

JSON as an ECMA standard

<https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>

JSON as an RFC standard

<https://datatracker.ietf.org/doc/html/rfc7159>

Harold, E. R., Means, W. S. (2004). *XML in a Nutshell*. 3rd edition. Chapters 1, 2, 4.1, 4.2, 16, and 21.

XML specification

<https://www.w3.org/TR/xml/>

Chapter 6

Wide column stores

Now that we have looked into some textual data formats (syntax) like CSV, JSON, XML, how do we store these CSV, JSON and XML files? A first obvious solution is: on the local disk, or on an object storage service (S3, Azure Blob Storage), or on a distributed file system (HDFS).

The problem with HDFS is its latency: HDFS works well with very large files (at least hundreds of MBs so that blocks even start becoming useful), but will have performance issues if accessing millions of small XML or JSON files.

Wide column stores were invented to provide more control over performance and in particular, in order to achieve high-throughput *and* low latency for objects ranging from a few bytes to about 10 MB, which are too big and numerous to be efficiently stored as so-called clob (character large objects) or blobs (binary large objects) in a relational database system, but also too small and numerous to be efficiently accessed in a distributed file system.

6.1 A sweet spot between object storage and relational database systems

The astute reader will argue that a large number of JSON or XML files would be handled quite well with an object storage service. But a wide column store has additional benefits:

- a wide column store will be more tightly integrated with the parallel data processing systems, which we will introduce in subsequent chapters. This is possible because the wide column store processes run on the same machines as the data processing processes, and it makes the entire system faster;

- wide column stores have a richer logical model than the simple key-value model behind object storage;
- wide column stores also handle very small values (bytes and kB) well thanks to batch processing.

Note that a wide column store is not a relational database management system:

- it does not have any data model for values, which are just arrays of bytes;
- since it efficiently handles values up to 10 MB, the values can be nested data in various formats, which breaks the first normal form;
- tables do not have a schema;
- there is no language like SQL, instead the API is on a lower level and more akin to that of a key-value store;
- tables can be very sparse, allowing for billions of rows and millions of columns at the same time; this is another reason why data stored in HBase is denormalized.

6.2 History

The good old relational database management systems (RDBMS) from the 1970s were built as monolithic engines that run on a single machine. They can only hold as much data as fits on one machine, and are only as fast as the CPU on that machine.

How can we handle more data or process it faster with such a system? This was a problem first encountered by search engines, which need to cache and index the Web (HTML pages, etc). As we previously saw, we could scale up by buying a bigger machine, with a more powerful CPU, with larger or more drives, and with more RAM. But if you remember, we also saw that scaling up has its limits because the price will not grow linearly, and what is feasible in a given year will also hit a hard limit.

Another approach is to scale out to several machines. In fact, this was attempted by several companies (some of the tech giants) in the early 2000s as the quantity of data they had to manage was challenging their infrastructure. The early systems attempting to scale out an RDBMS were rudimentary: several machines would be purchased, set up and connected with each other via the network and the same RDBMS software would be installed on each machine.

Then, the data would be spread over these different systems and possibly stored redundantly. In fact, the underlying idea is similar to how we partitioned file into blocks in HDFS, how the blocks are physically stored as several replica, and how the replicas are spread all over the cluster of machines.

However, the logic for partitioning the data across machines and for handling the replication had to be done with additional software, written in programming languages such as Java or C++, connecting as clients to the RDBMS instances. This was not only very difficult to set up, it was also very costly in terms of maintenance.

Furthermore, additional logic to query the data on the entire cluster was also needed: a program, outside the RDBMS, needed to figure out where to read the data from or where to write it to, how to reassemble it for the querying user, etc.

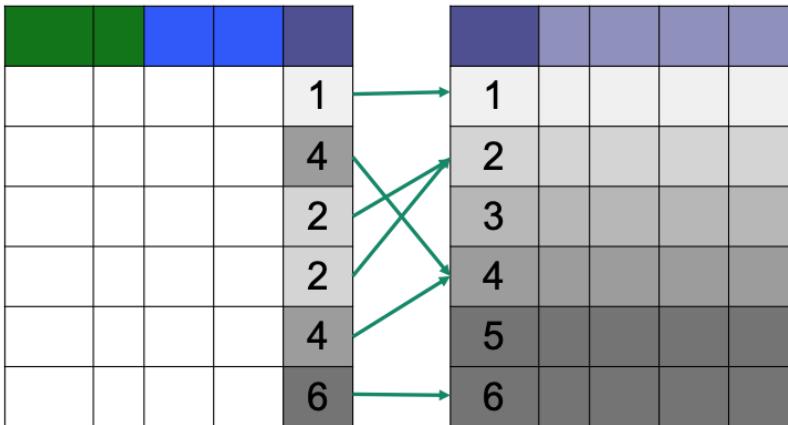
In fact, the engineers realized that they were really building an entirely new database management system, with an entirely new architecture designed for a cluster rather than for a single machine and optimized for clob and blobs. The company who came up with the first fully integrated product was Google, and this product was called BigTable. An open-source equivalent then followed as part of the Hadoop ecosystem, called HBase.

6.3 Logical data model

6.3.1 Rationale

The data model of HBase is based on the realization that joins are expensive, and that they should be avoided or minimized on a cluster architecture. Joins can be avoided if they are pre-computed, that is, instead of storing the data as separate tables, we store, and work on, the joined table.

Visually, what would look like this in a traditional RDBMS:



is instead stored like so in HBase:

					1				
					4				
					2				
					2				
					4				
					6				

There is a direct consequence of this change: the number of columns in a traditional RDBMS is limited, typically to somewhere around 256, or maybe in the low four digits if the columns have simple and compact types. But in a denormalized table, the number of columns can easily skyrocket, with many cells being in fact empty. The table is thus very wide and sparse, which is a completely different use case than what RDBMS software was designed for.

The second design principle underlying HBase is that it is efficient to store together what is accessed together. In the big picture, this is a flavor of batch processing, one of the overarching principles in Big Data. Batch processing reduces the impact of latency – remember that we saw

in Chapter 1 that latency barely improved in the past few decades in comparison to capacity and throughput – by involving fewer, larger requests instead of more, smaller requests.

These two design principles are tied to a specific usage pattern of the database management system: when the emphasis is reading and processing data in large amounts. This is, at first sight, in direct conflict with efficiently writing data. First, writing denormalized data is more cumbersome: one needs to deal with insertion, update and deletion anomalies. Second, writing data under the constraint of storing together what is accessed together is a challenging endeavor, because one cannot just “insert” new data at a specific physical location without moving the existing data around.

As we will see in this chapter, HBase provides an impressive solution for handling writes quite efficiently, while preserving a batch-processing paradigm. Moreover, the underlying idea also impacted the handling of intermediate data by MapReduce and Apache Spark, which we will cover in later chapters.

6.3.2 Tables and row IDs

From an abstract perspective, HBase can be seen as an enhanced key-value store, in the sense that:

- a key is compound and involves a row, a column and a version;
- keys are sortable;
- values can be larger (clob, blob), up to around 10 MB.

This is unlike traditional key-value stores, which have a flat key-value model, which (in the case of distributed hash tables) do not sort keys, which thus do not store “close” keys together, and which usually support smaller values.

On the logical level, the data is organized in a tabular fashion: as a collection of rows. Each row is identified with a row ID. Row IDs can be compared, and the rows are logically sorted by row ID:

Row ID	A	B	1	2	I
000					
002					
0A1					
1E0					
22A					
4A2					

A row ID is logically an array of bytes, although there is a library to easily create row ID bytes from specific primitive values (byte, short, int, long, string, etc).

6.3.3 Column families

The other attributes, called columns, are split into so-called column families. This is a concept that does not exist in relational databases and that allows scaling the number of columns. Very intuitively, one can think of column families as the tables that one would have if the data were actually normalized and the joins had not been pre-computed. On the picture above, we separated the columns into column families, using a different color and alphabet for each family. The name of a column family is a string, just like the name of a table. Often, we also use the terminology “column family” to refer to the name of the column family, i.e., we identify the column family with its name.

6.3.4 Column qualifiers

Columns in HBase have a name (in addition to the column family) called column qualifier, however unlike traditional RDBMS, they do not have a particular type. In fact, as far as HBase is concerned, all values are binary (arrays of bytes) and what the user does with it (string, integer, large objects, XML, JSON, HTML etc) is really up to them. There are many different frameworks that can be used in complement of HBase to add a type system (Avro, etc) and it is, in fact, very common to store large blobs of data in the cells. We will cover the paradigm for this in Chapter 7.

In fact, it goes further than that. Not only are there no column types: even the column qualifiers are not specified as part of the schema of an HBase table: columns are created on the fly when data is inserted, and the rows need not have data in the same columns, which natively

allows for sparsity. Column qualifiers are arrays of bytes (rather than strings), and as for row IDs, there is a library to easily create column qualifiers from primitive values.

Thus, on the logical level, columns come and go as the table lives its life:

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
22A									
4A2									

Unlike the values which can be large arrays of bytes (blobs), it is important to keep column families and column qualifiers short, because as we will see, they are repeated a gigantic number of times on the physical layer.

6.3.5 Versioning

HBase generally supports versioning, in the sense that it keeps track of the past versions of the data. As we will see, this is implemented by associating any value with a timestamp, also called version, at which it was created (or deleted). Users can also override timestamps with a value of their choice to have more control about versions.

6.3.6 A multidimensional key-value store

As we previously explained, one way to look at an HBase table is that it is an enhanced key-value store where the key is four-dimensional. Indeed, in HBase, the key identifying the values in the cells consists of:

- the row ID
- the column family
- the column qualifier
- the version

HBase is able to efficiently look up any cell, and even any version of any cell, given its key.

6.4 Logical queries

Having realized that an HBase table is nothing but a four-dimensional key-value store, it follows logically that the HBase API also resembles that of a key-value store: HBase supports four kinds of low-level queries: get, put, scan and delete. Unlike a traditional key-value store, HBase also supports querying ranges of row IDs and ranges of timestamps.

In comparison to a full-fledged RDBMS, this is quite limited and, as for data types, support for higher-level queries (such as SQL) is brought by additional frameworks that come as a complement of, and atop HBase (Apache Hive, Apache Phoenix, etc).

6.4.1 Get

With a get command, it is possible to retrieve a row specifying a table and a row ID.

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
22A									
4A2									

Optionally, it is also possible to only request some but not all of the columns, or to request a specific version, or the latest k versions (where k can be chosen) within a time range (interval of versions).

6.4.2 Put

With a put command, it is possible to put a new value in a cell by specifying a table, row ID, column family and column qualifier.

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
204									
22A									
4A2									

It is also possible to optionally specify the version. If none is specified, the current time is used as the version.

HBase offers a locking mechanism at the row level, meaning that different rows can be modified concurrently, but the cells in the same row cannot: only one user at a time can modify any given row.

6.4.3 Scan

With a scan command, it is possible to query a whole table or part of a table, as opposed to a single row.

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
204									
22A									
4A2									

It is possible to restrict the scan to specific columns families or even columns.

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
204									
22A									
4A2									

It is possible to restrict the scan to an interval of rows.

Row ID	A	B	C	1	2	I	II	III	IV
000									
002									
0A1									
1E0									
204									
22A									
4A2									

It is possible to run the scan at a specific version, or on a time range.

Scans are fundamental for obtaining high throughput in parallel processing.

6.4.4 Delete

With a delete command, it is possible to delete a specific value with a table, row ID, column family and qualifier. Optionally, it is also possible to delete the value with a specific version, or all values with a version less or equal to a specific version.

6.5 Physical architecture

6.5.1 Partitioning

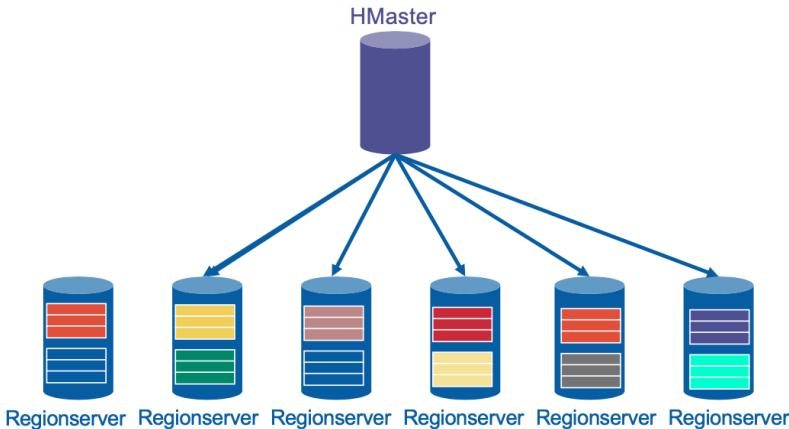
A table in HBase is physically partitioned in two ways: on the rows and on the columns.

The rows are split in consecutive regions. Each region is identified by a lower and an upper row key, the lower row key being included and the upper row key excluded.

A partition is called a store and corresponds to the intersection of a region and of a column family.

6.5.2 Network topology

HBase has exactly the same centralized architecture as HDFS

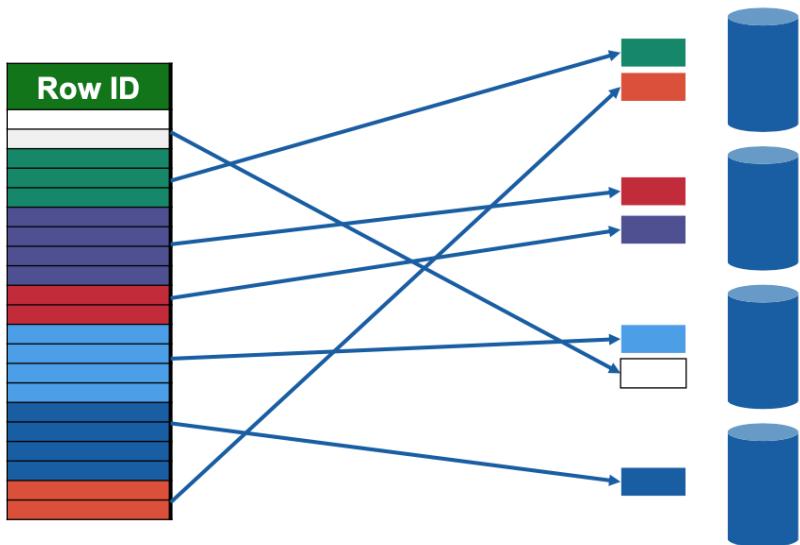


The HMaster and the RegionServers should be understood as processes running on the nodes, rather than the nodes themselves, even though it is common to use “HMaster” to designate the node on which the HMaster process runs, and “RegionServer” to designate a node on which a RegionServer process runs.

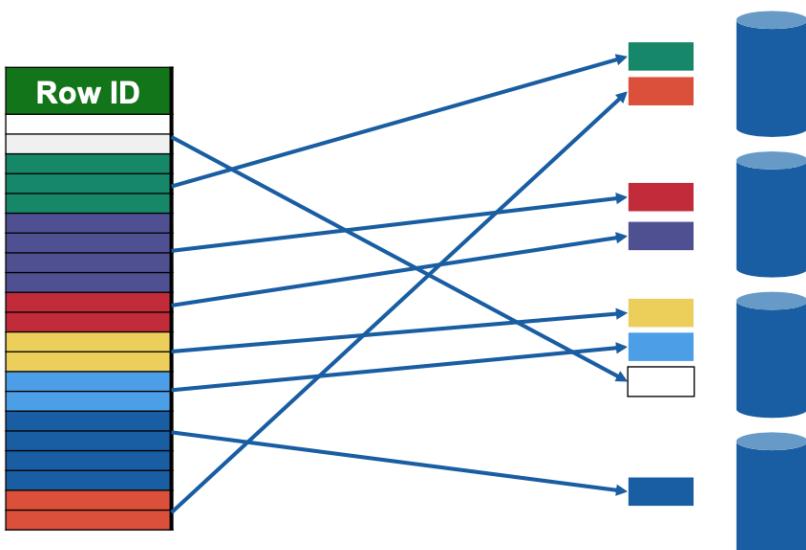
It is common for the HMaster process to run on the same node as the NameNode process. Likewise, it is common for the RegionServer processes to run on those same nodes on which the DataNode processes run.

The HMaster assigns responsibility of each region to one of the RegionServers. This does mean that, for a given region (remember: interval of row IDs), all column families – each one within this region being a store – are handled by the same RegionServer.

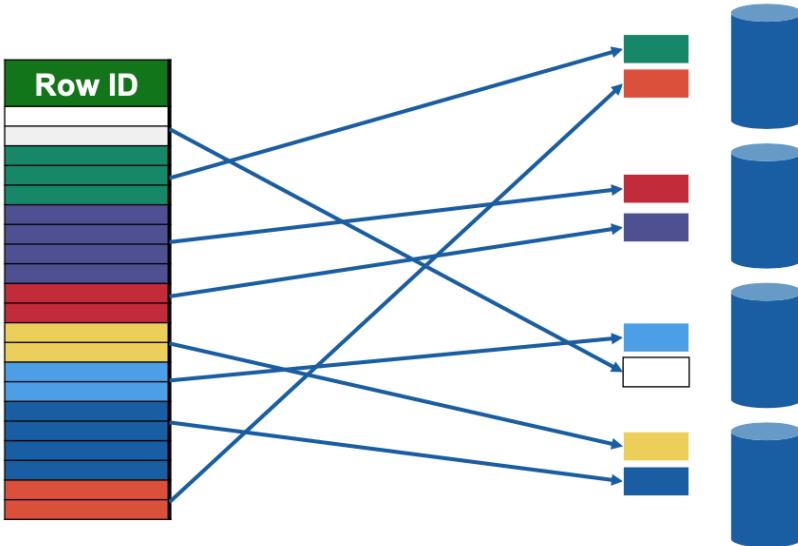
There is no need to attribute the responsibility of a region to more than one RegionServer at a time because, as we will see soon, fault tolerance is already handled on the storage level by HDFS.



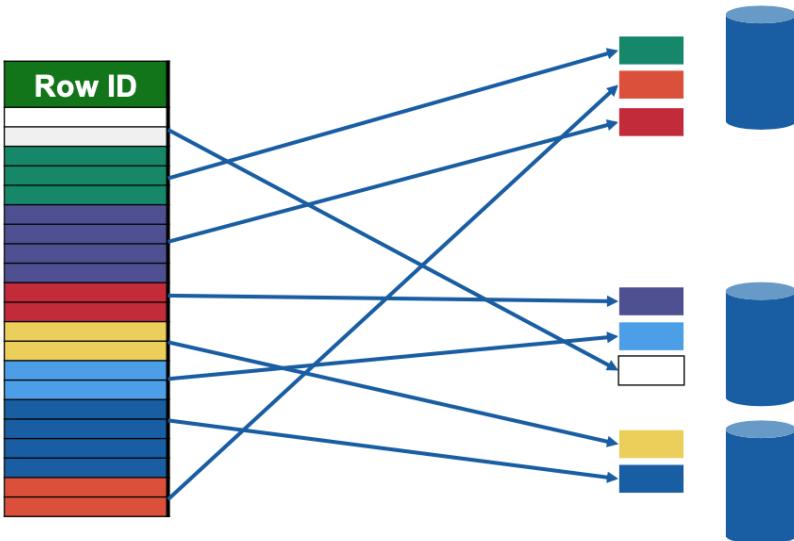
If a region grows too big, for example because of many writes in the same row ID interval, then the region will be automatically split by the responsible RegionServer. Note, however, that concentrated writes (“hot spots”) might be due to a poor choice of row IDs for the use case at hand. There are solutions to this such as salting or using hashes in row ID prefixes.



If a RegionServer has too many regions compared to other RegionServers, then the HMaster can reassign regions to other RegionServers.

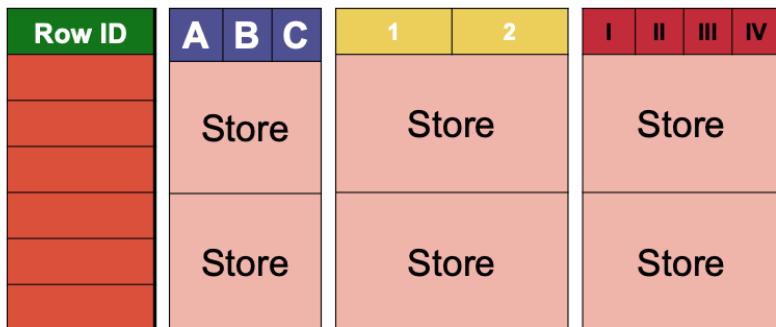


Likewise, if a RegionServer fails, then the HMaster can reassign all its regions to other RegionServers.



6.5.3 Physical storage

Let us now dive into the actual physical storage. As we saw, the data is partitioned in stores, so we need to look at how each store is physically stored and persisted.



The store is, physically, nothing less than an organized set of cells:

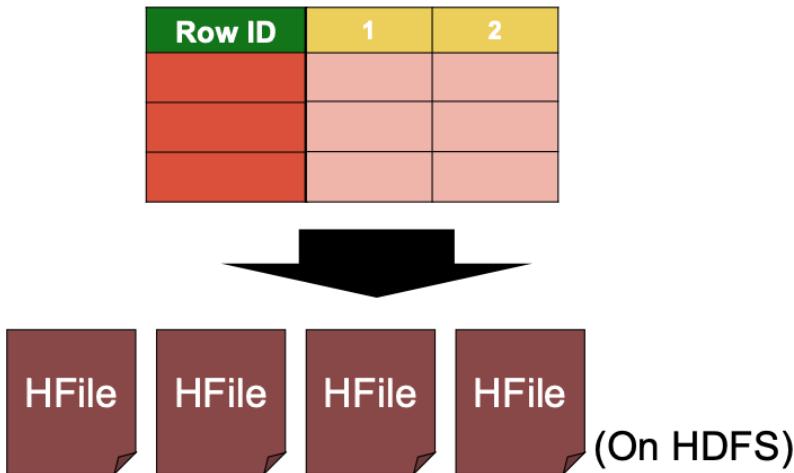
Row ID	1	2
	Cell	

Each value in a cell is identified by a row ID (within the region handled by the store), a column family (the one handled by the store), a column qualifier (arbitrary) and a version (arbitrary). The version is often implicit as several versions of the same cell can co-exist with the latest one being current, but it is an important component in the identification of a value in a cell. This tuple of four values will be referred to as the key (of the value in the cell).

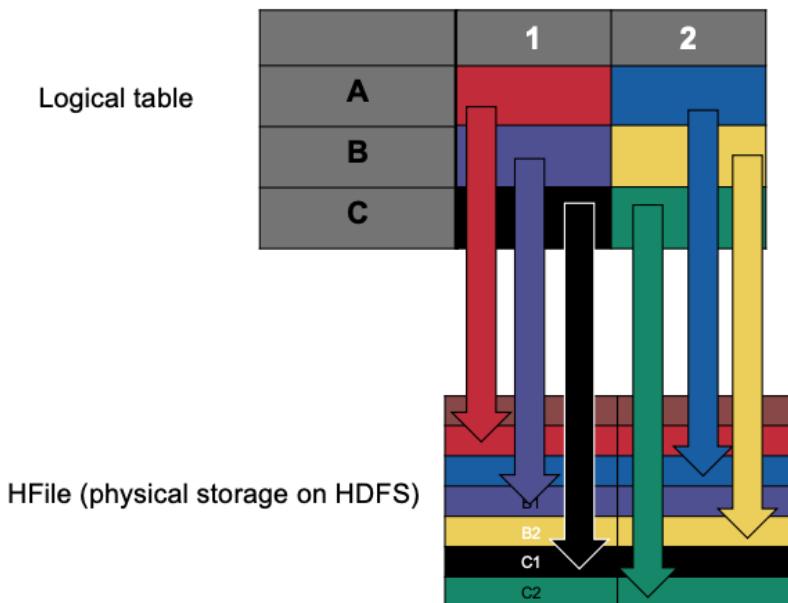
Each value in a cell is thus handled physically as a key-value pair where the key is a (row ID, column family, column qualifier, version)

tuple and the value is its content. On the physical level, a key-value pair is often referred to in CamelCase as a `KeyValue` to disambiguate from other contexts in which key-values might appear within HBase.

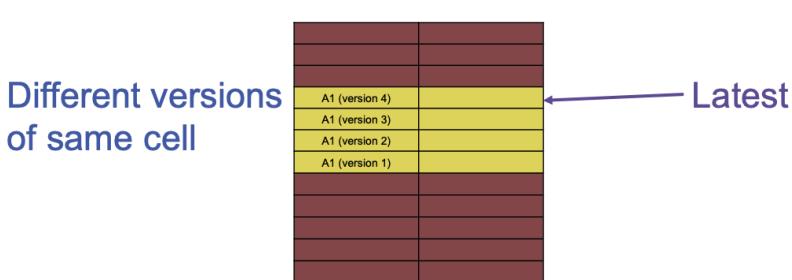
All the cells within a store are eventually persisted on HDFS, in files that we will call `HFfiles`.



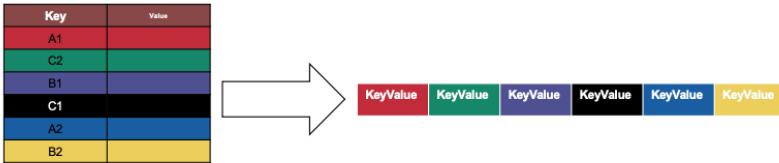
An `HFile` is, in fact, nothing else than a (boring) flat list of `KeyValues`, one per version of a cell. What is important is that, in an `HFile`, all these `KeyValues` are sorted by key in increasing order, meaning, first sorted by row ID, then by column family (trivially unique for a given store), then by column qualifier, then by version (in decreasing order, recent to old).



This means that all versions of a given cell that are in the same HFile are located together, and one of the values (within this HFile) is the latest:

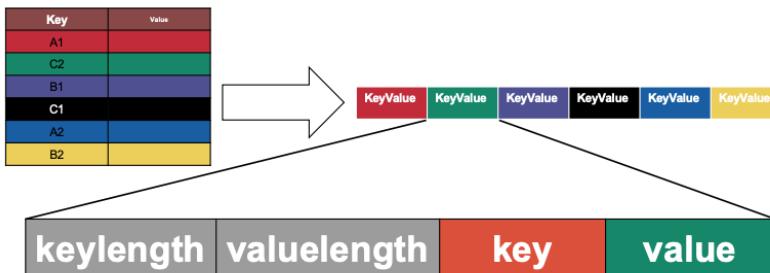


Of course, on the disk, a file is a sequence of 0s and 1s with no tabular structure, so that what in fact happens is that the KeyValues are stored sequentially, like so:



Now if we zoom in at the bit level, a KeyValue consists of four parts:

- The length of the keys in bits (this length is encoded on a constant, known number of bits)
- The length of the value in bits (this length is encoded on a constant, known number of bits)
- The actual key (of variable length)
- The actual value (of variable length)

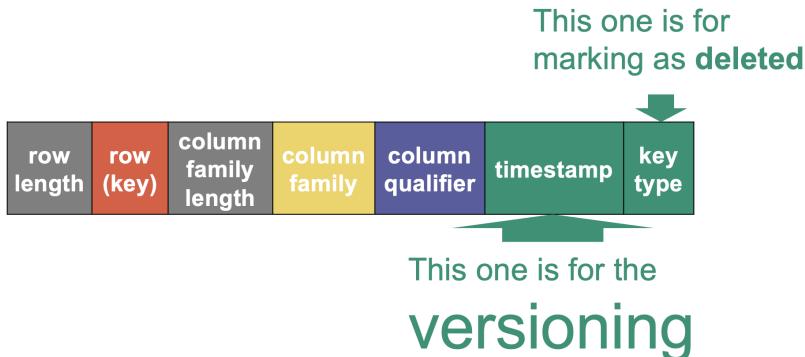


Why do we not just store the key and the value? This is because their length can vary. If we do not know their length, then it is impossible to know when they stop just looking at the bits. Thus, the trick is to start with saving the length of the keys, this length being itself always stored as 32 bits. Once the engine has obtained the length of the key (say, n_k bytes) and the length of the value (say, n_v bytes) from the first 64 bits, it can then look at the next n_k bytes (remember, 1 byte is 8 bits) and stop. We have the key. Then the engine looks at the next n_v bytes and stop. We have the value. And then proceed to the next KeyValue.

Zooming in, the key is itself made of the row ID, the column family, the column qualifier and the timestamp. We need also a row ID length and a column family length (similar to the key length and the value

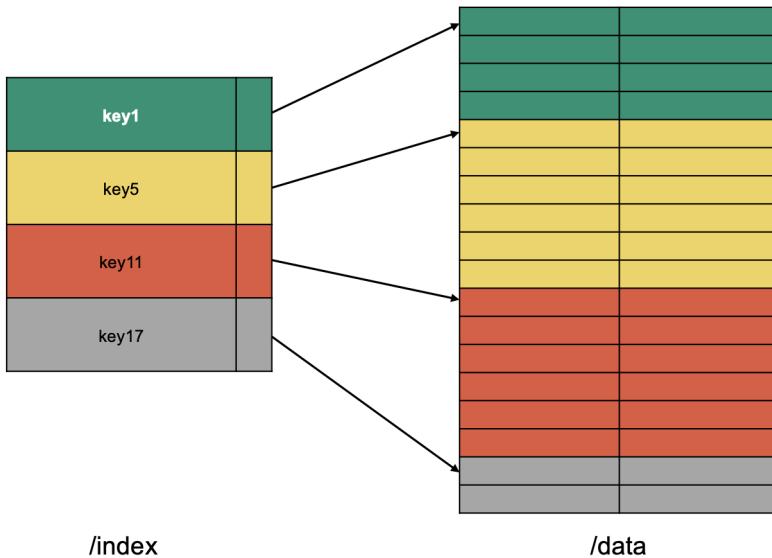
length). The timestamp has a fixed length (64 bits) and does not need additional input on its length. Finally the last byte (named “key type” for some reason) is mostly used as a deletion flag that indicates that the content of the cell, as of this version, was deleted.

Why does the column qualifier not have an additional column qualifier length? This is because we know the key length, so the column qualifier length would be superfluous as it can be deduce with simple arithmetics.



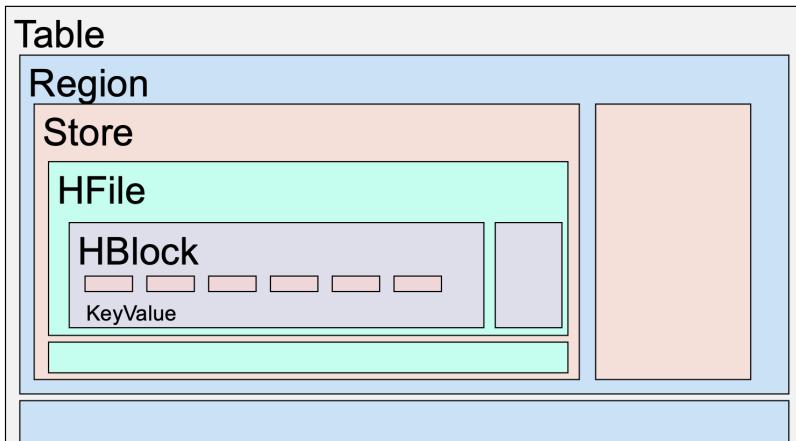
The next thing to know is that KeyValues, within an HFile, are organized in blocks. But to not confuse them with HDFS blocks, we will call them HBlocks. HBlocks have a size of 64 kB, but this size is variable: if the last KeyValue goes beyond this boundary, then the HBlock is simply longer and stops whenever the last KeyValue stops. This is in particular the case for large values exceeding 64 kB, which will be “their own HBlock.”

The HFile then additionally contains an index of all blocks with their key boundaries.



This separate index is loaded in memory prior to reading anything from the HFile. It can then be kept in memory for subsequent reads. Thanks to the index, it is possible to efficiently find out in which HBlock the KeyValues with a specific key (or within a specific key range) are to be read. We will study such indices in more details in Chapter 11.

The following is a summary of the entire physical storage hierarchy of KeyValues on HDFS:

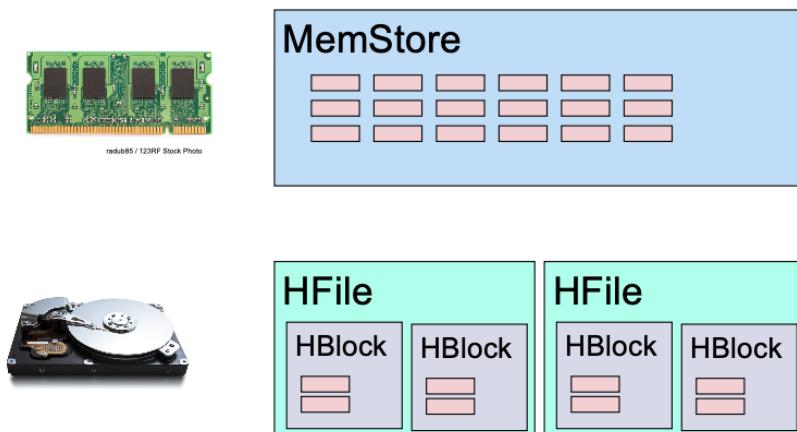


At this point, the astute reader may be wondering: but wait, if all the KeyValues within an HFile are sorted, and it is not possible to modify files on HDFS randomly, how is it even possible to insert any new KeyValues when the HBase user puts values into new cells?

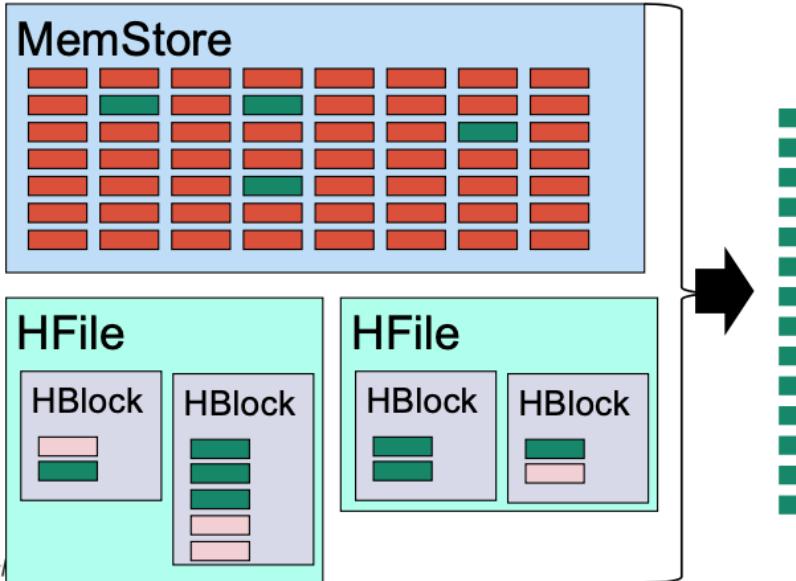
6.5.4 Log-structured merge trees

Before we dive into writing and persisting new data, it is important to understand where the data is read from.

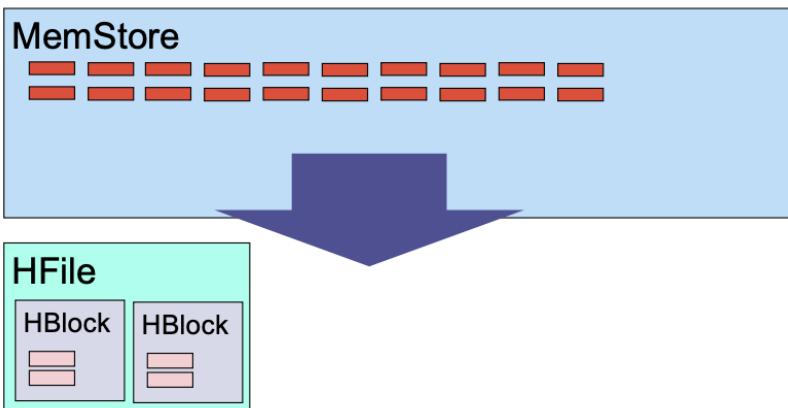
Generally, “old” data is persisted to HDFS in HFiles, while “fresh” data is still in memory on the RegionServer node, and has not been persisted to HDFS yet.



Thus, when accessing data, HBase needs to generally look everywhere for cell values (i.e., physically, KeyValues) to potentially return: in every HFile, and in memory.

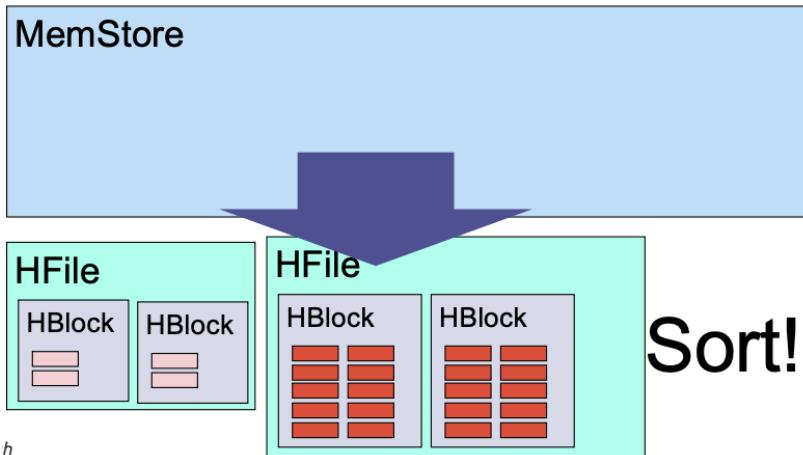


As long as there is room in memory, freshly created KeyValues are added in memory. At some point, the memory becomes full (or some other limits are reached). When this happens, all the KeyValues need to be flushed to a brand new HFile.



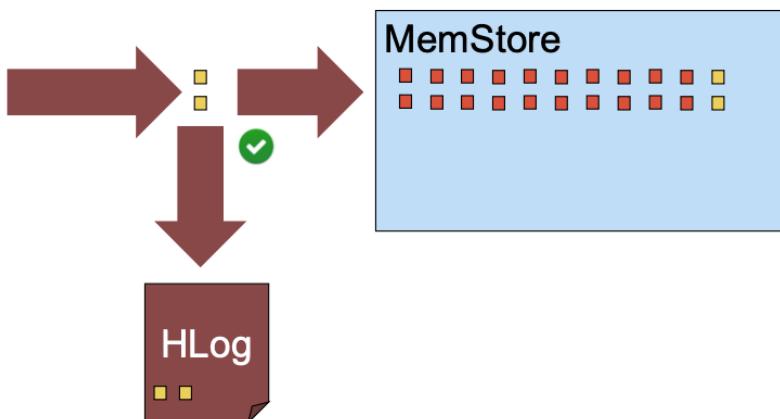
Upon flushing, all KeyValues are written sequentially to a new HFile in ascending key order, HBlock by HBlock, concurrently building the index structure. In fact, sorting is not done in the last minute when flushing. Rather, what happens is that when KeyValues are added to

memory, they are added inside a data structure that maintains them in sorted order (such as tree maps) and then flushing is a linear traversal of the tree.

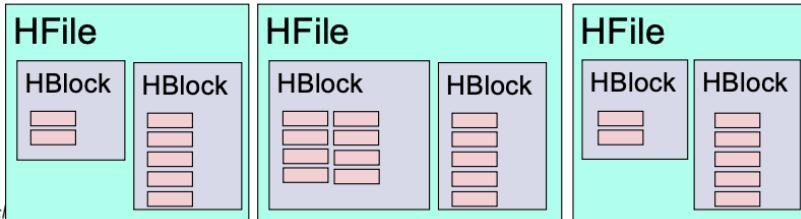


What happens if the machine crashes and we lose everything in memory? We have a so-called write-ahead-log for this. Before any fresh KeyValues are written to memory, they are written in sequential order (append) to an HDFS file called the HLog. There is one HLog per RegionServer. A full write-ahead-log also triggers a flush of all KeyValues in memory to a new HFile.

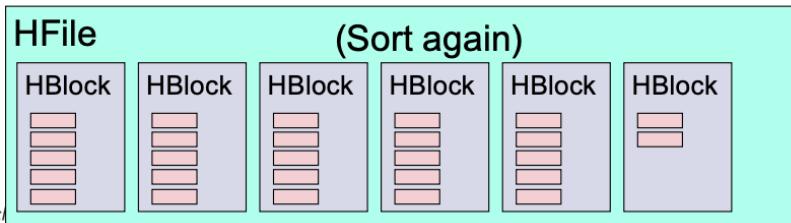
If there is a problem and the memory is lost, the HLog can be retrieved from HDFS and “played back” in order to repopulate the memory and recreate the sorting tree structure.



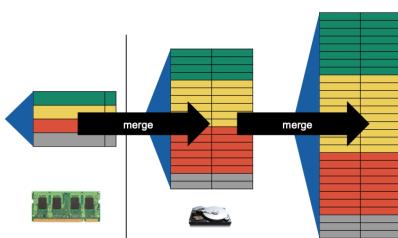
After many flushes, the number of HFiles to read from grows and becomes impractical. For this reason, there is an additional process called compaction that takes several HFiles:



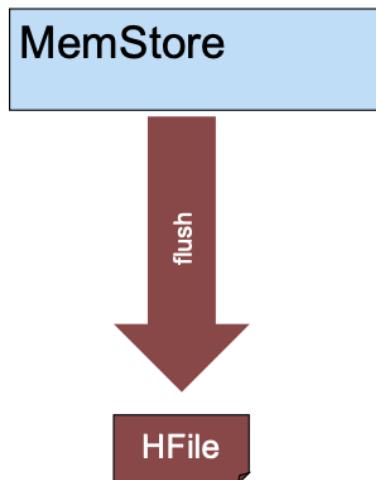
And outputs a single, merged HFile. Since the KeyValues within each HFile are already sorted, this can be done in linear time, as this is essentially the merge part of the merge-sort algorithm.



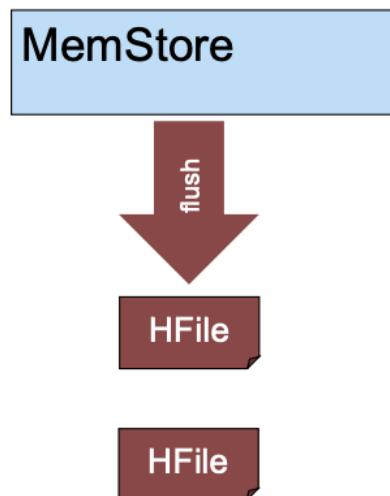
With flushing and compaction, we are starting to see some cycle of persistence, as illustrated below. On a first level, the KeyValues in memory, on a second level, the KeyValues that have been flushed, on the third level, the KeyValues that have been flushed and compacted once, etc.



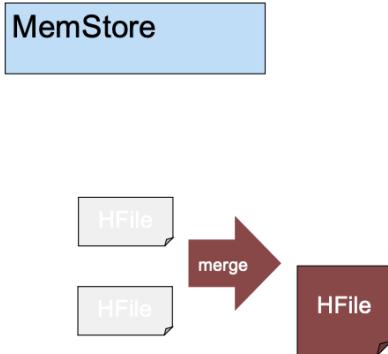
Compaction is not done arbitrarily but follows a regular, logarithmic pattern. Let us go through it. In a fresh HBase store, the memory becomes full at some point and a first HFile is output in a flush.



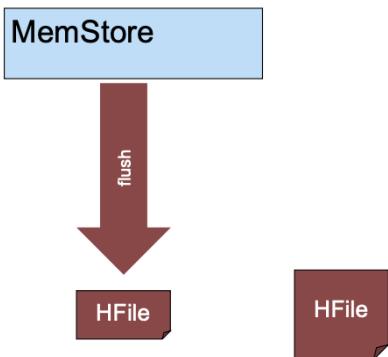
Then the memory, which was emptied, becomes full again and a second HFile is output in a flush.



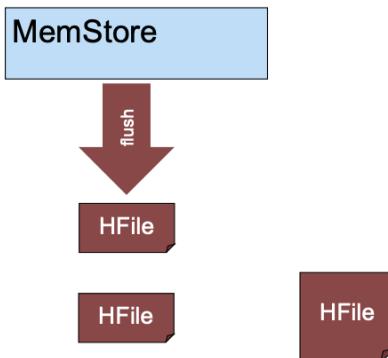
This results in two Hfiles of “standard size” that are immediately compacted to one Hfile, twice as large.



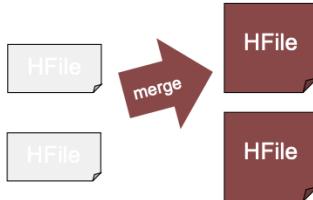
Then the memory, which was emptied, becomes full again and a new “standard-size” HFile is output in a flush.



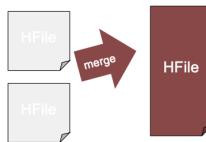
Then the memory, which was emptied, becomes full again and a second “standard-size” HFile is output in a flush.



This results in two Hfiles of “standard size” that are immediately compacted to one Hfile, twice as large.



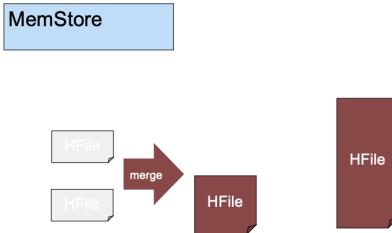
This results in two HFiles of “double size” that are immediately compacted to one HFile, four times as large as the standard size:



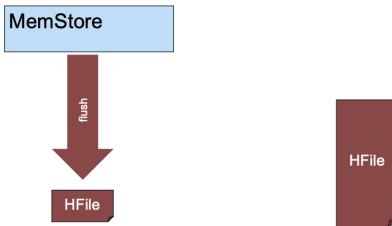
Then the memory, which was emptied, becomes full again and a new “standard-size” HFile is output in a flush.



By now, the process should be clear: when the memory is flushed again, an standard-size HFile is written and the two standard-size HFiles are immediately compacted to a double-size HFile.



When the memory is flushed again, an standard-size HFile is written, and so on, and so on.

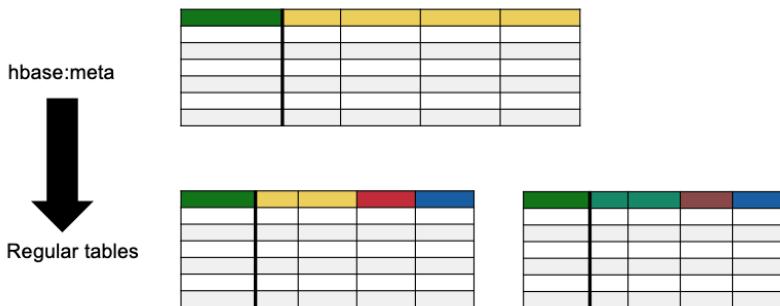


If you paid attention, you will have noticed that the pattern of the HFiles, looked at in a mirror, is simply counting in base 2: 1, 10, 11, 100, 101, 110, 111, and so on. In fact, this number in base two times the size of a standard HFile gives you the total persisted size on HDFS.

6.6 Additional design aspects

6.6.1 Bootstrapping lookups

In order to know which RegionServer a client should communicate with to receive KeyValues corresponding to a specific region, there is a main, big lookup table that lists all regions of all tables together with the coordinates of the RegionServer in charge of this region as well as additional metadata (e.g. to support splitting regions, etc).



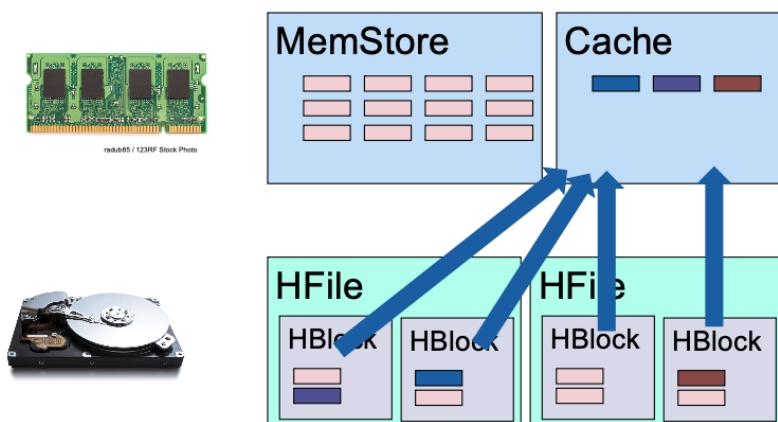
This big table is, in fact, also an HBase table, but it is special because this one fits on just one machine, known to everybody. Thus, the clients use this so-called meta table to know which RegionServers to communicate with.

There also exists an alternate design (commonly found in early versions of BigTable or HBase) with two levels of meta tables in order to scale up the number of tables.

To create, delete or update tables, clients communicate with the HMaster.

6.6.2 Caching

In order to improve latency, KeyValues that are normally persisted to HFiles (and thus no longer in memory) can be cached in a separate memory region, with the idea of keeping in the cache those KeyValues that are frequently accessed. We will not go into much details about the cache here and refer to computer architecture textbooks.

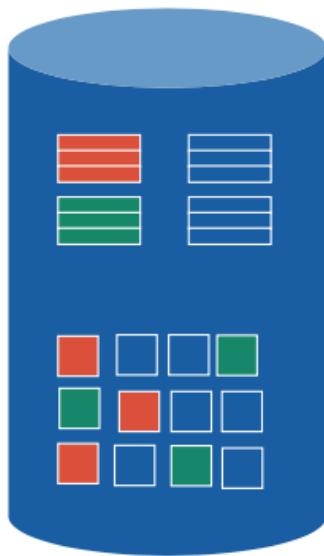


6.6.3 Bloom filters

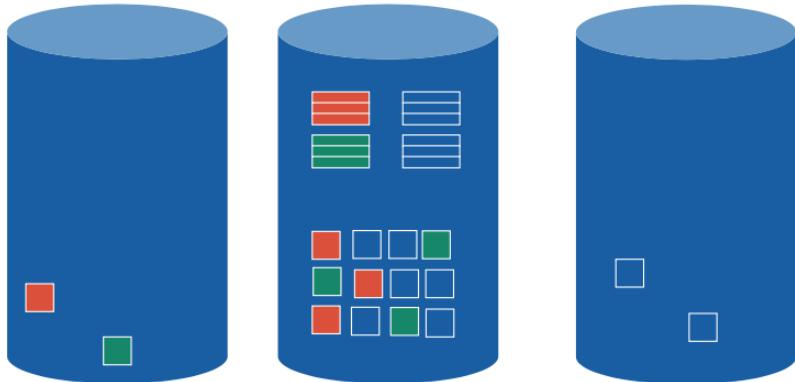
HBase has a mechanism to avoid looking for KeyValues in *every* HFile. This mechanism is called a Bloom filter. It is basically a black box that can tell with absolute certainty that a certain key does not belong to an HFile, while it only predicts with good probability (albeit not certain) that it does belong to it. A Bloom filter is implemented using a multiple hashing mechanism that sets Boolean flags in an array, which is very efficient. By maintaining Bloom filters for each HFile (or even each column), HBase can know with certainty that some HFiles need not be read when looking up certain keys.

6.6.4 Data locality and short-circuiting

It is informative to think about the interaction between HBase and HDFS. In particular, recollect when we said that HDFS outputs the first replica of every block on the same (DataNode) machine as the client. Who is the client here? The RegionServer, which does co-habit with a DataNode. Now the pieces of the puzzle should start assembling in your mind: this means that when a RegionServer flushes KeyValues to a new HFile, a replica of each (HDFS) block of the HFile is written, by the DataNode process living on the same machine as the RegionServer process, to the local disk. This makes accessing the KeyValues in future reads by the RegionServer extremely efficient, because the RegionServer can read the data locally without communicating with the NameNode: this is known as short-circuiting in HDFS.



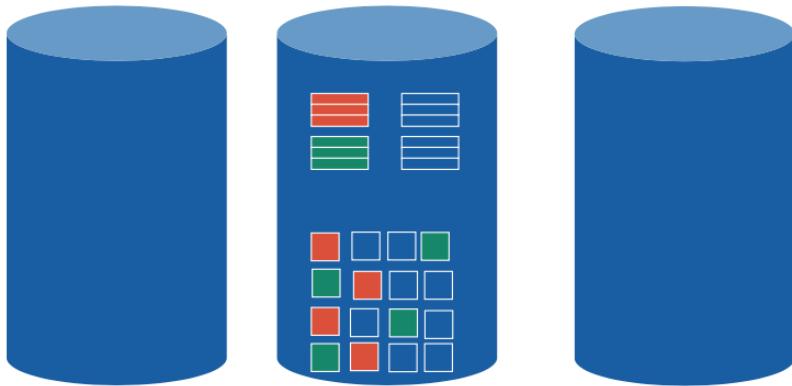
However, as time flies and the HDFS cluster lives its own life, some replicas might be moved to other DataNodes when rebalancing, making short-circuiting not (always) possible¹.



This, however, is not a problem, because with the log-structured merge tree mechanism, compactions happen regularly. And with every

¹Trick to remember: quantum physicists might see here something akin to a wave function spreading to a broader support over time!

compaction, the replicas of the brand new HFile are again written on the local disk².



²Trick to remember: quantum physicists might see here something akin to a quantum measurement with wave function collapse!

6.7 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain the limitations of the traditional relational model?
- b. Can you explain the differences and similarities between a wide column store and the traditional relational model? And between a wide column store and object storage?
- c. Can you explain why wide column stores are called wide column stores, in particular on the storage level?
- d. Do you know and can you contrast the two ways data can be distributed (partitioning, replicating)?
- e. Can you explain the data model behind wide column stores, in particular, rows, columns, column families, versions, as well as cells and their values?
- f. Can you explain the motivation behind data getting denormalized into several column families?
- g. Can you explain what aspects of a table in a wide column store must be typically known in advance, and which can be changed on the fly?
- h. Can you name a few big players, in particular the one behind the initial founding paper?
- i. Can you explain why HBase is based on HDFS, yet low-latency?
- j. Can you explain what regions are for wide column stores?
- k. Do you know how to identify a region based on the content of the wide column store?
- l. Do you know the four basic kinds of (low-level) queries in HBase?
- m. Can you describe the physical architecture of a wide column store like HBase, and compare it with that of a distributed file system like HDFS?
- n. Do you know the physical layers of HBase (table, region, Store, Memstore, HFile, HBase block (aka HBlock), Write-Ahead-Log (aka WAL, HLog), KeyValue)?
- o. Do you know the difference between an HDFS block and an HBase block (aka HBlock) as well as their typical sizes?

- p. Can you explain how new cells are written to HBase via the cell store?
- q. Can you explain how cells are read from HBase via both the stored HFiles and the Memstore?
- r. Can you explain what compaction and flushing is?
- s. Can you sketch the contents of an HFile and know what it corresponds to in an HBase table?
- t. Can you explain why it is crucial not to have too long column family names?
- u. Can you populate data into, and execute queries on top of HBase?
- v. Can you use the HBase shell to type simple commands?

6.8 Literature and recommended readings

The following is a list of recommended material for further reading and study.

- Chang, F. et al. (2006). *Bigtable: A Distributed Storage System for Structured Data*. In OSDI.
- George, L. (2011). *HBase: The Definitive Guide*. 1st edition. O'Reilly. Chapters 1, 3, 8.
- White, T. (2015). *Hadoop: The Definitive Guide*. 4th edition. O'Reilly. Chapter 20.

Chapter 7

Data models and validation

Even though the data is physically stored as bits – or as text directly encoded to bits in the case of XML and JSON – it would not be appropriate to directly manipulate the data at the bit or text level. This is, in fact, in the spirit of data independence to abstract away. Doing so is called data modelling.

A data model is an abstract view over the data that hides the way it is stored physically. For example, a CSV file should be abstracted logically as a table. This is because CSV enforces at least relational integrity as well as atomic integrity. As for domain integrity, this can be considered implicit since an entire column can be interpreted as a string in the case of incompatible literals.

Logical view Data Model	ID	Last name	First name	Theory
	1	Einstein	Albert	General, Special Relativity
	2	Gödel	Kurt	"Incompleteness" Theorem

This is a data model

Physical view

Syntax

```
ID,Last name,First name/Theory,  
1,Einstein,Albert,"General, Special Relativity"  
2,Gödel,Kurt,"""Incompleteness"" Theorem"
```

So how do we do the same for JSON and XML?

7.1 The JSON Information Set

Obviously, a model based on tables is not appropriate for JSON. This is because, unlike CSV, JSON enforce neither relational integrity, nor atomic integrity, nor domain integrity. In fact, we will see that the appropriate abstraction for any JSON document is a tree.

The nodes of that tree, which are JSON logical values, are naturally of six possible kinds: the six syntactic building blocks of JSON.

These are the four leaves corresponding to atomic values:

- Strings
- Numbers
- Booleans
- Nulls

As well as two intermediate nodes (possibly leaves if empty):

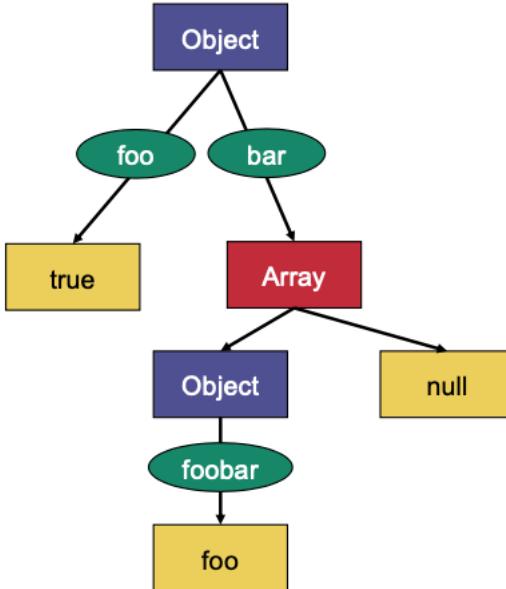
- Objects (String-to-value map)
- Arrays (List of values)

Formally, and not only for JSON but for all tree-based models, these nodes are generally called *information items* and form the logical building blocks of the model, called *information set*.

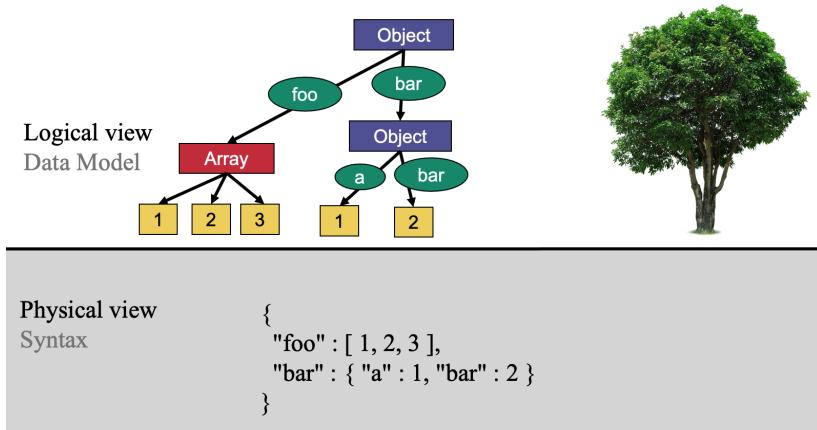
Let us take the following example.

```
{
  "foo" : true,
  "bar" : [
    {
      "foobar" : "foo"
    },
    null
  ]
}
```

It is possible to draw this document as a logical tree, where each information item (node) corresponds to each one of the values present in the document: two objects, one array, and three atomics. Note that the information items are the rectangles; the ovals are not information items but labels on the edges connecting the information items. The ovals correspond to object keys.



It is possible to do so for any JSON document. Thus, we have now obtained a similar logical/physical mapping to what we previously did with CSV and tables, except that this is now with JSON and trees:

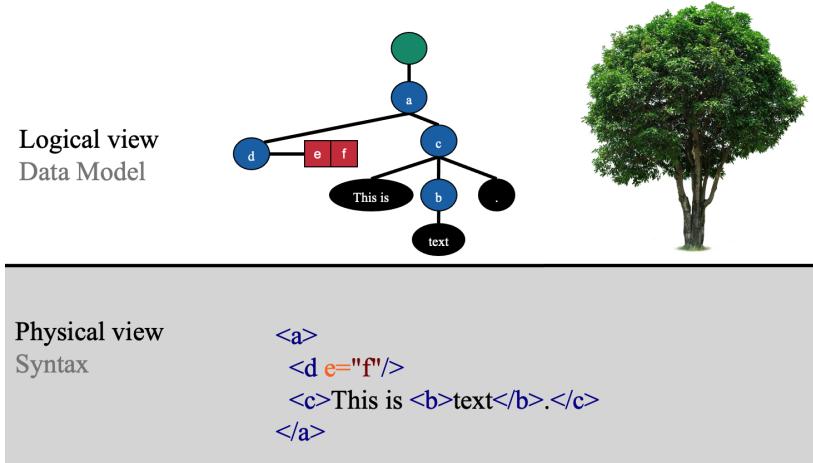


When a JSON document is being parsed by a JSON library, this tree is built in memory, the edges being pointers, and further processing will be done on the tree and not on the original syntax.

Conversely, it is possible to take a tree and output it back to JSON syntax. This is called *serialization*.

7.2 The XML Information Set

It is possible to do the same logical abstraction, also based on trees, with XML, where information items correspond to elements, attributes, text, etc:



A fundamental difference between JSON trees and XML trees is that for JSON, the labels (object keys) are on the *edges* connecting an object information item to each one of its children information items. In XML, the labels (these would be element and attribute names) are on the *nodes* (information items) directly. Another way to say it is that a JSON information item *does not know* with which key it is associated in an object (if at all), while an XML element or attribute information item *knows* its name.

Let us dive more into details. In XML, there are many more information items:

- Document information items
- Element information items
- Attribute information items
- Character information items
- Comment information items

- Processing instruction information items
- Namespace information items
- Unexpanded entity reference information items
- DTD information items
- Unparsed entity information items
- Notation information items

For the purpose of this course, though, we will only go into the most important ones from a data perspective: documents, elements, attributes, and characters. We will leave comments and namespaces aside to keep things simple, even though we saw what they look like syntactically, and will also skip all other information items, for which we have not studied the syntax.

Let us take this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
<metadata>
    <title
        language="en"
        year="2019">Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

which we can also highlight with colors to ease the read (any editor will do this with a well-formed XML document):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
<metadata>
    <title
        language="en"
        year="2019">Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Formally, the XML Information Set is defined in a standard of the World Wide Web consortium (W3C). Each kind of information item has specific properties, and some of these properties link it to other information items, building the tree.

Let us go through the information items for the above document and list some of its properties.

7.2.1 Document information item

The document information item is just the root of an XML tree. It does not correspond to anything syntactically or, if at all, it would correspond to the text and doctype declarations.

The documentation information has two important properties:

- [children] Element information item *metadata*
- [version] 1.0

7.2.2 Element information items

There is one element information item for each element. Here we have three.

The element information item metadata has four important properties:

- [local name] metadata
- [children] Element information item title, element information item publisher
- [attributes] (empty)
- [parent] Document information item

The element information item title has four important properties:

- [local name] title
- [children] Character information items (Systems Group)
- [attributes] Attribute information item language, Attribute information item year
- [parent] Element information item metadata

The element information item publisher has four important properties:

- [local name] publisher

- [children] Character information items (ETH Zurich)
- [attributes] (empty)
- [parent] Element information item metadata

7.2.3 Attributes information items

There is one attribute information item for each attribute. Here we have two.

The attribute information item language has three important properties:

- [local name] language
- [normalized value] en
- [owner element] Element information item title

The attribute information item year has three important properties:

- [local name] year
- [normalized value] 2019
- [owner element] Element information item title

7.2.4 Character information items

There are as many character information items as characters in text (between tags). For example, for the S in Systems Group:

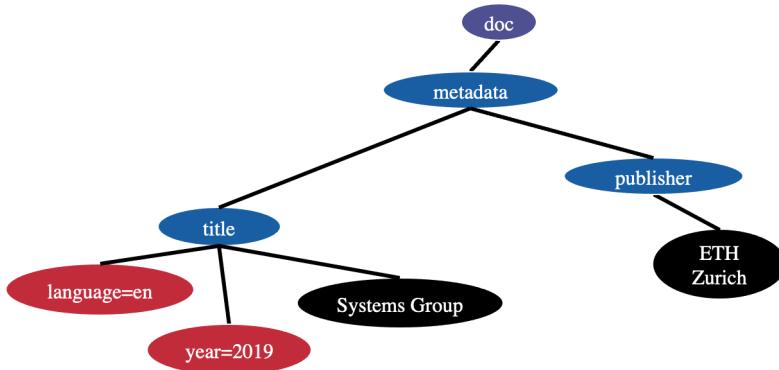
- [character code] the unicode code point for the letter S
- [parent] Element information item title

It is sometimes simpler to group them into a single (non standard) “text information item”:

- [characters] S y s t e m s G r o u p
- [parent] Element information item title

7.2.5 The entire tree

All information items built previously can finally be assembled and drawn as a tree. The edges, corresponding to children and parent (or owner element) properties, will correspond to pointers in memory when the tree is built by the XML library:



When an XML document is being parsed by a XML library, this tree is built in memory, the edges being pointers, and further processing will be done on the tree and not on the original syntax.

Conversely, it is possible to take a tree and output it back to XML syntax. This is called *serialization*.

7.3 Validation

Once documents, JSON or XML, have been parsed and logically abstracted as a tree in memory, the natural next step is to check for further structural constraints.

For example, you could want to check whether your JSON documents all associate key “name” with a string, or if they all associate “years” with an array of positive integers. Or you could want to check whether your XML documents all have root elements called “persons,” and whether the root element in each document has only children elements called “person,” all with an attribute “first” and an attribute “last.”

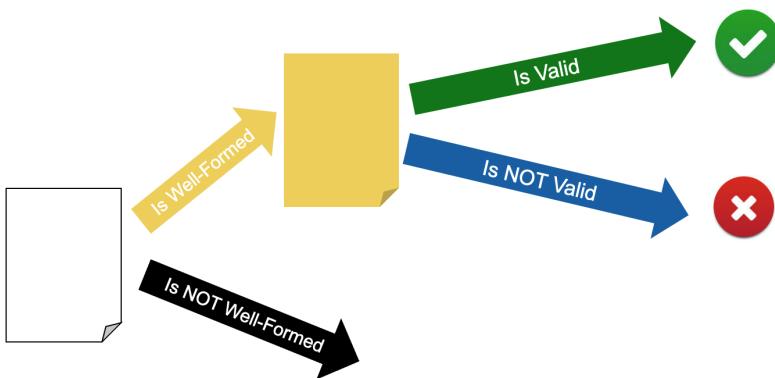
This might remind the reader of schemas in a relational database, but with a major difference: in a relational database, the schema of a table is defined before any data is populated into the table. Thus, the data in the table is guaranteed, at all times, to fulfil all the constraints of the schema. The exact term is that the data is guaranteed to be

valid against the schema, because the schema was enforced at write time (schema on write).

But in the case of a collection of JSON and XML documents, this is the other way round. A collection of JSON and XML documents out there can exist without any schema and contain arbitrary structures. Validation happens “ex post,” that is, only after reading the data (schema on read).

Thus, it means that JSON and XML documents undergo two steps:

- a well-formedness check: attempt to parse the document and construct a tree representation in memory
- (if the first step succeeded) a validation check given a specific schema



Thus, a text document can be either not well-formed, or well-formed and invalid against a specific schema, or well-formed and valid against a specific schema.

Note that, unlike well-formedness, validation is schema dependent: a given well-formed document can be valid against schema A and invalid against schema B.

Validation is often performed on much more than a document at a time: an entire collection. Thus, we distinguish between heterogeneous collections, whose documents are not valid against any particular schema, and homogeneous collections, whose documents are all valid against a specific schema.



To give an intuitive feeling, this is what a not-validated, “messy” document looks like. Notice in particular how values in the same array are of different types.

```
{  
    "a" : 1,  
    "b" : [  
        "foo",  
        true,  
        null,  
        {  
            "foo" : "bar"  
        }  
    ],  
    "c" : {  
        "d" : { "foo" : null },  
        "e" : [ 1, 2, [ 3, 4 ] ],  
        "f" : 3.14  
    }  
}
```

This, on the other hand, is a document that has more structure and could easily be validated against an appropriate schema: for example, the array associated with key “c” only has object elements, which all have a “foo” key associated with a string and an optional “bar” key associated with an array of integers.

```
{
  "a" : 1,
  "b" : true,
  "c" : [
    { "foo":"bar1", "bar":[1,2] },
    { "foo":"bar2", "bar": [ 3,4,5 ] },
    { "foo" : "bar3" }
  ]
}
```

7.4 Item types

A fundamental aspect of validation is the type system. A well-designed type system, in turn, allows for storing the data in much more efficient, binary formats tailored to the model.

There are many different technologies and type systems for arborescent, denormalized data, well beyond only JSON and XML. But there is good news: all these type systems are very similar to each other and have a lot in common. Therefore, in this section, we will present an overview of an agnostic type system that is very representative of common practice. After reading this, learning a new validation or storage format technology will then amount to reading the documentation and immediately recognizing familiar patterns, to then focus on the small deviations and peculiarities of each specific technology.

The first aspect that almost all, if not all type systems, have in common, is the distinction between atomic types and structured types. In fact, this distinction is so universal that even object-oriented languages like Java and Python make such a distinction.

The distinction should also not really come as a surprise to the reader, because we have seen it several times before: in Chapter 5 and in this one, when we looked at JSON and XML.

7.4.1 Atomic types

Atomic types correspond to the leaf of a tree data model: these are types that do not contain any further nestedness.

The kinds of atomic types available are also relatively standard and common to most technologies.

Also, all atomic types have in common that they have a logical value space and a lexical value space. The logical value space corresponds to the actual logical abstraction of the type (e.g., a mathematical set of integers), while the lexical value space corresponds to the representation of logical values in a textual format (such as the decimal, or binary, or hexadecimal representation of an integer).

An atomic type also has a (not necessarily injective) mapping from its lexical value space to its logical value space (e.g., mapping the hex-adecimal literal `x10` to the mathematical integer sixteen), and often, a canonical mapping from the logical value space to the lexical value space (e.g., mapping the mathematical integer sixteen to its canonical decimal representation 16).

Atomic types can be in a subtype relationship: a type is a subtype of another type if its logical value space is a subset of the latter. Normally it means that the same holds for the lexical value spaces and the related mappings should be consistent with each other. The subtype relationship over types organizes the types as a hierarchy, called the type hierarchy. The type hierarchy gives a good visual to get a quick overview of all available types in a specific technology.

Let us take a tour.

Strings

Strings are simply finite sequences of (usually printable) characters. Formally, strings form a monoid under concatenation, where the neutral element is the empty string.

These are three examples:

```
foo
Zurich
Ilsebill salzte nach.
```

Often, the lexical representation of a string is double-quoted, sometimes also single-quoted.

```
"foo"
"Zurich"
'Ilsebill salzte nach.'
```

The difference between the lexical representation and the logical value of a string becomes immediately apparent when escaping is used. For example, the lexical representation

```
"\\\""
```

corresponds to the (logical) string `\\"`.

In “pure computer science” textbooks, strings are often presented as structured values rather than as atomic values because of their complexity on the physical layer. However, for us data scientists, strings are atomic values.

Numbers: integers

Integers correspond to finite cardinalities (counting) as well as their negative counterparts. These are decimal numbers without anything after the decimal period, or fractions with a denominator of 1.

In older programming languages, support for integers used to be bounded. This is why classical types, still in use today, correspond to 8-bit (often called byte), 16-bit (often called short), 32-bit (often called int) and 64-bit integers (often called long). This means that, expressed in base 2, they use 8, 16, 32 or 64 bits (binary digits).

However, in modern databases, it is customary to support unbounded integers. In XML, the corresponding type is simply called integer, as opposed to int. Engines can optimize computations for small integers, but might become less efficient with very large integers.

The other, restricted integer types are called *subtypes* of the integer type, because their logical value space is a subset of the set of all integers.

Other commonly found integer subtypes include positive integers, non-negative integers (also called unsigned integers), negative integers, non-positive integers.

The lexical representation of integers is usually done in base 10, in the familiar decimal system, even though base 2, 8 or 16 can be found, too. Leading 0s are optional, but when an logical integer value is canonically serialized, it is done without a leading 0.

Note that the exact names of the types can vary! Some systems might use integer for 32-bit integers or int for the entire value space. You might need to read the documentation of the specific technology you are using if in doubt.

Numbers: decimals

Decimals correspond to real numbers that can be written as a finite sequence of digits in base 10, with an optional decimal period. Equivalently, these are fractions that can be expressed with a power of 10 in the denominator.

Many modern databases or storage formats support the entire logical decimal value space with no restriction on how large, small or precise a decimal number is.

The lexical representation of integers is usually done in base 10, it is not common to use other bases for decimals.

Numbers: floating-point

Support for the entire decimal value space can be costly in performance. In order to address this issue, a floating-point standard (IEEE 754) was invented and is still very popular today. These are the types known

as float and double in many programming languages. They can be processed natively by processors.

Floating-point numbers are limited both in precision and magnitude (both upper and lower) in order to fit on 32 bits (float) or 64 bits (double). Floats have about 7 digits of precision and their absolute value can be between roughly 10^{-37} and 10^{37} , while doubles have 15 digits of precision and their absolute value can be between roughly 10^{-307} and 10^{308} .

Double and float types also cover additional special values: NaN (not a number), positive and negative infinity, and negative zero (in addition to the “positive” 0).

Note that the exact names of the types can vary! Some systems might use float for double, or decimal for double, etc. You might need to read the documentation of the specific technology you are using if in doubt.

The lexical representation of floats and doubles often use the scientific notation:

`-12.34E-56`

And the lexical values corresponding to the special logical values look like so:

`NaN`
`INF`
`-INF`
`-0`

Booleans

The logical value space for the Boolean type is made of two values: true and false as in NoSQL queries, two-valued logic is typically assumed.

The corresponding lexical values are typically:

`true`
`false`

If an unknown value is needed in the spirit of three-valued logic, the null can be used (see below) or it can be left absent.

Dates and times

Dates and times are a very important component of databases because they are heavily needed by users, albeit often neglected or forgotten by some formats.

Dates are commonly using the Gregorian calendar (with some technologies possibly supporting more) with a year (BC or AD), a month and a day of the month.

Times are expressed in the hexagesimal (60) basis with hours, minutes, seconds, where the seconds commonly go all the way to microseconds (six digits after the decimal period).

Datetimes are expressed with a year, a month, a day of the month, hours, minutes and (decimal) seconds.

Some technologies offer types that support time zones (UTC, CEST, PDT...) or the explicit absence of any time zone for dates, times and datetimes.

The timestamp type corresponds to a datetime with a timezone, but treating datetimes as equivalent if they express the same point in time (formally, it means that it is a timezoned datetime quotiented with an equivalence relation). Timestamp values are typically stored as longs (64-bit integers) expressing the number of milliseconds elapsed since January 1, 1970 by convention.

The lexical values can also vary, although many technologies follow the ISO 8601 standard, where lexical values look like so (with many parts optional):

2022-08-07T14:18:00.123456+02:00

2022-08-07

14:18:00.123456

14:18:00.123456Z

The names of date, time, datetime and timestamp types vary largely between technologies and formats. For the purpose of this lecture, we will focus on the standardized XML Schema types, which are also the same as in JSound and the JSONiq language that we will study later. The types are called date, time, dateTime and dateTimeStamp. XML Schema additional supports values made of just a year (gYear), just a month (gMonth), just a day of the month (gDay), or a year and month (gMonthYear), or a month and day of the month (gMonthDay). XML Schema, JSound and JSONiq follow the ISO 8601 standard.

Durations

Durations can be of many different kinds, generally a combination of years, months, days, hours, minutes and (possibly with decimals) seconds.

What is important to understand is that there is a “wall” between months and days: what is the duration “1 month and 1 day?” It could be 29, 30, 31, or 32 days and should thus be avoided. Thus, most durations, for the sake of being unambiguous, are either involving years

and/or months, or are involving days and/or hours and/or minutes and/or seconds.

The lexical representation can vary, but there is a standard defined by ISO 8601 as well, starting with a P and prefixing sub-day parts with a T.

For example 2 years and 3 months:

P2Y3M

4 days, 3 hours, 2 minutes and 1.123456 seconds:

P4DT3H2M1.123456S

3 hours, 2 minutes and 1.123456 seconds:

PT3H2M1.123456S

2022-08-07T14:18:00.123456+02:00

2022-08-07

14:18:00.123456

14:18:00.123456Z

XML Schema, JSound and JSONiq, used in this course, follow the ISO 8601 standard.

Binary data

Binary data is, logically, simply a sequence of bytes.

There are two main lexical representations used in data: hexadecimal and base64.

Hexadecimal expresses the data with two hexadecimal digits per byte, like so:

0123456789ABCDEF

which would correspond to the bit sequence:

0000000100100011010001010110011110001001101010111100110111
101111

Base 64, formally, does the same but in the base 64, which “wastes” less lexical space in the text. It does so by encoding the bits six by six, encoding each sequence of six bits with one base-64 digit. Equivalently, it means that each group of three bytes is encoded with four base-64 digits. The base-64 digits are, by convention, decimal digits, all uppercase latin alphabet letters, all lowercase latin alphabet letters, as

well as + and /. = is used for padding if the length of the base-64 string is not a multiple of four.

This is the base-64 lexical representation of the same binary data as above, which is textually more compact than the hexadecimal version:

```
ASNfZ4mrze8=
```

In XML Schema, JSound and JSONiq, this is covered with two types hexBinary and base64Binary.

Note that the string type could also be considered to provide an additional lexical representation of a binary type (e.g., with the UTF-8 encoding), although this can cause issues with non-printable characters.

Null

Many technologies and formats also provide support for null values, although how this is done largely varies.

Some technologies allow null to appear as a (valid) value for *any* type. A schema can either allow, or disallow the null value. Often, the terminology used is that a type can be nullable (or nillable) or not.

Other technologies consider that there is a singleton-valued null type, containing only the null value with the lexical representation

```
null
```

Allowing nulls is done by taking the union of the desired type with the null type.

Yet other technologies consider null when it appears as a value in an object to be semantically equivalent by the absence of a value and then, allowing or disallowing null is achieved by (e.g. in JSON) making the field required or optional.

It is important to understand that the latter technologies are unable to distinguish between the following two JSON objects, so that information in the input dataset is lost upon validating:

```
{ }
{ "foo" : null }
```

This can be problematic with datasets where this distinction is semantically relevant.

XML also supports null values, but calls them “nil” and does so with a special attribute and no content rather than with a lexical representation

```
<foo
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:nil="true"/>
```

7.4.2 Structured types

Most technologies and formats offer four kinds of structured types (or a subset thereof).

Lists

Lists correspond to JSON arrays and are ordered sequences of (atomic or structured) values.

With a schema, it is possible to defines types that are, for example, lists or integers, or lists of strings, or lists of lists of doubles, etc.

Records

Records, or structs, correspond to JSON objects and are maps from strings to values.

With a schema, it is possible to restrict the type accepted for every key, e.g., name must be a string, birthday must be a date, etc.

Maps

Maps (not be confused with records, which are similar) are maps from any atomic value to any value, i.e., generalize objects to keys that are not necessarily strings (e.g., numbers, dates, etc).

With a schema, it is possible to restrict the type of the keys, as well as the type of the values. However, unlike records, the type of the values must be the same for all keys. For example, a map from dates to numbers.

Sets

Sets are rarer and supported by rather few technologies and formats, but they exist.

Sets are like lists, but without any specific ordering, and without duplicate values.

XML elements and attributes

XML Schema stands apart from most other technologies and formats, in that it does not offer specific support for records and maps; it offers some limited support for lists, but considers them to be simple types, which are “inbetween” atomic types and structured types. In XML Schema, structure is obtained, instead, with elements and attributes, and the machinery for elements and attributes is highly specific to XML. Elements and attributes can in particular easily emulate records and lists, but are more powerful (i.e., elements can be repeated, and intermixing text with elements is allowed).

Most formats besides XML are “JSON-like” from a modelling perspective and structure their data at the very minimum with lists and records.

Type names

For convenience, we provide below a summary of many types over various technologies and languages and how they correspond to each other. The most important thing to see here is that on the high level, atomic types are almost always the same everywhere, even though the names can vary.

JSound/JSONiq	JSON Schema	XML Schema	SparkSQL	PostgreSQL	Python
decimal	number	xs:decimal	DecimalType	numeric(..)	-
integer	integer	xs:integer	-	numeric(.)	int
byte	integer +range	xs:byte	ByteType	-	-
short	integer +range	xs:short	ShortType	smallint	-
int	integer +range	xs:int	IntegerType	int	-
long	integer +range	xs:long	LongType	bigint	-
double	number	xs:double	DoubleType	double precision	float
float	-	xs:float	FloatType	real	-
string	string	xs:string	StringType	text	str
boolean	boolean	xs:boolean	BooleanType	boolean	bool
hexBinary	string +pattern	xs:hexBinary	BinaryType	bytea	bytes/bytarray
base64Binary	-	xs:base64Binary	BinaryType	-	bytes/bytarray
date	string +format	xs:date	DateType	date	datetime.date
dateTimeStamp	string +format	xs:dateTimeStamp	TimestampType	timestamp	- (int)
object	object	-	StructType	(composite types)	dict
array	array	-	ArrayType	[]	list

71

7.5 Sequence types

7.5.1 Cardinality

In the context of data querying but also of nested lists and arrays, items (single values) rarely appear alone. They often appear as a sequence of many values. Thus, many type system give options regarding the number of occurrences of items in a sequence.

There are four main occurrence indicators:

- just once (often implicit): exactly one item
- optional: zero or one item. Often represented with a question mark (?).
- any occurrence: zero, one or more items. Often represented with a Kleene star (*).

- at least once: one or more items. Often represented with a Kleene plus (+).

The symbols commonly used (?, *, +) correspond to those used in regular expressions¹.

There is a variety of other ways to specify such indicators: some technologies use keywords such as “repeated,” “required,” “optional,” etc. Finally, some technologies even allow specifying a precise interval (e.g., between 2 and 5 items).

7.5.2 Collections vs. nested lists

There are different kinds of sequences of items. It is common to distinguish between collections of items, and lists (or arrays) of items.

A collection of items is on the outer level, and can be massively large (billions, trillions of items). A collection of items corresponds to a relational table, i.e., a relational table can be seen as a collection of flat object items in the context of this chapter.

A list (or array) of items, however, usually refers to a nested structure, for example an array nested inside a document or object. Such lists of items are usually restricted in size for reasons of performance and scalability. Many technologies do not allow items to exceed a two-digit number of Megabytes (e.g., 10 MB, 16 MB, etc), or if they do allow them, might become slow and inefficient with too large items. Thus, an array of items will usually not exceed a few thousand items, a million at best.

It is thus important to keep this subtle difference in mind, in particular, do not confuse a collection of integers with a collection that contains a single array of integers.

¹for non-computer scientists, regular expressions are formulas that give pattern for accepting strings, for example for an integer: $-?[1-9][0-9]*$

7.6 JSON validation

Let us now look at how to use what we have learned so far to validate JSON documents. First, it is important to remember that we can only attempt to validate well-formed JSON documents. If a document cannot be parsed as JSON, and thus, cannot be represented as a tree in memory, then validation makes no sense on it.

7.6.1 Validating flat objects

JSound is a schema language that was designed to be simple for 80% of the cases, making it particularly suitable in a teaching environment. It is independent of any programming language.

The compact syntax of JSound, which we focus on here, is very close to how the original documents look like. Let us take an example:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "age" : 142
}
```

Let us say we want to validate the above document, in the sense that “name” should be a string, and “first” should be a string. The corresponding JSound schema looks like so:

```
{
  "name" : "string",
  "first" : "string",
  "age" : "integer"
}
```

“string” can be replaced with any other named type, in particular taken from the table shown in the former section. Let us list them here:

- Strings: string, anyURI (for strings containing a URI);
- Numbers: decimal, integer, float, double, long, int, short, byte, negativeInteger, positiveInteger, nonNegativeInteger, nonPositiveInteger, unsignedByte, unsignedShort, unsignedInt, unsignedLong;
- Dates and times: date, time, dateTime, gYearMonth, gYear, gMonth, gDay, gMonthDay, dateTimeStamp;
- Time intervals: duration, yearMonthDuration, dayTimeDuration;
- Binary types: hexBinary, base64Binary

- Booleans: boolean
- Nulls: null

This is a standardized list of types defined by the W3C².

JSON Schema is another technology for validating JSON documents. A JSON Schema against which the same document as above is valid would be:

```
{
  "type" : "object",
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  }
}
```

The available JSON Schema types are string, number, integer, boolean, null, array and object. This closely matches the original JSON syntax with the only exception that numbers have this additional integer subtype. The type system of JSON Schema is thus less rich than that of JSound, but extra checks can be done with so-called formats, which include date, time, duration, email, and so on including generic regular expressions. Like JSound, JSON Schema also allow restricting the length of a string, constraining numbers to intervals, etc.

7.6.2 Requiring the presence of a key

By default, the presence of a key is optional, so that each one of the following objects is also valid against the previous schema:

```
{ "name" : "Einstein" }
{ "first" : "Albert" }
{ "age" : 142 }
{ "name" : "Einstein", "age" : 142 }
{ }
```

It is possible to require the presence of a key by adding an exclamation mark, like so.

```
{
  "!name" : "string",
  "!first" : "string",
```

²In the context of XML schema, however this list is universal enough to not be tied with XML.

```
"age" : "integer"
}
```

which would make the following two documents valid:

```
{ "name" : "Einstein", "first" : "Einstein", "age" : 142 }
{ "name" : "Einstein", "first" : "Einstein" }
```

This is the equivalent JSON Schema, which uses a “required” property associated with the list of required keys to express the same:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  }
}
```

7.6.3 Open and closed object types

In the JSound compact syntax, extra keys are forbidden, i.e., this document is valid against neither of the previous two schemas:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "profession" : "physicist"
}
```

The schema is said to be closed. There are ways to define JSound schemas to allow arbitrary additional keys (open schemas), with a more verbose syntax.

Unlike JSound, in JSON Schema, extra properties are allowed by default, i.e., this document is also valid against previous schemas:

```
{
  "name" : "Einstein",
  "first" : "Albert",
  "profession" : "physicist"
}
```

JSON Schema then allows to forbid extra properties with the “`additionalProperties`” property, like so:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : "string",
    "first" : "string",
    "age" : "number"
  },
  "additionalProperties" : false
}
```

7.6.4 Nested structures

What about nested structures? Let us consider this document, which contains a nested array of integers.

```
{
  "numbers" : [ 1, 2, 6, 2, 7, 1, 57, 4 ]
}
```

This document is valid against the following schema (where of course, “integer” can be replaced with any other type):

```
{ "numbers" : [ "integer" ] }
```

This also works with multiple dimensions

```
{ "matrix" : [ [ "decimal" ] ] }
```

for validating matrices:

```
{ "matrix" : [ [ 0, 1 ], [ 1, 0 ] ] }
```

With the JSound compact syntax, object and array types can nest arbitrarily:

```
{
  "datapoints" : [
    {
      "features" : [ "double" ],
      "label" : "integer"
    }
  ]
}
```

The following document is valid against the above schema:

```
{
  "datapoints" : [
    {
      "features" : [ 1.2, 3.4, 5.6 ],
      "label" : 0
    },
    {
      "features" : [ 9.3, 2.6, 2.4 ],
      "label" : 1
    },
    {
      "features" : [ 1.1, 4.3, 6.5 ],
      "label" : 0
    }
  ]
}
```

The same document can also be validated against a more complex JSON Schema with nested arrays and objects, like so:

```
{
  "type" : "object",
  "properties" : {
    "datapoints" : {
      "type" : "array",
      "items" : {
        "type" : "object",
        "properties" : {
          "features" : {
            "type" : "array",
            "items" : {
              "type" : "number"
            }
          },
          "label" : {
            "type" : "integer"
          }
        }
      }
    }
  }
}
```

As we will see shortly, the “shape” of a collection of documents captured with a compact JSound schema is of particular relevance in

the context of efficient data processing, and a collection of valid JSON documents with such a shape is known as a data frame.

Every schema can be given a name, turning into a type. When a document is valid against a schema, it is typical to also *annotate* the document, which means that its tree representation in memory contains additional type information and values are stored natively in their type, enabling efficient processing and space efficiency.

JSound allows for the definition not only of arbitrary array and object types as shown above, but also of additional atomic types, by imposing some constraint on existing types (e.g., airport codes by restricting the length of a string to 3 and requiring all three characters to be uppercase letters, shoe sizes with intervals, etc). These are called user-defined types.

7.6.5 Primary key constraints, allowing for null, default values

There are a few more features available in the compact JSound syntax (not in JSON Schema) with the special characters @, ? and =:

```
{
  "datapoints" : [
    {
      "@id" : "int",
      "features" : [ "double" ],
      "label?" : "integer",
      "set" : "string=training"
    }
  ]
}
```

The question mark (?) allows for null values (which are not the same as absent values). Technically, it creates a union of the specified type with the null type.

The arobase (@) indicates that one or more fields are primary keys for a list of objects that are members of the same array. In this case, it means all id fields must be different for the datapoints array of each document.

The equal sign (=) is used to indicate a default value that is automatically populated if the value is absent. In this case, if the field “set” is missing, then upon annotating the document after its validation, it will be added with a value “training”.

The following document is valid against the above schema. Note that some values are quoted, which does not matter for validation: validation only checks whether lexical values are part of the type’s lexical space.

```
{  
    "datapoints" : [  
        {  
            "id" : "10",  
            "features" : [ 1.2, 3.4, 5.6 ],  
            "label" : null,  
            "set" : "training"  
        },  
        {  
            "id" : 11,  
            "features" : [ "9.3", 2.6, 2.4 ],  
            "label" : 1  
        },  
        {  
            "id" : 12,  
            "features" : [ 1.1, 4.3, 6.5 ],  
            "label" : "0",  
            "set" : "test"  
        }  
    ]  
}
```

And, after annotating it, it will look like so (except it will be represented efficiently in memory, and no longer in actual JSON syntax).

```
{  
    "datapoints" : [  
        {  
            "id" : 10,  
            "features" : [ 1.2, 3.4, 5.6 ],  
            "label" : null,  
            "set" : "training"  
        },  
        {  
            "id" : 11,  
            "features" : [ 9.3, 2.6, 2.4 ],  
            "label" : 1,  
            "set" : "training"  
        },  
        {  
            "id" : 12,  
            "features" : [ 1.1, 4.3, 6.5 ],  
            "label" : 0,  
            "set" : "test"  
        }  
    ]  
}
```

```
    ]
}
```

7.6.6 Accepting any values

Accepting any values in JSound can be done with the type “item”, which contains all possible values, like so:

```
{
  "!name" : "item",
  "!first" : "item",
  "age" : "number"
}
```

In JSON Schema, in order to declare a field to accept any values, you can use either true or an empty object in lieu of the type, like so:

```
{
  "type" : "object",
  "required" : [ "name", "first" ]
  "properties" : {
    "name" : {},
    "first" : true,
    "age" : "number"
  },
  "additionalProperties" : false
}
```

The following document validates successfully against the above JSound and JSON Schemas:

```
{
  "name" : [ "Ein", "st", "ein" ],
  "first" : "Albert",
}
```

JSON Schema additionally allows to use false to forbid a field:

```
{
  "type" : "object",
  "properties" : {
    "name" : "string",
    "first" : false,
  }
}
```

Making this document invalid:

```
{
  "name" : "Einstein",
  "first" : "Albert",
}
```

7.6.7 Type unions

In JSON Schema, it is also possible to combine validation checks with Boolean combinations.

First, disjunction (logical or) is done with

```
{
  "anyOf" : [
    { "type" : "string" },
    { "type" : "array" }
  ]
}
```

JSound schema allows defining unions of types with the vertical bar inside type strings, like so:

"string|array"

7.6.8 Type conjunction, exclusive or, negation

In JSON Schema only (not in JSound), it is also possible to do a conjunction (logical and) with

```
{
  "allOf" : [
    { "type" : "string", "maxLength" : 3 },
    { "type" : "string", "minLength" : 2 }
  ]
}
```

as well as exclusive or (xor):

```
{
  "oneOf" : [
    { "type" : "number", "minimum" : 2 },
    { "type" : "number", "multipleOf" : 2 }
  ]
}
```

as well as negation:

```
{
  "not" : { "type" : "array" }
}
```

7.7 XML validation

XML validation is also supported by several technologies. We will briefly show how one of them works, XML Schema.

Similar to how a JSound schema or a JSON Schema is a JSON document, an XML Schema is also an XML document.

7.7.1 Simple types

This is an example (well-formed) XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>This is text.</foo>
```

And this is an XML Schema against which the above XML document is valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xselement name="foo" type="xs:string"/>
</xsschema>
```

So what is going on here? First, you notice that all elements in an XML Schema are in a namespace, the XML Schema namespace. We explained the namespace machinery in Chapter 5, so that the `xmlns:xs` construct should be familiar to you. The namespace is prescribed by the XML Schema standard and must be this one. It is recommended to stick to the prefix `xs`, or `xsd`, which is also quite popular. We do not recommend declaring the XML Schema namespace as a default namespace, because it can create confusion in several respects.

Next, you will notice that the top element in an XML Schema document is the `xsschema` element, and inside there is an element declaration done with the `xselement` element. It has two attributes: one defines the name of the element to validate (`foo`) and the other one specifies its type (`xs:string`). The list of predefined atomic types is the same as in JSound, except that in XML Schema, all these predefined types live in the XML Schema namespace and thus bear the prefix `xs` as well. In fact, formally, this list of predefined types is standardized by XML Schema (along with more XML-specific types that are less known).

Let us try to change the type. `integer` (prefixed with `xs!`) needs no introduction...

```
<?xml version="1.0" encoding="UTF-8"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xselement name="foo" type="xs:integer"/>
</xsschema>
```

This document is then valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>142857</foo>
```

Note that extra whitespaces, newlines and indentation are fine:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  142857
</foo>
```

7.7.2 Builtin types

Let us list again the most important predefined types here, almost the same as JSound, but with the xs prefix:

- Strings: xs:string, xs:anyURI (for strings containing a URI);
- Numbers: xs:decimal, xs:integer, xs:float, xs:double, xs:long, xs:int, xs:short, xs:byte, xs:negativeInteger, xs:positiveInteger, xs:nonNegativeInteger, xs:nonPositiveInteger, xs:unsignedByte, xs:unsignedShort, xs:unsignedInt, xs:unsignedLong;
- Dates and times: xs:date, xs:time, xs:dateTime, xs:gYearMonth, xs:gYear, xs:gMonth, xs:gDay, xs:gMonthDay, xs:dateTimeStamp;
- Time intervals: xs:duration, xs:yearMonthDuration, xs:dayTimeDuration;
- Binary types: xs:hexBinary, xs:base64Binary
- Booleans: xs:boolean
- Nulls: does not exist as a type in XML Schema (JSON specific).

XML Schema allows you to define user-defined atomic types, for example restricting the length of a string to 3 for airport codes, and then use it with an element (there is no prefix because, to keep things simple for teaching, we are not working with any namespaces for our own declared elements and types):

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="airportCode">
    <xs:restriction base="xs:string">
      <xs:length value="3"/>
    </xs:restriction>
```

```
</xs:simpleType>
<xs:element name="foo" type="airportCode"/>
</xs:schema>
```

With this valid document:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
    ZRH
</foo>
```

7.7.3 Complex types

It is also possible to constrain structures and the element/attribute/-text hierarchy with complex types applying to element nodes. There are four main kinds of complex types:

- complex content: there can be nested elements, but there can be no text nodes as direct children.
- simple content: there are no nested elements: just text, but attributes are also possible.
- empty content: there are neither nested elements nor text, but attributes are also possible.
- mixed content: there can be nested elements and it can be intermixed with text as well.

This is an example of complex content:

```
<foo>
    <twotofour>foobar</twotofour>
    <twotofour>foobar</twotofour>
    <twotofour>foobar</twotofour>
    <zeroorone>true</zeroorone>
</foo>
```

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="complex">
        <xs:sequence>
            <xs:element
                name="twotofour"
                type="xs:string"
                minOccurs="2"
```

```

        maxOccurs="4"/>
<xs:element
    name="zeroorone"
    type="xs:boolean"
    minOccurs="0"
    maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
<xs:element name="foo" type="complex"/>
</xs:schema>
```

Note how children elements can be repeated, and the number of occurrences can be constrained to some interval with minOccurs and maxOccurs attributes in the schema. Of course, this all works recursively, i.e., the nested elements can also have complex types with complex content and so on (even though in this example they have simple types).

This is an example of simple content:

```
<foo country="Switzerland">2014-12-02</foo>
```

which is valid against this schema:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="dateCountry">
        <xs:simpleContent>
            <xs:extension base="xs:date">
                <xs:attribute name="country" type="xs:string"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:element name="foo" type="dateCountry"/>
</xs:schema>
```

Note how a complex type with simple content is defined as the extension of a simple type, adding one or more attributes to it. If there are no attributes, of course, there is no need to bother with a complex type: a simple type does the trick as shown before.

This is an example of empty content:

```
<foo/>
```

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="complex">
    <xs:sequence/>
  </xs:complexType>
  <xs:element name="foo" type="complex"/>
</xs:schema>
```

As you can see, empty content is “boring”, in that it is defined just like complex content, but with no nested elements at all (attributes, though, can absolutely be added).

Finally, this is an example of mixed content:

```
<foo>Some text and some <b>bold</b> text.</foo>
```

which is valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="mixedContent" mixed="true">
    <xs:sequence>
      <xs:element
        name="b"
        type="xs:string"
        minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo" type="mixedContent"/>
</xs:schema>
```

Mixed content is also “boring” to define: all it takes is a structure just like complex content, plus an extra attribute in the xs:complexType declaration called “mixed” and set to true.

7.7.4 Attribute declarations

Finally, all types of content can additionally contain attributes. Attributes always have a simple type. An example with empty content involving one attribute:

```
<foo country="Switzerland"/>
```

which is valid against:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="withAttribute">
    <xs:sequence/>
    <xs:attribute name="country"
      type="xs:string"
      default="Switzerland"/>
  </xs:complexType>
  <xs:element name="foo" type="withAttribute"/>
</xs:schema>
```

The default attribute of the attribute declaration will automatically add an attribute with the corresponding name and specified value in memory in case it was missing in the original instance. This works just like in JSound.

7.7.5 Anonymous types

Finally, it is not mandatory to give a name to all types. It is possible, instead of the type attribute of an element or attribute declaration, to instead nest a type declaration with no name attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="c">
    <xs:complexType>
      <xs:sequence/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Be careful: if there is neither a type attribute nor a nested type declaration, then anything is allowed!

7.7.6 Miscellaneous

XML Schema has many more features than described here: primary keys, constraints, support for namespaces, etc.

Finally, XML Schema documents are themselves XML documents, and can thus be validated against a “schema or schemas”, itself written as an XML Schema. This schema has the wonderful property of being valid against itself, which will delight aficionados of Douglas Hofstadter’s Pulitzer-prize-winning Gödel, Escher, Bach book.

7.8 Data frames

7.8.1 Heterogeneous, nested datasets

Now that we have defined data models for JSON and XML, and added a mechanism (schema validation) to enforce additional constraint on collections of JSON objects or of XML documents, we can take a step back and look at the bigger picture.

First, it should now be clear to you that, in both XML and JSON, datasets are simply collections of trees. If we now consider the particular case of JSON, which is growing in popularity, we can express a collection of JSON trees formally as a list of maps.

Now, we have already seen lists of maps before: relational tables are lists of maps, too, at least in the list semantics, as opposed to the set or bag semantics of the relational algebra. But in the context of relational tables, we called these maps tuples, or rows. The reason is that relational tables are not *any* lists of maps: they have constraints, as we explained in Chapter 2. These constraints include atomic integrity, relational integrity, and domain integrity.

Casually explained, atomic integrity means that relational tables are *flat*.

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

A counterexample of flatness (or of atomic integrity) is the presence of nested tables:

ID	Profession	People	
1	Physicist	Name	Living
		Einstein	false
		Penrose	true
2	Computer Scientist	Name	Living
		Turing	false
		Dean	true

Casually explained, relational integrity and domain integrity means that relational tables are *homogeneous*.

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

A counterexample of homogeneity (or of relational and domain integrity) is the presence of missing or additional columns, and of values with different types in the same columns:

ID	Name	Living	Profession
1	Einstein	false	Physicist
2	Penrose	true	CS
3	Turing		
4	Dean	true	

Finally, we can even provide a counterexample of a dataset that is neither flat, nor homogeneous: it is both nested and heterogeneous.

ID	Profession	People		Comment
1	Physicist	Name	Living	
		Einstein	false	
		Penrose		
2	Computer Scientist	Name	Living	They rock
		Turing		
		Dean	true	

The beauty of the JSON data model is that, unlike the relational model and the CSV syntax, it supports nested, heterogeneous datasets, while also supporting *as a particular case* flat, homogeneous datasets like so:

```
{"ID":1, "Name": "Einstein", "Living" : false}
{"ID":2, "Name": "Penrose", "Living" : true}
 {"ID":3, "Name": "Turing", "Living" : false}
 {"ID":4, "Name": "Dean", "Living" : true}
```

This is now an example of nested (but homogeneous) collection in JSON syntax, corresponding to the nested visual above:

```
{
  "ID":1,
  "Profession": "Physicist",
  "People": [
    {"Name": "Einstein", "Living" : false},
    {"Name": "Penrose", "Living" : true}
  ]
}
{
  "ID":2,
  "Profession": "Computer Scientist",
  "People": [
    {"Name": "Turing", "Living" : false},
    {"Name": "Dean", "Living" : true}
  ]
}
```

This is now an example of heterogeneous (but flat) collection in JSON syntax, corresponding to the heterogeneous visual above:

```
{
  "ID":1,
  "Name": "Einstein",
  "Living" : false,
  "Profession" : "Physicist"
}
{
  "ID":2,
  "Name": "Penrose",
  "Living" : true,
  "Profession" : "CS"
}
{
  "ID":3,
  "Name": "Turing"
}
{
  "ID":4,
  "Name": "Dean",
  "Living" : true
}
```

And finally, this is an example of heterogeneous and nested collection in JSON syntax, corresponding to the heterogeneous and nested visual above:

```
{
  "ID":1,
  "Profession": "Physicist",
  "People": [
    {"Name": "Einstein", "Living" : false},
    "Penrose"
  ]
}
{
  "ID":2,
  "Profession": "Computer Scientist",
  "People": [
    {"Name": "Turing"},
    {"Name": "Dean", "Living" : true}
  ],
  "Comment": "They rock"
}
```

For completeness, this shows how the same can be expressed in XML. However, XML is more powerful as we explained before, so that

in the remainder of the course, we will tend to focus much more on JSON.

```
<professions>
  <profession id="1">
    <name>physicist</name>
    <persons>
      <person>
        <name>Einstein</name>
        <living>false</living>
      </person>
      Penrose
    </persons>
  </profession>
  <profession id="2">
    <name>Computer Scientist</name>
    <persons>
      <person>
        <name>Turing</name>
      </person>
      <person>
        <name>Dean</name>
        <living>true</living>
      </person>
    </persons>
    <comment>They rock</comment>
  </profession>
</professions>
```

7.8.2 Dataframe visuals

There is a particular subclass of semi-structured datasets that are very interesting: valid datasets, which are collections of JSON objects valid against a common schema, with some requirements on the considered schemas.

The datasets belonging to this particular subclass are called data frames, or dataframes³

Specifically, for the dataset to qualify as a data frame, firstly, we forbid schemas that allow for open object types, that is, schemas must disallow any additional attributes, and, secondly, we forbid schemas

³Unlike German, compound words in English are normally space-separated or dash-separated. However, when a compound word is used very often, at some point, people start pasting them together in the German style: doormat, today, keyboard, etc. In the case of dataframes though, this might also be due to the CamelCase used in many libraries: DataFrames.

that allow for object or array values to be too permissive and allow *any* values, that is, we ask that schemas require specific types such as integers, strings, dates, objects representing a person, arrays of binaries, etc. We, however, include schemas that allow for null values and/or absent values.

Under the above conditions, we call the collection of objects a data frame. It should be immediate to the reader that relational tables are data frames, while data frames are not necessarily relational tables: data frames can be (and are often) nested, but they are still relatively homogeneous to some extent. Relatively, because schemas can still allow for a value to be missing.

Data frames have the nice property that they can be drawn visually in structures that look like generalized relational tables and that look a bit nicer and more structured than the previous visuals with nested tables. Further, JSound compact schemas provides a natural syntax for constraining data frames, because object types in this syntax are always closed, and it allows for requiring or not values, and for including or not null values. Thus, we can now give a few examples of JSound schemas and their corresponding visuals. Let us start with the example of a “flat” JSound schema:

```
{  
    "ID" : "integer",  
    "Name" : "string",  
    "Living" : "boolean"  
}
```

This schema could also be described as a SQL CREATE TABLE statement just as well. In fact, for this use case, a relational database might make more sense than a datalake altogether. Concretely, this means the data can be drawn as a table, like so:

ID	Name	Living
1	Einstein	false
2	Penrose	true
3	Turing	false
4	Dean	true

But things get interesting if we denormalize and define one of the fields to be an array of strings, for example, like so:

```
{  
  "ID" : "integer",  
  "Name" : [ "string" ],  
  "Living" : "boolean"  
}
```

Then, the Data frame visual becomes:

ID	Name	Living
1	Albert	false
	Einstein	
2	Penrose	true
3	Alan	false
	Turing	
4	Dean	true

Thus, Data frames are a generalization of (normalized) relational tables allowing for (organized and structured) nestedness.

Data frames also can have nested objects, as described by the following schema:

```
{
  "ID" : "integer",
  "Name" : {
    "First" : "string",
    "Last" : "string"
  },
  "Living" : "boolean"
}
```

A valid instance can then be drawn like so:

ID	Name		Living
	First	Last	
1	Albert	Einstein	false
2	Roger	Penrose	true
3	Alan	Turing	false
4	Jeff	Dean	true

Finally, a very common use case modelling “tables in tables” involves objects nested in arrays, like so:

```
{
  "ID" : "integer",
  "Who" : [
    {
      "Name" : "string",
      "Type" : "string"
    }
  ],
  "Living" : "boolean"
}
```

Leading to the following visual:

ID	Who		Living
	Name	Type	
1	Albert	first	false
	Einstein	last	
2	Penrose	last	true
3	Alan	first	false
	Turing	last	
4	Dean	last	true

Note that the former visual could also match the following different, but less natural, schema:

```
{
  "ID" : "integer",
  "Who" : {
    "Name" : [ "string" ],
    "Type" : [ "string" ]
  },
  "Living" : "boolean"
}
```

This schema structure is, however, much less common even though it can be used by savvy users to optimize the layout under specific circumstances (an example where this can be seen is in high-energy physics datasets found at CERN). Beware in particular that this alternate schema structure does not enforce that there must be the same number of items in each array.

7.9 Data formats

Having now seen models for XML and in particular JSON, and having discovered data frames, it should be clear to the reader that the data structures in memory have nothing to do with the original syntax any more.

In fact, if the data is structured as a (valid) data frame, then there are many, many different formats that it can be stored in, and in a way that is much more efficient than JSON. These formats are highly optimized and typically stored in binary form, for example Parquet, Avro, Root, Google’s protocol buffers, etc. If you see a JSON dataset that you are able to validate against a (e.g., JSound) schema that is “data-frame friendly”, i.e., such that the data can be visualized as shown in the previous section, then you are highly encouraged to immediately build this schema, and convert the dataset to, say, Parquet. This gives you two immediate advantages:

- space efficiency: the file will be considerably smaller, meaning you can fit much more data even on your local laptop and it is also faster to transfer to and from the cloud to share with others.
- performance efficiency: the smaller binary file will be much faster to read from disk when you write queries. In fact, many optimizers are able to skip entire sections of the data based on the query (projecting away a column, etc), making it even faster than it already was.

Also, at the risk of repeating myself, if the schema does not involve any nested structures and contains only closed object types, and the data is less than a Terabyte, then a relational database with SQL queries is very often the best way to go.

Why is it possible to store the data more efficiently when it is valid and data-frame-friendly? One important reason is that the schema can be stored as a header in the binary format, and the data can be stored without repeating the fields in every record (as is done in textual JSON). Furthermore, there are plenty of techniques that allow compressing homogeneous sequences of values and, because of constant size, perform direct lookups (e.g., directly return the field “Name” of the 100th record) without having to scan through all the dataset. In fact, Parquet is called that way because the data is stored in a columnar fashion, grouping all the values (across objects) together when they are associated with the same key, which might remind you of the parquet floor of your living room.

Generally, data formats can be classified along three dimensions:

- whether they require validity against a data frame compatible schema (Parquet, protocol buffers, etc.) or not (JSON, XML, YAML, etc.).
- whether they allow for nestedness (Parquet, etc.) or not (CSV).
- whether they are textual (CSV, XML, JSON, etc.) or binary (Parquet, etc.), even though typically, formats that require a schema will be binary because of the “performance free lunch.”

We finish this chapter with good news: we will not give more details about the formats. This is because if you look the documentation (take Parquet as an exercise for example), you will immediately notice that the modelling will be very close to what we studied in this chapter: you only need to learn the terminology specific to the format (e.g., int64 instead of long, UTF8 instead of string, LIST instead of array, etc.). For Swiss readers, going from JSound+JSON to Parquet or Avro comes down to speaking Zurich German and going to Basel or to Bern and adapting to the different dialect: similar grammar, with some different words. XML then comes down to going to Wallis, a trip really worth it because it has stunning landscapes, too.

7.10 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain the difference between syntax and data models?
- b. Can you explain why and how (collections of) trees can be used to model data that is denormalized (i.e., not in first normal form)?
- c. Can you relate syntaxes to data models (e.g., CSV to tables, XML/JSON to trees, ...)?
- d. Can you explain why trees modeling XML (Infoset) have labels on the nodes, while trees modeling JSON have labels on edges?
- e. Can you name a few data models for XML (Infoset, PSVI, JDM)?
- f. Are you able, given an XML document or a JSON document, to sketch a tree representing that data (e.g., for XML, with a document node, elements, attributes, text, ...)?
- g. Do you know the difference between an atomic type (or value), and a structured type (or value), regardless of the exact data model?
- h. Can you name a few atomic types found across a broad number of data models? Can you classify them (numbers, dates, binary...)?
- i. Do you know the difference between the lexical space and the value space of an atomic type?
- j. Can you tell the difference between structured types based on lists (e.g., JSON array) vs. maps (e.g., JSON object)?
- k. Can you give examples of type cardinalities, and their associated symbols?
- l. Can you explain the difference between well-formedness and validity? Do you know what extra information you need to assess validity?
- m. Are you able to design simple XML Schemas to validate XML? Can you restrict simple types to allow specific values or value fulfilling specific criteria (length, pattern...)? Can you explain how to declare elements with simple types? Can you explain how to declare elements with complex types and various content models (empty, simple content, complex content, mixed content)?
- n. Can you tell, given an XML Schema, whether an XML document is valid against it, with software (oXygen) but also with your own eyes?

- o. Are you able to design simple JSound Schemas to validate JSON? Can you explain how to declare object and array types? Do you know the most important builtin atomic types (widely shared with those of XML Schema)?
- p. Are you able to design simple JSON Schemas to validate JSON? Can you explain how to declare object and array types?
- q. Can you name (without details) further data modeling technologies and formats for tree-like data? (Parquet, Avro, protocol buffers, ...)
- r. Do you understand that a table in first normal form can be seen as a homogeneous collection of flat trees (one per row and all with the same attributes)?
- s. Can you explain why valid JSON data (provided the schema has no open object types, and no heterogeneity in field types) can be represented as DataFrames? Are you able to draw them given JSON data?
- t. Do you understand that DataFrames can be stored in more efficient ways (time wise and space wise) than JSON and can you explain why it can make sense to convert valid JSON datasets to formats like Parquet before querying them?

7.11 Literature and recommended readings

The following is a list of recommended material for further reading and study.

JSound specifications and tutorial.

<http://www.jsound-spec.org/>

Droettboom, M (2015). *Understanding JSON Schema*.

Melnik, S., et al. (2011). *Dremel: Interactive Analysis of Web-Scale Datasets*. In CACM, 54(6).

White, T. (2015). *Hadoop: The Definitive Guide*. 4th edition. O'Reilly. Chapters 12 (Avro) and 13 (Parquet).

Harold, E. R., Means, W. S. (2004). *XML in a Nutshell*. O'Reilly. Chapter 17 on XML Schema.

Chapter 8

Massive Parallel Processing

Now that we know how and where to store datasets; and we know how to model nested, heterogeneous datasets; and we know how to validate them; and we know how to build data frames (when applicable); and we know how they can be stored in various optimized binary formats, it is time to finally move on to the truly awesome part of Big Data: actually processing gigantic amounts of data.

8.1 Counting cats

8.1.1 The Input

As I would like for this textbook to be used to also teach in other universities, it needs to involve cats. This is where Erwin, who lived in Zurich, comes into play. Erwin has hundreds of cats of ten various kinds, specifically he has:

- Persian cats
- Siamese cats
- Bengal cats
- Scottish folds
- American shorthairs
- Maine coons
- British shorthairs
- Sphynx cats
- Ragdolls

- Norwegian forest cats

However, Erwin lost track of counts, and would like to know how many cats of each kind he has with him. He could, of course, count them one by one, but Erwin thinks there has to be a better way.

8.1.2 The Map

Fortunately, his 22 grandchildren are visiting him this week-end, and Erwin has a plan.

First, he will distribute the cats across all 17 rooms of his large mansion. Then, he will send 17 of his grandchildren to each room, one per room, and ask them to count the cats, by kind, of the room they are assigned to, and to then write the counts of each kind on a piece of paper. This is an example for one room:

Persian cats	12
Siamese cats	23
Bengal cats	14
Scottish folds	23
American shorthairs	36
Maine coons	3
British shorthairs	5
Sphynx cats	54
Ragdolls	2
Norwegian forest cats	63

8.1.3 The Shuffle

Meanwhile, the other 5 grandchildren have been each assigned several kinds of cats. Each kind of cat will be processed by one, and exactly one, of these 5 grandchildren. Mathias, who is the oldest grandchild, will process 3 kinds of cats. Leane, Anna and Elliot will process 2 kinds each, and Cleo, the youngest one, will process 1 kind of cats, making it 10 in total.

Persian cats	Mathias
Siamese cats	Anna
Bengal cats	Cleo
Scottish folds	Mathias
American shorthairs	Leane
Maine coons	Mathias
British shorthairs	Elliot
Sphynx cats	Leane
Ragdolls	Anna
Norwegian forest cats	Elliot

By the time this is done, the 17 grandchildren are coming back with their sheets of paper. Erwin gives them scissors and asks them to divide these sheets of paper into stripes: each stripe with a kind of cat and its count. Then, the group of 17 and the group of 5 start walking in all directions, where each stripe of paper is handed over by the grandchild (from the group of 17) who created it, to the one supposed to process it (from the group of 5).

Needless to say, the mansion is quite noisy at that time and it takes quite a while until all stripes of paper are finally in the hands of Mathias, Leane, Anna, Elliot, and Cleo. Cleo received 16 stripes with Bengal cats counts; indeed, in one of the 17 rooms, there was no Bengal cats at all. Mathias received 17 stripes of Persian cat counts, 15 stripes of Scottish fold counts, and 14 stripes of Maine coons, and so on.

8.1.4 The Reduce

Now, Erwin asks the 5 grandchildren to add the counts for each kind of cat. Cleo has one big addition to do with all the numbers on her stripes:

Bengal cats	12
Bengal cats	13
Bengal cats	23
Bengal cats	3
Bengal cats	1
Bengal cats	6
Bengal cats	13
Bengal cats	6
Bengal cats	14
Bengal cats	9
Bengal cats	7
Bengal cats	7
Bengal cats	9
Bengal cats	11
Bengal cats	12
Bengal cats	11

Cleo proudly creates a new stripe with her grand total:

Bengal cats	157
-------------	-----

8.1.5 The Output

Mathias, Anna, Elliot and Leane do the same and give back their newly created stripes to Erwin, who now has the overview of his cat counts:

Persian cats	412
Siamese cats	233
Bengal cats	157
Scottish folds	233
American shorthairs	36
Maine coons	351
British shorthairs	153
Sphynx cats	236
Ragdolls	139
Norwegian forest cats	176

What did just happen? This is simply an instance of a MapReduce computation. The first phase, with the 17 separate rooms, is called the Map phase. Then came the Shuffle phase, when the stripes were handed over. And finally came the Reduce phase, with the computation of the final grand totals.

8.2 Patterns of large-scale query processing

8.2.1 Textual input

We saw that the data we want to query can take many forms. First, it can be billions of lines of text (this is from Sherlock Holmes, which came into public domain a few years ago, which in turn explains why there are so many movies and series about Sherlock Holmes these days):

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties.

It can also be plenty of CSV lines (these will likely be familiar to residents of Zurich):

```
Year,Date,Duration,Guest
2022,2019-04-25,00:37:59,UR
2021,2019-04-19,00:12:57,UR
```

```
2020,NULL,NULL,NULL
2019,2019-04-08,00:17:44,Strassburg
2018,2018-04-16,00:20:31,BS
2017,2017-04-24,00:09:56,GL
2016,2016-04-18,00:43:34,LU
...

```

Or maybe some use-case-specific textual format (common for weather data, for example):

```
20222019-04-2500:37:59UR
20212019-04-1900:12:57UR
20200000-00-0000:00:0000
20192019-04-0800:17:44FR
20182018-04-1600:20:31BS
20172017-04-2400:09:56GL
20162016-04-1800:43:34LU
...

```

Such a format can also be made of a key-value pattern, here, with key and value separated by a space character:

```
2022 00:37:59.000000
2021 00:12:57.000000
2020 00:00:00.000000
2019 00:17:44.000000
2018 00:20:31.000000
2017 00:09:56.000000
2016 00:43:34.000000
...

```

A popular format that is more machine readable is the JSON Lines format, with one JSON object per line (to fit it on this page, we had to write them on two lines, but in the real file, it would be a single line). Note that it can be heterogeneous:

```
{
  "year": 2022, "date": "2022-04-25",
  "duration": "00:37:59", "canton": "UR"
}
{
  "year": 2021, "date": "2021-04-19",
  "duration": "00:12:57", "canton": "UR"
}
{
  "year": 2020, "duration": null
}
{
  "year": 2019, "date": "2019-04-08",
  "duration": "00:17:44", "city": "Strasbourg"
}
{
  "year": 2018, "date": "2018-04-16",
  "duration": "00:20:31", "canton": "BS"
}
{
  "year": 2017, "date": "2017-04-24",
  ...
}
```

```
        "duration": "00:09:56", "canton": "GL" }  
{ "year": 2016, "date": "2016-04-18",  
        "duration": "00:43:34", "canton": "LU" }
```

8.2.2 Other input formats

Some other formats (e.g., Parquet, ...) can be binary, as we saw in Chapter 7. Personally, I am not able to decode it directly, but luckily computers can:

We also encountered HFiles in Chapter 6, which are lists of key-value pairs. In fact, Hadoop has another such kind of key-value-based format called Sequence File, which is simply a list of key-values, but not necessarily sorted by key (although ordered) and with keys not necessarily unique. This is because, as we will see shortly, this format is used for intermediate data generated with MapReduce.

Sequence Files also have a flavour in which values are compressed, and another flavour in which their blocks (akin to what we called HBlocks in HFfiles) are compressed.

8.2.3 Shards

How do we store Petabytes of data on online cloud storage, such as S3 or Azure blob storage, where the maximum size of a file is limited? Simply by spreading the data over many files. It is very common to have datasets lying in a directory spread over 100 or 1,000 files. Often, these files are named incrementally: part-0001, part-0002, etc. These files are often also called “shards” of the dataset.

What about HDFS? Technically, HDFS would make it possible to have a gigantic, single file, automatically partitioned into blocks. However, also for HDFS, it is common to have a pattern with a directory

containing many files named incrementally. The size of these files is typically higher than that of a block, for example 10 GB files.

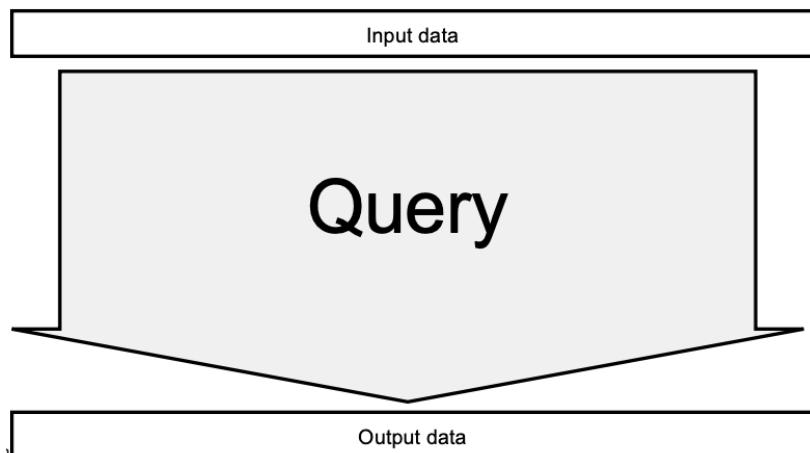
Note that the size of the files do not constrain parallelism: with HDFS, even with 10 GB files, the 128 MB blocks within the same file can still be processed in parallel. S3 is also compatible with intra-file parallelism. There are several practical motivations for the many-files pattern even in HDFS:

- As we will see, it is much more convenient to output several shards from a framework like MapReduce than it is to create a single, big final file (which would require a way to concatenate it properly).
- It is considerably easier to download or upload datasets that are stored as many files. This is because network issues happen once in a while, and you can simply retry with only the files that failed. With a single 10 PB file, the time it takes is so long that it is extremely likely that a network issue will force you to start it all over and over.

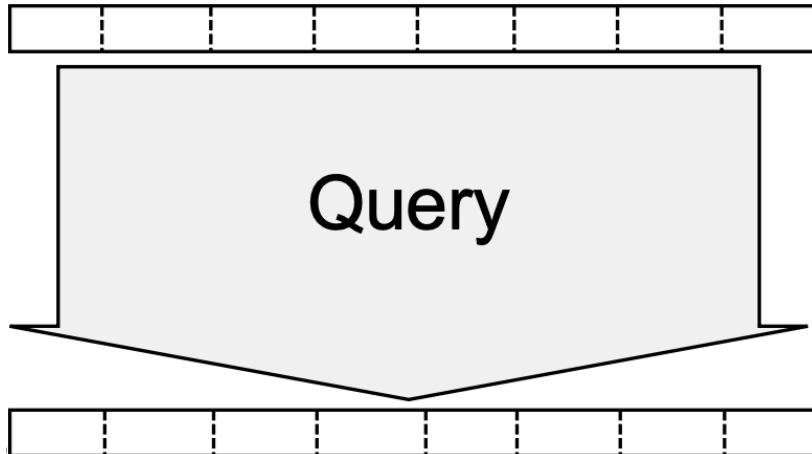
Of course, the data does not need to be stored directly as files on cloud storage or a distributed file system: it could be stored as a relational table in a relational database, or in a wide column store like HBase. In fact, it is quite common to store XML, HTML or JSON data in HBase cells. MapReduce can also process data on these higher-level stores.

8.2.4 Querying pattern

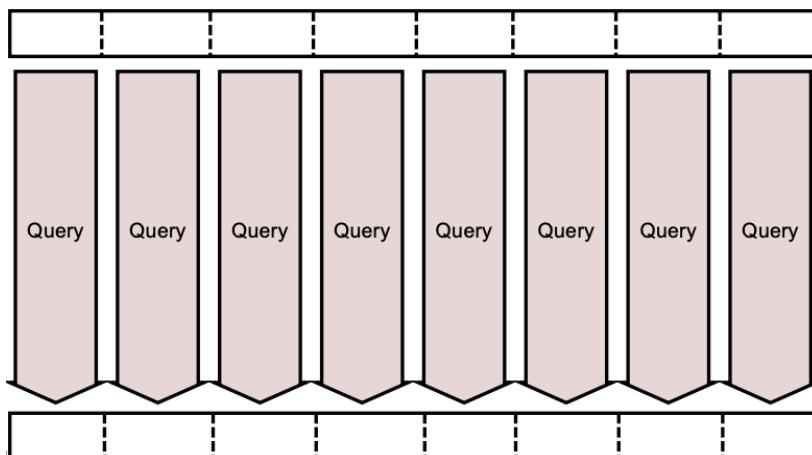
Now that we have the data, how does querying it look like? On the very high-level, it converts some input to some output like so:



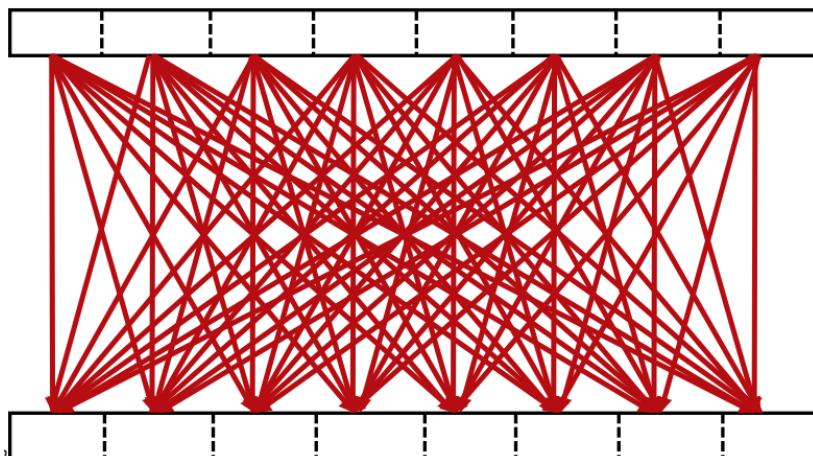
However, because the underlying storage supports parallelism (via shards, blocks, regions, etc), the input as well as the output are partitioned:



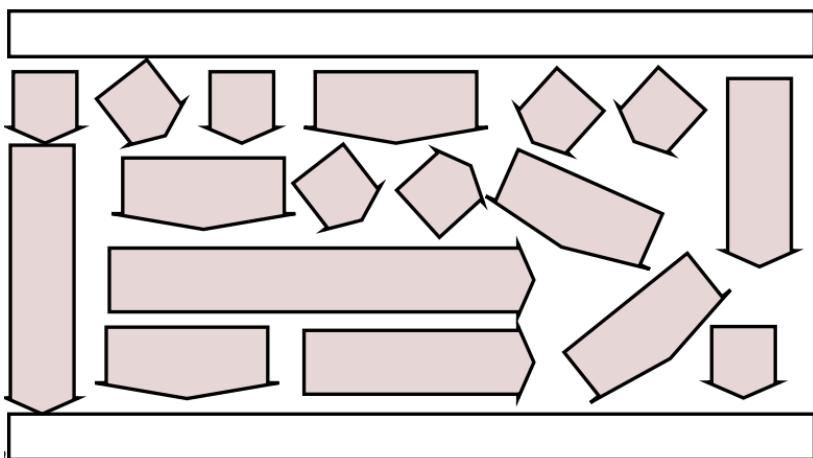
So far, that does not really help us. But what if we are lucky and the query can, in fact, be reexpressed equivalently to simply map every input partition to an output partition, like so?



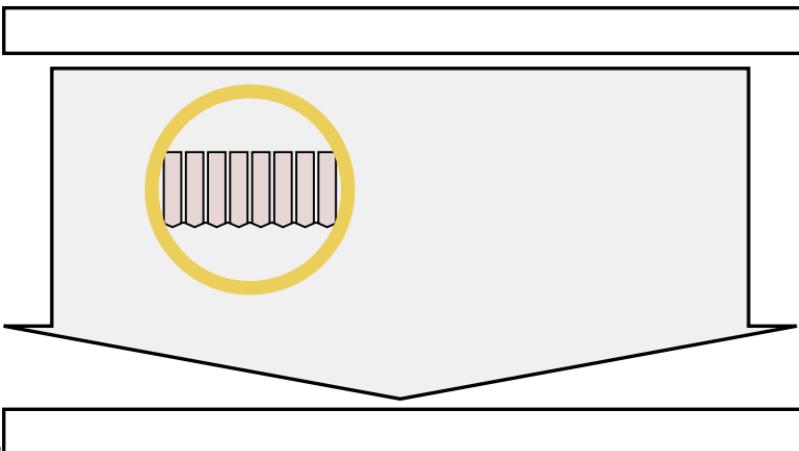
Sadly, in reality, this is what you might find instead something that looks more like spaghetti:



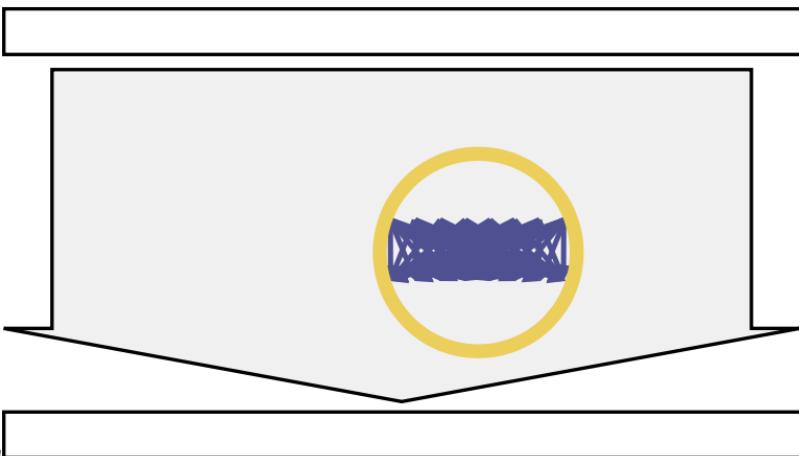
Fortunately, what happens often is somewhere in the middle, with some data flow patterns:



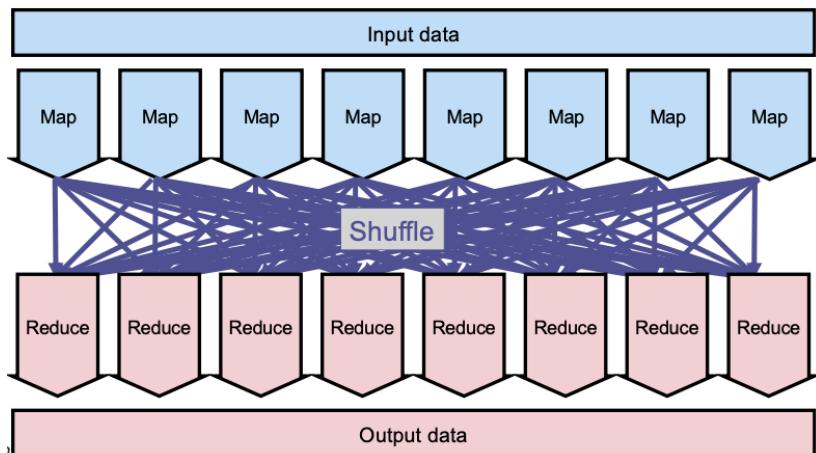
Some places have data flowing in parallel (map-like):



While some others are more spaghetti-y (shuffle-like):



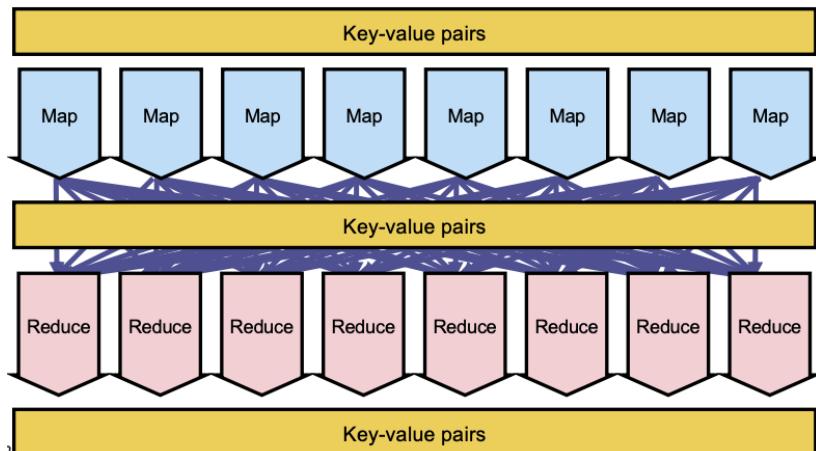
This is the motivation behind the standard MapReduce pattern: a map-like phase on the entire input, then a shuffle phase on the intermediate data, then another map-like phase (called reduce) producing the entire output:



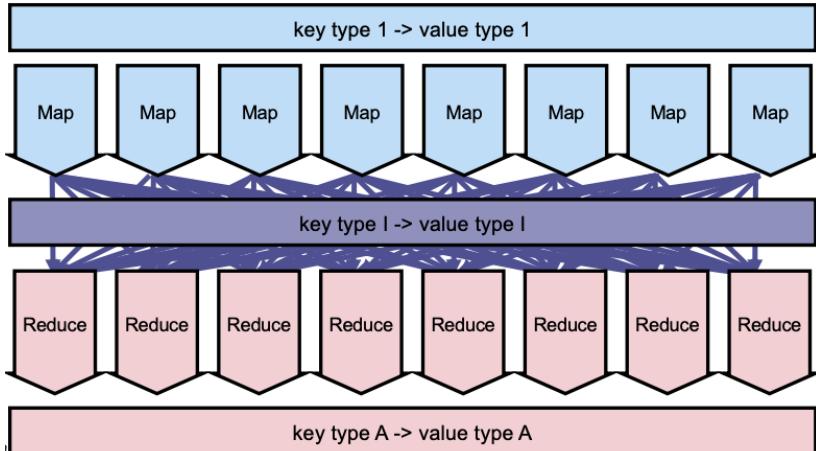
This seems restrictive, but in reality, it is extremely powerful and generic to accommodate for many use cases, in particular if you chain MapReduce jobs with each other.

8.3 MapReduce model

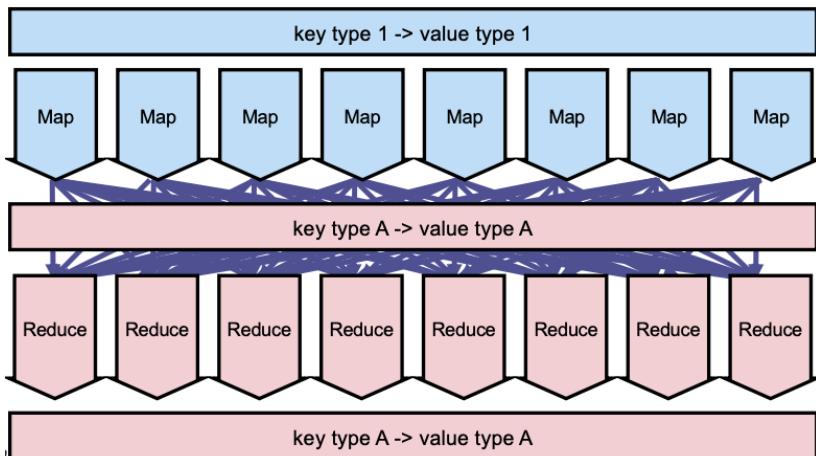
In MapReduce, the input data, intermediate data, and output data are all made of a large collection of key-value pairs (with the keys not necessarily unique, and not necessarily sorted by key):



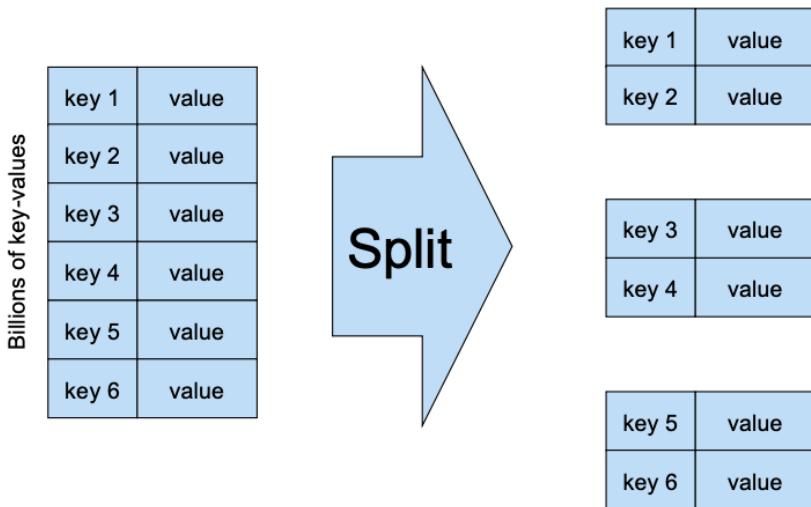
The types of the keys and values are known at compile-time (statically), and they do not need to be the same across all three collections:



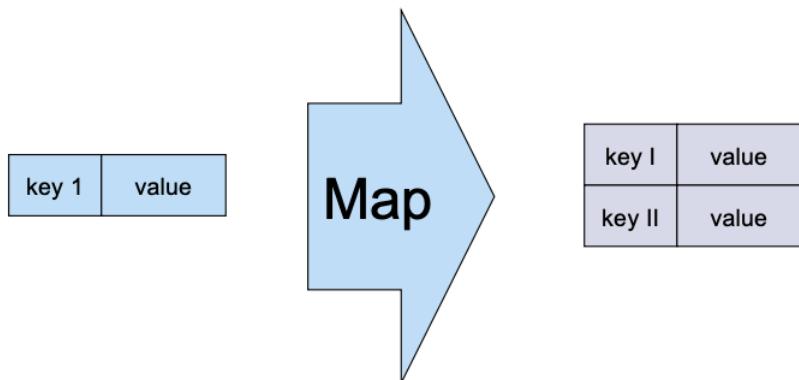
In practice, however, it is quite common that the type of the intermediate key-value pairs is the same as that of the output key-value pairs:



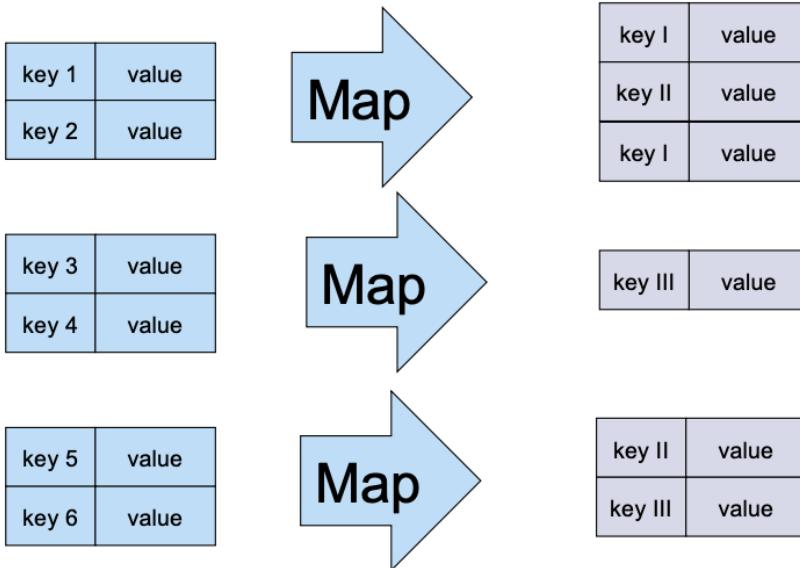
Let us now walk through a MapReduce query, but on the logical level for now. Everything starts with partitioning the input. MapReduce calls the partitions “splits”:



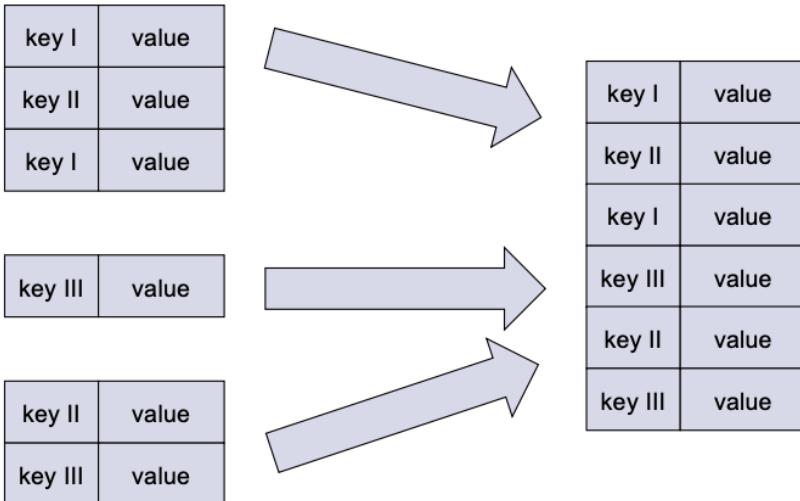
Each key-value will be fed into a map function:



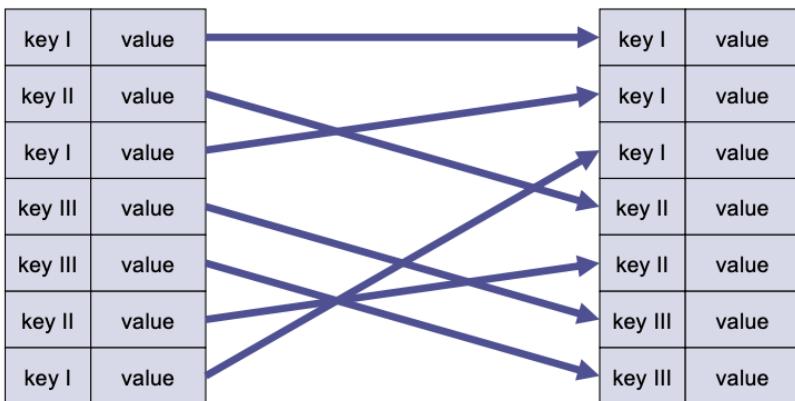
However, as you can imagine, it would be extremely inefficient to do it pair by pair, thus, the map function is called in batches, on each split:



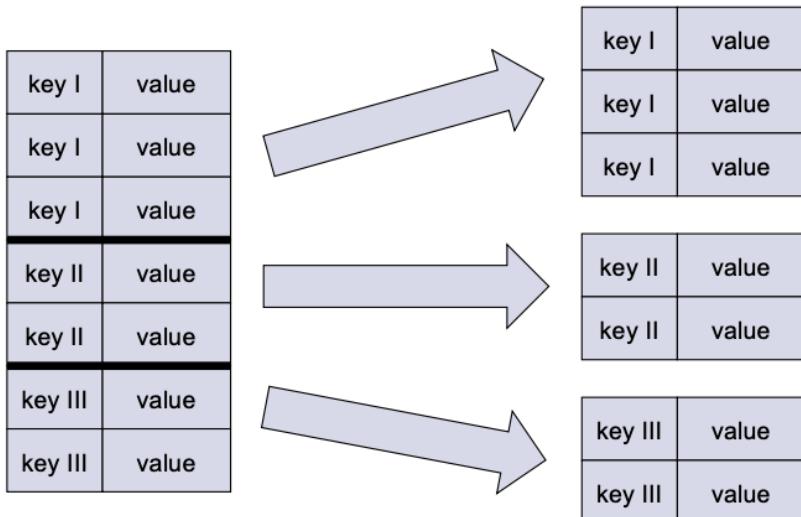
The intermediate pairs can be seen, logically, as a single big collection:



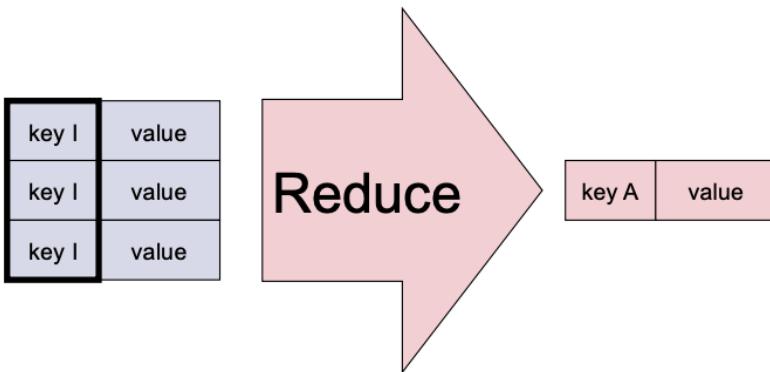
This collection can be thought of as logically sorted:



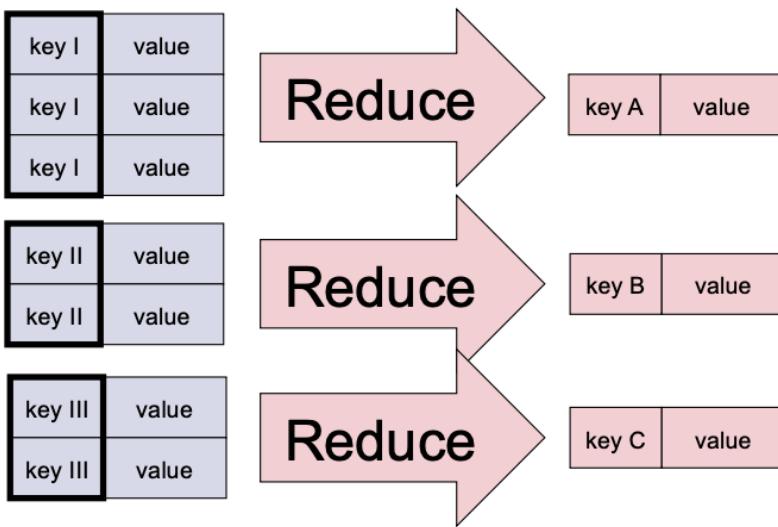
And then partitioned again, making sure the pairs with the same key are always in the same partition (but a partition can have pairs with several keys):



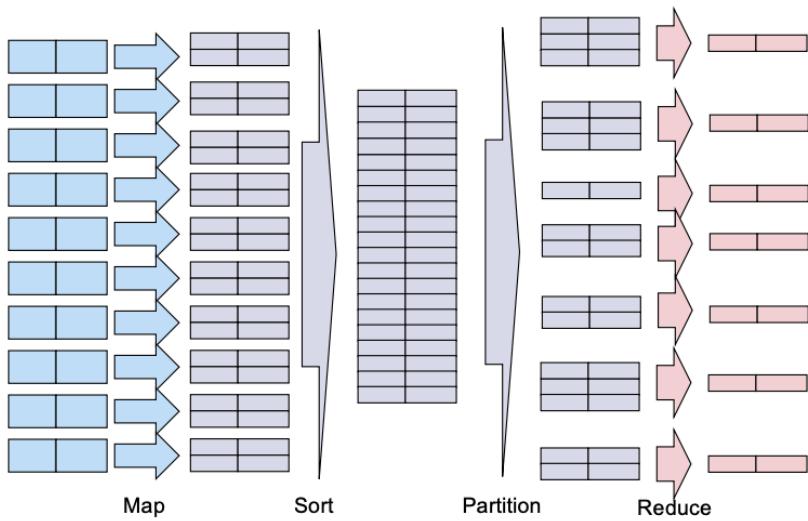
The reduce function must then be called, for every unique key, on all the pairs with that key, outputting zero, one or more output pairs (here, we just drew one, which is the most common case):



Just like the map function, the reduce function is called in batches, on each intermediate partition (multiple calls, one per unique key):



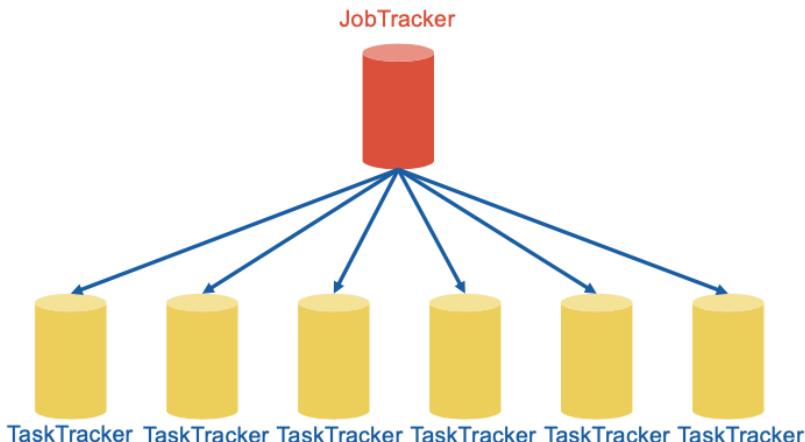
If we recap, this is how the entire process looks like on the high (and logical) level:



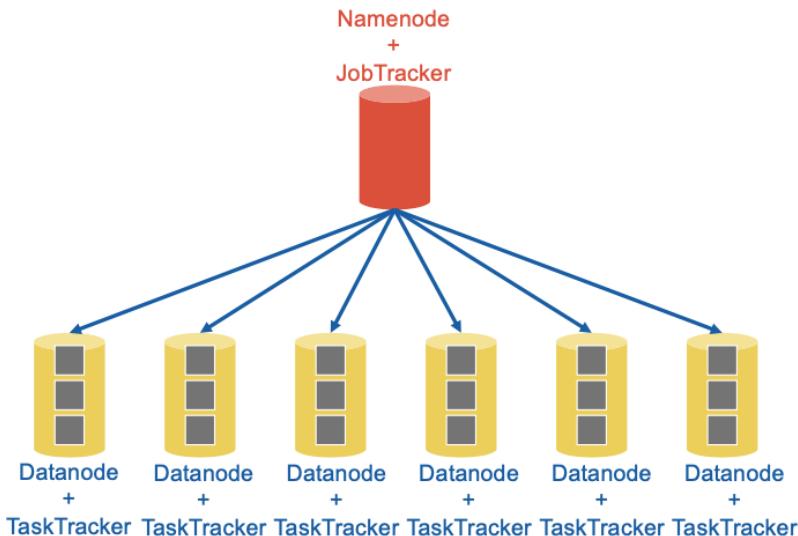
8.4 MapReduce architecture

MapReduce can read its input from many places as we saw: a cloud storage service, HDFS, a wide column store, etc. The data can be Petabyte-sized, with thousands of machines involved.

On a cluster, the architecture is centralized, just like for HDFS and HBase. In the original version of MapReduce, the main node is called JobTracker, and the worker nodes are called TaskTrackers.



In fact, the JobTracker typically runs on the same machine as the NameNode (and HMaster) and the TaskTrackers on the same machines as the DataNodes (and RegionServers). This is called “bring the query to the data.” If using HDFS, then most of the time, for the map phase, things will be orchestrated in such a way that there is a replica of the block corresponding to the split on the same machine (since it is also a DataNode...), meaning that it is a local read and not a network connection.



As the map phase progresses, there is a risk that the memory becomes full. But we have seen this before with HBase: the intermediate pairs on that machine are then sorted by key and flushed to the disk to a Sequence File. And as more flushes happen, these Sequence Files can be compacted to less of them, very similarly to HBase’s Log-Structured Merge Trees.

When the map phase is over, each TaskTracker runs an HTTP server listening for connections, so that they can connect to each other and ship the intermediate data over to create the intermediate partitions ensuring that the same keys are on the same machines (we will look at this again with more precise terminology later in this chapter). This is the phase called shuffling. Then, the reduce phase can start.

Note that shuffling can start before the map phase is over, but the reduce phase can only start after the map phase is over (why?).

When the reduce phase is completed, each output partition will be output to a shard (as we saw, a file named incrementally) in the output destination (HDFS, S3, etc) and in the desired format.

8.5 MapReduce input and output formats

8.5.1 Impedance mismatch

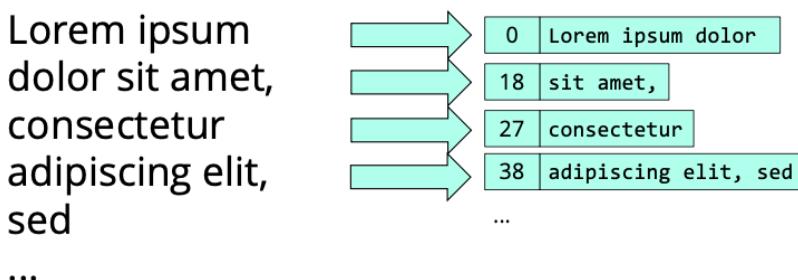
Let us now get back to the input received by and the output produced by MapReduce. We looked at various examples of textual and binary formats in Section 8.2, and saw in particular that MapReduce can read its input from files lying in a data lake as well as directly from a database system such as HBase or a relational database management system.

As the reader will have noticed, MapReduce only reads and writes lists of key-value pairs, where keys may be duplicates and need not appear in order. However, the inputs we considered are not key-value pairs. So we need an additional mechanism that allows MapReduce to interpret this input as key-value pairs.

For tables, whereas relational or in a wide column stores, this is relatively easy: indeed, tables have primary keys, consisting of either a single column or multiple columns. Thus, each tuple can be interpreted as a key-value pair, where the key is the (sub)tuple containing all the values associated with columns that are part of the primary key, while the value is the (sub)tuple containing the values associated with all other columns.

8.5.2 Mapping files to pairs

Let us thus focus on files. How do we read a (possibly huge) text file as a list of key-value pairs? The most natural way to do so is to turn each line of text in a key value pair¹: the value is the string corresponding to the entire line, while the key is an integer that expresses the position (as a number of characters), or offset, at which the line starts in the current file being read, like so:

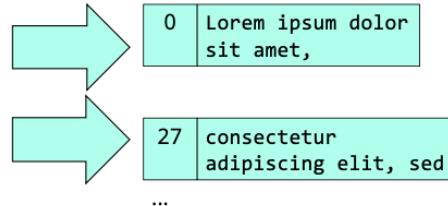


¹This means we take \n, or \r \n as a special separator, depending on the operating system.

A small variation consists in reading N lines at a time, mapping them to a single key-value:

**Lorem ipsum
 dolor sit amet,
 consectetur
 adipiscing elit,
 sed**

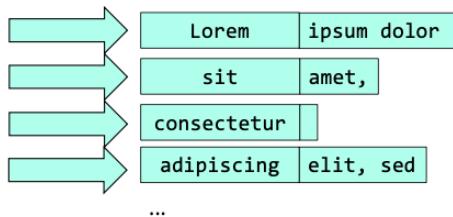
...



Another variation consists of treating a character (picked by the user) specially, as the separator between the key and the value:

**Lorem•ipsum
 dolor sit•amet,
 consectetur
 adipiscing•elit,
 sed**

...



Since MapReduce is not aware of JSON or any syntax, a JSON Line file will be read as text as explained above, and it is the responsibility of the user to parse this text to JSON. This is, of course, not very convenient, as it means MapReduce pushes the burden of doing so to the user, but we will see in subsequent chapters that there are additional layers that will come on top of the technology stack, which free the user from having to deal with these details.

Sequence Files are easier to handle: since they already contain lists of key-value pairs in a format native to MapReduce, their interpretation is straightforward. In fact, intermediate data spilled to disk will be written and read back in this format.

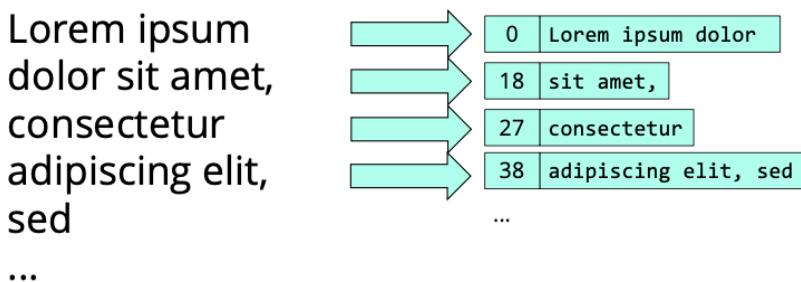
8.6 A few examples

8.6.1 Counting words

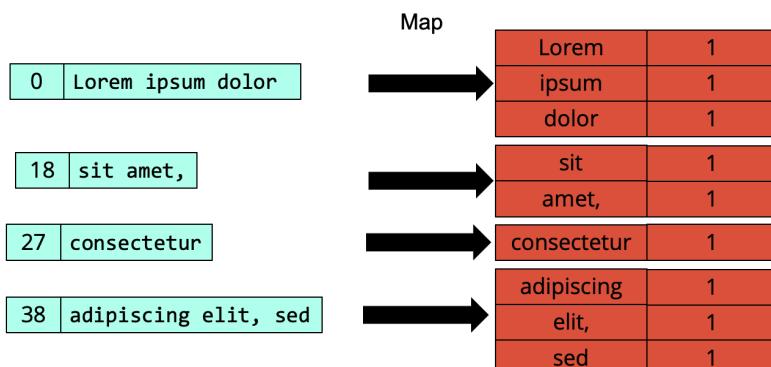
Let us use a concrete example and count the number of occurrences of each word in this document

```
 Lorem ipsum
 dolor sit amet,
 consectetur adipiscing elit,
 sed
 ...
```

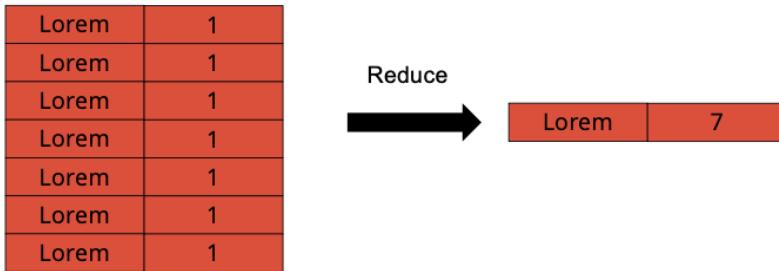
We already saw how this is interpreted as key-value pairs:



We can count the words within each line similar to our motivation example with the cats, by mapping each line key-value to several key-values, one per word and with a count of 1. This gives us our map function:



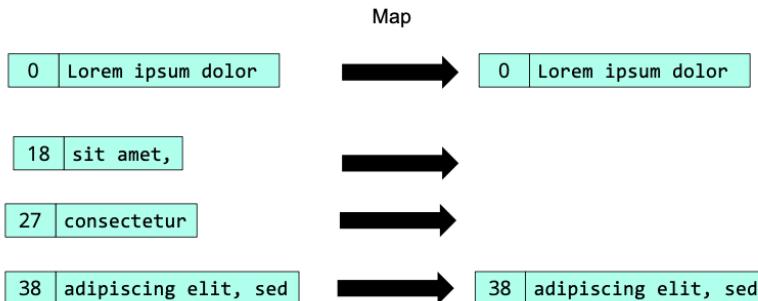
The reduce function is then obtained by summing the values with the same key, and keeping the same key:



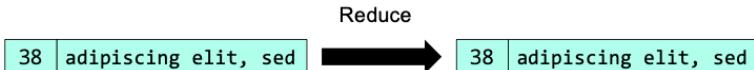
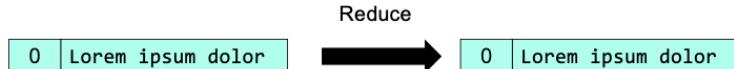
The output will then consist of a list of unique key-values, with one key for each word, and the number of its occurrences as the associated value.

8.6.2 Selecting

How about, say, filtering the lines containing a specific word? This is something easily done by having a map function that outputs a subset of its input, based on some predicate provided by the user:



Here we notice that the output of the map phase already gives us the desired result; we still need to provide a reduce function, which is taken trivially as the identity function. This is not unusual (and there are also examples where the map function is trivial, and the reduce function is doing the actual processing):

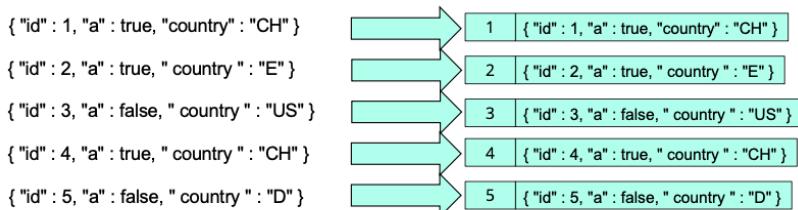


In fact, what we have just implemented in MapReduce is nothing else than a selection operator from the relational algebra. So now, you know how to implement the following SQL query on text, seen as a one-column table, on top of MapReduce, and in a way that scales to billions of lines!

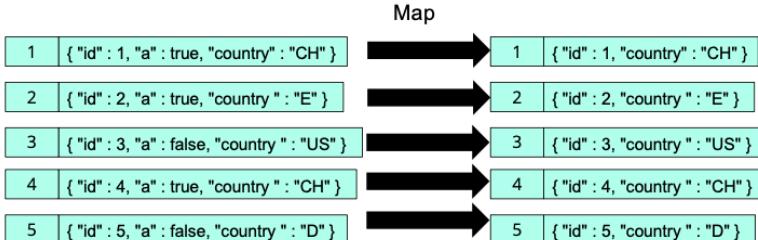
```
SELECT text
FROM input
WHERE text LIKE '\%foobar\%',
```

8.6.3 Projecting

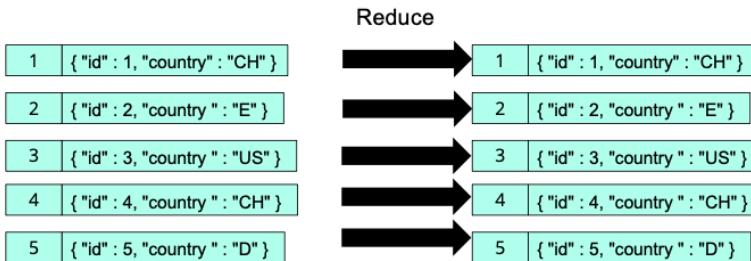
What about projection on some input in the JSON Lines format? Well, MapReduce doesn't know anything about attributes. So it is up to the user to parse, in their code, each line to a JSON object (e.g., if using Python, to a dict), like so:



Then, the map function can project this object to an object with less attributes:



And, like previously, we use a trivial identity function for the reduce function:



MapReduce will then output the results as one or several output files in the JSON Lines format. What we have just implemented on billions of records is the following equivalent SQL query:

```
SELECT id, country
FROM input
```

8.6.4 MapReduce and the relational algebra

As an exercise, try to figure out how to implement a GROUP BY clause and an ORDER BY clause. What about the HAVING clause?

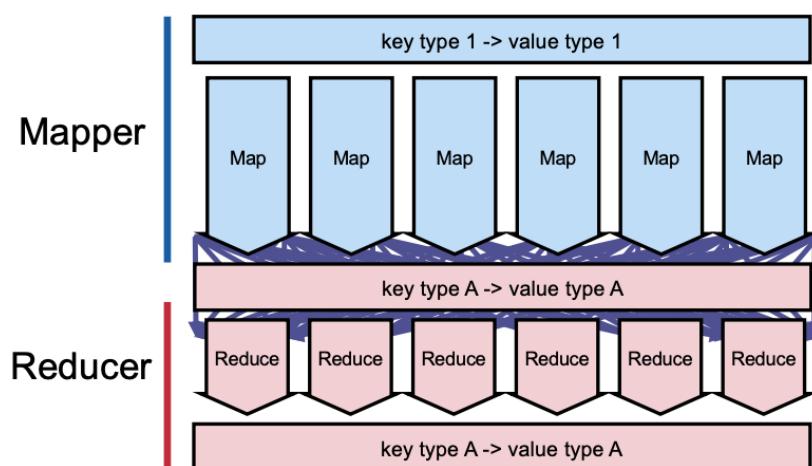
Do take a few seconds to realize what we have achieved here: previously, in the 2000s, the execution of such queries was only possible on inputs that fit on a single machine, e.g., in a PostgreSQL installation. This means maybe millions of records, perhaps one billion with today's machines, but nothing beyond this. But now, we know how to handle trillions of them on a cluster.

Naturally, executing these queries directly in MapReduce is very cumbersome because of the low-level code we need to write, and understanding how this is done has a pedagogical purpose more than a

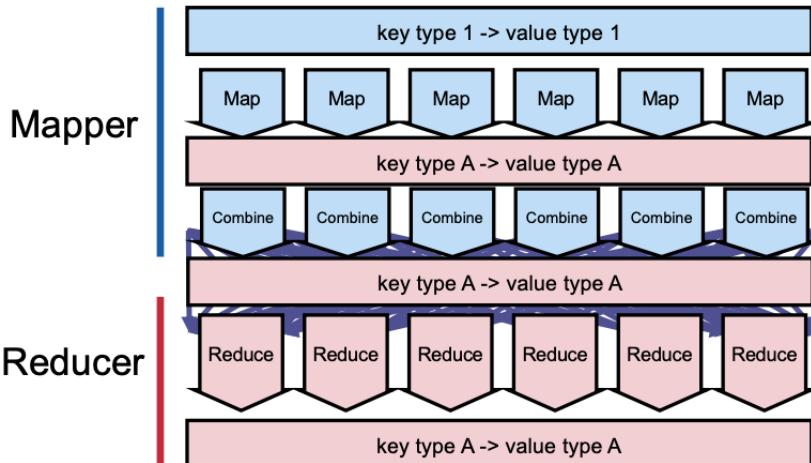
practical purpose. If I did my job correctly as a teacher, most of my students will never use MapReduce directly (except Data Engineers maybe, implementing this inside a larger system). But in real life, this is nothing to worry about, because all of this complexity will be again hidden behind SQL and similar languages, following the data independence paradigm.

8.7 Combine functions and optimization

Let us look again at the MapReduce phases:



Is there any way we can optimize things and run even faster? In fact, there is. In our counting example, we created an intermediate key-value for each occurrence of a word with a value set to 1. But what if a word appears 5 times on the same line? In this case, we can replace the corresponding key-value pairs with just one pair, with the value 5. Doing so is called combining and happens during the map phase:

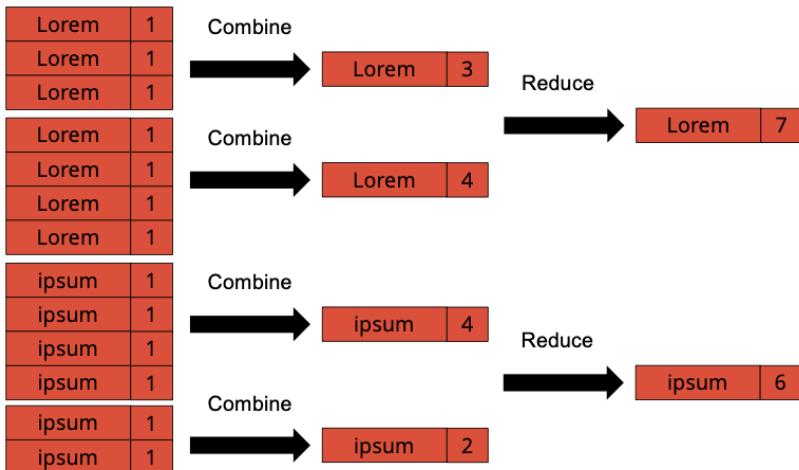


Thus, in addition to the map function and the reduce function, the user can supply a combine function. This combine function can then be called by the system during the map phase as many times as it sees fit to “compress” the intermediate key-value pairs. Strategically, the combine function is likely to be called at every flush of key-value pairs to a Sequence File on disk, and at every compaction of several Sequence Files into one.

However, there is no guarantee that the combine function will be called at all, and there is also no guarantee on how many times it will be called. Thus, if the user provides a combine function, it is important that they think carefully about a combine function that does not affect the correctness of the output data. In fact, in most of the cases, the combine function will be identical to the reduce function, which is generally possible if the intermediate key-value pairs have the same type as the output key-value pairs, and the reduce function is both associative and commutative. This is the case for summing or multiplying values² as well as for taking the maximum or the minimum, but not for an unweighted average (why?). As a reminder, associativity means that $(a + b) + c = a + (b + c)$ and commutativity means that $a + b = b + a$.

In our example, the reduce function fulfils these criteria and can then be used as a combine function as well:

²Caution is advised for IEEE754 floats and doubles: it is then not strictly commutative, although for many users, it is commutative “for all practical purposes.”



8.8 MapReduce programming API

Let us now move on to the concrete use of MapReduce in a computer program.

8.8.1 Mapper classes

In Java, the user needs to define a so-called Mapper class that contains the map function, and a Reducer class that contains the reduce function.

A map function takes in particular a key and a value. Note that it outputs key-value pairs via the call of the write method on the context, rather than with a return statement. That way, it can output zero, one or more key-values. A Mapper class looks like so:

```
import org.apache.hadoop.mapreduce.Mapper;
public class MyOwnMapper extends Mapper<K1, V1, K2, V2>{
    public void map(K1 key, V1 value, Context context)
        throws IOException, InterruptedException
    {
        ...
        K2 new-key = ...
        V2 new-value = ... context.write(new-key, new-value); ...
    }
}
```

8.8.2 Reducer classes

A reduce function takes in particular a key and a list of values. Note that it outputs key-value pairs via the call of the write method on the context, rather than with a return statement. That way, it can output zero, one or more key-values. A Reducer class looks like so:

```
import org.apache.hadoop.mapreduce.Reducer;
public class MyOwnReducer extends Reducer<K2, V2, K3, V3> {
    public void reduce (
        K2 key,
        Iterable<V2> values,
        Context context)
        throws IOException, InterruptedException
    {
        ...
        K3 new-key = ...
        V3 new-value = ... context.write(new-key, new-value); ...
    }
}
```

8.8.3 Running the job

Finally, a MapReduce job can be created and invoked by supplying a Mapper and Reducer instance to the job, like so:

```
import org.apache.hadoop.mapreduce.Job;
public class MyMapReduceJob {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");

        job.setMapperClass(MyOwnMapper.class);
        job.setReducerClass(MyOwnReducer.class);

        FileInputFormat.addInputPath(job, ...);
        FileOutputFormat.setOutputPath(job, ...);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

A combine function can also be supplied with the setCombinerClass method (passing, for example, the same Reducer instance).

It is also possible to use Python rather than Java, via the so-called Streaming API. The Streaming API is the general way to invoke MapReduce jobs in other languages than Java (or than JVM languages,

like Scala). This is done by creating two files, say, mapper.py and reducer.py. These two files take input (or intermediate) key-value pairs from standard input, and write intermediate (or output) key-value pairs to standard output. They are then invoked on the command line:

```
$ hadoop jar hadoop-streaming*.jar \
-files mapper.py,reducer.py \
-mapper mapper.py \
-reducer reducer.py \
-input input \
-output output
```

Without going too much in details, the input formats and output formats can also be specified in all programming languages. In Java, this is in the form of picking Java classes inheriting from InputFormat (DBInputFormat for a relational database, TableInputFormat for HBase, KeyValueTextInputFormat, SequenceFileInputFormat, TextInputFormat, FixedLengthInputFormat, NLineInputFormat...) or OutputFormat (DBOutputFormat, TableOutputFormat, SequenceFileOutputFormat, TextOutputFormat, MapFileOutputFormat...).

8.9 Using correct terminology

8.9.1 A warning

Let us now have a word of warning: the terminology “Mapper” and “Reducer” should only be used in the context of naming classes and files, but never when describing the MapReduce architecture. Even less so with “Combiner.”

This is because this terminology is very imprecise and makes it very difficult to comprehend the MapReduce architecture. Sadly, many resources in books and on the Web do refer to mappers, reducers and combiners. We hope the reader will find what follows enlightening and helpful to truly understand what is going on in a MapReduce cluster.

Rather than “mapper”, we encourage the reader to use “map function”, “map task”, “map slot” or “map phase” depending on what is meant. Rather than “reducer”, we encourage the reader to use “reduce function”, “reduce task”, “reduce slot” or “reduce phase” depending on what is meant. Rather than “combiner”, we encourage the reader to use “combine function” – there is no such thing as a combine task, a combine slot or a combine phase.

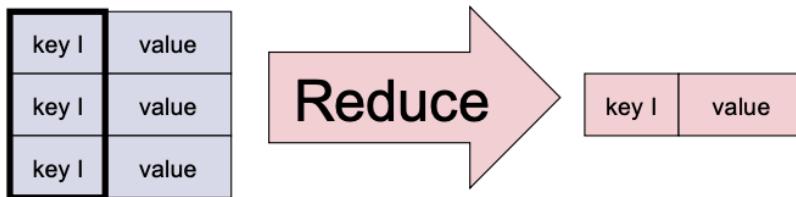
8.9.2 Functions

Let us start with functions.

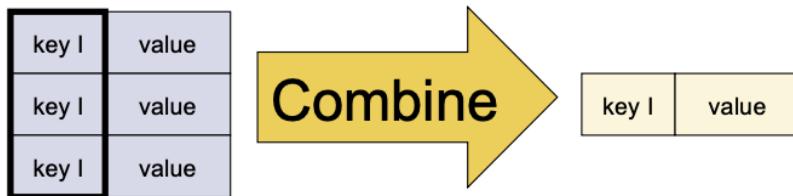
A map function is a mathematical, or programmed, function that takes one input key-value pair and returns zero, one or more intermediate key-value pairs.



A reduce function is a mathematical, or programmed, function that takes one or more intermediate key-value pairs and returns zero, one or more output key-value pairs.



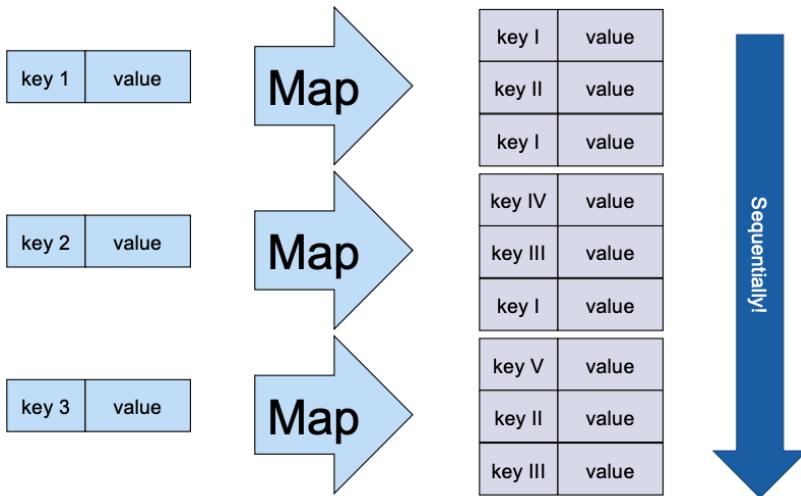
A combine function is a mathematical, or programmed, function that takes one or more intermediate key-value pairs and returns zero, one or more intermediate key-value pairs.



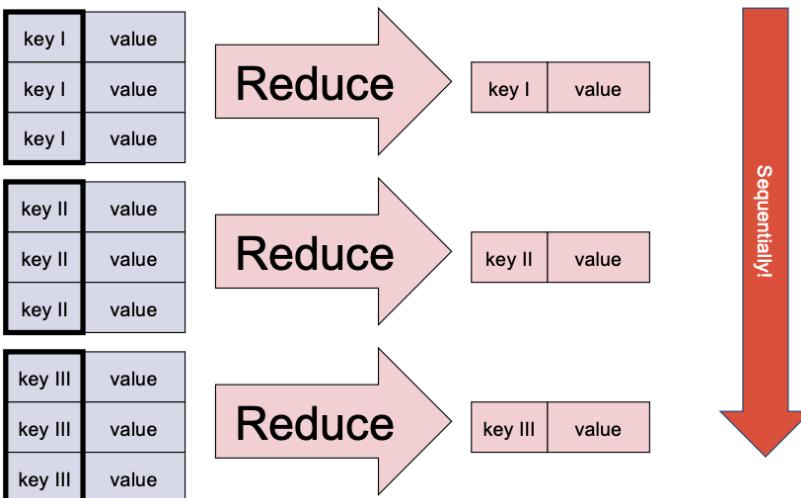
8.9.3 Tasks

Then, a map task is an *assignment* (or “homework”, or “TODO”) that consists in a (sequential) series of calls of the map function on a subset

of the input. There is one map task for every input split, so that there are many map tasks as partitions of the input.



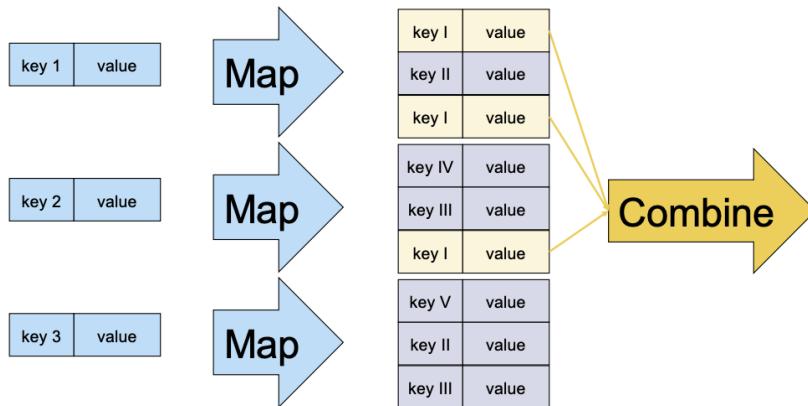
A reduce task is an assignment that consists in a (sequential) series of calls of the reduce function on a subset of the intermediate input. There are as many reduce tasks as partitions of the list of intermediate key-value pairs.



We insist that the calls within a task are sequential, meaning that there is no parallelism at all within a task. You can think of it as a for

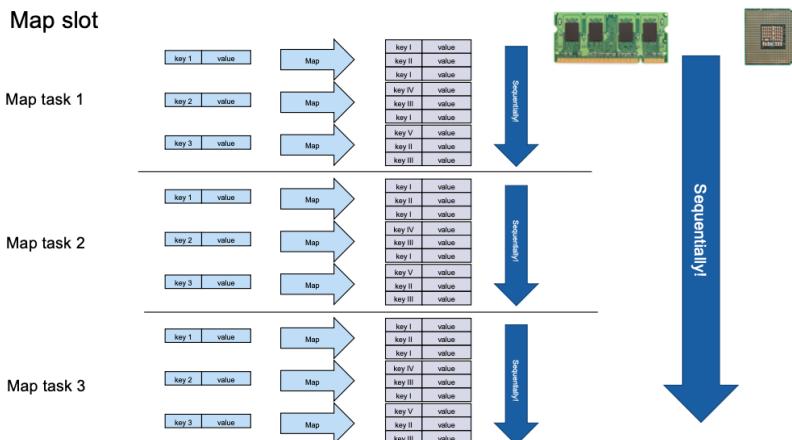
loop calling the function repeatedly, with the size of the for loop being, in a typical setting, between 1,000 and 1,000,000 calls.

There is no such thing as a combine task. Calls of the combine function are not planned as a task, but is called ad-hoc during flushing and compaction.

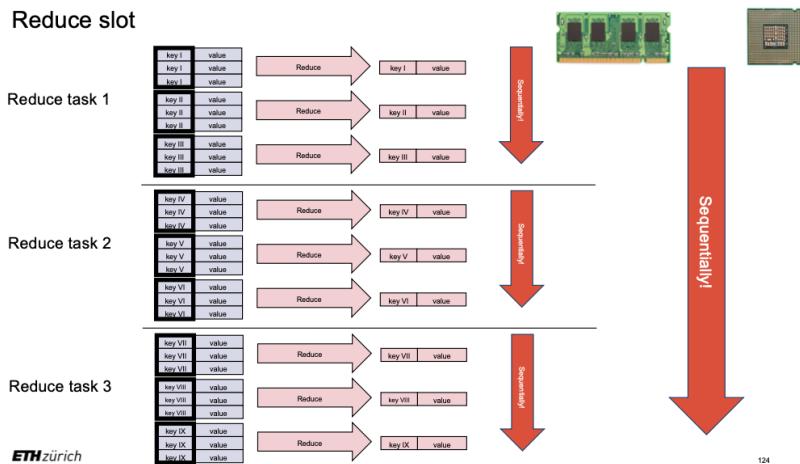


8.9.4 Slots

The map tasks are processed thanks to compute and memory resources (CPU and RAM). These resources are called map slots. One map slot corresponds to one CPU core and some allocated memory. The number of map slots is limited by the number of available cores. Each map slot then processes one map task at a time, sequentially. This means that the same map slot can process several map tasks.



The resources used to process reduce tasks are called reduce slots. Again, one reduce slot corresponds to one CPU core and some allocated memory. The number of reduce slots is limited by the number of available cores. Each reduce slot then processes one reduce task at a time, sequentially. This means that the same reduce slot can process multiple reduce tasks.

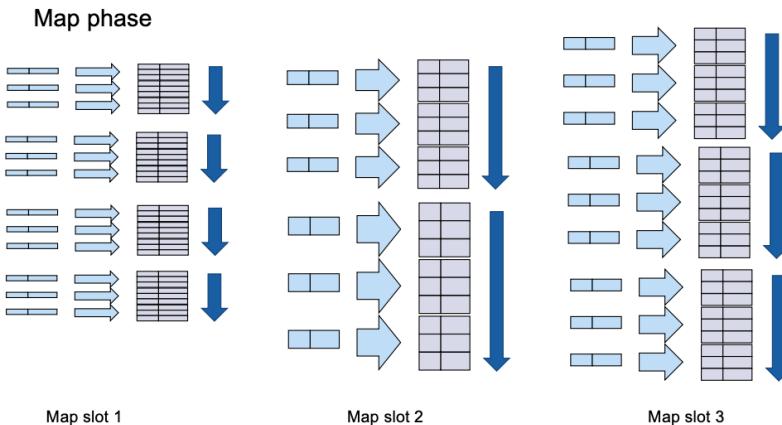


So, there is no parallelism either within one map slot, or one reduce slot. In fact, parallelism happens across several slots. In a typical MapReduce job, there will be more tasks than slots. Initially, each slot will receive one task, and the other tasks are kept pending. Every time a slot is done processing a task, it receives a new task from the pending list, and so on, until no task is left: then, some slots will remain idle until all tasks have been processed³. If a task fails, it can be reassigned to another slot.

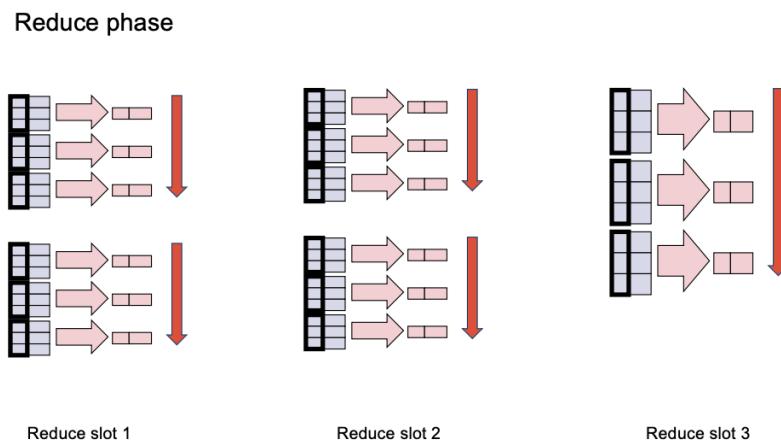
8.9.5 Phases

The map phase thus consists of several map slots processing map tasks in parallel:

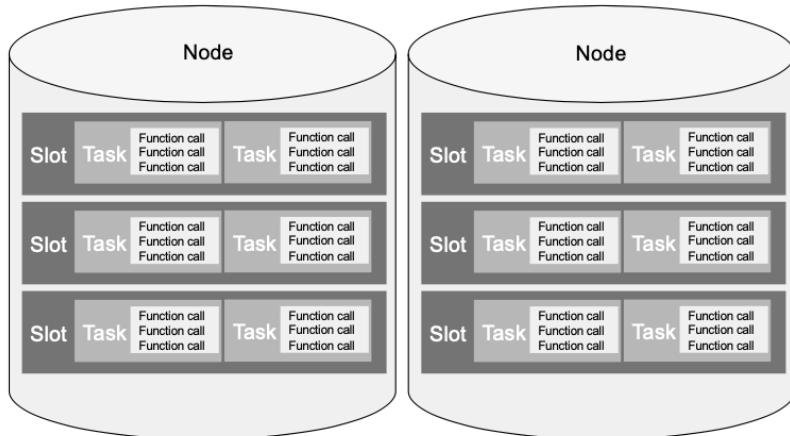
³For those who have already volunteered to count votes after an election or rotation, this will feel familiar, as votes are also processed in batches of, say, 100 ballots, by multiple groups across several tables. Next time you do, tell them they are actually using the MapReduce framework!



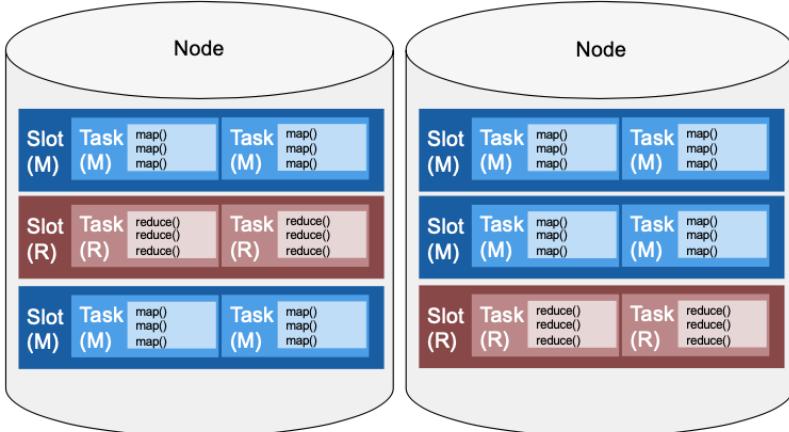
And the reduce phase consists of several reduce slots processing reduce tasks in parallel:



This is a summary of how functions, tasks, slots and phases fit together and within cluster nodes:



In the very first version of MapReduce (with a JobTracker and TaskTrackers), map slots and reduce slots are all pre-allocated from the very beginning, which blocks parts of the cluster remaining idle in both phases. We will see in the next Chapter that this can be improved.

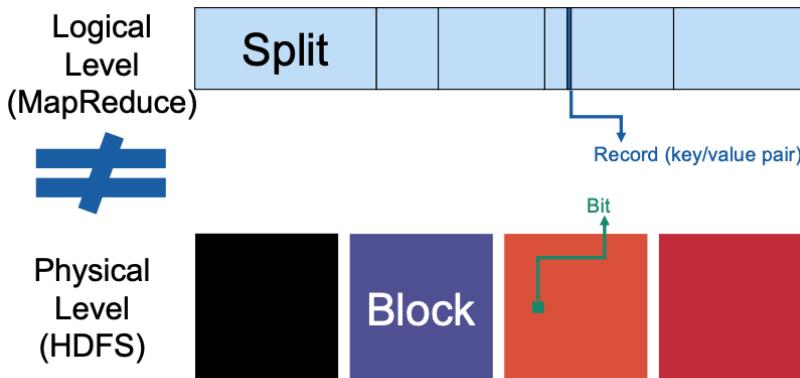


8.10 Impedance mismatch: blocks vs. splits

We finish this chapter with a more in-depth discussion of how MapReduce and HDFS interact. Now that we have the necessary terminology to express it, we can say that the “bring the query to the data”

paradigm means that the data belonging to the split that is the input of a map task resides on the same machine as the map slot processing this map task. How can it be? This is because the DataNode process of HDFS and the TaskTracker process of MapReduce are on the same machine. Thus, getting a replica of the block containing the data necessary to the processing of the task is as simple as a local read. This is called short-circuiting, the same name we gave to the local read of HFiles in HBase as well.

But there is something important to consider: HDFS blocks have a size of (at most) 128 MB. In every file, all blocks but the last one have a size of exactly 128 MB. Splits, however, only contain full records: a key-value pair will only belong to one split (and thus be processed by one map task). This means that, while most key-value pairs will be in the same block, the first and/or last key-value pair in a split will be spread across two blocks. This means that, while most of the data is obtained locally, getting the first and/or last record in full will require a remote read over the HDFS protocol. This, in turn, is also the reason why the HDFS API gives the ability to only read a block partially.



8.11 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain the patterns that appear in large-scale data processing: map, shuffle?
- b. Can you explain how the mastery of these patterns can bring significant improvements in performance?
- c. Can you explain the MapReduce model, also in terms of the patterns depicted above (map, shuffle, reduce)?
- d. Can you describe the physical architecture of MapReduce (map tasks, reduce tasks as well as data flow)? Can you explain version 1 of the architecture (JobTracker, TaskTrackers) and its limits?
- e. Can you outline what a map function looks like, and what a reduce function looks like?
- f. Do you understand the difference between a map function, a map task, a map slot and a map phase?
- g. Do you understand the difference between a reduce function, a reduce task, a reduce slot and a reduce phase?
- h. Do you know the main input and output formats of MapReduce?
- i. Can you explain how combining improve MapReduce's performance?
- j. Do you understand what a combine function is? Is there anything at all called a “combine task”, or a “combine slot” or a “combine phase”?
- k. Can you state the assumptions behind reusing the reduce function as a combine function?
- l. Do you understand, in some simple cases, how to design a combine function that would make a MapReduce job faster, even if the combine function is not the exact same as the reduce function? (Example: computing an average, which requires keeping track of the weights in the output of the combine function).
- m. Can you explain why MapReduce shines on top of a distributed file system: “Bring the query to the data”?
- n. Can you explain why MapReduce especially makes sense when the bottleneck is the speed of reading and writing data from the disk (as opposed to other bottlenecks such as storage capacity or CPU usage)?

- o. Can you explain how MapReduce splits differ from HDFS blocks, what impedance mismatches arise and how they are addressed?
- p. Do you know what the Java API of MapReduce looks like on a high level (Version 2 of the API that we covered, not to be confused with Version 2 of MapReduce running on YARN)?

8.12 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Dean, J. et al. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI.

White, T. (2015). *Hadoop: The Definitive Guide*. 4th edition. O'Reilly. Chapters 2, 6, 7, 8, 9.

MapReduce tutorial (on hadoop.apache.org/docs/current, in the documentation page)

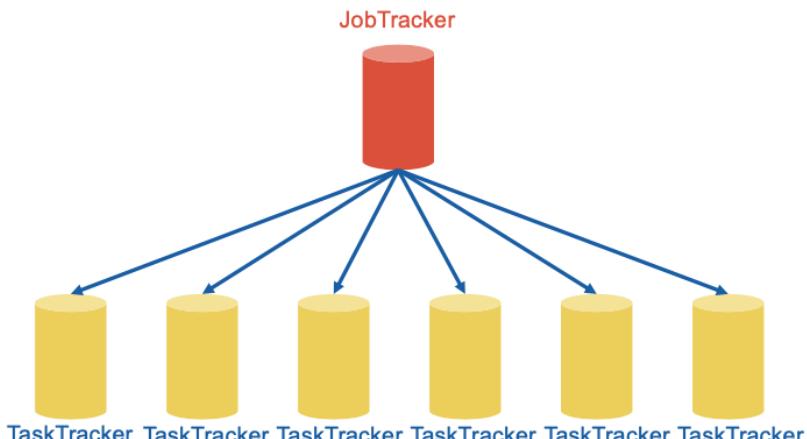
Chapter 9

Resource management

The first version of MapReduce is inefficient in several respects. For this reason, the architecture was fundamentally changed by adding a resource management layer to the stack, adding one more level of decoupling between scheduling and monitoring. A resource management system, here YARN, is a very important building block not only for a better MapReduce, but also for many other technologies running on a cluster (Generalized Parallel Processing, e.g., with Apache Spark, which we will cover in the next chapter, Message-Passing Interface, Graph computing, etc.).

9.1 Limitations of MapReduce in its first version

Recollect that the first version of MapReduce is based on a centralized architecture with a JobTracker, the coordinator node, and TaskTrackers, the worker nodes.



The JobTracker has a lot on its shoulders! It has to deal with resource management, scheduling, monitoring, the job lifecycle, and fault tolerance.

The first consequence of this is scalability: things start breaking beyond 4,000 nodes and/or 40,000 tasks.

The second consequence is the bottleneck that this introduces at the JobTracker level, which slows down the entire system.

The third issue is that it is difficult to design a system that do many things well: “Jack of all trades, master of none”.

The fourth issue is that resources are statically allocated to the Map or the Reduce phase, meaning that parts of the cluster remain idle during both phases.

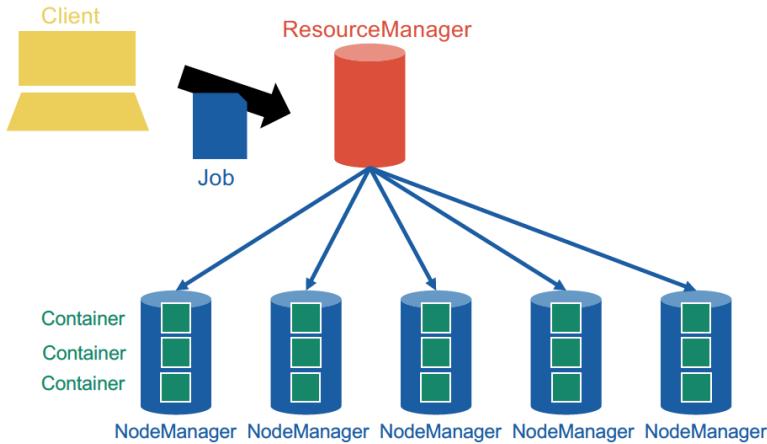
The fifth issue is the lack of fungibility between the Map phase and the Reduce phase: the system is closely tied to the two-phase mechanism of MapReduce, in spite of these two phases having a lot in common in terms of parallel execution.

9.2 YARN

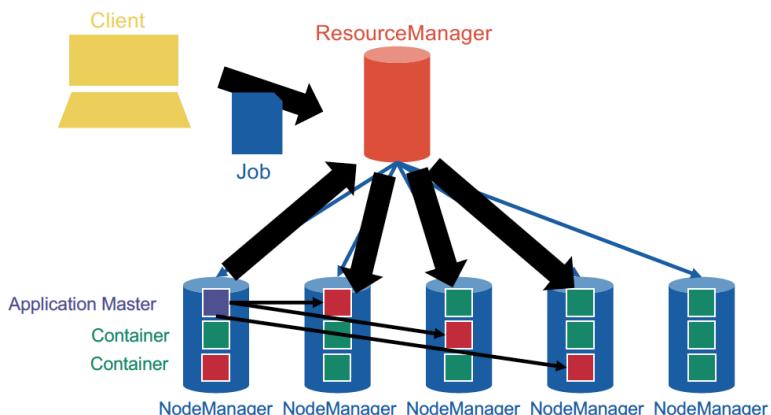
9.2.1 General architecture

YARN means Yet Another Resource manager. It was introduced as an additional layer that specifically handles the management of CPU and memory resources in the cluster.

YARN, unsurprisingly, is based on a centralized architecture in which the coordinator node is called the ResourceManager, and the worker nodes are called NodeManagers. NodeManagers furthermore provide slots (equipped with exclusively allocated CPU and memory) known as containers.



YARN provides generic support for allocating resources to any application and is application-agnostic. When the user launches a new application, the ResourceManager assigns one of the container to act as the ApplicationMaster which will take care of running the application. This is a fundamental change from the initial MapReduce architecture, in which the JobTracker was *also* taking care of running the MapReduce job. The ApplicationMaster can then communicate with the ResourceManager in order to book and use more containers in order to run jobs.



Thus, YARN cleanly separates between the general management of resources and bootstrapping new applications, which remains central-

ized on the coordinator node, and monitoring the job lifecycle, which is now delegated to one *or more* ApplicationMasters running concurrently. This means, in particular, that several applications can run concurrently in the same cluster. This ability, known as multi-tenancy, is very important for large companies or universities in order to optimize the use of their resources. In fact, contrary to popular belief, this ecosystem was not meant to optimize performance, but to optimize resource use and costs. Technologies like MapReduce or Apache Spark are often described as slow, but this is a misunderstanding of the fact that these technologies were never designed for a *single user* running *just one job* in their own cluster. When there is only one user, this user typically has more control over the cluster and can optimize things in better ways than they can in a shared environment (an example of that is High-Performance Computing).

9.2.2 Resource management

In resource management, one abstracts away from hardware by distinguishing between four specific resources used in a distributed database system. These four resources are:

- Memory
- CPU
- Disk I/O
- Network I/O

Most resource management systems (including YARN), today, focus mainly on allocating and sharing memory and CPU, but there is also a lot of ongoing work and process to allocate and share disk and network I/O.

ApplicationMasters can request and release containers at any time, dynamically. A container request is typically made by the ApplicationMasters with a specific demand (e.g., “10 containers with each 2 cores and 16 GB of RAM”). If the request is granted by the ResourceManager fully or partially, this is done indirectly by signing and issuing a container token to the ApplicationMaster that acts as proof that the resource was granted.

The ApplicationMaster can then connect to the allocated NodeManager and send the token. The NodeManager will then check the validity of the token and provide the memory and CPU granted by the ResourceManager. The ApplicationMaster ships the code (e.g., as a jar file) as well as parameters, which then runs as a process with exclusive use of this memory and CPU.

To bootstrap a new application, the ResourceManager can also issue application tokens to external clients so they can start the ApplicationMaster.

9.2.3 Job lifecycle management and fault tolerance

Version 2 of MapReduce works on top of YARN by leaving the job lifecycle management to an ApplicationMaster. The ApplicationMaster requests containers for the Map phase, and sets these containers up to execute Map tasks. As soon as a container is done executing a Map task, the ApplicationMaster will assign a new Map task to this container from the remaining queue, until no Map tasks are left.

Note that there is a subtle distinction between a slot and a container. Indeed, containers that have several cores can run several Map tasks concurrently (sharing the same memory space). In other words, a container in the Map phase can contain several Map slots. Sharing memory and containers across slots in this way improves the overall efficiency, because setting up a container adds latency. It is thus more efficient to allocate 10 containers of each 4 cores, compared to 40 containers of each 1 core. Likewise, sharing memory does not mean that the threads executing the Map tasks within the same container communicate or are entangled in any way (they each live their own life), but rather allows a better “spread” of the difference of memory usage across tasks.

When the end of the Map phase approaches, the ApplicationMaster then starts allocating containers for the Reduce phase and initiates shuffling. When the Map phase is complete, these new slots can then start processing Reduce tasks and the Map containers can be released. When the output has been written, the Reduce containers are also freed and returned to YARN for other users to book.

In the event of a container crashing during the Map phase, the ApplicationMaster will handle this by re-requesting containers and restarting the failed tasks. In the case that some data is lost in the Reduce phase, it is possible that the entire job must be restarted, because this is the only way to recreate the intermediate data is to re-execute the Map tasks.

Finally, version 2 of MapReduce supports 10,000 nodes and 100,000 tasks, which is an improvement on version 1.

9.2.4 Scheduling

The ResourceManager decides whether and when to grant resource requests based on several factors: capacity guarantees, fairness, service level agreements (remember the numbers with plenty of 9s?) and with the goal to maximize cluster utilization. It provides both an interface to

clients (users) and maintains a queue of applications with their status, as well as statistics. It also provides an admin interface to configure the queue.

The ResourceManager keeps track of the list of available NodeManagers (who can dynamically come and go) and their status. Just like in HDFS, NodeManagers send periodic heartbeats to the ResourceManager to give a sign of life.

The ResourceManager also offers an interface so that ApplicationMasters can register and send container requests. ApplicationMasters also send periodic heartbeats to the ResourceManager.

It is important to understand that, unlike the JobTracker, the ResourceManager does not monitor tasks, and will not restart slots upon failure. This job is left to the ApplicationMasters.

9.3 Scheduling strategies

There are several possible scheduling strategies, common to many systems with multi-tenancy.

9.3.1 FIFO scheduling

In FIFO scheduling, there is one application at a time running on the entire cluster. When it is done, the next application runs again on the entire cluster, and so on.



9.3.2 Capacity scheduling

In capacity scheduling, the resources of the cluster are partitioned into several sub-clusters of various sizes: these can correspond, for example, to subdivisions of a company or university: the CS department, the Mathematics department, etc.

Each one of these sub-clusters has its own queue of applications running in a FIFO fashion within this queue.

It is also possible to have more hierarchical levels: sub-sub-clusters. This is called a hierarchical queue. Applications then run with FIFO scheduling on the leaves of the hierarchical queue, which shows that scheduling strategies can mix with each other in a recursive fashion.

Capacity scheduling also exists in a more “dynamic flavour” in which, when a sub-cluster is not currently used, its resources can be temporarily lent to the other sub-clusters. This is also in the spirit of usage maximization, so that the company as a whole will not waste unused resources.



9.3.3 Fair scheduling

Fair scheduling involves more complex algorithms that attempt to allocate resources in a way fair to all users of the cluster and based on the share they are normally entitled to. Fair scheduling also typically includes economic and game theory thinking, which is necessary to ensure a smooth cluster management with a large number of users within the same company.

Fair scheduling should be understood in a dynamic fashion: the cluster has, at any point in time, users from various departments running their applications. Applications are dynamically and regularly requesting many containers with specific memory and CPU requirements, and releasing them again. Thus, fair scheduling consists on making dynamic decisions regarding which requests get granted and which requests have to wait.



Fair scheduling combines several ways to compute cluster shares:

- Steady fair share: this is the share of the cluster officially allocated to each department. The various department agree upon this with each other in advance (e.g., based on their financial contribution to the maintenance of the cluster). This number is thus static and rarely changes. It is the theoretical share that each department would normally get if all departments constantly use the cluster.
- Instantaneous fair share: this is the fair share that a department should ideally be allocated (according to economic and game theory considerations) at any point in time. This is a dynamic number that changes constantly, based on departments being idle: if a department is idle, then the instantaneous fair share of others department becomes higher than their steady fair shares.
- Current share: this is the actual share of the cluster that a department effectively uses at any point in time. This is highly dynamic. The current share does not necessarily match the instantaneous fair share because there is some inertia in the process: a department might be using more resources while another is idle. When the other department later stops being idle, these resources are not immediately withdrawn from the first department; rather, the first department will stop getting more resources, and the second department will gradually recover these resources as they get released by the first department. This behavior can be changed with preemption and overrides with priority levels, in which case the resources are forcibly and immediately taken back from the

first department (which typically leads to failed tasks or even failed jobs) and handed over to the second department.

The easiest case of fair scheduling is when only one resource is considered: for example, only CPU cores, or only memory. In this case, the algorithm is relatively simple: the requests by users who are significantly below their instantaneous fair share get prioritized over those who are above (or closer to) their instantaneous fair share.

Things become more complicated when several resources are considered, in practice both CPU cores and memory. This is because one application may request many containers with, say, each one core and 100 GB of memory, while another might request many containers with each 4 cores and 10 GB of memory. How can we, then, define what the “current share” is, as they are based on only one dimension?

This problem was solved game-theoretically with the Dominant Resource Fairness algorithm. The two (or more) dimensions are projected again to a single dimension by looking at the dominant resource for each user. For example, imagine the total size of the cluster is 1000 cores and 10 TB of memory. Then, a container with one core and 100 GB of memory would use 0.1% of the cluster cores and 1% of the cluster memory. Its dominant resource is memory and it is thus the 1% that is considered. Another container with 4 cores and 10 GB of memory would use 0.4% of the CPU cores and 0.1% of the memory. Then, CPU cores are the dominant resource, and it is 0.4% that is considered. Let us take the example where both users have an instantaneous fair share of 50% each. In this case, the fair ratio of the containers should be 2 to 5: each time the first user gets 2 containers of one core and 100 GB of memory, the other user gets 5 containers of 4 cores and 10 GB of memory. That way, both have the same “dominant resource usage”: they each get 2% of the cluster every time the first user gets 2 containers (2 times 1%) and the second user gets 5 containers (5 times 0.4%)¹. The algorithm grants requests in order to get as close as possible to this fair ratio in cruise mode.

¹Note that the sum of the actual dominant resource percentages might exceed 100% in full cluster use. Try to figure out why!

9.4 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain how YARN works, and how it can be used to improve MapReduce, and to support other technologies like Spark?
- b. Can you quickly describe the YARN components?
- c. Do you know what a ResourceManager does?
- d. Do you know what a NodeManager does?
- e. Do you know what and where a Container is?
- f. Do you know what an ApplicationMaster is and does?
- g. Can you list the main resources that are managed in a cluster? (Disk storage, memory, CPU, network bandwidth).
- h. Can you explain, in simpler words, what the added value of YARN is? Can you explain what it is an improvement over the first version of MapReduce, which was taking care of resource management on its own, and had issues with this?

Chapter 10

Generic Dataflow Management

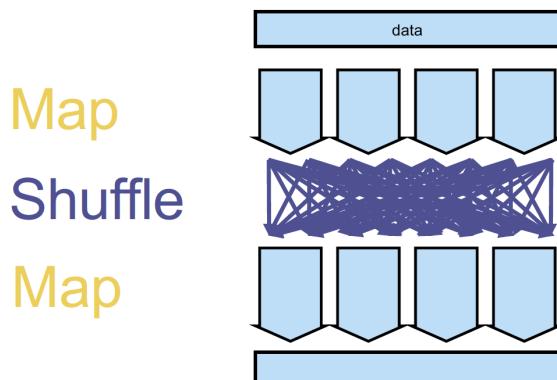
MapReduce is very simple and generic, but many more complex uses involve not just one, but a sequence of several MapReduce jobs. Furthermore, the MapReduce API is low-level, and most users need higher-level interfaces, either in the form of APIs or query languages.

This is why, after MapReduce, another generation of distributed processing technologies were invented. The most popular one is the open source Apache Spark.

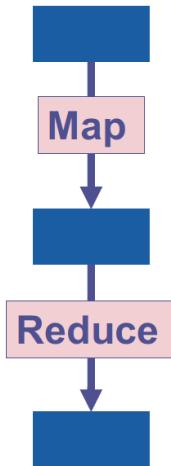
10.1 A more general dataflow model

MapReduce consists of a map phase, followed by shuffling, followed by a reduce phase. In fact, the map phase and the reduce phase are not so different: both involve the computation of tasks in parallel on slots.

One could exaggerate a bit by considering that MapReduce is in fact a map phase, then a shuffle, then a map phase (the reduce phase on the already shuffled data) again:

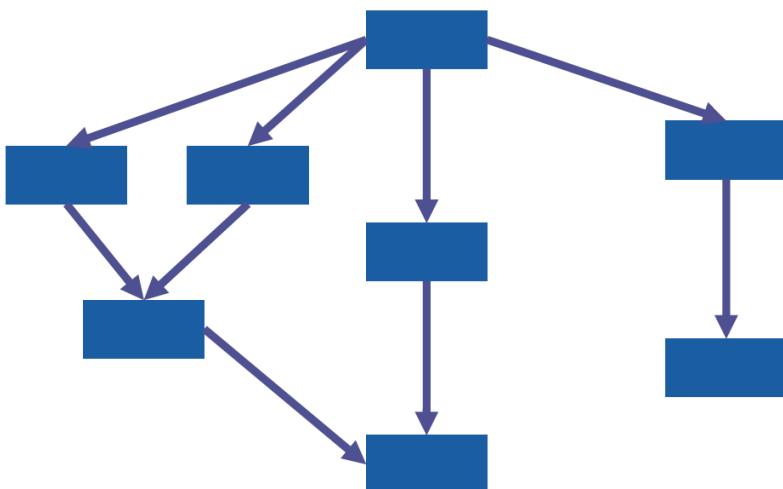


Or, if we abstract away the partitions and consider the input, intermediate input and output as blackboxes that these phases act on:



10.2 Resilient distributed datasets

The first idea behind generic dataflow processing is to allow the dataflow to be arranged in any distributed acyclic graph (DAG), like so:



All the rectangular nodes in the above graph correspond to intermediate data. They are called resilient distributed datasets, or in short, RDDs. Resilient means that they remain in memory or on disk on a “best effort” basis, and can be recomputed if need be. Distributed means that, just like the collections of key-value pairs in MapReduce, they are partitioned and spread over multiple machines.

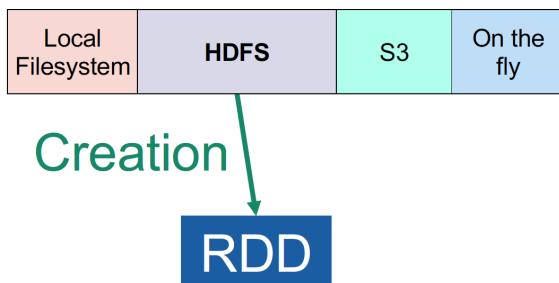
A major difference with MapReduce, though, is that RDDs need not be collections of pairs. In fact, RDDs can be (ordered) collections of just about *anything*: strings, dates, integers, objects, arrays, arbitrary JSON values, XML nodes, etc. The only constraint is that the values within the same RDD share the same static type, which does not exclude the use of polymorphism.

Since a key-value pair is a particular example of possible value, RDDs are a generalization of the MapReduce model for input, intermediate input and output.

10.3 The RDD lifecycle

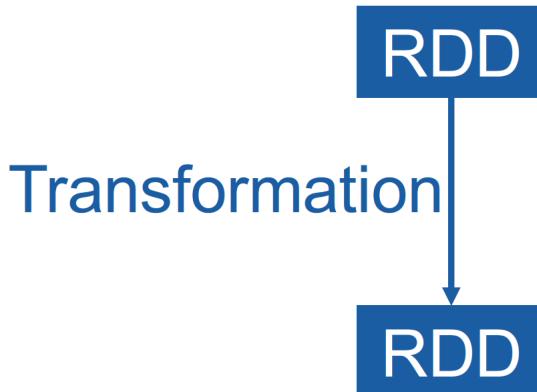
10.3.1 Creation

RDDs undergo a lifecycle. First, they get created. RDDs can be created by reading a dataset from the local disk, or from cloud storage, or from a distributed file system, or from a database source, or directly on the fly from a list of values residing in the memory of the client using Apache Spark:



10.3.2 Transformation

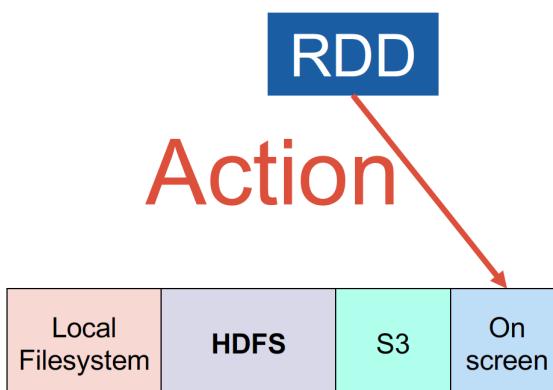
Then, RDDs can be transformed into other RDDs.



Mapping or reducing, in this model, become two very specific cases of transformations. However, Spark allows for many, many more kinds of transformations. This also includes transformations with several RDDs as input (think of joins or unions in the relational algebra, for example).

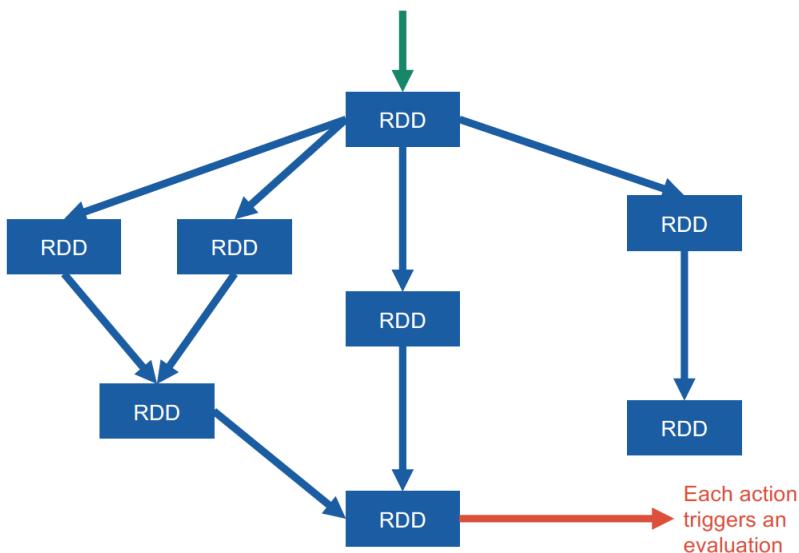
10.3.3 Action

RDDs can also undergo a final action leading to making an output persistent. This can be by outputting the contents of an RDD to the local disk, to cloud storage, to a distributed file system, to a database system, or directly to the screen of the user (assuming there is not intractably much data to display).



10.3.4 Lazy evaluation

Another important aspect of the RDD lifecycle is that the evaluation is lazy: in fact, creations and transformations on their own do nothing. It is only with an action that the entire computation pipeline is put into motion, leading to the computation of all the necessary intermediate RDDs all the way down to the final output corresponding to the action.



10.3.5 A simple example

Let us give a very simple example of RDD lifecycle. One easy way to start with Spark is by using a shell. Let us use a scala shell.

First, we can create an RDD with a simple list of two strings, and assign it to a variable, with the parallelize creation function, like so:

```
val rdd1 = sc.parallelize(
  List("Hello, World!", "Hello, there!")
)
```

Note, for completeness, that it would also be possible to read from an input dataset (let us assume a sharded dataset in some directory in HDFS configured as the default file system) with

```
val rdd1 = sc.textFile("/user/hadoop/directory/data-*txt")
```

Note that, at this point, nothing actually gets set into motion. But, for pedagogical purpose, this is what *will* be generated when the computation starts:

Value
Hello, World!
Hello, there!

Next, we can transform the RDD by tokenizing the strings based on spaces with a flatMap transformation. We assign the resulting RDD to another variable.

```
val rdd2 = rdd1.flatMap(
    value => value.split(" "))
)
```

Note that, at this point, still nothing actually gets set into motion, but `rdd2` would correspond do:

Value
Hello,
World!
Hello,
there!

Finally, we invoke the action `countByValue()`, which will return a (local) map structure (object, dict...) associating each value with a count. This is when the computations are actually triggered:

```
rdd2.countByValue()
```

The result will be displayed on the screen of the shell:

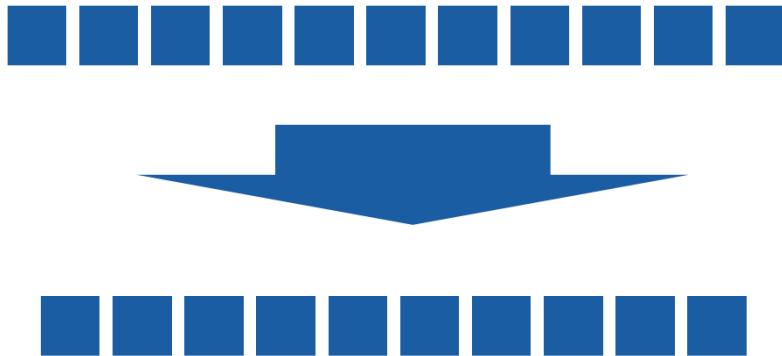
Key	Value
Hello,	2
there!	1
World!	1

10.4 Transformations

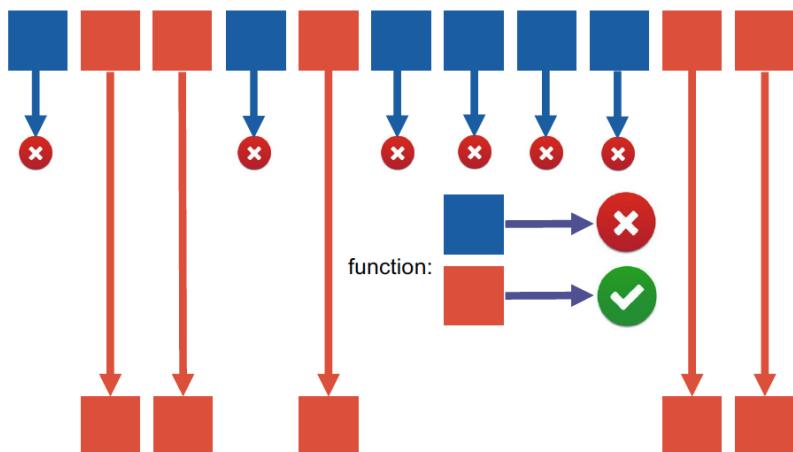
Transformations, in Spark, are like a large restaurant menu to pick from. We now go through the most important transformations.

10.4.1 Unary transformations

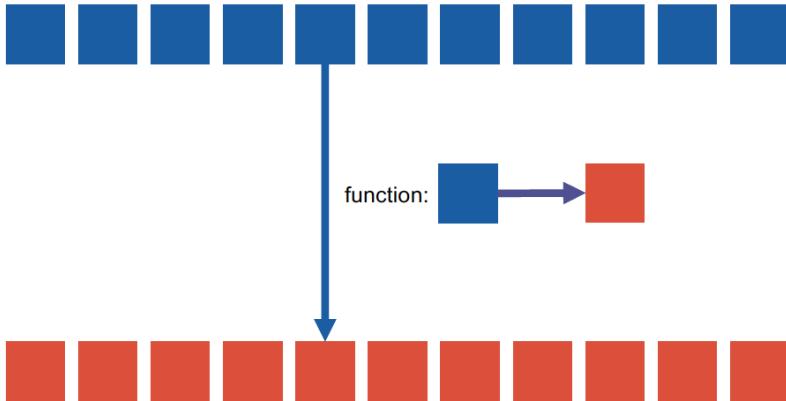
Let us start with unary transformations, i.e., those that take a single RDD as their input.



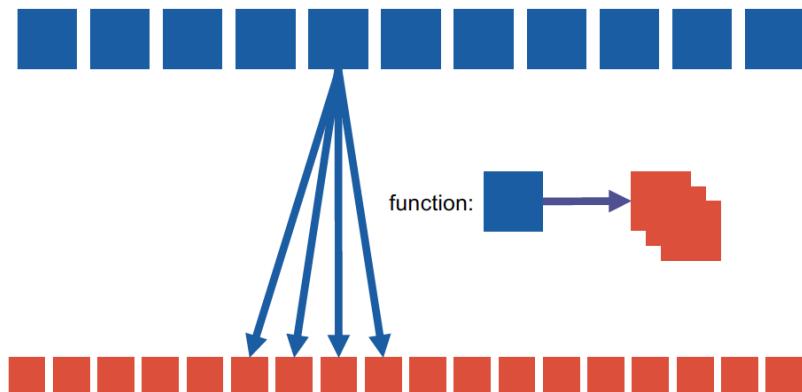
The filter transformation, provided a predicate function taking a value and returning a Boolean, returns the subset of its input that satisfies the predicate, preserving the relative order.



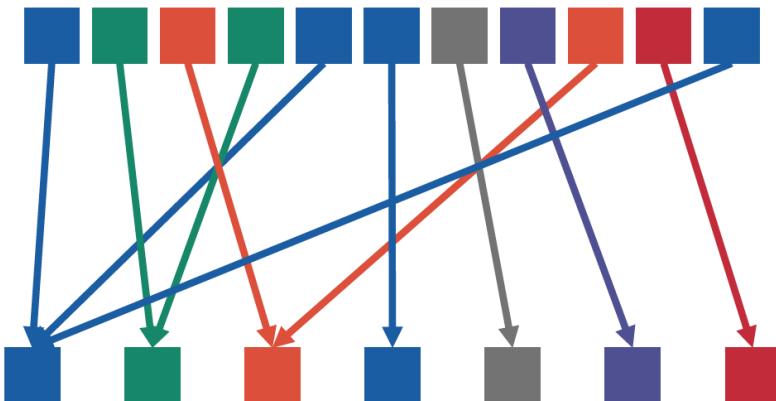
The map transformation, provided a function taking a value and returning another value, returns the list of values obtained by applying this function to each value in the input.



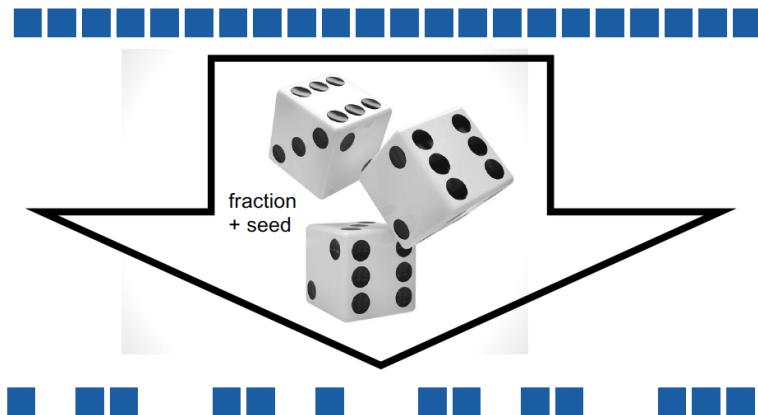
The flatMap transformation, provided a function taking a value and returning zero, one or more values, returns the list of values obtained by applying this function to each value in the input, flattening the obtained values (i.e., the information on which values came from the same input value is lost). The flatMap transformation, in fact, corresponds to the MapReduce map phase (in the special case that the RDDs involved are RDDs of key-value pairs).



The distinct transformation eliminates duplicates in the input. It is either possible to supply a comparison function, or to make sure that the class (type) of the input values implements the appropriate Comparable interface.

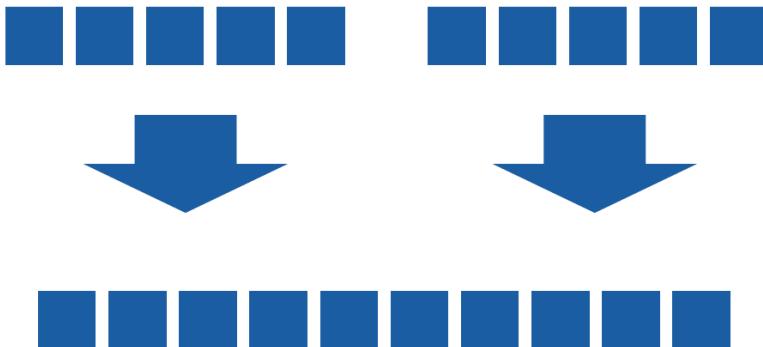


The sample transformation samples the input to a smaller list. It is possible to specify the sampling percentage.

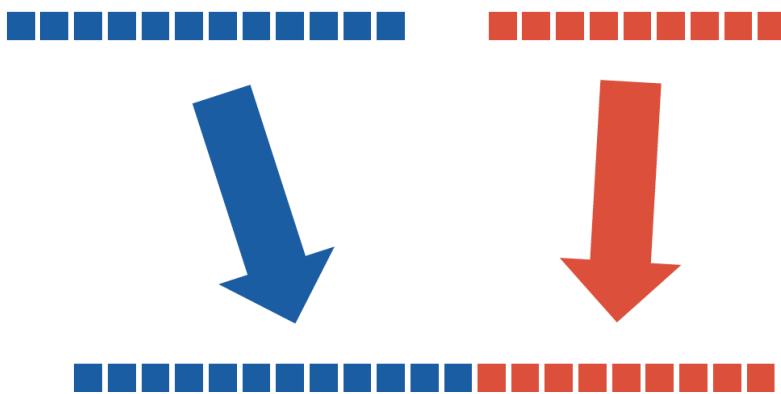


10.4.2 Binary transformations

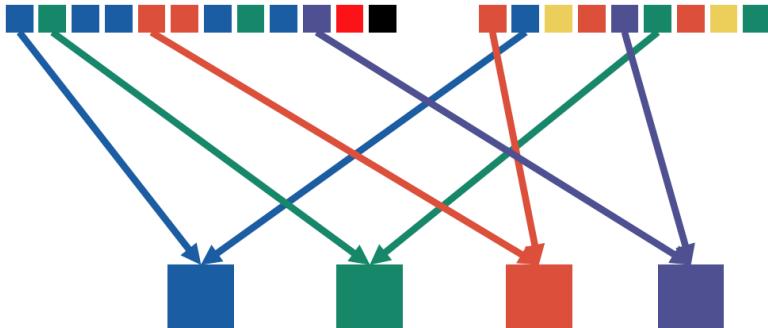
There are also transformations that take two RDDs as input.



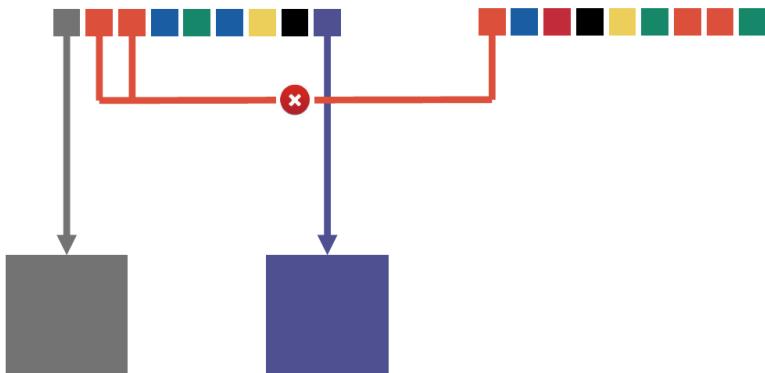
Three of them correspond to set operators in the relational algebra: for example, taking the union of two RDDs (with the same value type).



It is also possible to take the intersection.



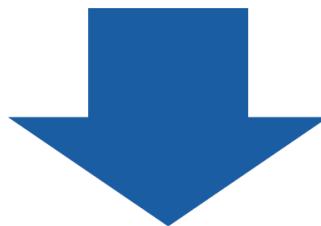
It is also possible to take the subtraction, i.e., only keep the elements from the left RDD that do not appear in the right RDD.



10.4.3 Pair transformations

Since RDDs can have values of any type, they can also in particular be pairs. Spark has transformations specifically tailored for RDDs of key-value pairs. Note that all other transformation also work on RDDs of key-value pairs, as they accept any values.

The key transformation returns a new RDD with only the keys:



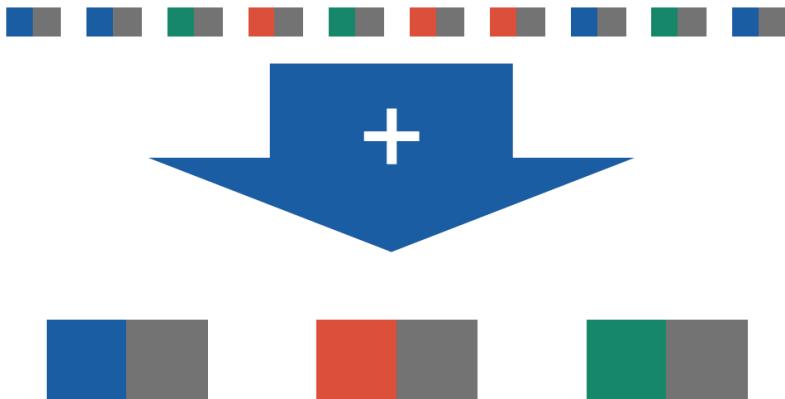
The values transformation returns a new RDD with only the values:



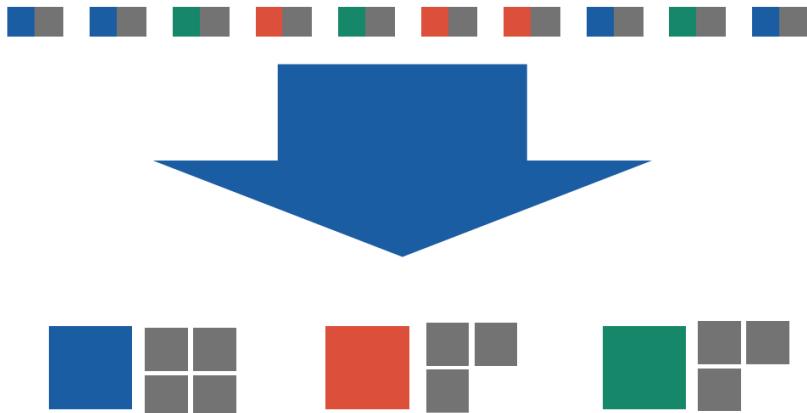
The reduceByKey transformation, given a (normally associative and commutative) binary operator under which the input data type is closed, invokes and chains this operator on all values of the input RDD that share the same key. For example, if values v_1, v_2, \dots, v_n are associated with key k each, in their own input pair, then the pair $(k, (v_1 + v_2 + \dots + v_n))$ is output assuming $+$ is the operator.

It is normally also required to tell Spark what the neutral element is (e.g., 0 for addition, 1 for multiplication, etc) so that it can use it for empty partitions during the parallel computation.

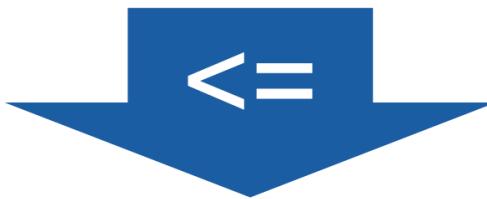
The reduceByKey transformation corresponds to the reduce phase of MapReduce.



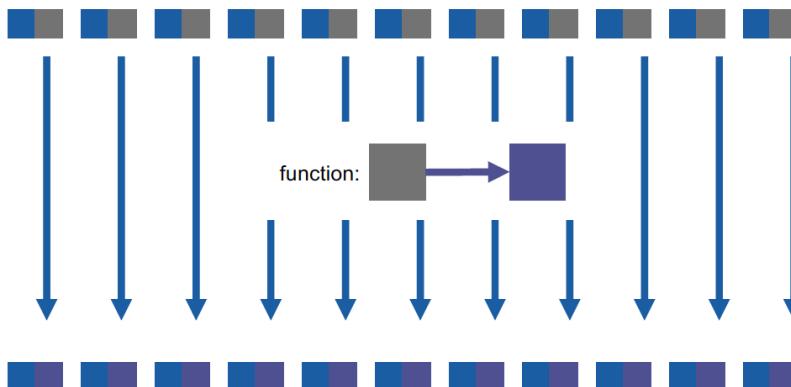
The `groupByKey` transformation groups all key-value pairs by key, and outputs a single key-value for each key, where the value is an array (or list) of all the values that were associated with this key in the input. For example, if the input contains pairs (k, a) and (k, b) and there is no other pair with key k in the input, then the output value corresponding to key k will be $(k, [a, b])$.



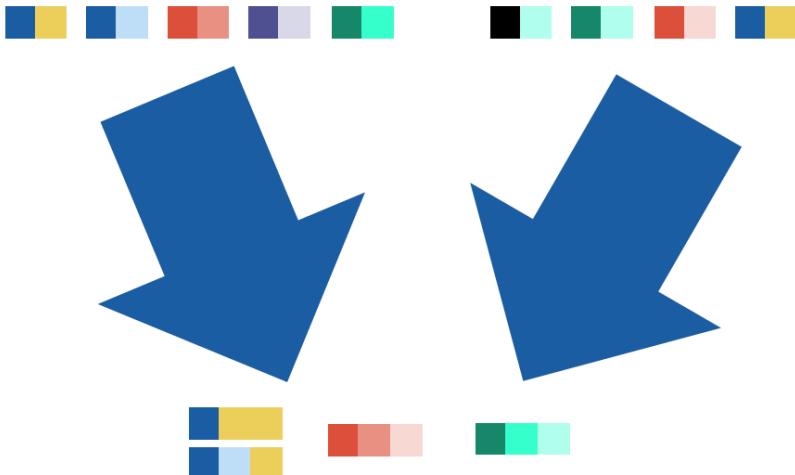
The `sortByKey` transformation, given the specification of an order or comparison operator on the key type, outputs the same pairs as in the input RDD, but reordered by key.



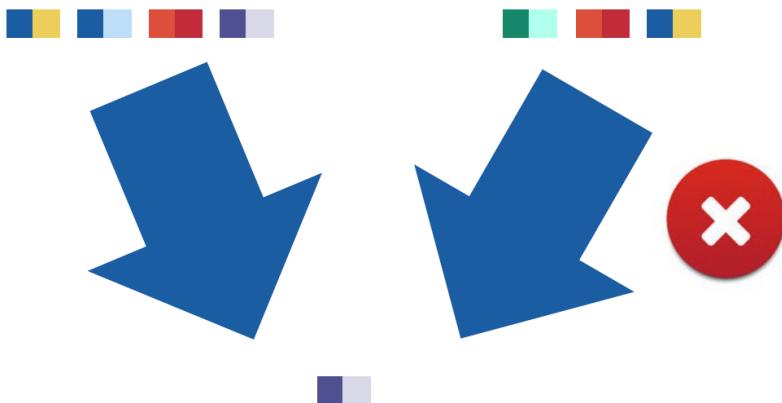
The mapValues transformation is similar to the map transformation, except that the map function is only applied to the value (and the key is kept).



The join transformation works on two input RDDs or key-value pairs. It matches the pairs on both sides that have the same key and outputs, for each match, an output pair with that shared key and a tuple with the two values from each side. If there are multiple values with the same key on any side (or both), then all possible combinations are output.



The `subtractByKey` transformation also takes two input RDDs of key-value pairs. It outputs all pairs of the left, except those who have a key present on the right.



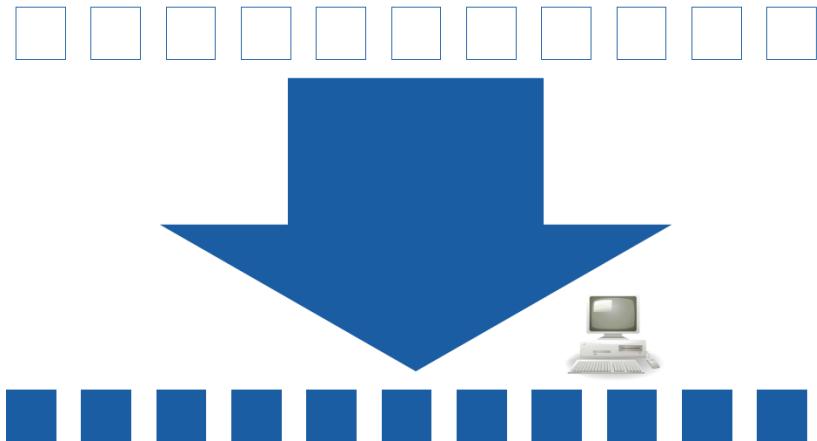
10.5 Actions

Let us now go through a few actions.

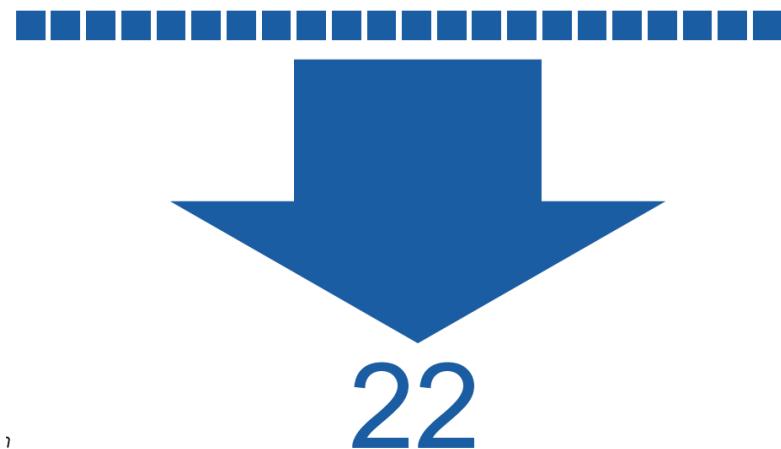
10.5.1 Gathering output locally

The `collect` action downloads all values of an RDD on the client machine and outputs them as a (local) list. It is important to only call this action

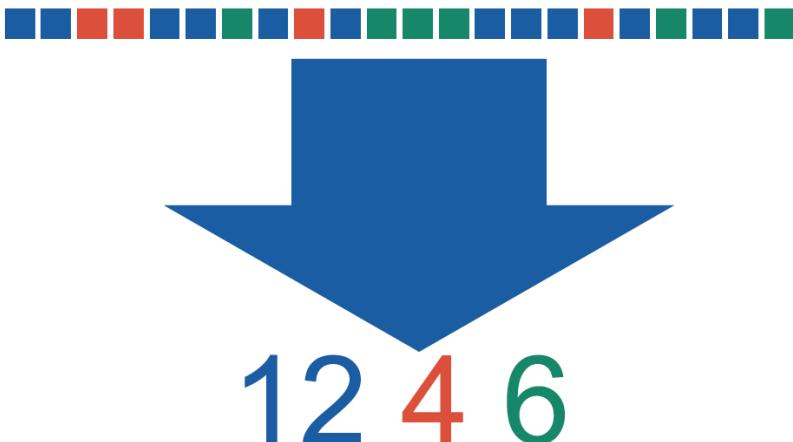
on an RDD that is known to be small enough (e.g., after filtering) to avoid a memory overflow.



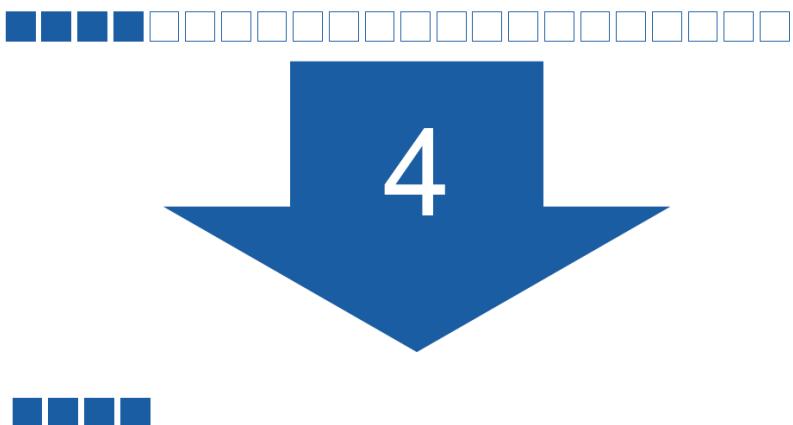
The count action computes (in parallel) the total number of values in the input RDD. This one is safe even for RDDs with billions of values, as it returns a simple integer to the client.



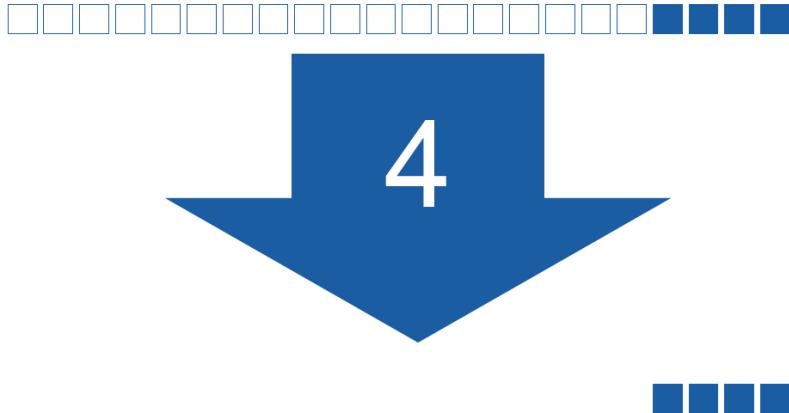
The countByValue action computes (efficiently, in parallel) the total number of occurrence of each distinct value in the input RDD. It is important to only call this action on an RDD that is known to have a small enough number of distinct values (e.g., after filtering) to avoid a memory overflow.



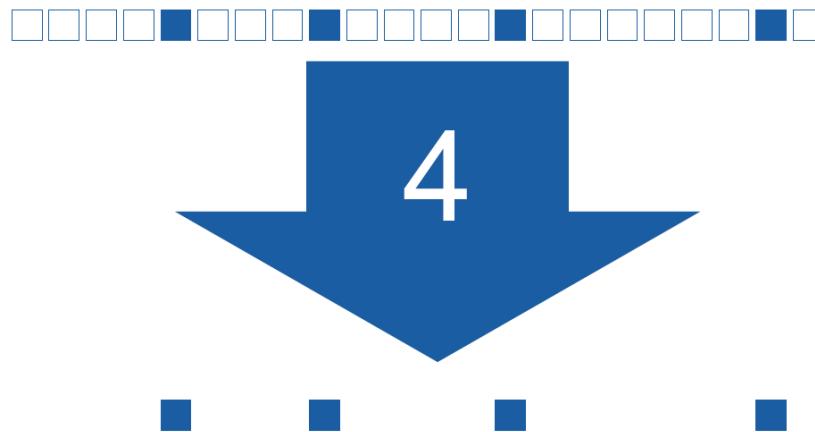
The `take` action returns, as a local list, the first n (where n can be chosen) values in the input RDD.



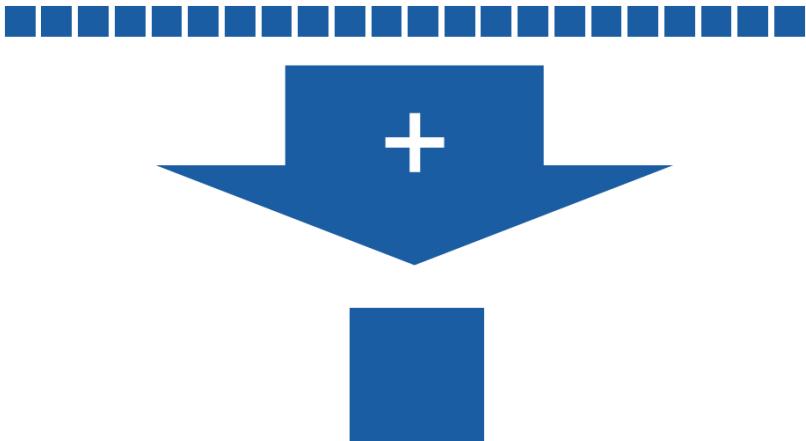
The `top` action returns, as a local list, the last n (where n can be chosen) values from the input RDD.



The `takeSample` action returns, as a local list, n randomly picked values from the input RDD.



The `reduce` action, given a (normally associative and commutative) binary operator under which the input data type is closed, invokes this operator on all values of the input RDD ($v_1 + v_2 + \dots + v_n$ if $+$ is the operator) and outputs the resulting value. It is normally also required to tell Spark what the neutral element is (e.g., 0 for addition, 1 for multiplication, etc) so that it can use it for empty subsets.



10.5.2 Writing to sharded datasets

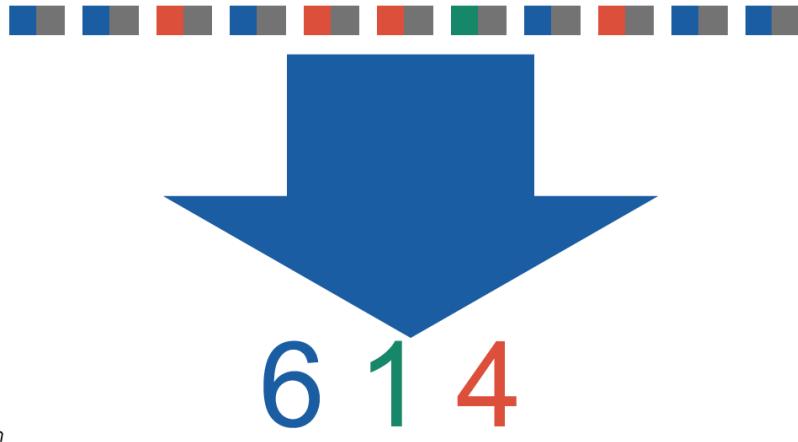
There is also an action called `saveAsTextFile` that can save the *entire* RDD to a sharded dataset on Cloud storage (S3, Azure blob storage) or a distributed file system (HDFS). It is very natural in Spark, like in MapReduce, to output sharded datasets, because this means the slots can write their own shards independently and without interfering or synchronizing with each other.

Binary outputs can be saved with `saveAsObjectFile`. Note that Spark, at least in its RDD API, is not aware of any particular format or syntax, i.e., it is up to the user to parse and serialize values to the appropriate text or bytes, typically with code in the host programming language (Python, Java, Scala...). See Chapter 7 for syntaxes and formats.

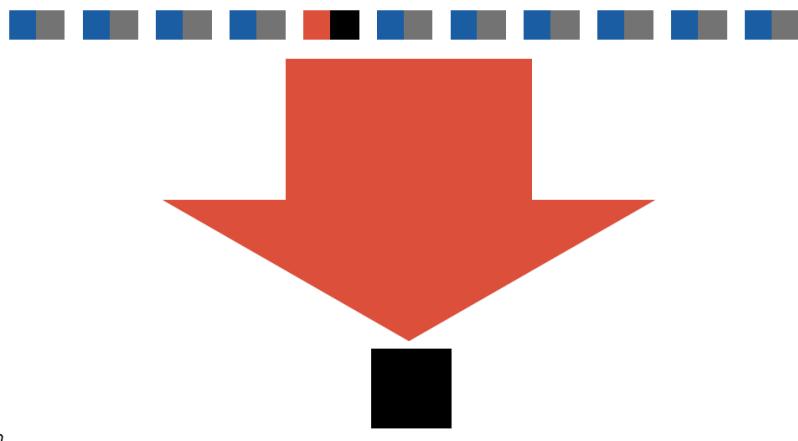
10.5.3 Actions on Pair RDDs

There are actions specifically working on RDDs of key-value pairs. Note that all other actions will also work on RDDs of key-value pairs, as pairs are values, too.

The `countByKey` action outputs, locally, each key together with the number of values in the input that are associated with this key. This results in a local list of key-value pairs. This requires care if there is a large number of distinct keys.



The lookup action outputs, locally, the value or values associated with a specific key.



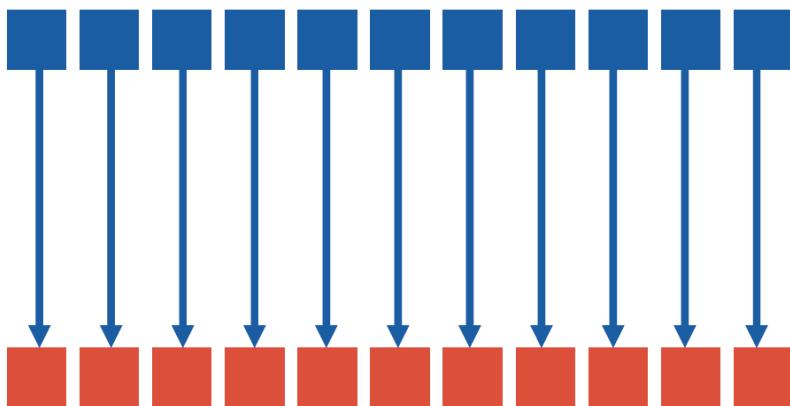
10.6 Physical architecture

Let us now take a look at the physical architecture of Spark.

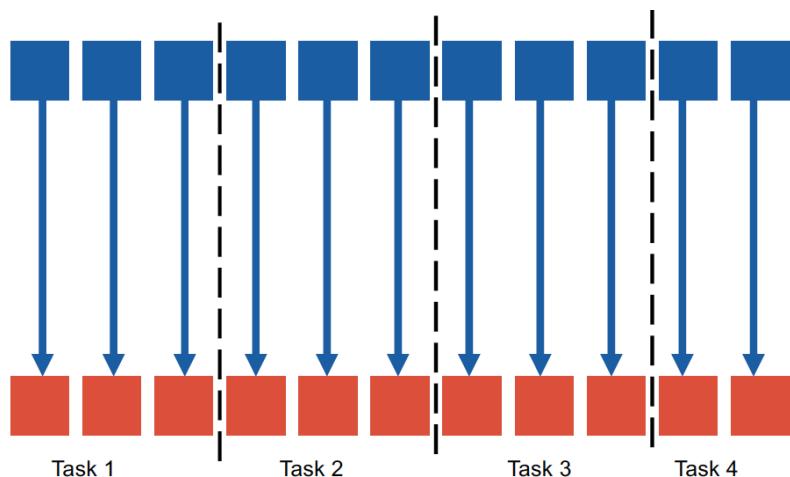
10.6.1 Narrow-dependency transformations

There are two kinds of transformations: narrow-dependency transformations and wide-dependency transformations.

In a narrow-dependency transformation, the computation of each output value involves a single input value.



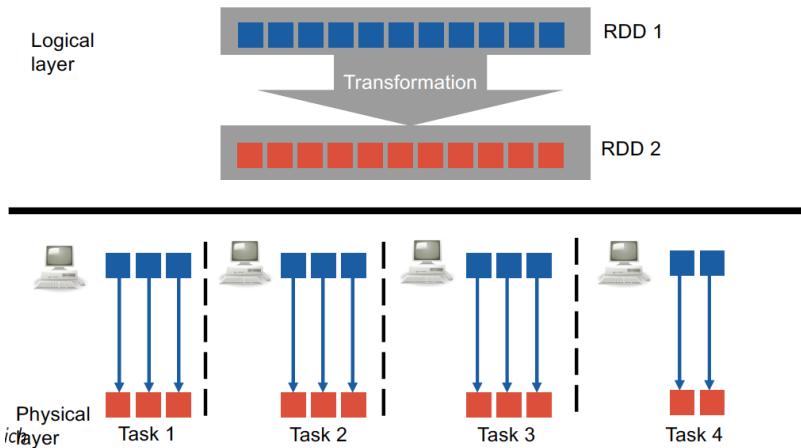
Thus, a narrow-dependency transformation is comparable to the map phase of MapReduce, and is easily parallelizable: if the input is partitioned and spread across nodes in the cluster, then the output can also be computed in partitions with no need to communicate between the nodes.



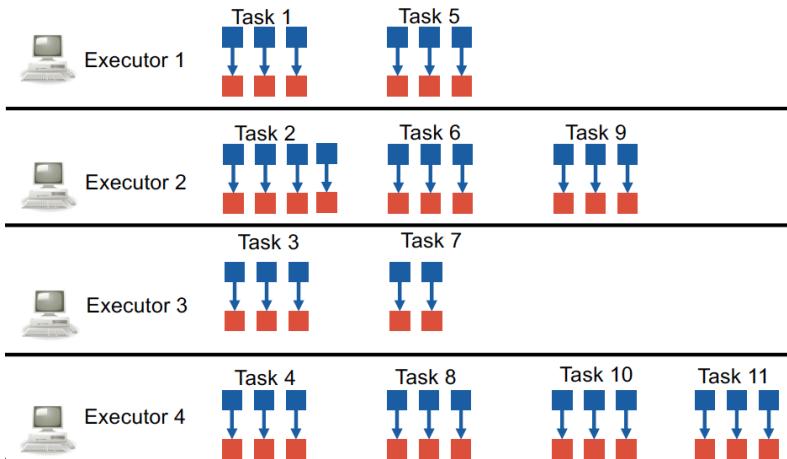
Examples of narrow-dependency transformation include map, flatMap, filter, etc. As an exercise, you can try to classify the transformations covered previously

By default, if the transformation is applied to an RDD freshly created from reading a dataset from HDFS, each partition will correspond to an HDFS block. Thus, the computation of the narrow-dependency transformation mostly involves local reads by short-circuiting HDFS (even though, as we say in Chapter 8, a few remote reads may be needed).

One can look at these transformations on the logical layer (as a transformation taking one RDD as input and returning one RDD, which is one edge in the DAG of RDDs), or on the physical layer, as the parallel computation on separate partitions.

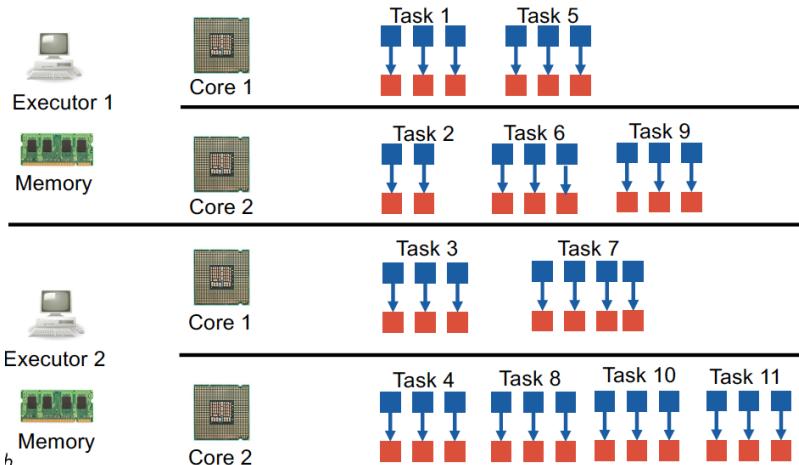


In fact, the way this works is quite similar to MapReduce: the sequential calls of the transformation function (whichever it is: a filter predicate, a map function, a flatMap function, etc) on each input value within a single partition is called a task. And just like MapReduce, the tasks are assigned to slots. These slots correspond to cores within YARN containers. YARN containers used by Spark are called executors. This is an example if we assume that each executor has one core.



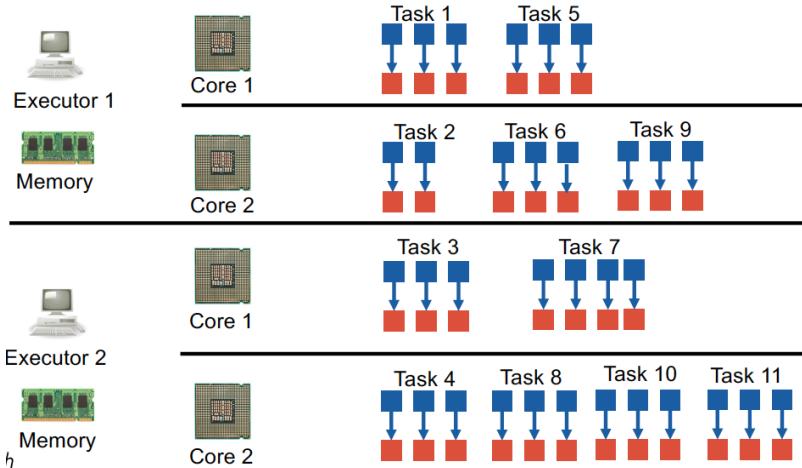
As you can see, the processing of the tasks is sequential within each executor, and tasks are executed in parallel across executors. And like in MapReduce, a queue of unprocessed tasks is maintained, and everytime a slot is done, it gets a new task. When all tasks have been assigned, the slots who are done become idle and wait for all others to complete.

Below, we show the same, but with two cores per executor, i.e., two slots per executor.

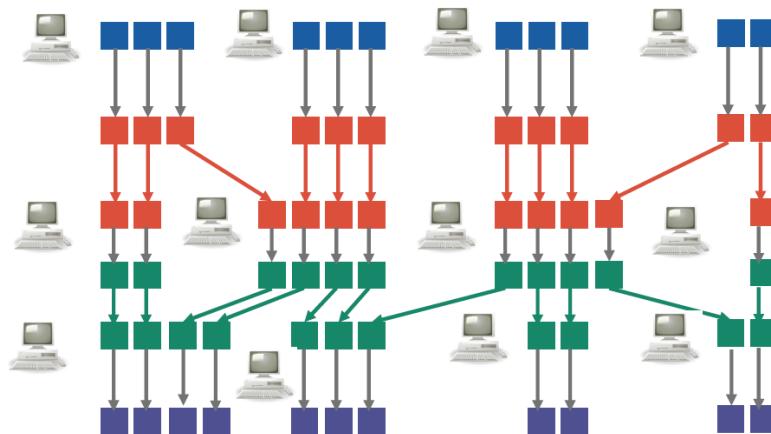


10.6.2 Chains of narrow-dependency transformations

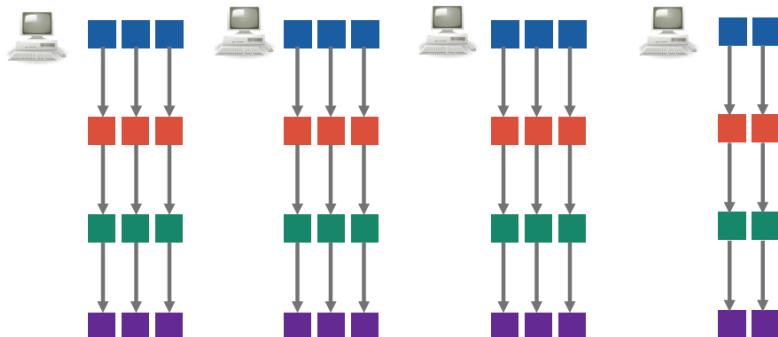
Let us now consider the case of a chain of several narrow-dependency transformations, executed in turn.



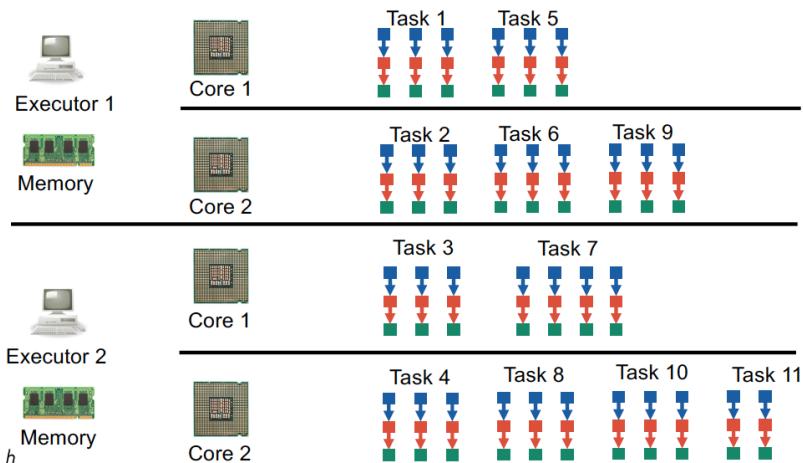
Naively, one could expect the execution to look like this (oversimplifying and assuming there is one task per slot):



However, this would be very inefficient because it requires shipping intermediate data over the network. But if all transformations are narrow-dependency transformations, it is possible to chain them without having data leaving the machines:



In fact, on the physical level, the physical calls of the underlying map/filter/etc functions are directly chained on each input value to directly produce the corresponding final, output value, meaning that the intermediate RDDs are not even materialized anywhere and exist purely logically. This means in particular that there is a single set of tasks, one for each partition of the input, for the entire chain of transformations.



Such a chain of narrow-dependency transformations executed efficiently as a single set of tasks is called a *stage*, which would correspond to what is called a phase in MapReduce.

10.6.3 Physical parameters

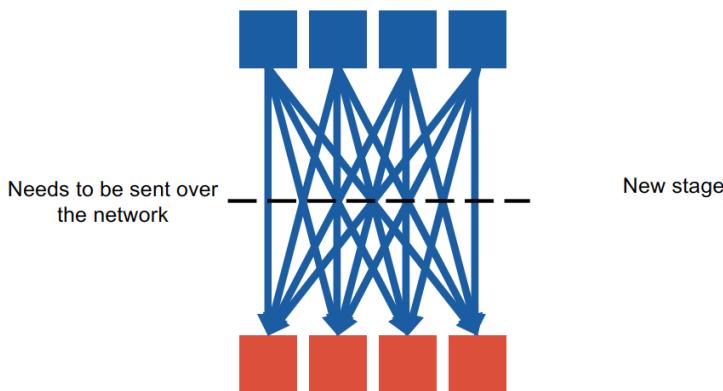
Users can parameterize how many executors there are, how many cores there are per executor and how much memory per executor (remember that these then correspond to YARN container requests). In most recent version of Spark, the number of executors can also be dynamically allocated by the cluster.

Below is an example of command to execute a Spark job (supplied within a jar file) with 42 executors, each having 2 cores and 3 GB of memory.

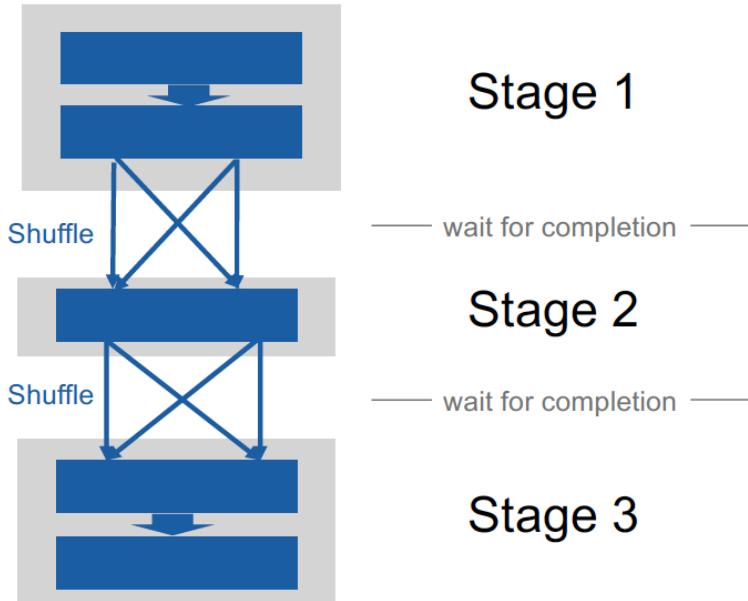
```
spark-submit
--num-executors 42
--executor-cores 2
--executor-memory 3G
my-application.jar
```

10.6.4 Shuffling

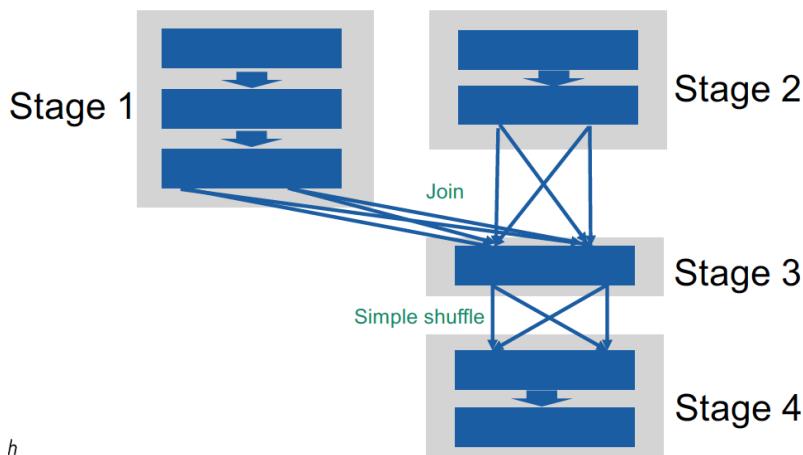
What about wide-dependency transformations? They involve a shuffling of the data over the network, so that the data necessary to compute each partition of the output RDD is physically at the same location.



Thus, on the high-level of a job, the physical execution consists of a sequence of stages, with shuffling happening everytime a new stage begins:



Even though one can imagine a physical implementation in which two stages (corresponding to different, independent parts of the DAG of RDDs) are executed at the same time on two sub-parts of the cluster (with each their own executors), a more typical setting is a linear succession of stages on the physical level.



Given a DAG, it is always possible to list the nodes in a linear order compatible with the partial order relation underlying the DAG; this is called a linearization of the DAG. In the above example, Stage 1 can be executed on the entire cluster, then Stage 2, then Stage 3, then Stage 4.

10.6.5 Optimizations

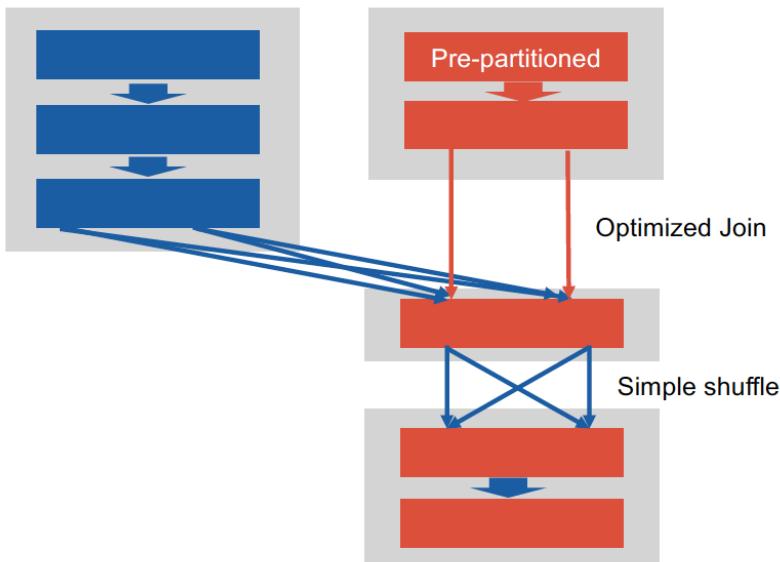
Some understanding and knowledge of the internals of Spark can be useful in order to get more performance from it. There are in particular two aspects worthy of a mention.

Pinning RDDs

Everytime an action is triggered, all the computations of the "reverse transitive closure" (i.e., all the way up the DAG through the reverted edges) are set into motion. In some cases, several actions might share subgraphs of the DAG. It makes sense, then, to "pin" the intermediate RDD by persisting it. It is possible to ask for persistence in memory and/or on disk. In some circumstances, persistence is not possible because of a lack of resources, in which case the full computation can be retriggered.

Pre-partitioning

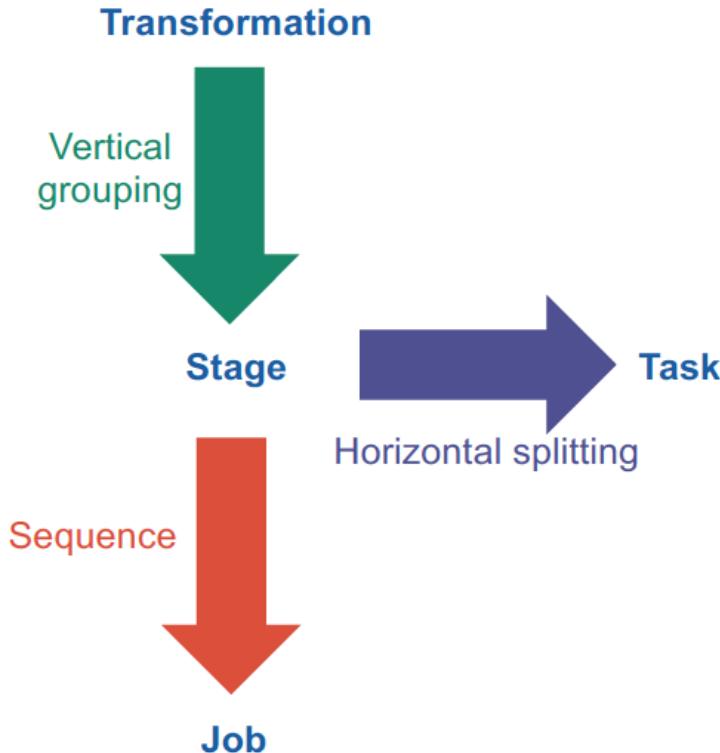
Shuffle is needed to bring together the data that is needed to jointly contribute to individual output values. If, however, Spark knows that the data is *already* located where it should be, then shuffling is not needed. A simple example is when data is sorted before being grouped with the same keys as for sorting: then, Spark has the knowledge that the groups are already consistent with their physical location, and that no additional shuffling is needed between sorting and grouping.



Users can control how the data is partitioned by asking Spark to repartition, or pre-partition, the data in a certain way.

10.6.6 Summary

Here is a high-level summary of how tasks, transformations, stages, and jobs relate:



10.7 DataFrames in Spark

10.7.1 Data independence

RDDs, the primary citizen in Spark's model and the nodes in the dataflow DAG, are very flexible. In particular, the values can be of any (static) type, while it is possible to use polymorphism in the host language to allow for different dynamic types. Thus, Spark has no problem dealing with heterogeneous and even nested datasets.

However, the complexity of dealing with these datasets is a burden that is placed on the user, because the consequence of this flexible model is that it is also a low-level model. This causes a lot of extra work for the end user: unlike a relational database that has everything right off-the-shelf, with RDDs, the user has to re-implement all the primitives they need. For example, when reading a JSON Lines dataset, the user must

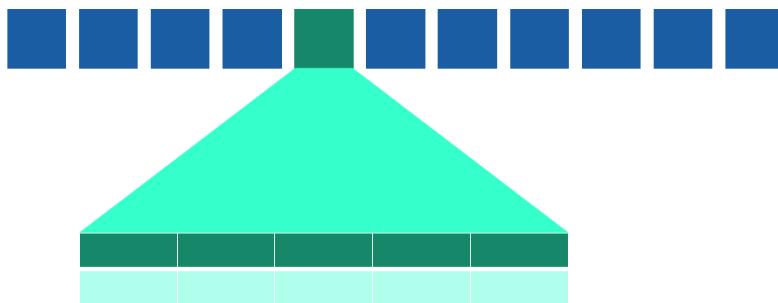
manually read the input as an RDD of strings, then invoke a JSON parsing function, and only then work with the language primitives. An example is shown below.

```
rdd = spark.sparkContext.textFile('hdfs:///dataset.json')
rdd2 = rdd.filter(lambda l: parseJSON(l))
rdd3 = rdd2.filter(lambda l: l['key'] == 0)
rdd4 = rdd3.map(lambda l: (l['key'], l['otherfield']))
result = rdd4.countByKey()
```

The reader might remember that we discussed data independence in Chapter 2. What is described above is a breach of the data independence principle. This was, of course, on the minds of the developers behind Spark, because they addressed this issue in a subsequent version of Spark by extending the model with support for DataFrames and Spark SQL, bringing back a widely established and popular declarative, high-level language into the ecosystem.

10.7.2 A specific kind of RDD

A DataFrame can be seen as a specific kind of RDD: it is an RDD of rows (equivalently: tuples, records) that has relational integrity, domain integrity, but not necessarily (as the name “Row” would otherwise fallaciously suggest) atomic integrity. We also saw in Chapter 7 that DataFrames can also be characterized as homogeneous collections of valid JSON objects (which are the rows) against a schema. It is thus only logical that, in Spark, every DataFrame has a schema.



The immediate consequence is that, in the particular case of flat rows, an RDD of (flat) rows is a relational table. Thus, it is very

natural to think of SQL as the natural language to query them. This also was on the minds of the developers, who added support for a dialect of SQL, Spark SQL, for querying DataFrames.

10.7.3 Performance impact

DataFrames are not only useful because of the higher, more convenient, level of abstraction that they provide to the user, enhancing their productivity. DataFrames also have a positive impact on performance, because Spark can do a much better job of optimizing the memory footprint and the processing, leveraging decades of knowledge on the matter. In particular, DataFrames are stored column-wise in memory, meaning that the values that belong to the same column are stored together. Furthermore, since there is a known schema, the names of the attributes need not be repeated in every single row, as would be the case with raw RDDs. DataFrames are thus considerably more compact in memory than raw RDDs.

Generally, Spark converts Spark SQL to internal DataFrame transformation and eventually to a physical query plan. An optimizer known as Catalyst is then able to find many ways of making the execution faster, as knowing the DataFrame schema is invaluable information for an optimizer, as opposed to generic RDDs of which one knows little statically. In fact, an optimizer like Catalyst contains more than 50 years of knowledge on optimizing relational queries.

As an example, since the data is stored by columns, whenever Spark knows that some columns are not used by subsequent transformations and actions, it can silently drop these unused columns with no consequence. This is called “projecting away” and it is one of the most important optimizations in large-scale databases. Projecting away can even be done already at the disk level, i.e., when reading a Parquet file, it is possible to read only the columns that are actually needed, which significantly reduces the I/O bottleneck and accelerates the job.

10.7.4 Input formats

Below is an example of code in which a dataset is directly read as JSON, a view is created, and a SQL query is then evaluated.

```
df = spark.read.json('hdfs://dataset.json')
df.createOrReplaceTempView("dataset")
df2 = df.sql("SELECT * FROM dataset "
"WHERE guess = target "
"ORDER BY target ASC, country DESC, date DESC")
result = df2.take(10)
```

The first 10 rows are then collected. Note that Spark automatically infers the schema from discovering the JSON Lines file, which adds a static performance overhead that does not exist for raw RDDs: there is no free lunch.

```
{ "foo" : 1, "bar" : true}
{ "foo" : 2, "bar" : true}
{ "foo" : 3, "bar" : false}
{ "foo" : 4, "bar" : true}
{ "foo" : 5, "bar" : true}
{ "foo" : 6, "bar" : false}
{ "foo" : 7, "bar" : true}
```



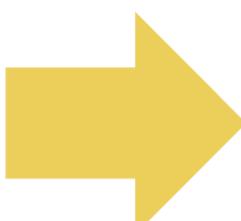
dataset.json

foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

DataFrame

CSV also come with a schema discovery, although this one is optional (as by default, one can treat all values as strings).

```
foo,bar
1,true
2,true
3,false
4,true
5,true
6,false
7,true
```



dataset.csv

foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

DataFrame

Some formats like Parquet, however, come with a declared schema, making this schema discovery step superfluous.

Builtin input formats can be specified, when creating a DataFrame from a dataset, with a simple method call after the read command. Non-builtin formats are supplied as a string parameter to a format() method. This is extensible and it is possible to “make” a format builtin via extension libraries.

```
df = spark.read.json("hdfs:///dataset.json")
df = spark.read.parquet("hdfs:///dataset.parquet")
df = spark.read.csv("hdfs:///dir/*.csv")
df = spark.read.text("hdfs:///dataset[0-7].txt")
df = spark.read.jdbc(
    "jdbc:postgresql://localhost/test?user=fred&password=secret",
    ...
)
df = spark.read.format("avro").load("hdfs:///dataset.avro")
```

It is also possible to create DataFrames on the fly by validating RDDs, however this is very complex, in particular when not using Scala, the native Spark language. The opposite, converting a DataFrame to an RDD of rows, is relatively easy.

10.7.5 DataFrame transformations

It is also possible, instead of Spark SQL, to use a transformation API similar to (but distinct from) the RDD transformation API. This API can be seen as a lower-level, query-plan-like counterpart to Spark SQL. It is also similar to a Python library known as Pandas. This API is more complex to manipulate than Spark SQL and is typically more suitable to power users or data engineers who implement libraries and engines, exposing a higher-level API or language to their end users.

```
df = spark.read.json('hdfs:///dataset.json')
df2 = df.filter(df['name'] = 'Einstein')
df3 = df.sortBy(asc("theory"), desc("date"))
df4 = df.select('year')
result = df4.take(10)
```

Unlike the RDD transformation API, there is no guarantee that the execution will happen as written, as the optimizer is free to reorganize the actual computations.

10.7.6 DataFrame column types

Let us now discuss the types that DataFrame columns can have. As expected if you read Chapter 7, there are atomic and structured types.

Atomic types include:

- numbers: byte, short, int, long, float, double, decimal

- non-number types: string, Boolean, binary data, date, timestamps

Below is a correspondence table mapping these types to two host languages, Java and Python.

DataFrame	Java	Python	[JSON IQ]
ByteType	byte	int or long	byte
ShortType	short	int or long	short
IntegerType	int	int or long	int
LongType	long	long	long
FloatType	float	float	float
DoubleType	double	float	double
BooleanType	boolean	bool	boolean
StringType	String	string	string
DecimalType(38,19)	java.math.BigDecimal	decimal.Decimal	decimal
DecimalType(38,0)	java.math.BigDecimal	decimal.Decimal	integer
TimestampType	java.sql.Timestamp	date.datetime	dateTimeStamp
DateType	java.sql.Date	date.date	date
BinaryType	byte[]	bytearray	hexBinary, base64Binary

DataFrames also support, without surprise, the three structured types we previously covered in Chapter 7: arrays, structs, and maps. As a reminder, structs have string keys and arbitrary value types and correspond to generic JSON objects, while in maps, all values have the same type. Thus, structs are more common than maps. Arrays correspond to JSON arrays.

DataFrame	Java	Python	[JSON IQ]
ArrayType	java.util.List	list or tuple or array	array
MapType	java.util.Map	dict	-
StructType	Row	list or tuple	object

Note that, to conclude, there may and will be impedance mismatches with the various input and output formats, in the sense that the type mappings might not be perfect and information can be lost (e.g., an output format might not support the full decimal value space, or an input format might have a time type).

Mapping types with each other when interfacing technologies is an important skill for data engineers and it is always a good idea to carefully document type mappings when interconnecting technologies.

10.7.7 The Spark SQL dialect

We mentioned that Spark SQL is a dialect of SQL. Spark SQL has a few limitations (i.e., there is no OFFSET clause) but also comes with a few convenient extensions, in particular to deal with nested data.

In particular, it is straightforward to get started with Spark SQL as a full-blown query with the usual clauses in the usual order will work – with the exception, as we said, of the OFFSET clause, which is not implemented but, we hope, will be one day.

```
SELECT first_name, last_name
FROM persons
WHERE age >= 65
GROUP BY country
HAVING COUNT(*) >= 1000
ORDER BY country DESC NULLS FIRST
LIMIT 100
```

Note, en passant, that both GROUP BY and ORDER BY will trigger a shuffle in the system, even though this can be optimized as the grouping key and the sorting key are the same.

Let us now look at extensions. The SORT BY clause can sort rows within each partition, but not across partitions, i.e., does not induce any shuffling.

```
SELECT first_name, last_name
FROM persons
WHERE age >= 65
GROUP BY country
HAVING COUNT(*) >= 1000
SORT BY country DESC NULLS FIRST
```

The DISTRIBUTE BY clause forces a repartition by putting all rows with the same value (for the specified field(s)) into the same new partition.

```
SELECT first_name, last_name
FROM persons
WHERE age >= 65
GROUP BY country
HAVING COUNT(*) >= 1000
DISTRIBUTE BY country
```

It is possible to use both SORT and DISTRIBUTE:

```
SELECT first_name, last_name
FROM persons
WHERE age >= 65
GROUP BY country
HAVING COUNT(*) >= 1000
SORT BY country DESC NULLS FIRST
DISTRIBUTE BY country
```

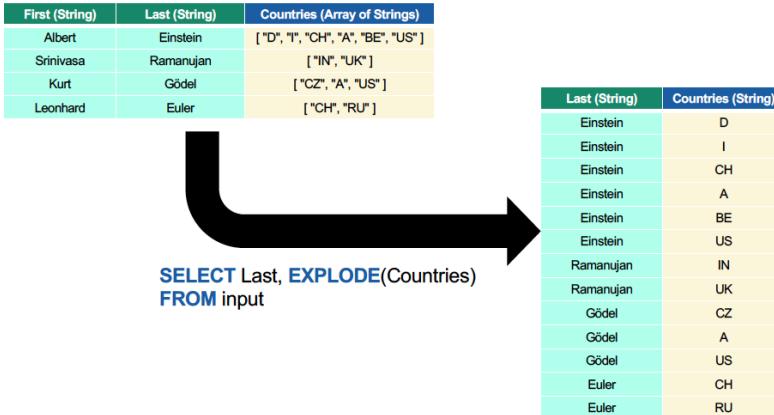
but this is equivalent to the use of another clause, CLUSTER BY:

```
SELECT first_name, last_name
FROM persons
WHERE age >= 65
GROUP BY country
HAVING COUNT(*) >= 1000
CLUSTER BY country
```

A word of warning must be given on SORT, DISTRIBUTE and CLUSTER clauses: they are, in fact, a breach of data independence, because they expose partitions, which goes against the fundamental principle set by Edgar Codd in 1970, according to which the physical layer should not be seen by the end user. In the real world, though, data independence is difficult to strictly enforce because optimization is a difficult and complex problem. In fact, it is even undecidable. As a consequence, it is desirable to leave some control to the end user on the physical execution, to compensate for the (current) inability of optimizers to make the system truly data independent. This might change in the future as technology continues to evolve.

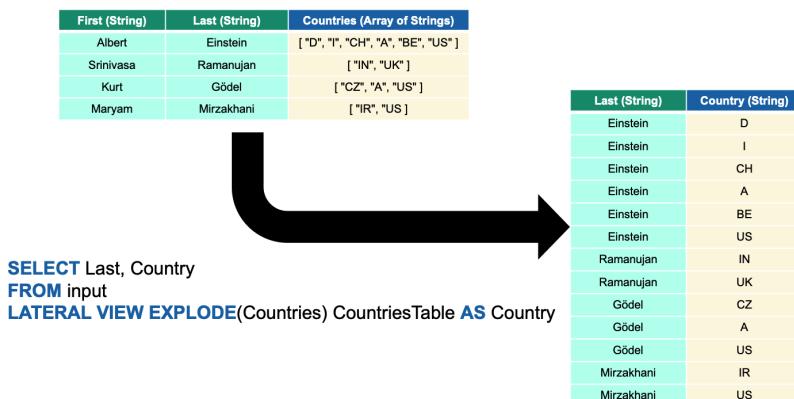
Spark SQL also comes with language features to deal with nested arrays and objects.

First, nested arrays can be navigated with the explode() function, like so:



The effect of the explode() function would probably have been called a “spooky action at a distance” with a smile by Albert Einstein, as it is not just a function call. Its effect is that the rows are duplicated into as many rows as items in the array, and one particular item of the array is placed on each duplicated row in the corresponding column. Thus, in the case of a single level of nestedness, explode() can be seen as normalizing the table by bringing it into first normal form, at the cost of data duplication.

Explode() cannot deal with all use cases, though, which is why there is another, more generic construct known as LATERAL VIEW:



A lateral view can be intuitively described this way: the array mentioned in the LATERAL VIEW clause is turned into a second, virtual

table with the rest of the original table is joined. The other clauses can then refer to columns in both the original and second, virtual table.

Lateral views are more powerful and generic than just an explode() because they give more control, and they can also be used to go down several levels of nesting, like so:

The diagram illustrates the transformation of a single DataFrame into two separate DataFrames using LATERAL VIEW EXPLODE. On the left, a source DataFrame contains a column of nested structures. An arrow points from this source to a query that uses LATERAL VIEW EXPLODE to flatten the nested structures into separate columns. On the right, the resulting flattened DataFrame is shown.

First (String)	Last (String)	Continents (Array of Structs)
Albert	Einstein	[{"C": "Europe", "Countries": ["D", "I", "CH", "A", "BE"]}, {"C": "America", "Countries": ["US"]}]
Srinivasa	Ramanujan	[{"C": "Asia", "Countries": ["IN"]}, {"C": "Europe", "Countries": ["UK"]}]
Kurt	Gödel	[{"C": "Europe", "Countries": ["CZ", "A"]}, {"C": "America", "Countries": ["US"]}]
Maryam	Mirzakhani	[{"C": "North America", "Countries": ["US"]}, {"C": "Asia", "Countries": ["IR"]}]]

Last (String)	Continent (String)	Country (String)
Einstein	Europe	D
Einstein	Europe	I
Einstein	Europe	CH
Einstein	Europe	A
Einstein	Europe	BE
Einstein	American	US
Ramanujan	Asia	IN
Ramanujan	Europe	UK
Gödel	Europe	CZ
Gödel	Europe	A
Gödel	America	US
Mirzakhani	N. America	US
Mirzakhani	Asia	IR

It is also possible to navigate and “unnest” nested structs (objects) using dots, like so:

The diagram illustrates the transformation of a single DataFrame into two separate DataFrames using dot notation to access nested object fields. On the left, a source DataFrame contains a column of nested objects. An arrow points from this source to a query that uses dot notation to extract specific fields. On the right, the resulting flattened DataFrame is shown.

Name (Object)	Countries (Int)
{"First": "Albert", "Last": "Einstein"}	6
{"First": "Srinivasa", "Last": "Ramanujan"}	2
{"First": "Kurt", "Last": "Gödel"}	3
{"First": "John", "Last": "Nash"}	1
{"First": "Alan", "Last": "Turing"}	1
{"First": "Leonhard", "Last": "Euler"}	2

First (String)	Last (String)
Albert	Einstein
Srinivasa	Ramanujan
Kurt	Gödel
John	Nash
Alan	Turing
Leonhard	Euler

Does Spark SQL solve the problem of querying denormalized data? In fact, only partially. First, dealing with nested structures is easy if there is not too much nestedness, but the verbosity of the query becomes quickly overwhelming with more nestedness and more complex use cases. SQL was designed for normalized tables and was never meant to be used on denormalized data. Using “SQL-with-dots” languages helps a bit but does not fully address the issue.

Second, DataFrames and Spark SQL do support data that is not in first normal form, but they do not support at all data that does

not fulfill relational integrity or domain integrity. It is possible to read as input a JSON Lines dataset with inconsistent value types in the same columns, however the schema discovery phase will simply result in a column with type “string”, which puts all the burden of dealing with the polymorphism of the column to the end user, leading to many additional lines of code in the host language. Sadly, messy data is commonly found in the real world and it is one of the top challenges to clean up and prepare data for machine learning pipelines.

```
{ "foo" : 1, "bar" : true}
{ "foo" : 2, "bar" : true}
{ "foo" : [3, 4], "bar" : false}
{ "foo" : 4, "bar" : true}
{ "foo" : 5, "bar" : true}
{ "foo" : 6, "bar" : false}
{ "foo" : 7, "bar" : true}
```



foo	bar
string	boolean
"1"	true
"2"	true
"[3, 4]"	false
"4"	true
"5"	true
"6"	false
"7"	true

dataset.json

DataFrame

We will see, in subsequent chapters, that this problem can be solved by using query languages more adequate, because specifically designed, for denormalized data, both valid (DataFrames) or unvalidated.

10.8 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain how Spark is more powerful than MapReduce on a data model level?
- b. Can you explain what a Resilient Distributed Dataset (RDD) is?
- c. Do you know the difference between an action and a transformation?
- d. Do you know the main actions and transformations available in Spark?
- e. Are you able to classify transformations and actions in various buckets (those that work on any value, those that work on key-value pairs, unary vs. binary transformations, ...)?
- f. Can you describe how transformations run physically (tasks, stages...)?
Can you explain the similarities (and differences) with MapReduce on the physical level?
- g. Can you explain when and how a series of transformations can be optimized by keeping the same set of machines with no network communication between the transformations?
- h. Can you explain what a stage is? Can you relate it to transformations? To tasks? To jobs?
- i. Can you tell why and when shuffling is needed? In other words, can say whether a transformation has a narrow dependency or a wide dependency?
- j. Can you easily draw a directly acyclic graph for a Spark job, and mark the stages?
- k. Do you understand why keeping an RDD persisted can be useful and can improve performance?
- l. Can you explain how controlling the way data is partitioned can make execution faster because we can influence stages?
- m. Can you write Spark jobs in Python (PySpark) manipulating RDDs, given a formulation of the query in English?
- n. Can you explain what a DataFrame is? What does it improve in Spark?
- o. Can you explain what the limitations of DataFrames are? What can they do well? What can they do less well?

- p. Can you write Spark jobs in Spark SQL manipulating DataFrames, given a formulation of the query in English?

10.9 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Zaharia, M. et al. (2012). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. In NSDI.

Armbrust, M., et al. (2015). *Spark SQL: Relational Data Processing in Spark*. In SIGMOD.

Karau, H. et al. (2015). *Learning Spark*. 1st edition. Chapters 3, 4, 5, 6, 8.

Chapter 11

Document stores

In the journey towards processing large-scale, real world datasets, a few compromises had to be made in the early systems. In particular, the ACID paradigm was replaced with the more lenient CAP-theorem paradigm with, in particular eventual consistency. Another compromise is that many of the early Big Data systems offer a low-level API in an imperative host language rather than a query language. And finally, many systems work as data lakes where users throw all their datasets, rather than a fully integrated database management system that takes over the control of, and hides, the physical layout.

All of these compromises are a high price to pay because it sends us back, as far as data independence is concerned, in the 1960s where people wrote programs to directly read data from their file system. This is something that the database community is well aware of, and for this reason, there are attempts to bring back all these data independence bells and whistles (ACID, query languages, data management).

Document stores are an example of step in this direction: a document store, unlike a data lake, manages the data directly and the users do not see the physical layout.

11.1 Relational databases

As a reminder, in relational databases, everything is a table. We saw that a table can be seen as a set of maps (from attributes to values) that fulfils three constraints: relational integrity, domain integrity, and atomic integrity.

We can, of course, process tables through a data lake: we could upload CSV files to S3 or HDFS and then query them via Spark or, even better, Spark SQL.

But a relational database management system will offer more than this: it can optimize the layout of the data on disk and build addi-

tional structures (indices) to accelerate SQL queries without the need to modify them, and it can handle transactions.

Can we rebuild a similar system for collections of trees, in the sense that we drop all three constraints: relational integrity, domain integrity, and atomic integrity?

Document stores bring us one step in this direction.

11.2 Challenges

11.2.1 Schema on read

Data that fulfills relational integrity, domain integrity, and atomic integrity always comes with a schema. In a relational database management system, it is not possible to populate a table without having defined its schema first.

We saw in Chapter 7 that schemas can be extended to data that break relational integrity (optional fields, open objects), or domain integrity (union types or use of the “item” topmost type), or atomic integrity (nested arrays and objects). We also saw that the special case of valid data that only breaks atomic integrity (or relational integrity in reasonable amounts, i.e., optional fields but no open objects) is described with the dataframes framework.

However, when encountering such denormalized data, in the real world, there is often no schema. In fact, one of the important features of a system that deals with denormalized data is the ability to *discover* a schema, i.e., offer query functionality to find out which keys appear in the data, what kind of value is associated with each key, etc; or even functionality that directly infers a schema, as we saw is the case with Apache Spark.

11.2.2 Making trees fit in tables

A first thought when trying to build a system that supports denormalized data, such as collections of JSON or XML objects, is to force-fit it into tables. In fact, it is a very natural thing to do if the collection is flat and homogeneous, i.e., respects the three fundamental integrity constraints.

For example, a flat JSON object can naturally be seen as the row of a relational table:

```
{
  "foo":      1,
  "bar":      "foo",
  "foobar" : true,
  "a"   :      "bar",
  "b"   :      3.14
}
```



foo	bar	foobar	a	b
1	foo	true	bar	3.14

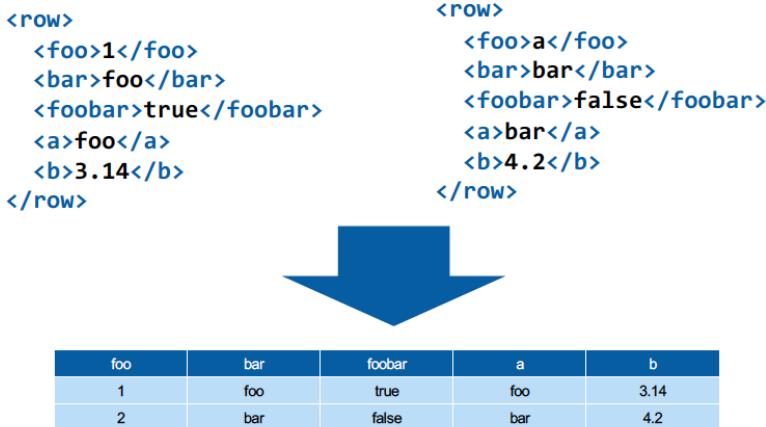
Likewise, a flat XML element can naturally be seen as the row of a relational table:

```
<row>
  <foo>1</foo>
  <bar>foo</bar>
  <foobar>true</foobar>
  <a>foo</a>
  <b>3.14</b>
</row>
```



foo	bar	foobar	a	b
1	foo	true	foo	3.14

Thus, several XML elements (or, likewise, several JSON objects) can be naturally mapped to a relational table with several rows:



The corresponding XML Schemas can also be transformed (modulo an appropriate data type mapping, as explained in Chapter 10) naturally to a relational schema:

```
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="row">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="foo" type="xs:integer"/>
        <xss:element name="bar" type="xs:string"/>
        <xss:element name="foobar" type="xs:boolean"/>
        <xss:element name="a" type="xs:string"/>
        <xss:element name="b" type="xs:decimal"/>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
</xss:schema>
```

The diagram illustrates the transformation of an XML Schema into a relational schema. At the top, an XML Schema snippet is shown:

```
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="row">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="foo" type="xs:integer"/>
        <xss:element name="bar" type="xs:string"/>
        <xss:element name="foobar" type="xs:boolean"/>
        <xss:element name="a" type="xs:string"/>
        <xss:element name="b" type="xs:decimal"/>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
</xss:schema>
```

A large blue downward-pointing arrow is positioned below the schema snippet, indicating the transformation process.

Below the arrow is a table representing the relational schema:

foo	bar	foobar	a	b
integer	string	boolean	string	decimal

The same goes for JSound or JSON Schemas:

```
{
    "foo" : "integer",
    "bar" : "string",
    "foobar" : "boolean",
    "a" : "string",
    "b" : "decimal"
}
```



foo	bar	foobar	a	b
integer	string	boolean	string	decimal

Is this not great? Does it mean we actually have nothing to do: JSON and XML collections, more generally semi-structured collections, just fit elegantly in relational tables? At the risk of raining on the party, the matter is more complex than this. This is because semi-structured data can generally be nested and heterogeneous.

So it is tempting to map nestedness, then:

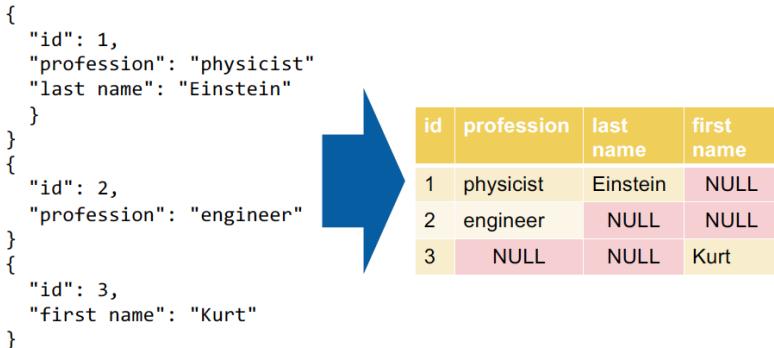
```
{
    "category": 1,
    "job": "mathematician",
    "name": [ {
        "last": "Ramanujan",
        "first": "Srinivasa"
    },
    {
        "last": "Gödel",
        "first": "Kurt"
    }
],
{
    "category": 2,
    "job": "physicist",
    "name": [ {
        "last": "Einstein",
        "first": "Albert"
    }
]
...
}
```



category	job
1	mathematician
2	physicist

category	name.last	name.first
1	Ramanujan	Srinivasa
1	Gödel	Kurt
2	Einstein	Albert

as well as heterogeneity:



Doing so quickly leads to the realization that such mapping will at best have to be done for *every single dataset*, and requires in most cases a schema, whereas we are looking for a generic solution for semi-structured data with no a-priori schema information. And at worst, dealing with heterogeneity will lead to an intractable number of columns and nestedness to an intractable number of tables.

This is why we have...

11.3 Document stores

Document stores provide a native database management system for semi-structured data. Document stores also scale to Gigabytes or Terabytes of data, and typically millions or billions of records (a record being a JSON object or an XML document).

A document store typically specializes in either JSON or XML data, even though some companies (e.g., MarkLogic) offer support for both.

Document stores work on collections of records, generalizing the way that relational tables can be seen as collections of rows. Such a collection can look like this in the case of JSON:

```
{
  "foo": 1,
  "bar": [ "foo", "bar" ],
  "foobar" : true,
  "a" : { "foo" : null, "b" : [ 3, 2 ] },
  "b" : 3.14
}
{
  "foo": 1,
  "bar": "foo"
```

```

}
{
  "foo": 2,
  "bar": [ "foo", "foobar" ],
  "foobar" : false,
  "a" : { "foo" : "foo", "b" : [ 3, 2 ] },
  "b" : 3.1415
}
...

```

Records, in a document store, are called, wait for it, documents. It is important to understand that document stores are optimized for the typical use cases of many records of small to medium sizes. Typically, a collection can have millions or billions of documents, while each single document weighs no more than 16 MB (or a size in a similar magnitude). Some document stores strictly enforce a maximum size and will not allow larger individual documents.

Finally, a collection of documents need not have a schema: it is possible to insert random documents that have dissimilar structures with no problem at all. Most document stores, however, do provide the ability to add a schema. If they do, it is then possible to validate the documents in a collection. This can be done *before* adding documents (schema-on-write), to ensure validity, or validation can be attempted *after* adding the schema to a previously schemaless collection, or while processing a collection (schema-on-read).

As you can see, this model with collection of trees generalizes the relational model to nested and heterogeneous data elegantly, while retaining the ability to enforce a minimum structure thanks to schemas.

Document stores can generally do selection, projection, aggregation and sorting quite well, but many of them are typically not (yet) optimized for joining collections. In fact, often, their language or API does not offer any joining functionality at all, which pushes the burden to reimplement joins in a host language to the users. The same applies for complex queries that push the API to its limits and force users to write a significant part of their code in the host language, rather than push it down to the document store. This is a serious breach of data independence.

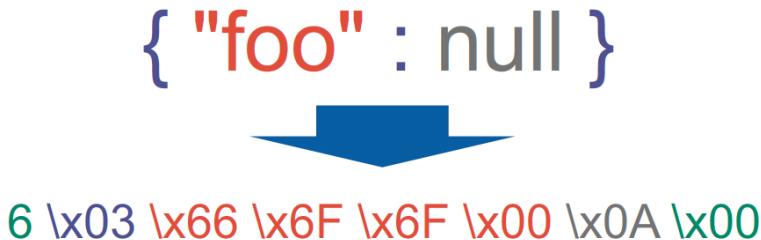
11.4 Implementations

There is a very large number of products in the document store space for both JSON and XML, let us mention for example MongoDB, CouchDB, ElasticSearch, Cloudant, existDB, ArangoDB, BaseX, MarkLogic, DocumentDB, CosmosDB, and so on. We will focus, as an example, on MongoDB.

11.5 Physical storage

In a data lake, the files are stored “as is” in the cloud or on a distributed file system. In a database management system, the storage format is typically proprietary and optimized for performance. It is hidden from the user. Just like the storage format is optimized for tabular data in a relational database management system, it is optimized for tree-like data in a document store.

In MongoDB, the format is a binary version of JSON called BSON. BSON is basically based on a sequence of tokens that efficiently encode the JSON constructs found in a document, like so:



The immediate benefit of BSON is that it takes less space in storage than JSON stored as a text file: for example, null, true and false literals need four or five bytes in text format at best, while they can be efficiently encoded as single bytes in BSON.

Furthermore, BSON supports additional types that JSON does not have, such as dates. These types are exposed to the user.

11.6 Querying paradigm (CRUD)

The API of MongoDB, like many document stores, is based on the CRUD paradigm. CRUD means Create, Read, Update, Delete and corresponds to low-level primitives similar to those we covered in Chapter 6 for HBase.

MongoDB supports several host languages to query collections via an API. This includes in particular JavaScript and Python, but many other languages are supported via drivers. We will use JavaScript here because this is the native host language of MongoDB. It is important to note that these APIs are not query languages; this is because they provide functionality to the user allowing the dynamic creation of a query plan (even though the engine will, of course, optimize and modify it automatically for performance), while a full-blown query language

completely hides the concept of query plan from the user and creates it in the background.

MongoDB also provides access to the data via a shell called mongo or, newly, mongosh. This is a simple JavaScript interface wrapped around the MongoDB's node.js driver. We will give our example using queries on the mongo shell, and draw the attention of the reader that, if they try to look for MongoDB documentation via a search engine, they might land on pages showing queries written with a different driver (e.g., Python). The syntax might thus differ, but the general API look and feel is the same.

11.6.1 Populating a collection

Creating and populating collections is more straightforward in a document store than in a relational database system, because no schema is required. Thus, to create a collection, one can simply insert a document in it, and it will be automatically created if it does not exist.

Generally, collections in MongoDB are accessed in JavaScript with

```
db.scientists
```

where "scientists" is the name of the collection.

In order to insert one document, the method `insertOne()` should be called, like so:

```
db.scientists.insertOne(  
  {  
    "Last" : "Einstein",  
    "Theory" : "Relativity"  
  }  
)
```

In order to insert several documents at the same time, the method `insertMany()` should be called, with the documents to insert provided in an array, like so:

```
db.scientists.insertMany(  
  [  
    {  
      "Last" : "Lovelace",  
      "Theory" : "Analytical Engine"  
    },  
    {  
      "Last" : "Einstein",  
      "Theory" : "Relativity"  
    }  
)
```

```
    ]  
})
```

MongoDB automatically adds to every inserted document a special field called “_id” and associated with a value called an Object ID and with a type of its own. An Object ID can simply be thought as a 12-byte binary value. Object IDs are convenient for deleting or updating a specific document with no ambiguity.

11.6.2 Querying a collection

In the following examples, for easing the learning, we will also attempt to give a SQL equivalent, but please understand this with caution as document stores are not relational databases and using SQL with them would be an impedance mismatch.

Scan a collection

Asking for the contents of an entire collection is done with a simple `find()` call on the previous object, like so:

```
db.collection.find()
```

An equivalent SQL query would be:

```
SELECT *  
FROM collection
```

This function does not in fact return the entire collection; rather, it returns some pointer, called a cursor, to the collection; the user can then iterate on the cursor in an imperative fashion in the host language (this is another reason why this is not a query language).

Selection

It is possible to perform a selection on a collection by passing a parameter to `find()` that is a JSON object (the JavaScript syntax is similar, in fact this is where the JSON syntax comes from):

```
db.collection.find({ "Theory" : "Relativity" })
```

The above query returns all documents in the collection that have a key called “Theory” associated with the string value “Relativity”. It would correspond, in SQL, to a WHERE clause, like so:

```
SELECT *  
FROM collection  
WHERE Theory = 'Relativity'
```

Note that the MongoDB query is not written in any query language; rather, it is an API in an imperative host language (JavaScript) used to create a (high-level) query plan via chains of method calls (`find()` and others).

What is different from a relational database, then? In a document store, it is possible that some documents have this key, while others do not. The latter are excluded from the results. It is also possible that some documents have the key, but the value is something else than a string (a number, a date, etc). These documents would also be excluded from the results.

It is possible to select on several field values simply by adding more in the parameter object, like so:

```
db.collection.find(  
  {  
    "Theory": "Relativity",  
    "Last": "Einstein"  
  }  
)
```

A disjunction (OR) uses a special MongoDB keyword, prefixed with a dollar sign, like so:

```
db.collection.find(  
  {  
    "$or" : [  
      { "Theory": "Relativity" },  
      { "Last": "Einstein" }  
    ]  
  }  
)
```

The special “\$or” field is associated with an array, and the members of the array are (recursively) all the filtering predicates to take the disjunction of.

MongoDB offers many other keywords, for example for comparison other than equality:

```
db.collection.find(  
  {  
    "Publications" : { "$gte" : 100 }  
  }  
)
```

Note that quoting keys, in MongoDB, is optional and this is also true with dollar keywords; however, we recommend caution with growing this as a habit, as this is not well-formed JSON.

Projection

By default, MongoDB returns the entire objects.

Projections are made with the second parameter of this same find() method. This is done in form of a JSON object associating all the desired keys in the projection with the value 1.

```
db.scientists.find(
  { "Theory" : "Relativity" },
  { "First" : 1, "Last": 1 }
)
```

equivalent to:

```
SELECT First, Last
FROM scientists
WHERE Theory = "Relativity"
```

By default, the object ID, in the field “_id” is always included in the results. It is possible to project it away with a 0:

```
db.scientists.find(
  { "Theory" : "Relativity" },
  { "First" : 1, "Last" : 1, "_id" : 0 }
)
```

It is also possible to project fields away in the same way with 0s, however 1s and 0s cannot be mixed in the projection parameter, except in the specific above case of projecting away the object ID.

```
db.scientists.find(
  { "Theory" : "Relativity" },
  { "First" : 0, "Last" : 0 }
)
```

Counting

Counting can be done by chaining a count() method call, like so:

```
db.scientists.find(
  { "Theory" : "Relativity" }
).count()
```

equivalent to:

```
SELECT COUNT(*)
FROM scientists
WHERE Theory = "Relativity"
```

Sorting

Sorting can be done by chaining a sort() method call, like so:

```
db.scientists.find(
  { "Theory" : "Relativity" },
  { "First" : 1, "Last" : 1 }
).sort(
{
  "First" : 1,
  "Name" : -1
})
)
```

equivalent to:

```
SELECT First, Last
FROM scientists
WHERE Theory = "Relativity"
ORDER BY First ASC, Name DESC
```

1 is for ascending order and -1 for descending order, which is also an artefact and arbitrary convention due to the use of an API rather than a query language.

It is also possible to add limits and offsets to paginate results also by chaining more method calls:

```
db.scientists.find(
  { "Theory" : "Relativity" },
  { "First" : 1, "Last" : 1 }
).sort(
{
  "First" : 1,
  "Name" : -1
})
.skip(30).limit(10)
```

equivalent to:

```
SELECT First, Last
FROM scientists
WHERE Theory = "Relativity"
ORDER BY First ASC, Name DESC
LIMIT 10
OFFSET 30
```

Note that, contrary to intuition, the order of the calls does not matter, as this is really just the creation of a query plan by providing parameters (in any order).

Duplicate elimination

It is possible to obtain all the distinct values for one field with a `distinct()` call:

```
db.scientists.distinct("Theory")
```

11.6.3 Querying for heterogeneity

Absent fields

Absent fields can be filtered with:

```
db.scientists.find(
  { "Theory" : null }
)
```

which is equivalent to the following SQL query (as NULLs, in relational database, provide limited support for heterogeneity when used parsimoniously):

```
SELECT *
FROM scientists
WHERE Theory IS NULL
```

Filtering for values across types

Querying for several values with different types and in the same field can easily be made with a disjunctive query:

```
db.collection.find(
{
  "$or" : [
    { "Theory": "Relativity" },
    { "Theory": 42 },
    { "Theory": null }
  ]
}
```

MongoDB also provides an alternate syntax for doing the same with the `$in` keyword:

```
db.scientists.find(
{
  "Theory" : {
    "$in" : [

```

```
    "Relativity",
    42,
    null
]
}
}
)
```

MongoDB is also able to sort on fields that have heterogeneous data types. It does so by first order by type in some (arbitrary, but documented) order, and then within each type. The documented order to sort across types is (ascending):

1. null values
2. numbers
3. strings
4. objects
5. arrays
6. binary data
7. Object IDs
8. Booleans
9. dates
10. timestamps
11. regular expressions

There are also special “types” and (singleton) values called the min key and the max key whose sole purpose is for the value to come first or last.

11.6.4 Querying for nestedness

Nestedness in MongoDB is handled in several ad-hoc ways for specific use cases (a fully generic mechanism for this would require a query language native to denormalized data, which we will cover in a subsequent chapter).

Values in nested objects

We saw how to select documents based on values associated with top-level keys. What about values that are not at the top-level, but in nested objects? Let us put arrays aside for now and assume only nested objects.

The first solution that might come to mind is something like this:

```
db.scientists.find({
  "Name" : {
    "First" : "Albert"
  }
})
```

However, this query will not have the behavior many would have expected: instead of finding documents that have a value “Albert” associated with the key “First” in an object itself associated with the top-level key ”Name”, this query looks for an *exact match* of the *entire object*

```
{
  "First" : "Albert"
}
```

associated with key “Name”, and will not include documents such as, for example:

```
{
  "Name" : {
    "First" : "Albert",
    "Some" : "Other field"
  }
}
```

In order to include documents such as above, MongoDB uses a dot syntax. This means that, like the dollar sign, dots are treated in a special way in MongoDB queries. This query will return the document above:

```
db.scientists.find({
  "Name.First" : "Albert"
})
```

Values in nested arrays

MongoDB allows to filter documents based on whether a nested array contains a specific value, like so:

```
db.scientists.find({  
  "Theories" : "Special relativity"  
})
```

The query above will return this document, as expected:

```
{  
  "Name" : "Albert Einstein",  
  "Theories" : "Special relativity"  
}
```

but it will also return documents like this:

```
{  
  "Name" : "Albert Einstein",  
  "Theories" : [  
    "Special relativity",  
    "General relativity"  
  ]  
}
```

that is, a single value in a query will also be matched with arrays that contain it.

11.6.5 Deleting objects from a collection

Objects can be deleted from a collection either one at a time with `deleteOne()`, or several at a time with `deleteMany()`:

```
db.scientists.deleteMany(  
  { "century" : "15" }  
)
```

will delete *all* documents matching the criteria, which uses the same syntax as selection in `find()` queries.

```
db.scientists.deleteOne(  
  { "century" : "15" }  
)
```

will delete just one document matching the criterion, leaving any other documents matching the same criterion unchanged in the original collection. If there is no such document, nothing happens.

11.6.6 Updating objects in a collection

Documents can be updated with `updateOne()` and `updateMany()` by providing both a filtering criterion (with the same syntax as the first parameter of `find()`) and an update to apply. The command looks like so:

```
db.scientists.updateMany(
  { "Last" : "Einstein" },
  { $set : { "Century" : "20" } }
)
```

In addition to `$set`, there are also `$unset` to remove a value and `$replaceWith` to completely change an entire document.

The granularity of updates is per document, that is, a single document can be updated by at most one query at the same time. However, within the same collection, several different documents can be modified concurrently by different queries in parallel.

11.6.7 Complex pipelines

For grouping and such more complex queries, MongoDB provides an API in the form of aggregation pipelines, for which the syntax looks like so:

```
db.scientists.aggregate(
  { $match : { "Century" : 20 },
  { $group : {
      "Year" : "$year",
      "Count" : { "$sum" : 1 }
    }
  },
  { $sort : { "Count" : -1 } },
  { $limit : 5 }
)
```

Since we covered Apache Spark in Chapter 10, we believe the above code requires not much explanation, as this is quite akin to a sequence of Spark Transformation followed by a collect action.

For completeness, this would be a similar SQL query:

```
SELECT Year, SUM(Count) AS Count
FROM scientists
WHERE Century = 20
GROUP BY Year
ORDER BY SUM(Count) DESC
LIMIT 5
```

11.6.8 Limitations of a document store querying API

The API provided by MongoDB and similar document stores is very powerful and goes beyond what SQL can do in functionality especially in the context of nested, heterogeneous datasets.

Simple use cases are straightforward to handle, however more complex use cases require a lot of additional code in the host language, be it JavaScript or Python. An example is that joins must be taken care of by the end user in the host language: the burden of implementing more complex use cases is pushed to the end user. This leads to code that is complex to write and to read, and also potentially to performance issues in cases that the optimizer does not catch non optimal code written by the user.

Some document stores recently added higher-level query languages on top of their API in order to address this issue. We will cover query languages for denormalized data in Chapter 12.

11.7 Architecture

The architecture of MongoDB follows similar principles to what we covered before: scaling out the hardware to multiple machine, and sharding as well as replicating the data.

11.7.1 Sharding collections

Collections in MongoDB can be sharded. Shards are determined by selecting one or several fields. (Lexicographically-ordered) intervals over these fields then determine the shards. This is similar in spirit to regions in HBase covered in Chapter 6, which are sharded by Row ID intervals.

Shards then are stored in different physical locations.

The fields used to shard must be organized in a tree index structure. Indices are described in Section 11.8.

11.7.2 Replica sets

A replica set is a set of several nodes running the MongoDB server process. The nodes within the same replica set all have a copy of the same data.

Each shard of each collection is assigned to exactly one replica set. Note that this architecture is not the same as that of HDFS, in which the replicas are spread over the entire cluster with no notion of “walls” between replica sets and no two DataNodes having the exact same block replicas. It is also not the same as HBase, in which nodes receive the responsibility of handling specific regions, but do not necessarily store them physically as this is done on the HDFS level.

11.7.3 Write concerns

When writing (be it delete, update or insert) to a collection, more exactly, to a specific shard of a collection, MongoDB checks that a specific minimum number of nodes (within the replica set that is responsible for the shard) have successfully performed the update. This is similar in spirit to the W parameter in DynamoDB. Once the minimum number of replication is reached, the user call returns (synchronous). Then replication continues in the background to more (asynchronous).

11.8 Indices

11.8.1 Motivation

A document store, unlike a data lake, manages the physical data layout. This has a cost: the need to import (ETL) data before it is possible to query it, but this cost comes with a nice benefit: index support, just like relational database management systems.

An index is an auxiliary structure that can accelerate certain queries.

Imaging for example a large collection with billions of objects like this:

```
{"Name": "Einstein", "Profession": "Physicist"}  
{"Name": "Lovelace", "Profession": "Computer scientist"}  
{"Name": "Gödel", "Profession": "Mathematician"}  
{"Name": "Ramanujan", "Profession": "Mathematician"}  
{"Name": "Pythagoras", "Profession": "Mathematician"}  
{"Name": "Curie", "Profession": "Chemist"}  
{"Name": "Turing", "Profession": "Computer Scientist"}  
{"Name": "Church", "Profession": "Computer Scientist"}  
{"Name": "Nash", "Profession": "Economist"}  
{"Name": "Mirzakhani", "Profession": "Mathematician"}  
{"Name": "Euler", "Profession": "Mathematician"}  
{"Name": "Bohm", "Profession": "Physicist"}  
{"Name": "Galileo", "Profession": "Astrophysicist"}  
{"Name": "Germain", "Profession": "Mathematician"}  
{"Name": "Lagrange", "Profession": "Mathematician"}  
{"Name": "Gauss", "Profession": "Mathematician"}  
{"Name": "du Chatelet", "Profession": "Computer Scientist"}  
{"Name": "Thales", "Profession": "Mathematician"}  
...
```

A typical use case is performing a selection on this collection. A selection query can be very specific and pinpoint exactly one document. This is called a point query. Assuming the values associated with the Names field are unique, the following is a point query:

```
find({"Name":"Euler"})
```

With no index, the document would have no other choice than scanning the full collection. It can easily take minutes, if not hours before the appropriate document is located.

With an index, this same query, with no change, will execute in just a few milliseconds.

Even selection queries that are less specific will be considerably faster with an index, for example:

```
find("Profession":"Mathematician")
```

There is another important class of selection queries called range queries. Range queries select an interval of values on an ordered field.

Imagine for example this collection:

```
{"Name": "Einstein", "Year":1879}
{"Name": "Lovelace", "Year":1815}
{"Name": "Gödel", "Year":1906}
{"Name": "Ramanujan", "Year":1887}
{"Name": "Pythagoras", "Year":-570}
{"Name": "Curie", "Year":1867}
{"Name": "Turing", "Year":1912}
{"Name": "Church", "Year":1903}
{"Name": "Nash", "Year":1928}
{"Name": "Mirzakhani", "Year":1977}
{"Name": "Euler", "Year":1707}
{"Name": "Bohm", "Year":1917}
{"Name": "Galileo", "Year":1564}
{"Name": "Germain", "Year":1777}
{"Name": "Lagrange", "Year":1736}
{"Name": "Gauss", "Year":1777}
{"Name": "du Chatelet", "Year":1706}
{"Name": "Thales", "Year":-624}
```

Then the following query is a range query, which selects all years from 1900 onward.

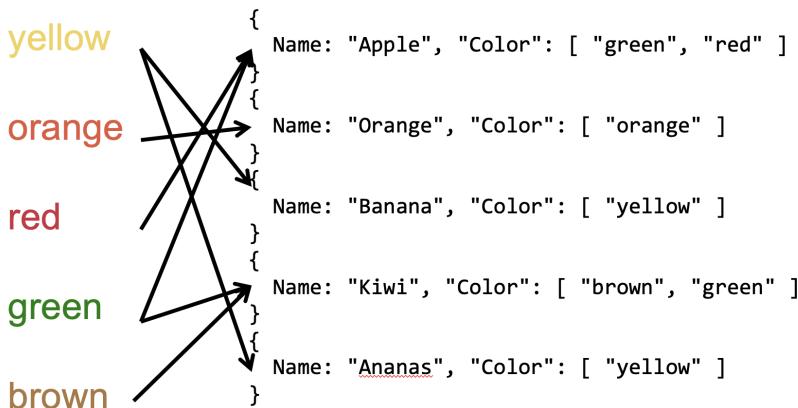
```
find("Year":{$gte":1900})
```

How can an index make queries faster? Let us start with a visual on a simple example. Let us consider the collection

```
{
  Name: "Apple", "Color": [ "green", "red" ]
}
```

```
{
  Name: "Orange", "Color": [ "orange" ]
}
{
  Name: "Banana", "Color": [ "yellow" ]
}
{
  Name: "Kiwi", "Color": [ "brown", "green" ]
}
{
  Name: "Ananas", "Color": [ "yellow" ]
}
```

and imagine we would like to quickly locate fruits that have a specific colour. Then an index would visually look like so:



The general idea is that by picking a color on the left (which can be done quickly – how quickly depends on the index chosen), it then suffices to follow the pointer(s) to the documents with that color, which is just a disk read away.

We will look into two kinds of indices commonly used: hash indices and tree indices.

11.8.2 Hash indices

Hash indices are used to optimize point queries and more generally query that select on a specific value of a field.

The general idea is that all the values that a field takes in a specific collection can be hashed to an integer. The value, together with pointers to the corresponding documents, is then placed in a physical array

in memory, at the position corresponding to this integer (modulo the overall size of the array).

	Value	Records	Scientists
$h(20)=0$	20		("Name": "F": "Albert", "L": "Einstein"), "Country": "Switzerland", "Century": 20)
			("Name": "Gödel", "Country": "Austria", "Century": 20)
			{"Name": "Ramanujan", "Country": "India", "Century": 19}
			{"Name": "Euclid", "Country": "Greece", "Century": -4}
			{"Name": "Pythagoras", "Country": "Greece", "Century": -6}
			{"Name": "Turing", "Country": "UK", "Century": 20}

The primitives involved in this are all in O(1): computing the hash leverages cryptographic tools by means of a so-called hash function. Hash functions fulfill useful criteria such as making collisions unlikely, spreading values uniformly in the index array, etc. They are, in fact, more powerful than what we need, but most importantly *enough* powerful for what we need.

The lookup of a value at a specific position in array is also happening in O(1): this is really just adding the integer to a pointer address to obtain a new address.

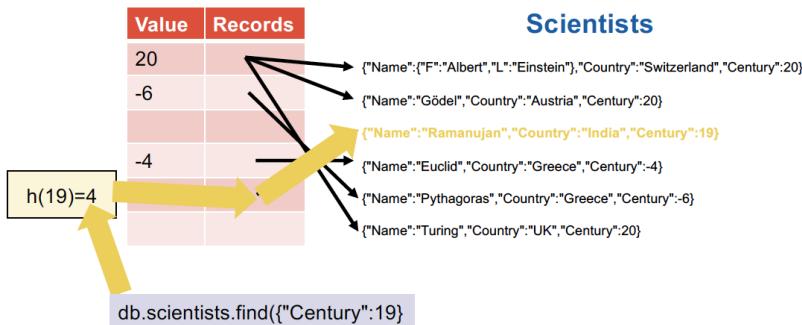
Finally, following the pointer(s) to the document(s) is also happening in O(1), although the constant can vary: if the collection fits in memory and is loaded in memory, then this is just access to the RAM. If the desired documents are not loaded in memory, this is a disk access, but even in this case, disks support random access, so that it is fast too (compared to scanning the entire collection...).

However, there is no free lunch: before an index can be used, it must be built. Building an index consists in the creation of the array structure, and then its population by sequentially scanning through the entire collection, computing the hash of the value, and adding to the index an entry and a pointer to the document, document by document.

	Value	Records	Scientists
	20		("Name": "F": "Albert", "L": "Einstein"), "Country": "Switzerland", "Century": 20)
	-6		("Name": "Gödel", "Country": "Austria", "Century": 20)
	-4		{"Name": "Ramanujan", "Country": "India", "Century": 19}
	19		{"Name": "Euclid", "Country": "Greece", "Century": -4}
			{"Name": "Pythagoras", "Country": "Greece", "Century": -6}
			{"Name": "Turing", "Country": "UK", "Century": 20}

Indices are built at the request of the user, by executing a command. Building an index can either happen synchronously, in the sense that the entire collection (or data store) remains unavailable during build time. Or it can happen asynchronously, meaning that the collection (or data store) remains available, but will be slower until the index is completely built.

The following illustrates how the index is used to perform a point query:



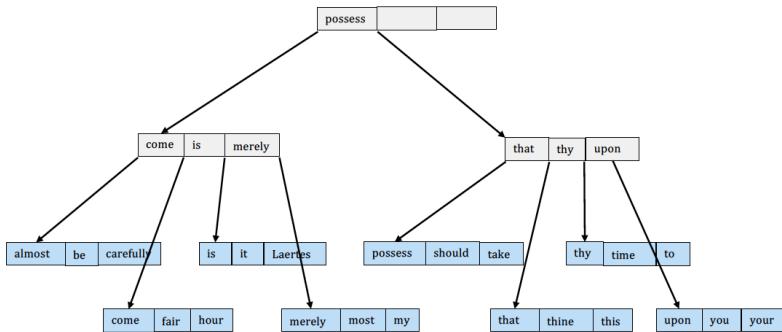
11.8.3 Tree indices

Hash indices are great and fast, but have limitations: first, they consume space. The more one wants to avoid hash collisions, the larger the array needs to be. Collisions can be handled by nesting additional lists in the array entries for all values that land into the same hash value, however if the array is too small these lists grow with the size of the collection and lookups become slower.

But more importantly, hash indices cannot support range queries. This is because hashes do not preserve the order of the values and distribute them “randomly” in the index array structure. At best, evaluating range queries with a hash index might work with discrete value spaces, by looking up each possible value in the sought interval, one by one. This is, however, intractable and inefficient for large ranges, and impossible for decimal numbers or double values.

Range queries are supported with tree indices. Instead of an array, tree indices use some sort of tree structure in which they arrange the possible values of the indexed field, such that the values are ordered when traversing the tree in a depth-first-search manner.

More precisely, the structure is called a B+-tree. Unlike a simple binary tree, nodes have a large number of children. The intent is that the leaves (and nodes) of the tree are large enough to match roughly a disk block, in order to optimize disk latency when fetching the nodes. In many cases, indices are so large (or numerous) that they do not fit in memory, and the database system only loads the parts of the index it needs. With nodes the size of a block, fewer disk accesses are needed.



There are a few constraints in a B+-tree. First, all leaves must be at exactly the same depth. This depth grows with larger collections. Second, the number of children of each node must be within a specific interval, generally parameterized as between $d + 1$ and $2d + 1$ with some choice of d . For pedagogical purposes, we use a small value of d on our examples, but in practice d is larger. The only exception is the root node, which is not subject to the $d + 1$ minimum. The root can have less children, but at least two (had it only one child, it would be useless and could be removed).

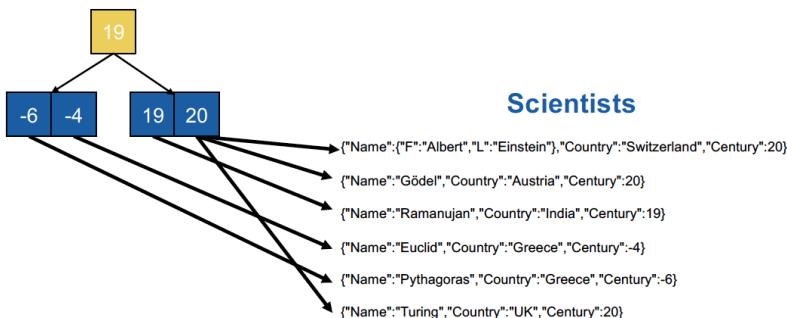
The non-leaf nodes in the tree contain a list of increasing values, interlaced with pointers to the children. These values are compared to the actually sought value in order to locate the pointer that must be followed in order to resolved this sought value. There is exactly one less value on a node than its number of children. Thus, each node has between d and $2d$ values.

In a B+-tree, all possible values appear on the leaves together with a pointer to the documents that contain them. The values can be “repeated” on non-leaf nodes, but values on non-leaf nodes are only used for comparison purposes. A B+-tree also typically chains all its leaves with pointers, for efficient full-traversals in ascending value order. This allows resolving a range query by only looking up the bounds in the B+-tree, and then traversing from the minimum to the maximum value.

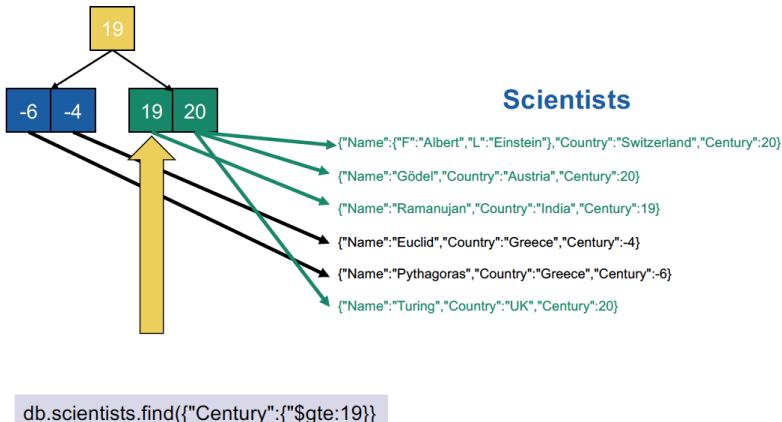
This is unlike B-trees, in which non-leaf nodes can also contain pointers to documents.

Insertion and deletion of values in a B+-tree are a non-trivial matter, which we will not dive into. The idea is that one navigates down the tree and compares the new value to the existing value, then inserts the new value in a free spot. If this breaks the constraint that there must be at most $2d$ values on the same node, then the node is split into two nodes. This, in turn, can cause the number of values on the parent node to exceed $2d$ values, in which case the splitting recursively propagates up the tree, potentially even causing the depth to increase by one. Deletion is the other round: if nodes become too small, they are merged, and this can also cause the depth of the tree to shrink.

Below is a visual of a very small B+-tree index structure on a collection:



And the example of a lookup. The lookup is in logarithmic time ($O(\log n)$) with the size of the collection, which is less efficient than a hash index, but nevertheless very good compared to a linear full scan.



11.8.4 Secondary indices

By default, MongoDB always builds a tree index for the `_id` field. Users can request to build hash and tree indices for more fields. These indices are called secondary indices. The choice of which indices to build depends on the collection and expected queries, and is both a science and an art. Too many indices can be counterproductive and slow down the system.

The command for building a hash index looks like so:

```
db.scientists.createIndex({
  "Name.Last" : "hash"
})
```

And for a tree index (1 means in ascending order, -1 would be descending):

```
db.scientists.createIndex({
  "Name.Last" : 1
})
```

Secondary tree indices can also involve several fields, as opposed to just one for hash indices. When several fields are involved, the value tuples are sorted in lexicographic order.

```
db.scientists.createIndex({
  "Name.Last" : 1
  "Name.First" : -1
})
```

(Note, en passant, that the order in which the fields are given is thus important, which is a breach of the JSON object model in which key-value pairs are not supposed to be ordered – this shows the limits of this and similar APIs compared to high-level query languages, which we will study in a later chapter).

What is important to understand is that, if one builds a tree index on fields A, B, C and D, then a tree index on field A is superfluous, and so is a tree index on fields A and B, and so is a tree index on fields A, B and C. This is because looking up documents with a specific value for A only, or for A and B, or for A, B and C can be efficiently done on the index on all four fields. Why this is, is left as a very interesting exercise (hint: this is because of the lexicographic ordering).

It is a common mistake by document store users to build superfluous indices in this way, which wastes valuable space.

11.8.5 When are indices useful

When building indices, it is important to get a feeling for whether a query will be faster or not with this index.

Some cases are obvious. For example, if we built a hash index on Name.Last, then this query will be instantaneous:

```
db.scientists.find({
  "Name.Last" : "Einstein"
})
```

Let us now imagine we built a hash index on Profession. Then, this query will be faster, but less instantaneous:

```
db.scientists.find({
  "Profession" : "Physicist",
  "Theories" : "Relativity"
})
```

This is because the index can be used to pre-filter a superset of the results (which have the correct value for Profession), but some post-processing is still needed, in memory, to also filter those documents that have the correct value for Theories.

Now, let us look at range queries. If we consider this index:

```
db.scientists.createIndex({
  "Birth" : 1,
  "Death" : -1
})
```

Then this query will be very fast:

```
db.scientists.find({  
    "Birth" : 1887,  
    "Death" : 1946  
})
```

However, since the index is a tree index, an interval lookup will also be faster than a full scan (note that this works because Birth is the first field of the index, it would not work if it were the second):

```
db.scientists.find({  
    "Birth" : { "$gte": 1946 }  
})
```

This query will also be faster, but requires additional post-processing for the Death field in memory

```
db.scientists.find({  
    "Birth" : { "$gte": 1946 },  
    "Death" : 1998  
})
```

11.9 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Do you understand how collections in documents store generalize the concept of a relational table?
- b. Can you explain what documents store can do, but that relational databases cannot do (e.g., heterogeneous collections, schema-less collections, data denormalized into trees...)?
- c. Do you know how to write MongoDB “queries” (i.e., with the JavaScript API) on a low level? Do you know the parameters of the `find()` function? (query, then projection, then sorting)?
- d. Can you explain how, in a document store, the documents can be sharded and replicated? Can you contrast the architecture with that of HDFS or HBase?
- e. Do you understand how indices can make queries faster, like in relational databases?
- f. Do you know what kinds of indices there are (hash, B+-trees)? Do you know how efficient they are, and what each of them can and cannot do?
- g. Are you capable of telling if an index is useful to a given query, for simple settings (for example, an index on a single field and a query that selects on that field)?
- h. Do you know that, and why, a compound index (e.g., on keys a and b) can also be used as an index on any prefix of the compound key (e.g., on key a only) ”for free”?
 - i. Can you explain the limitations of a document store like MongoDB?
 - j. Can you write simple queries in MongoDB’s JSON-based JavaScript API?

11.10 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Chodorow, K. (2013). *MongoDB: The Definitive Guide*. 3rd edition. O’Reilly. Chapters 1, 2, 3, 4, 5, 7.

Chapter 12

Querying denormalized data

12.1 Motivation

12.1.1 Where we are

In several previous chapters, we studied the storage and processing of denormalized data: the syntax, the models, the validation, data frames.

We also looked at document stores, which are database systems that manage denormalized data not as a data lake, but as ETL-based collections with an optimized storage format hidden from the user and additional managed features such as indices that make queries faster and document-level atomicity.

If we now look at the stack we have built for denormalized data, both as data lake and as managed database systems, this is not fully satisfactory. Indeed, an important component is still missing, which is the query language. Indeed, with what we have covered, users are left with two options to handle denormalized datasets:

- They can use an API within an imperative host language (e.g., Pandas in Python, or the MongoDB API in JavaScript, or the Spark RDD API in Java or Scala).
- Or they can push SQL, including ad-hoc extensions to support nestedness, to its limits.

APIs are unsatisfactory for complex analytics use cases. They are very convenient and suitable for Data Engineers that implement more data management layers on top of these APIs, but they are not suitable for end users who want to run queries to analyse data.

There is agreement in the database community that SQL is more satisfactory for the case that data is flat and homogeneous (relational tables). Take the following query for example:

```
SELECT foo
FROM input
WHERE bar = "foobar"
```

which is much simpler to write than the following lower-level equivalent in APIs. With Spark RDDs:

```
rdd1 = sc.textFile("hdfs:///input.json")
rdd2 = rdd1.map(line => parseJSON(line))
rdd3 = rdd2.filter(obj => obj.bar = "foobar")
rdd4 = rdd3.map(obj => obj.foo)
rdd4.saveAsTextFile("hdfs:///output.json")
```

With the Spark DataFrame API:

```
df1 = spark.read.json("hdfs:///input.json")
df2 = df1.filter(df1['bar'] = "foobar")
df3 = df2.select(df2['foo'])
df3.show()
```

Or even if nesting SQL in a host language, there is still additional logic needed to access the collection:

```
df1 = spark.read.json("hdfs:///input.json")
df1.createGlobalTempView("input")
df2 = df1.sql("SELECT foo
FROM input
WHERE bar = 'foobar'
")
df2.show()
```

SQL, possibly extended with a few dots, lateral view syntax and explode-like functions, will work nicely for the most simple use cases. But as soon as more complex functionality is needed, e.g., the dataset is nested up to a depth of 10, or the user would like to denormalize a dataset from relational tables to a single, nested collection, or the user would like to explore and discover a dataset that is heterogeneous, this approach becomes intractable. At best, this leads to gigantic and hard-to-read SQL queries. At worst, there is no way to express the use case in SQL. In both cases, the user ends up writing most of the code in an imperative language, invoking the lower-level API or nesting and chaining simple blocks of SQL. A concrete example that such is the case in the real world is the high-energy-physics community, who

are working with dataframes APIs rather than SQL in spite of their (nested) data being homogeneous.

Here are a few examples of use cases that are simple enough to be manageable in Spark SQL, although they require some effort to be read and understood:

```
SELECT *
FROM person
LATERAL VIEW EXPLODE(ARRAY(30, 60)) tableName AS c_age
LATERAL VIEW EXPLODE(ARRAY(40, 80)) AS d_age;
```

and

```
SELECT key, values, collect_list(value + 1) AS values_plus_one
FROM nested_data
LATERAL VIEW explode(values) T AS value
GROUP BY key, values
```

But let us look at another use case that is simple to express: in the GitHub dataset, for each event, what are all the commits by the top-committer within this event?

In Spark SQL, this is what the query looks like:

```
WITH Commits AS (
    SELECT
        FORMAT('%s-%i', created_at, ROW_NUMBER()
            OVER(PARTITION BY created_at))
        AS event_id,
        payload.shas
    FROM `bigquery-public-data.samples.github_nested`
    WHERE ARRAY_LENGTH(payload.shas) > 0),
CommitterFrequency AS (
    SELECT
        Commits.event_id AS event_id,
        actor_email,
        COUNT(*) commit_count
    FROM Commits, UNNEST(shas)
    GROUP BY event_id, actor_email),
MaxCommitterFrequency AS (
    SELECT
        event_id,
        MAX(commit_count) AS commit_count
    FROM CommitterFrequency
    GROUP BY event_id),
TopCommitters AS (
    SELECT
        c.event_id,
        ANY_VALUE(c.actor_email) AS actor_email
    FROM
        CommitterFrequency c,
        MaxCommitterFrequency m
    WHERE c.event_id = m.event_id AND
        c.commit_count = m.commit_count
    GROUP BY c.event_id),
TopCommitterCommits AS (
    SELECT c.event_id, commits
    FROM
        Commits c,
        UNNEST(shas) AS commits,
        TopCommitters AS tc
    WHERE c.event_id = tc.event_id AND
        commits.actor_email = tc.actor_email
)
SELECT ARRAY_AGG(commits) shas
FROM TopCommitterCommits
GROUP BY event_id
```

In the language we will study in this chapter for denormalized data, this is how the query looks like. As you can see, it is much more compact and easier to read:

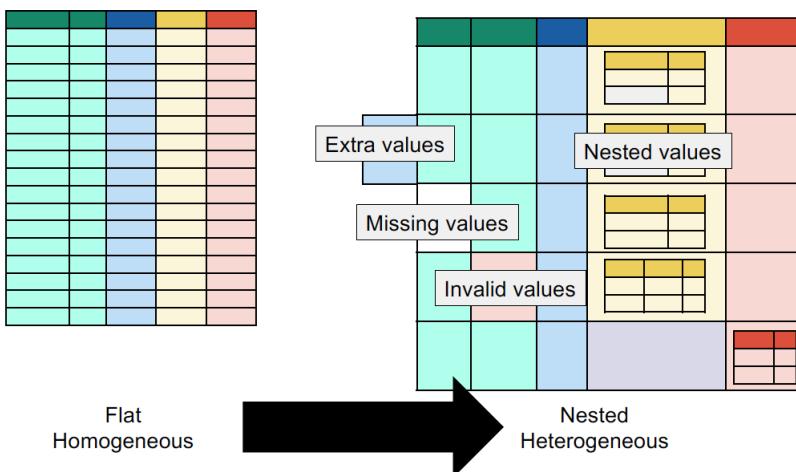
```
for $e in $events
let $top-committer := (
    for $c in $e.commits[]
        group by $c.author
        stable order by count($c) descending
        return $c.author)[1]
    return [
        $e.commits[][$$.author eq $top-committer]
    ]
```

This language is called JSONiq and it is tailor-made for denormalized data. It offers a data-independent layer on top of both data lakes and ETL-based, database management systems, similar to what SQL offers for (flat and homogeneous) relational tables.

98% of JSONiq is directly the same as a W3C standard, XQuery, which is a language offering this functionality for XML datasets. This functionality is the fruit of more than 20 years of work by a two-digit-sized working group from many different companies, many of them with extensive SQL experience (or themselves SQL editors) who carefully discussed every single corner case, leading to a long and precise, publicly available specification. JSONiq is basically XQuery without XML and with (instead) JSON, similar to how one could bake a blueberry cake by using a strawberry cake recipe and simply replacing the strawberries with blueberries. This is a reminder than JSON and XML are very similar when it comes to querying, because both models are based on tree structures. JSONiq was born during the working group discussions on how to add support for maps and arrays to the language and became a standalone language optimized specifically for JSON. XQuery in its latest version supports maps and arrays, and is best suitable in an environment where both XML and JSON co-exist, which is out of scope in this course.

12.1.2 Denormalized data

What do we mean with denormalized data? Let us simply remind that it is characterized with two features: nestedness, and heterogeneity.



Consider the following example of a JSON Lines dataset (note that the objects are displayed on multiple lines only so it fits on the printed page, in reality they would each be on a single line):

```
{
  "Name" : { "First" : "Albert", "Last" : "Einstein" },
  "Countries" : [ "D", "I", "CH", "A", "BE", "US" ]
}
{
  "Name" : { "First" : "Srinivasa", "Last" : "Ramanujan" },
  "Countries" : [ "IN", "UK" ]
}
{
  "Name" : { "First" : "Kurt", "Last" : "Gödel" },
  "Countries" : [ "CZ", "A", "US" ]
}
{
  "Name" : { "First" : "John", "Last" : "Nash" },
  "Countries" : "US"
}
{
  "Name" : { "First" : "Alan", "Last" : "Turing" },
  "Countries" : "UK"
}
{
  "Name" : { "First" : "Maryam", "Last" : "Mirzakhani" },
  "Countries" : [ "IR", "US" ]
}
```

```
{
  "Name" : "Pythagoras",
  "Countries" : [ "GR" ]
}
{
  "Name" : { "First" : "Nicolas", "Last" : "Bourbaki" },
  "Number" : 9,
  "Countries" : null
}
```

If one wants to put it in a DataFrame in order to use Spark SQL, this is what one will get:

Name (String)	Countries (String)	Number (Int)
{ "First" : "Albert", "Last" : "Einstein" }	["D", "I", "CH", "A", "BE", "US"]	null
{ "First" : "Srinivasa", "Last" : "Ramanujan" }	["IN", "UK"]	null
{ "First" : "Kurt", "Last" : "Gödel" }	["CZ", "A", "US"]	null
{ "First" : "John", "Last" : "Nash" }	US	null
{ "First" : "Alan", "Last" : "Turing" },	UK	null
{ "First" : "Maryam", "Last" : "Mirzakhani" }	["IR", "US"]	null
Pythagoras	["GR"]	null
{ "First" : "Nicolas", "Last" : "Bourbaki" }	null	9

As can be seen, the columns “Name” and “Countries” are typed as string, because the system could not deal with the fact that they contain a mix of atomic and structured types. What is in fact happening is that the burden of dealing with heterogeneity is pushed up to the end user, who will be forced to write a lot of additional code in the host language (Python, Java...) to attempt to parse back these strings one by one and decide what to do. This, in turn, is likely to force the user to make a heavy use of UDFs (User-Defined Functions), which are blackboxes that can be called from SQL and with user code inside. But UDFs are very inefficient compared to native SQL execution, because first they need to be registered and shipped to all nodes in the cluster (which not all distributed processing technologies can do efficiently), and second because the SQL optimizer has no idea of what there is inside, which prevents many (otherwise possible) optimizations from kicking in.

In fact, denormalized datasets should not be seen as “broken tables pushed to their limits”, but rather as collections of trees.

The GitHub archive dataset is a good illustration of this: it contains 2,900,000,000 events, each as a JSON document, taking 7.6 TB of space uncompressed. 10% of all the paths (you can think of them as “data frame columns” although, for a heterogeneous dataset, viewing it as a data frame is not very suitable) have mixed types. Furthermore, there are 1,300 such paths in total, although each event only uses 100 of them.

One could think of fitting this into relational tables or dataframes with 1,300 attributes, but 1,300 is already beyond what many relational database systems can handle reasonably well.

12.1.3 Features of a query language

A query language for datasets has three main features.

Declarative

First, it is declarative. This means that the users do not focus on how the query is computed, but on what it should return. Thus, the database engine enjoys the flexibility to figure out the most efficient and fastest plan of execution to return the results.

Functional

Second, it is functional. This means that the query language is made of composable expressions that nest with each other, like a Lego game. Many, but not all, expressions can be seen as functions that take as input the output of their children expressions, and send their output to their parent expressions. However, the syntax of a good functional language should look nothing like a simple chain of function calls with parentheses and lambdas everywhere (this would then be an API, not a query language; examples of APIs are the Spark transformation APIs or Pandas): rather, expression syntax is carefully and naturally designed for ease of write and read. In complement to expressions (typically 20 or 30 different kinds of expressions), a rich function library (this time, with actual function call syntax) completes the expressions to a fully functional language.

Set-based

Finally, it is set-based, in the sense that the values taken and returned by expressions are not only single values (scalars), but are large sequences of items (in the case of SQL, an item is a row). In spite of the set-based terminology, set-based languages can still have bag or list semantics, in that they can allow for duplicates and sequences might be ordered on the logical level.

12.1.4 Query languages for denormalized data

The landscape for denormalized data querying is very different from that of structured, relational data: indeed, for structured data, SQL is undisputed.

For denormalized data though, sadly, the number of languages keeps increasing: the oldest ones being XQuery, JSONiq, but then now also JMESPath, SpahQL, JSON Query, PartiQL, UnQL, N1QL, Object-Path, JSONPath, ArangoDB Query Language (AQL), SQL++, GraphQL, MRQL, Asterix Query Language (AQL), RQL. One day, we expect the market to consolidate.

But the good news is that these languages share common features. In this course, we focus on JSONiq for several reasons:

- It is fully documented;
- Most of its syntax, semantics, function library and type system relies on a W3C standard (XPath/XQuery), meaning that a group of 30+ very smart people with expertise and experience on SQL swept into every corner to define the language;
- It has several independent implementations.

Having learned JSONiq, it will be very easy for the reader to learn any one of the other languages in the future.

12.1.5 JSONiq as a data calculator

The smoothest start with JSONiq is to understand it as a data calculator.

In particular, it can perform arithmetics

Query	<code>1+1</code>
Result	2

Query	<code>3+2*4</code>
Result	11

but also comparison and logic:

Query	<code>2 < 5</code>
Result	true

It is, however, more powerful than a common calculator and supports more complex constructs, for example variable binding:

Query	<code>let \$i := 2 return \$i + 1</code>
Result	3

It also supports all JSON values. Any copy-pasted JSON value literally returns itself:

Query	[1, 2, 3]
Result	[1, 2, 3]

Query	{ "foo" : 1 }
Result	{ "foo" : 1 }

Things start to become interesting with object and array navigation, with dots and square brackets:

Query	{ "foo" : 1 }.foo
Result	1

Query	[3, 4, 5][[1]]
Result	3

Query	{ "foo" : [3, 4, 5] }.foo[[1]] + 3
Result	6

Another difference with a calculator is that a query can return multiple items, as a sequence:

Query	{ "foo" : [3, 4, 5] }.foo[]
Result	3 4 5

Query	1 to 4
Result	1 2 3 4

Query	for \$i in 3 to 5 return { \$i : \$i * \$i }
Result	{ "3" : 9 } { "4" : 16 } { "5" : 25 }

Query	for \$i in { "foo" : [3, 4, 5] }.foo[] return { \$i : \$i * \$i }
Result	{ "3" : 9 } { "4" : 16 } { "5" : 25 }

And, unlike a calculator, it can access storage (data lakes, the Web, etc):

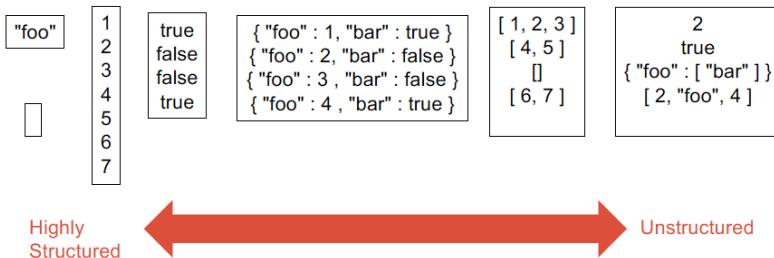
Query	<pre>keys(for \$i in json-file("s3://bucket/myfiles/json/*") return \$i)</pre>
Result	<pre>"foo" "bar"</pre>

Query	<pre>keys(for \$i in parquet-file("s3://bucket/myfiles/parquet") return \$i)</pre>
Result	<pre>"foo" "bar"</pre>

12.2 The JSONNiq Data Model

Every expression of the JSONNiq “data calculator” returns a sequence of items. Always.

Sequences of Items



An item can be either an object, an array, an atomic item, or a function item. For the purpose of this course, we ignore function items, but it might interest the reader to know that function items can be used to represent Machine Learning models, as JSONNiq can be used for training and testing models as well.

Atomic items can be any of the “core” JSON types: strings, numbers (integers, decimals, doubles...), booleans and nulls, but JSONiq has a much richer type system, which we covered in Chapter 7 in the context of both JSound and XML Schema.

Sequences of items are flat, in that sequences cannot be nested. But they scale massively and can contain billions or trillions of items. The only way to nest lists is to use arrays (which can be recursively nested). If you read the previous chapters, you can think of sequences as being similar to RDDs in Spark, or to JSON lines documents with billions of JSON objects.

Sequences can be homogeneous (e.g., a million integers, or a million of JSON objects valid against a flat schema, logically equivalent to a relational table) or heterogeneous (a messy list of various items: objects, arrays, integers, etc).

One item is logically the same as a sequence of one item. Thus, when we say that $1+1$ returns 2, it in fact means that it returns a singleton sequence with one item, which is the integer 2.

A sequence can also be empty. Caution, the empty sequence is not the same logically as a null item.

12.3 Navigation

A good angle to start with JSONiq is to take a large dataset and discover its structure. The online sandbox has examples of this for the GitHub archive dataset, which is continuously growing. Each hour of log can be downloaded from URIs with the pattern

<https://data.gharchive.org/2022-11-01-0.json.gz>

where you can pick the year, month, date and hour of the day.

For the purpose of this textbook, we will pick made-up data patterns to illustrate our point. Let us consider the following JSON document, consisting of a single, large JSON object (possibly on multiple line, as is common for single JSON documents). Let us assume it is named file.json.

```
{
  "o" : [
    {
      "a" : {
        "b" : [
          { "c" : 1, "d" : "a" }
        ]
      }
    },
  ],
},
```

```
{  
    "a" : {  
        "b" : [  
            { "c" : 1, "d" : "f" },  
            { "c" : 2, "d" : "b" }  
        ]  
    }  
},  
{  
    "a" : {  
        "b" : [  
            { "c" : 4, "d" : "e" },  
            { "c" : 8, "d" : "d" },  
            { "c" : 3, "d" : "c" }  
        ]  
    }  
},  
{  
    "a" : {  
        "b" : [ ]  
    }  
},  
{  
    "a" : {  
        "b" : [  
            { "c" : 3, "d" : "h" },  
            { "c" : 9, "d" : "z" }  
        ]  
    }  
},  
{  
    "a" : {  
        "b" : [  
            { "c" : 4, "d" : "g" }  
        ]  
    }  
},  
{  
    "a" : {  
        "b" : [  
            { "c" : 3, "d" : "l" },  
            { "c" : 1, "d" : "m" },  
            { "c" : 0, "d" : "k" }  
        ]  
    }  
}
```

```
        }  
    ]  
}
```

We can open this document and return its contents.

Query	json-doc("file.json")
Result	{ "o" : [{ "a" : { "b" : [{ "c" : 1, "d" : "a" }] } }, { "a" : { "b" : [{ "c" : 1, "d" : "f" }, { "c" : 2, "d" : "b" }] } }, { "a" : { "b" : [{ "c" : 4, "d" : "e" }, { "c" : 8, "d" : "d" }, { "c" : 3, "d" : "c" }] } }, { "a" : { "b" : [] } }, { "a" : { "b" : [{ "c" : 3, "d" : "h" }, { "c" : 9, "d" : "z" }] } },], }

(Continued)	<pre>{ "a" : { "b" : [{ "c" : 4, "d" : "g" }] } } }, { "a" : { "b" : [{ "c" : 3, "d" : "l" }, { "c" : 1, "d" : "m" }, { "c" : 0, "d" : "k" }] } }]</pre>
-------------	---

We are going to start our dataset exploration with JSON navigation. Navigating semi-structured data is several decades old and was pioneered on XML with XPath. JSON navigation uses similar ideas, but is considerably simpler than XML navigation. The general idea of navigation is that it is possible to “dive” into the nested data with dots and square brackets (originally, these were slashes with XPath) – all in parallel: starting with an original collection of objects (or, possibly, a single document), each *step* (i.e., for each dot and each pair of square brackets) goes down the nested structure and returns another sequence of nested items.

The experience feels like scanning the entire collection, moving down the nested structure in parallel¹. Some steps might massively increase the sequence size (i.e., when unboxing arrays); some other steps might on the contrary contract the sequence to a smaller one (i.e., when filtering, or in the presence of heterogeneity when parts of the collection go deeper than others).

12.3.1 Object lookups (dot syntax)

It is possible to navigate into objects with dots, similar to object-oriented programming. For example, this is how we can get the value associated with key *o* in the document (which is a sequence of one object).

¹In XPath, it is possible to also move up the nested structure (e.g., with the .. syntax, similar to what is used in file system paths to go to parent directories), because there are backpointers to parent nodes, or to move aside to siblings. This is achieved with so-called “axes.” This feature does not typically exist in JSON models, in which one only goes down the trees to the children and descendants.

Query	<code>json-doc("file.json").o</code>
Result	<pre>[{ "a" : { "b" : [{ "c" : 1, "d" : "a" }] } }, { "a" : { "b" : [{ "c" : 1, "d" : "f" }, { "c" : 2, "d" : "b" }] } } { "a" : { "b" : [{ "c" : 4, "d" : "e" }, { "c" : 8, "d" : "d" }, { "c" : 3, "d" : "c" }] } } { "a" : { "b" : [] } } { "a" : { "b" : [{ "c" : 3, "d" : "h" }] } } { "a" : { "b" : [{ "c" : 4, "d" : "g" }] } } } { "a" : { "b" : [{ "c" : 3, "d" : "l" }] } }]</pre>

This returned an array, more precisely, a sequence of *one* array item.

12.3.2 Array unboxing (empty square bracket syntax)

We can *unbox* the array, meaning, extract its members as a sequence of seven object items, with empty square brackets, like so:

Query	<code>json-doc("file.json").o[]</code>
Result	<pre>{ "a" : { "b" : [{ "c" : 1, "d" : "a" }] } } { "a" : { "b" : [{ "c" : 1, "d" : "f" }, { "c" : 2, "d" : "b" }] } } { "a" : { "b" : [{ "c" : 4, "d" : "e" }, { "c" : 8, "d" : "d" }, { "c" : 3, "d" : "c" }] } } { "a" : { "b" : [] } } { "a" : { "b" : [{ "c" : 3, "d" : "h" }] } } { "a" : { "b" : [{ "c" : 4, "d" : "g" }] } } } { "a" : { "b" : [{ "c" : 3, "d" : "l" }] } }</pre>

12.3.3 Parallel navigation

The dot syntax, in fact, works on sequences, too. It will extract the value associated with a key in every object of the sequence (anything else than an object is ignored and thrown away):

Query	json-doc("file.json").o[].a
Result	<pre>{ "b" : [{ "c" : 1, "d" : "a" }] } { "b" : [{ "c" : 1, "d" : "f" }, { "c" : 2, "d" : "b" }] } { "b" : [{ "c" : 4, "d" : "e" }, { "c" : 8, "d" : "d" }, { "c" : 3, "d" : "c" }] } { "b" : [] } { "b" : [{ "c" : 3, "d" : "h" }] } { "b" : [{ "c" : 4, "d" : "g" }] } { "b" : [{ "c" : 3, "d" : "l" }] }</pre>

Array unboxing works on sequences, too. Note how all the members are concatenated to a single, merged sequence, similar to a flatMap in Apache Spark.

Query	json-doc("file.json").o[].a.b[]
Result	<pre>{ "c" : 1, "d" : "a" } { "c" : 1, "d" : "f" } { "c" : 2, "d" : "b" } { "c" : 4, "d" : "e" } { "c" : 8, "d" : "d" } { "c" : 3, "d" : "c" } { "c" : 3, "d" : "h" } { "c" : 4, "d" : "g" } { "c" : 3, "d" : "l" }</pre>

12.3.4 Filtering with predicates (simple square bracket syntax)

It is possible to filter any sequence with a predicate, where `$$` in the predicate refers to the current item being tested. Let us only keep those objects that associate `c` with 3:

Query	json-doc("file.json").o[].a.b[] [\$\$.c = 3]
Result	<pre>{ "c" : 3, "d" : "c" } { "c" : 3, "d" : "h" } { "c" : 3, "d" : "l" }</pre>

It is also possible to access the item at position `n` in a sequence with this same notation: if what is inside the square brackets is a Boolean, then it acts as a filtering predicate; if it is an integer, it acts as a position:

Query	<code>json-doc("file.json").o[] .a.b[] [5]</code>
Result	<code>{ "c" : 8, "d" : "d" }</code>

12.3.5 Array lookup (double square bracket syntax)

To access the n-th member of an array, you can use double-square-brackets, like so:

Query	<code>json-doc("file.json").o[[2]].a</code>
Result	<code>{ "b" : [{ "c" : 1, "d" : "f" }, { "c" : 2, "d" : "b" }] }</code>

Like dot object navigation and unboxing, double square brackets (array navigation) work with sequences as well. For any array that has less elements than the requested position, as well as for items that are not arrays, no items are contributed to the output:

Query	<code>json-doc("file.json").o[] .a.b[[2]]</code>
Result	<code>{ "c" : 2, "d" : "b" }</code> <code>{ "c" : 8, "d" : "d" }</code>

12.3.6 A common pitfall: Array lookup vs. Sequence predicates

Do not confuse sequence positions (single square brackets) with array positions (double square brackets)! The difference is easy to see on a simple example involving a sequence of two arrays with two members each:

Query	<code>([1, 2], [3, 4])[2]</code>
Result	<code>[3, 4]</code>

Query	<code>([1, 2], [3, 4])[[2]]</code>
Result	<code>2</code> <code>4</code>

12.4 Schema discovery

We now go on with more simple querying functionality related to discovering datasets with an unknown structure.

12.4.1 Collections

While there exist files that contain a single JSON document (or a single XML document), many datasets are in fact found in the form of large collections of smaller objects (as in document stores).

Such collections are accessed with a function call together with a name or (if reading from a data lake) a path. The name of the function can vary and in this Chapter we will just use the W3C-standard *collection* function. In RumbleDB, a JSON Lines dataset is accessed with the function *json-line*, in a similar way.

Query	<pre>collection("https://www.rumbledb.org/samples/git-archive.jsonl")</pre>
Result	<pre>{ "id" : "7045118886", "type" : "PushEvent", ... { "id" : "7045118891", "type" : "PushEvent", ... { "id" : "7045118892", "type" : "PullRequestEvent", ... { "id" : "7045118894", "type" : "PushEvent", ... { "id" : "7045118895", "type" : "WatchEvent", ... { "id" : "7045118896", "type" : "PushEvent", ... { "id" : "7045118899", "type" : "GollumEvent", ... { "id" : "7045118900", "type" : "PushEvent", ... { "id" : "7045118901", "type" : "PullRequestEvent", ... { "id" : "7045118904", "type" : "PushEvent",</pre>

It is a good idea to look at the first object of a collection to get a rough idea of what the layout looks like (although there is always the risk of heterogeneity, and there is no guarantee all objects look the same):

Query	<pre>collection("https://www.rumbledb.org/samples/git-archive.jsonl")[1]</pre>
Result	<pre>{ "id" : "7045118886", "type" : "PushEvent", ...</pre>

One can also look at the top N objects using the *position* function in a predicate, which returns the position in the sequence of the current item being tested by the predicate (similar to the LIMIT clause in SQL):

Query	<pre>collection("https://www.rumbledb.org/samples/git-archive.jsonl") [position() le 5]</pre>
Result	<pre>{ "id" : "7045118886", "type" : "PushEvent", ... { "id" : "7045118891", "type" : "PushEvent", ... { "id" : "7045118892", "type" : "PullRequestEvent", ... { "id" : "7045118894", "type" : "PushEvent", ... { "id" : "7045118895", "type" : "WatchEvent",</pre>

12.4.2 Getting all top-level keys

The *keys* function retrieves all keys. It can be called on the entire sequence of objects and will return all unique keys found (at the top level) in that collection:

Query	<pre>keys(collection("https://www.rumbledb.org/samples/git-archive.jsonl"))</pre>
Result	<pre>"repo" "org" "actor" "public" "type" "created_at" "id" "payload"</pre>

12.4.3 Getting unique values associated with a key

With dot object lookup, we can look at all the values associated with a key like so:

Query	<pre>collection("https://www.rumbledb.org/samples/git-archive.jsonl") .type</pre>
Result	<pre>"PushEvent" "PushEvent" "PullRequestEvent" "PushEvent" "WatchEvent" "PushEvent" "GollumEvent" "PushEvent" "PullRequestEvent" .—</pre>

With *distinct-values*, it is then possible to eliminate duplicates and look at unique values:

Query	<pre>distinct-values(collection("https://www.rumbledb.org/samples/git-archive.jsonl") .type)</pre>
Result	<pre>"PullRequestEvent" "MemberEvent" "PushEvent" "IssuesEvent" "PublicEvent" "CommitCommentEvent" "ReleaseEvent" "IssueCommentEvent" "ForkEvent" "GollumEvent" "WatchEvent" "PullRequestReviewCommentEvent" "CreateEvent" "DeleteEvent" .—</pre>

12.4.4 Aggregations

Aggregations can be made on entire sequences with a single function call (this would be like a SQL GROUP BY clause but without grouping

key). The five basic functions are *count*, *sum*, *avg*, *min*, *max*. Obviously, the last four require numeric values and will otherwise throw an error.

Query	<code>count(distinct-values(collection("https://www.rumbledb.org/samples/git-archive.jsonl").type))</code>
Result	3597

Query	<code>count(collection("https://www.rumbledb.org/samples/git-archive.jsonl"))</code>
Result	36577

12.5 Construction

Let us now look into how to construct new values with JSONiq.

12.5.1 Construction of atomic values

Atomic values that are core to JSON can be constructed with exactly the same syntax as JSON.

Query	"foo"
Result	"foo"

Query	"This is a line\nand this is a new line"
Result	"This is a line\nand this is a new line"

Query	42
Result	42

Query	3.141592653589793238462650283279
Result	3.141592653589793238462650283279

Query	-6.022E23
Result	-6.022E23

Query	true
Result	true

Query	false
Result	false

Query	null
Result	null

For more specific types, a cast is needed. This works with any of the atomic types we covered in Chapter 7. There are two syntaxes for this:

Query	nonNegativeInteger("42")
Result	42

Query	"42" cast as nonNegativeInteger
Result	42

12.5.2 Construction of objects and arrays

Objects and arrays are constructed with the same syntax as JSON. In fact, one can copy-paste any JSON value, and it will always be recognized as a valid JSONiq query returning that value.

Query	[{ "foo" : "bar" }, { "bar" : [1, 2, true, null] }]
Result	[{ "foo" : "bar" }, { "bar" : [1, 2, true, null] }]

It is also possible to build objects and arrays dynamically (with computed values, not known at compile time), as will be shown shortly when we discuss composability of expressions.

12.5.3 Construction of sequences

Sequences can be constructed (and concatenated) using commas:

Query	[2, 3], true, "foo", { "f" : 1 }
Result	[2, 3] true "foo" { "f" : 1 }

Increasing sequences of integers can also be built with the *to* keyword:

Query	<code>1 to 10</code>
Result	1 2 3 4 5 6 7 8 9 10

Another way to build sequences is with FLWOR expressions, covered a bit further down.

12.6 Scalar expressions

Sequences of items can have any number of items. A few JSONiq expression (arithmetic, logic, value comparison...) work on the particular case that a sequence has zero or one items.

12.6.1 Arithmetic

JSONiq supports basic arithmetic: addition (+), subtraction (-), division (div)², integer division (idiv) and modulo (mod).

If both sides have exactly one item, the semantics is relatively natural.

Query	<code>1 + 1</code>
Result	2

Query	<code>42 - 10</code>
Result	32

Query	<code>6 * 7</code>
Result	42

Query	<code>42.3 div 7.2</code>
Result	5.875

Query	<code>42 idiv 9</code>
Result	4

²Mind that this is not a slash (/).

Query	<code>42 mod 9</code>
Result	6

If the data types are different, then conversions are made automatically:

- If one side is a double and the other side a float, then the float is converted to a double and a double is returned.
- If one side is a double and the other side a decimal (or integer, etc), then the decimal is converted to a double and a double is returned.
- If one side is a float and the other side a decimal (or integer, etc), then the decimal is converted to a float and a float is returned.

Note that an integer and a decimal are not considered different here, because an integer is a special case of decimal. Adding a decimal with an integer returns a decimal.

The empty sequence enjoys special treatment: if one of the sides (or both) is the empty sequence, then the arithmetic expression returns an empty sequence (no error):

Query	<code>() + 1</code>
Result	

If one of the two sides is null (and the other side is not the empty sequence), then the arithmetic expression returns null.

If one of the sides (or both) is not a number, null, or the empty sequence, then a type error is thrown.

12.6.2 String manipulation

String concatenation is done with a double vertical bar:

Query	<code>"foo" "bar"</code>
Result	"foobar"

Most other string manipulation primitives are available from the rich JSONiq builtin function library (itself relying on a W3C standard called XPath and XQuery Functions and Operators). The complete list of functions in this standard is available at <https://www.w3.org/TR/xpath-functions/>, although XML-related functions should be ignored, and JSONiq supports additional JSON-related functions (such as `keys()`, etc).

Query	<code>concat("foo", "bar")</code>
Result	"foobar"

Query	<code>string-join(("foo", "bar", "foobar"), "-")</code>
Result	"foo-bar-foobar"

Query	<code>substr(("foobar", 4, 3)</code>
Result	"bar"

Query	<code>string-length("foobar")</code>
Result	6

12.6.3 Value comparison

Sequences of one atomic item can be compared with eq (equal), ne (not equal), le (lower or equal), ge (greater or equal), lt (lower than) and gt (greater than).

Query	<code>1 + 1 eq 2</code>
Result	true

Query	<code>6 * 7 ne 21 * 2</code>
Result	false

Query	<code>234 gt 123</code>
Result	true

If one of the two sides is the empty sequence, then the value comparison expression returns an empty sequence as well.

Query	<code>() le 2</code>
Result	

If one of the two sides is null, then the value comparison expression returns null as well.

Query	<code>null le 2</code>
Result	null

12.6.4 Logic

JSONiq supports the three basic logic expressions and, or, and not. not has the highest precedence, then and, then or.

Query	<code>1 + 1 eq 2 and (2 + 2 eq 4 or not 100 mod 5 eq 0)</code>
Result	<code>true</code>

JSONiq also supports universal and existential quantification:

Query	<code>every \$i in 1 to 10 satisfies \$i gt 0</code>
Result	<code>true</code>

Query	<code>some \$i in 1 to 10 satisfies \$i gt 5</code>
Result	<code>true</code>

Note that unlike SQL, JSONiq logic expressions are two-valued and return either true or false.

If one of the two sides is not a sequence of a single Boolean item, then implicit conversions are made. This mechanism is called the Effective Boolean Value (EBV). For example, an empty sequence, or a sequence of one empty string, or a sequence of one zero integer, is considered false. A sequence of one non-empty string, or a sequence of one non-zero integer, or a sequence starting with one object (or array) is considered true.

12.6.5 General comparison

JSONiq has a shortcut for existential quantification on value comparisons. This is called general comparison.

For example, consider this query:

Query	<code>some \$i in (1, 2, 3, 4, 5) satisfies \$i eq 1</code>
Result	<code>true</code>

It can be abbreviated to the shorter:

Query	<code>(1, 2, 3, 4, 5) = 1</code>
Result	<code>true</code>

More generally,

Query	<code>some \$i in 1 to 5, \$j in 3 to 10 satisfies \$i gt \$j</code>
Result	<code>true</code>

can be abbreviated to:

Query	<code>1 to 5 > 3 to 10</code>
Result	<code>true</code>

In other words, `=` (resp. `!=`, `<`, `>`, `<=`, `>=`) is a shortcut for an existential quantification on both input sequences on the value comparison `eq` (resp. `ne`, `lt`, `gt`, `le`, `ge`).

In particular, errors are thrown for incompatible types, and `false` is returned if any side is the empty sequence.

General comparison is very convenient when scanning datasets and looking for matching values.

Query	<code>json-doc("file.json").o[] .a.b[] .c = 1</code>
Result	<code>true</code>

12.7 Composability

JSONiq, as a functional language, is modular. This means that expressions can be combined at will, exactly like one would combine addition, multiplication, etc, at will.

Any expression can appear as the operand of any other expression. Of course, if the output of an expression is not compatible with what the parent expression expects, an error is thrown.

We show below an example and the successive details of the evaluation.

Query	<code>(1 + (({"b": [{"a": 2+2}]}).b[] .a)) to 10</code>
Step 1	<code>(1 + (({"b": [{"a": 4}]}).b[] .a)) to 10</code>
Step 2	<code>(1 + ([{"a": 4}] [] .a)) to 10</code>
Step 3	<code>(1 + ({"a": 4}.a)) to 10</code>
Step 4	<code>(1 + 4) to 10</code>
Step 4	<code>5 to 10</code>
Result	<code>5</code> <code>6</code> <code>7</code> <code>8</code> <code>9</code> <code>10</code>

Here is another example:

Query	<pre>{ "attr" : string-length("foobar") "values" : [for \$i in 1 to 10 return long(\$i)] }</pre>
Result	<pre>{ "attr" : 6 "values" : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] }</pre>

Just like arithmetic, where multiplication has precedence over addition, expressions have a precedence order. Precedence can be easily overridden with parentheses. In practice, it is not realistic to know all precedences by heart, so that when in doubt, it is always a good idea to add parentheses to be on the safe side.

We list the expression in increasing order of precedence below. Beware, the low precedence of the comma is a common pitfall.

Precedence (low first)

Comma

Data Flow (FLWOR, if-then-else, switch...)

Logic

Comparison

String concatenation

Range

Arithmetic

Path expressions

Filter predicates, dynamic function calls

Literals, constructors and variables

Function calls, named function references, inline functions

12.7.1 Data flow

A few expressions give some control over the data flow by picking the output or this or that expression based on the value of another expression. These expression look quite close to their counterparts in imperative languages (Python, Java...), but it is important to understand that they have functional semantics here.

This includes conditional expressions. If the expression inside the if returns true (or if its Effective Boolean Value is true), the result of the expression in the then clause is taken, otherwise, the result of the expression in the else clause is taken.

Query	<pre>if(count(json-file("file.json")).o) gt 1000) then "Large file!" else "Small file."</pre>
Result	"Small file."

This also includes switch expressions. The expression inside the switch is evaluated and an error is thrown if more than one item is returned. Then, the resulting item is compared for equality with each one of the candidate values. The result of the expression corresponding to the first match is taken, and if there are no matches, the result of the default expression is taken.

Query	<pre>switch(json-file("file.json").o[[1]].a.b[[1]].c) case 1 return "one" case 2 return "two" default return "other"</pre>
Result	one

Note that we covered data types and cardinality indicators (*, +, ?) in Chapter 7.

This also includes try-catch expressions. If the expression in the try clause is successfully evaluated, then its results are taken. If there is an error, then the results of the expression in the first catch clause matching the error is taken (* being the joker).

Query	<pre>try { date(json-file("file.json").o[[1]].a.b[[1]].c) } catch * { "This is not a date!" }</pre>
Result	"This is not a date!"

12.8 Binding variables with cascades of let clauses

Let us go back to this example:

Query	<pre>json-doc("file.json").o[] .a.b[] .c = 1</pre>
Result	true

We can rewrite it by explicitly binding intermediate variables, like so:

Query	<pre>let \$a := json-doc("file.json") let \$b := \$a.o let \$c := \$b[] let \$d := \$c.a let \$e := \$d.b let \$f := \$d[] let \$g := \$f.c return \$g = 1</pre>
Result	true

Variables in JSONiq start with a dollar sign. This way of subsequently binding variables to compute intermediate results is typical of functional language: OCAML, F#, Haskell... have a similar feature. It is important to understand that this is not a variable assignment that would *change* the value of a variable. This is only a declarative binding.

It is possible to reuse the same variable name, in which case the previous binding is hidden. Again, this is not an assignment or a change of value.

Query	<pre>let \$a := json-doc("file.json") let \$a := \$a.o let \$a := \$a[] let \$a := \$a.a let \$a := \$a.b let \$a := \$a[] let \$a := \$a.c return \$a = 1</pre>
Result	true

Each variable is visible in all subsequent let clauses, as well as in the final return clause (unless/until it is hidden by another let clause with the same variable name). It is not visible to any parent expressions. In particular, this query returns an error because the last reference to variable \$a is not within the scope of the let \$a binding:

Query	<pre>(let \$a := json-doc("file.json") let \$b := \$a.o let \$c := \$b[] let \$d := \$c.a let \$e := \$d.b let \$f := \$d[] let \$g := \$f.c return \$g = 1) + \$a</pre>
-------	---

12.9 FLWOR expressions

One of the most important and powerful features of JSONiq is the FLWOR expression. It corresponds to SELECT-FROM-WHERE queries in SQL, however, it is considerably more expressive and generic than them in several aspects:

- In SQL, the clauses must appear in a specific order (SELECT, then FROM, then WHERE, then GROUP BY, then HAVING, then ORDER BY, then OFFSET, then LIMIT), although most are optional. In JSONiq the clauses can appear in any order with the exception of the first and last clause.
- JSONiq supports a let clause, which does not exist in SQL. Let clauses make it very convenient to write and organize more complex queries.
- In SQL, when iterating over multiple tables in the FROM clause, they “do not see each other”, i.e., the semantics is (logically) that of a Cartesian product. In JSONiq, for clauses (which correspond to FROM clauses in SQL), do see each other, meaning that it is possible to iterate in higher and higher levels of nesting by referring to a previous for variable. This is both easier to write and read than lateral views, and it is also more expressive.
- The semantics of FLWOR clauses is simple, clean, and inherently functional; it is based on tuple streams containing variable bindings, which flow from clause to clause. There is no “spooky action at a distance” such as the explode() function, which indirectly causes a duplication of rows in Spark SQL.

12.9.1 Simple dataset

For the purpose of illustration, we will use a very simple dataset consisting of two JSON Lines files:

products.json
<pre>{"pid":1, "type" : "tv", "store":1} {"pid":2, "type" : "tv", "store":2} {"pid":3, "type" : "phone", "store":2} {"pid":4, "type" : "tv", "store":3} {"pid":5, "type" : "teapot", "store":2} {"pid":6, "type" : "tv", "store":1} {"pid":7, "type" : "teapot", "store":2} {"pid":8, "type" : "phone", "store":4}</pre>

stores.json
{ "sid" : 1, "country" : "Switzerland" }
{ "sid" : 2, "country" : "Germany" }
{ "sid" : 3, "country" : "United States" }

Note that the Store ID 4 for stores is intentionally missing from stores.json, this is for the purpose of showing what happens if there are no matches.

12.9.2 For clauses

For clauses bind their variable in turn to each item of the provided expression. Here is an example:

Query	<pre>for \$x in 1 to 10 return { "number": \$x, "square": \$x * \$x }</pre>
Result	<pre>{ "number" : 1, "square" : 1 } { "number" : 2, "square" : 4 } { "number" : 3, "square" : 9 } { "number" : 4, "square" : 16 } { "number" : 5, "square" : 25 } { "number" : 6, "square" : 36 } { "number" : 7, "square" : 49 } { "number" : 8, "square" : 64 } { "number" : 9, "square" : 81 } { "number" : 10, "square" : 100 }</pre>

In the above query, the variable \$x is bound with 1, then with 2, then with 3, etc, and finally with 10. It is always bound with a sequence of exactly one item. It is, however, possible to bind it with an empty sequence if the expression returns no items. This is done with “allowing empty”.

Query	<pre>for \$x allowing empty in () return count(\$x)</pre>
Result	0

Note that, without “allowing empty”, if the expression in the for clause evaluates to an empty sequence, the variable would not bind to anything at all and the FLWOR expression would simply return an empty sequence.

Query	<code>for \$x in () return count(\$x)</code>
Result	

Each variable binding is also more generally called a tuple. In this above examples, there is only one variable binding in each tuple (\$x), but it is possible to build larger tuples with more clauses. For example, this FLWOR expression involves two for clauses. The tuples after the first for clause and before the second one only bind variable \$x (to 1, then to 2, then to 3), but the tuple after the second for clause and before the return clause bind variables \$x and \$y. There are six tuples in total, because the second for clause expands each incoming tuple to zero, one or more tuples (think of a flatMap transformation in Spark for an analogy).

Query	<code>for \$x in 1 to 3 for \$y in 1 to \$x return [\$x, \$y]</code>
Result	[1, 1] [2, 1] [2, 2] [3, 1] [3, 2] [3, 3]

Now if we use our small example dataset, we can iterate on all objects, say, products:

Query	<code>for \$product in json-file("products.json") return \$product.type</code>
Result	"tv" "tv" "phone" "tv" "teapot" "tv" "teapot" "phone" ...

It can thus be seen that the for clause is akin to the FROM clause in SQL, and the return is akin to the SELECT clause.

Projection in JSONiq can be made with a project() function call, with the keys to keep:

Query	<pre>for \$product in json-file("products.json") return project(\$product, ("type", "store"))</pre>
Result	<pre>{"type" : "tv", "store":1} {"type" : "tv", "store":2} {"type" : "phone", "store":2} {"type" : "tv", "store":3} {"type" : "teapot", "store":2} {"type" : "tv", "store":1} {"type" : "teapot", "store":2} {"type" : "phone", "store":4}</pre>

It is possible to implement a join with a sequence of two for clauses and a predicate (note that newlines in JSONiq are irrelevant, so we spread the for clause on two lines in order to fit the query on this page):

Query	<pre>for \$product in json-file("products.json") for \$store in json-file("stores.json") [\$\$.sid eq \$product.store] return { "product" : \$product.type, "country" : \$store.country }</pre>
Result	<pre>{"product" : "tv", "country":"Switzerland"} {"product" : "tv", "country":"Germany"} {"product" : "phone", "country":"Germany"} {"product" : "tv", "country":"United States"} {"product" : "teapot", "country":"Germany"} {"product" : "tv", "country":"Switzerland"} {"product" : "teapot", "country":"Germany"}</pre>

Note that allowing empty can be used to perform a left outer join, i.e., to account for the case when there are no matching records in the second collection:

Query	<pre>for \$product in json-file("products.json") for \$store allowing empty in json-file("stores.json") [\$\$.sid eq \$product.store] return { "product" : \$product.type, "country" : \$store.country }</pre>
Result	<pre>{"product" : "tv", "country": "Switzerland"} {"product" : "tv", "country": "Germany"} {"product" : "phone", "country": "Germany"} {"product" : "tv", "country": "United States"} {"product" : "teapot", "country": "Germany"} {"product" : "tv", "country": "Switzerland"} {"product" : "teapot", "country": "Germany"} {"product" : "phone", "country": null}</pre>

In the case of the last product, no matching record in stores.json is found and \$store is bound to the empty sequence for that tuple. When constructing the object in the return clause's expression, the empty sequence obtained from \$store.country is automatically replaced with a null value (because an object value cannot be empty). But if we add an array constructor around the country, we will notice the empty sequence:

Query	<pre>for \$product in json-file("products.json") for \$store allowing empty in json-file("stores.json") [\$\$.sid eq \$product.store] return { "product" : \$product.type, "country" : [\$store.country] }</pre>
Result	<pre>{"product" : "tv", "country": ["Switzerland"]} {"product" : "tv", "country": ["Germany"]} {"product" : "phone", "country": ["Germany"]} {"product" : "tv", "country": ["United States"]} {"product" : "teapot", "country": ["Germany"]} {"product" : "tv", "country": ["Switzerland"]} {"product" : "teapot", "country": ["Germany"]} {"product" : "phone", "country": []]}</pre>

12.9.3 Let clauses

As seen before, the let clause can be used to bind a variable with any sequence of items, also more than one. FLWOR expressions with just

a cascade of let clauses are quite popular.

Query	<code>let \$x := 2 return \$x * \$x</code>
Result	4

However, let clauses can also appear after other clauses, for example, after a for clause. Then, they will bind a sequence of items for each previous binding (tuple), like so:

Query	<code>for \$x in 1 to 10 let \$square := \$x * \$x return { "number": \$x, "square": \$square }</code>
Result	<code>{ "number" : 1, "square" : 1 } { "number" : 2, "square" : 4 } { "number" : 3, "square" : 9 } { "number" : 4, "square" : 16 } { "number" : 5, "square" : 25 } { "number" : 6, "square" : 36 } { "number" : 7, "square" : 49 } { "number" : 8, "square" : 64 } { "number" : 9, "square" : 81 } { "number" : 10, "square" : 100 }</code>

In the above example, `$square` is only bound with one item. Here is another example where it is bound with more than one:

Query	<pre>for \$x in 1 to 10 let \$square-and-cube := (\$x * \$x, \$x * \$x * \$x) return { "number": \$x, "square": \$square-and-cube[1] , "cube": \$square-and-cube[2] }</pre>
Result	<pre>{ "number" : 1, "square" : 1, "cube" : 1 } { "number" : 2, "square" : 4, "cube" : 8 } { "number" : 3, "square" : 9, "cube" : 27 } { "number" : 4, "square" : 16, "cube" : 64 } { "number" : 5, "square" : 25, "cube" : 125 } { "number" : 6, "square" : 36, "cube" : 216 } { "number" : 7, "square" : 49, "cube" : 343 } { "number" : 8, "square" : 64, "cube" : 512 } { "number" : 9, "square" : 81, "cube" : 729 } { "number" : 10, "square" : 100, "cube" : 1000 }</pre>

Note the difference with a for clause:

Query	<pre> for \$x in 1 to 10 for \$square-or-cube in (\$x * \$x, \$x * \$x * \$x) return { "number": \$x, "square or cube": \$square-or-cube } </pre>
Result	<pre> { "number" : 1, "square or cube" : 1 } { "number" : 1, "square or cube" : 1 } { "number" : 2, "square or cube" : 4 } { "number" : 2, "square or cube" : 8 } { "number" : 3, "square or cube" : 9 } { "number" : 3, "square or cube" : 27 } { "number" : 4, "square or cube" : 16 } { "number" : 4, "square or cube" : 64 } { "number" : 5, "square or cube" : 25 } { "number" : 5, "square or cube" : 125 } { "number" : 6, "square or cube" : 36 } { "number" : 6, "square or cube" : 216 } { "number" : 7, "square or cube" : 49 } { "number" : 7, "square or cube" : 343 } { "number" : 8, "square or cube" : 64 } { "number" : 8, "square or cube" : 512 } { "number" : 9, "square or cube" : 81 } { "number" : 9, "square or cube" : 729 } { "number" : 10, "square or cube" : 100 } { "number" : 10, "square or cube" : 1000 } </pre>

A let clause outputs exactly one outgoing tuple for each incoming tuple (think of a map transformation in Spark). Unlike the for clause, it does not modify the number of tuples.

Let us now showcase the use of a let clause with our dataset.

Now if we use our small example dataset, we can iterate on all objects, say, products:

Query	<pre>for \$product in json-file("products.json") let \$type := \$product.type return \$type</pre>
Result	<pre>"tv" "tv" "phone" "tv" "teapot" "tv" "teapot" "phone" ...</pre>

Let clauses also allow for joining the two datasets and denormalizing them by nesting the stores into the products. This would be considerably more difficult to do with (Spark) SQL, even with extensions. The results are pretty-printed for ease of read.

Query	<pre>for \$store in json-file("stores.json") let \$product := json-file("products.json") [\$store.sid eq \$\$.\$store] return { "store" : \$store.country, "available products" : [distinct-values(\$product.type)] }</pre>
Result	<pre>{ "store" : "Germany", "available products" : ["tv", "teapot", "phone"] } { "store" : "Switzerland", "available products" : ["tv"] } { "store" : "United States", "available products" : ["tv"] }</pre>

12.9.4 Where clauses

Where clauses are used to filter variable bindings (tuples) based on a predicate on these variables. They are the equivalent to a WHERE clause in SQL.

This is a simple example of its use in conjunction with a for clause:

Query	<pre>for \$x in 1 to 10 where \$x gt 7 return { "number": \$x, "square": \$x * \$x }</pre>
Result	<pre>{ "number" : 8, "square" : 64 } { "number" : 9, "square" : 81 } { "number" : 10, "square" : 100 }</pre>

A where clause can appear anywhere in a FLWOR expression, except that it cannot be the first clause (always for or let) or the last clause (always return).

Query	<pre>for \$x in 1 to 10 let \$square := \$x * \$x where \$square gt 60 for \$y in \$square to \$square + 1 return { "number": \$x, "y": \$y }</pre>
Result	<pre>{ "number" : 8, "y" : 64 } { "number" : 8, "y" : 65 } { "number" : 9, "y" : 81 } { "number" : 9, "y" : 82 } { "number" : 10, "y" : 100 } { "number" : 10, "y" : 101 }</pre>

A where clause always outputs a subset (or all) of its incoming tuples, without any alteration. In the case that the predicate always evaluates to true, it forwards all tuples, as if there had been no where clause at all. In the case that the predicate always evaluates to false, it outputs no tuple and the FLWOR expression will then return the empty sequence, with no need to further evaluate any of the remaining clauses.

Here is another example of use of the where clause with our datasets:

Query	<pre>for \$product in json-file("products.json") let \$store := json-file("stores.json") [\$\$.sid eq \$product.store] where \$store.country = "Germany" return \$product.type</pre>
Result	<pre>"tv" "phone" "teapot" "teapot"</pre>

12.9.5 Order by clauses

Order by clauses are used to reorganize the order of the tuples, but without altering them. They are the same as ORDER BY clauses in SQL.

Query	<pre>for \$x in -2 to 2 let \$square := \$x * \$x order by \$square return { "number": \$x, "square": \$square }</pre>
Result	<pre>{ "number" : 0, "square" : 0 } { "number" : -1, "square" : 1 } { "number" : 1, "square" : 1 } { "number" : -2, "square" : 4 } { "number" : 2, "square" : 4 }</pre>

It is also possible, like in SQL, to specify an ascending or a descending order. By default, the order is ascending.

Query	<pre>for \$x in -2 to 2 let \$square := \$x * \$x order by \$square ascending return { "number": \$x, "square": \$square }</pre>
Result	<pre>{ "number" : 0, "square" : 0 } { "number" : -1, "square" : 1 } { "number" : 1, "square" : 1 } { "number" : -2, "square" : 4 } { "number" : 2, "square" : 4 }</pre>

Query	<pre>for \$x in -2 to 2 let \$square := \$x * \$x order by \$square descending return { "number": \$x, "square": \$square }</pre>
Result	<pre>{ "number" : 2, "square" : 4 } { "number" : -2, "square" : 4 } { "number" : 1, "square" : 1 } { "number" : -1, "square" : 1 } { "number" : 0, "square" : 0 }</pre>

In case of ties between tuples, the order is arbitrary. But it is possible to sort on another variable in case there is a tie with the first one (compound sorting keys):

Query	<pre>for \$x in -2 to 2 let \$square := \$x * \$x order by \$square descending, \$x ascending return { "number": \$x, "square": \$square }</pre>
Result	<pre>{ "number" : -2, "square" : 4 } { "number" : 2, "square" : 4 } { "number" : -1, "square" : 1 } { "number" : 1, "square" : 1 } { "number" : 0, "square" : 0 }</pre>

It is possible to control what to do with empty sequences: they can be considered smallest or greatest.

Query	<pre>for \$x in 1 to 5 let \$y := \$x[\$\$ mod 2 = 1] order by \$y ascending empty greatest return [\$y]</pre>
Result	<pre>[1] [3] [5] [] []</pre>

Query	<pre>for \$x in 1 to 5 let \$y := \$x[\$\$ mod 2 = 1] order by \$y ascending empty least return [\$y]</pre>
Result	<pre>[] [] [1] [3] [5]</pre>

Here is another example of use of the order by clause with our datasets:

Query	<pre>for \$product in json-file("products.json") let \$store := json-file("stores.json") [\$\$.sid eq \$product.store] group by \$t := \$product.type order by count(\$store) descending, string-length(\$t) ascending return \$t</pre>
Result	<pre>"tv" "teapot" "phone"</pre>

12.9.6 Group by clauses

Group by clauses organize tuples in groups based on matching keys, and then output only one tuple for each group, aggregating other variables (count, sum, max, min...). This is similar to GROUP BY clauses in SQL.

Query	<pre>for \$x in 1 to 5 let \$y := \$x mod 2 group by \$y return { "grouping key" : \$y, "count of x" : count(\$x) }</pre>
Result	<pre>{ "grouping key" : 0, "count of x" : 2 } { "grouping key" : 1, "count of x" : 3 }</pre>

However, JSONiq's group by clauses are more powerful and expressive than SQL GROUP BY clauses: indeed, it is also possible to opt out of aggregating other (non-grouping-key) variables. Then, for a non-aggregated variable, the sequence of all its values within a group will

be rebound to this same variable as a single binding in the outcoming tuple. It is thus possible to write many more queries than SQL would allow, which is one of the reasons why a language like JSONiq should be preferred for nested datasets.

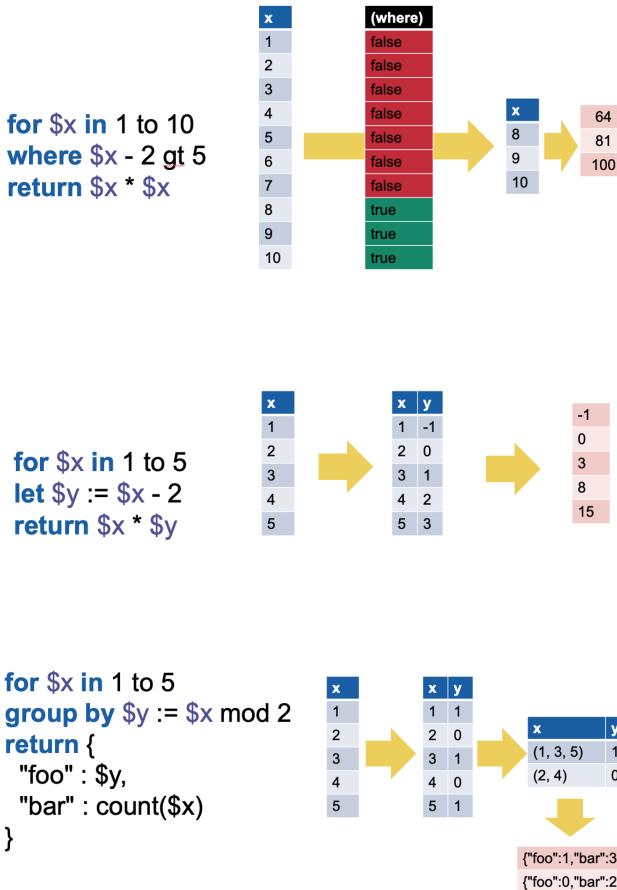
Query	<pre>for \$x in 1 to 5 let \$y := \$x mod 2 group by \$y return { "grouping key" : \$y, "grouped x values" : [\$x], }</pre>
Result	<pre>{ "grouping key" : 0, "grouped x values" : [2, 4] } { "grouping key" : 1, "grouped x values" : [1, 3, 5] }</pre>

Finally, here is an example of use of a group by clause with our example dataset.

Query	<pre>for \$product in json-file("products.json") group by \$sid := \$product.sid order by \$sid let \$store := json-file("stores.json") [\$\$.sid = \$sid] return { \$store, { "products" : [distinct-values(\$product.type)] } }</pre>
Result	<pre>{ "sid" : 1, "country" : "Switzerland", "products" ["tv"] } { "sid" : 2, "country" : "Germany", "products" : ["tv", "phone", "teapot"] } { "sid" : 3, "country" : "United States", "products" : ["tv"] }</pre>

12.9.7 Tuple stream visualization

Although it is unnecessary to write simple FLWOR expressions, a visualization can be helpful in order to understand how more complex FLWOR expressions are evaluated. We give below a few examples of how tuple streams within a FLWOR expression can be seen as tables (or DataFrames) in which each bound variable is represented in a column:



Note, however, that these tuple streams are not sequences of items, because clauses are not expressions; tuple streams are only a formal description of the semantics of FLWOR expressions and their visualization as DataFrames is pedagogical. Having said that, the reader may have guessed that tuple streams can be internally implemented as Spark DataFrames, and in fact, RumbleDB does just that (but it hides it from the user).

12.10 Types

The type system in JSONiq is consistent with what was covered in Chapter 7.

12.10.1 Variable types

It is possible to annotate any FLWOR variable with an expected type as shown below.

Query	<pre>let \$path as string := "git-archive-big.json" let \$events as object* := json-file(\$path) let \$actors as object* := \$events.actor let \$logins as string* := \$actors.login let \$distinct-logins as string* := distinct-values(\$logins) let \$count as integer := count(\$distinct-logins) return \$count</pre>
Result	53744

Since every value in JSONiq is a sequence of item, a sequence type consists of two parts: an item type, and a cardinality.

Item types can be any of the builtin atomic types (JSound) covered in Chapter 7, as well as “object”, “array” and the most generic item type, “item”. Cardinality can be one of the following four:

- Any number of items (suffix *); for example object*
- One or more items (suffix +); for example array+
- Zero or one item (suffix ?); for example boolean?
- Exactly one item (no suffix); for example integer

If it is detected, at runtime, that a sequence of items is bound to a variable but does not match the expected sequence type, either because one of the items does not match the expected item type, or because the cardinality of the sequence does not match the expected cardinality, then a type error is thrown and the query is not evaluated.

It is also possible to annotate variables in for clauses, however the cardinality of the sequence type of a for variable will logically be either one (no suffix), or zero-or-one (?) in the case that “allowing empty” is specified.

12.10.2 Type expressions

JSONiq has a few expressions related to types.

An instance of expression checks whether a sequences matches a sequence type, and returns true or false. This is similar to the homonymous expression in Java.

Query	<code>(3.14, "foo") instance of integer*, ([1], [2, 3]) instance of array+</code>
Result	<code>false</code> <code>true</code>

A cast as expression casts single items to an expected item type.

Query	<code>"3.14" cast as decimal</code>
Result	<code>3.14</code>

A cast as expression can also deal with an empty sequence, and supports the zero-or-more cardinality in the expected resulting type. But it will throw an error if the sequence has more than one item: you need to use a FLWOR expression if you want to cast every item in a sequence.

Query	<code>[1, 2, 3, 4] [\$\$ > 4] cast as string?</code>
Result	

A castable as expression tests whether a cast would succeed (in which case it returns true) or not (false).

Query	<code>"3.14" castable as decimal</code>
Result	<code>true</code>

A treat as expression checks whether its input sequence matches an expected type (like a type on a variable); if it does, the input sequence is returned unchanged. If not, an error is raised. This is useful in complex queries and for debugging purposes.

Query	<code>[1, 2, 3, 4] [] treat as integer+</code>
Result	<code>1</code> <code>2</code> <code>3</code> <code>4</code>

There are also typeswitch expressions. The expression inside the typeswitch is evaluated. Then, the resulting sequence is type-matched with each one of the sequence types. The result of the expression

corresponding to the first match is taken, and if there are no matches, the result of the default expression is taken.

Query	<pre>typeswitch(json-file("file.json").o[[1]].a.b[[1]].c) case integer+ return "integer" case string return "string" default return "other"</pre>
Result	integer

12.10.3 Types in user-defined functions

JSONiq supports user-defined functions. Parameter types can be optionally specified, and a return type can also be optionally specified.

Query	<pre>declare function is-big-data(\$threshold as integer, \$objects as object*) as boolean { count(\$objects) gt \$threshold }; is-big-data(1000, json-file("git-archive.json"))</pre>
Result	true

But also:

Query	<pre>declare function is-big-data(\$threshold, \$objects) { count(\$objects) gt \$threshold }; is-big-data(1000, json-file("git-archive.json"))</pre>
Result	true

12.10.4 Validating against a schema

It is possible to declare a schema, associating it with a user-defined type, and to validate a sequence of items against this user-defined type.

Query	<pre> declare type local:histogram as { "commits" : "short", "count" : "long" }; validate type local:histogram* { for \$event in json-file("git-archive-big.json") group by \$nb-commits := (size(\$event.payload.commits), 0)[1] order by \$nb-commits return { "commits" : \$nb-commits, "count" : count(\$event) } } </pre>
Result	<pre> { "commits" : 0, "count" : 94554 } { "commits" : 1, "count" : 92094 } { "commits" : 2, "count" : 9951 } { "commits" : 3, "count" : 3211 } { "commits" : 4, "count" : 1525 } { "commits" : 5, "count" : 877 } { "commits" : 6, "count" : 688 } { "commits" : 7, "count" : 426 } { "commits" : 8, "count" : 383 } { "commits" : 9, "count" : 259 } { "commits" : 10, "count" : 274 } { "commits" : 11, "count" : 193 } { "commits" : 12, "count" : 146 } </pre>

If the results of a JSONiq query have been validated against a JSound schema, under specific conditions (the same covered in Chapter 7 for a schema to be DataFrame compatible), then it is possible to save the output of the query in other formats than JSON, such as Parquet, Avro, or (if there is no nestedness) CSV.

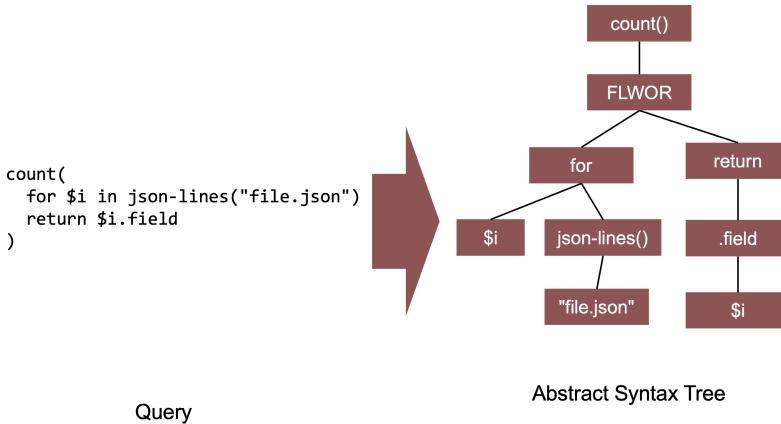
12.11 Architecture of a query engine

We now cover the physical architecture and implementation of a query engine such as RumbleDB.

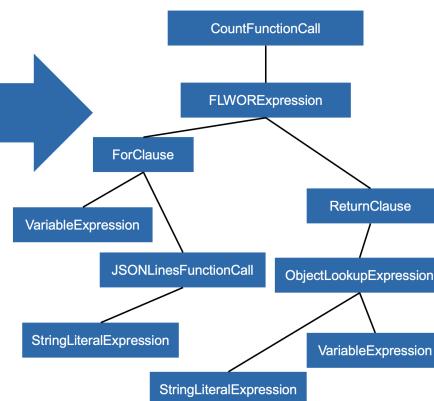
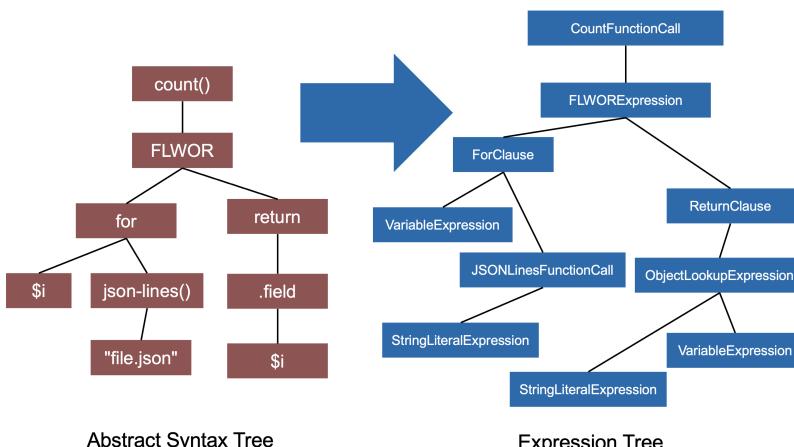
12.11.1 Static phase

When a query is received by an engine, it is text that needs to be parsed. The theory and techniques for doing this (context-free gram-

mars, EBNF...) are covered in compiler design courses. The output of this is a tree structure called an Abstract Syntax Tree.



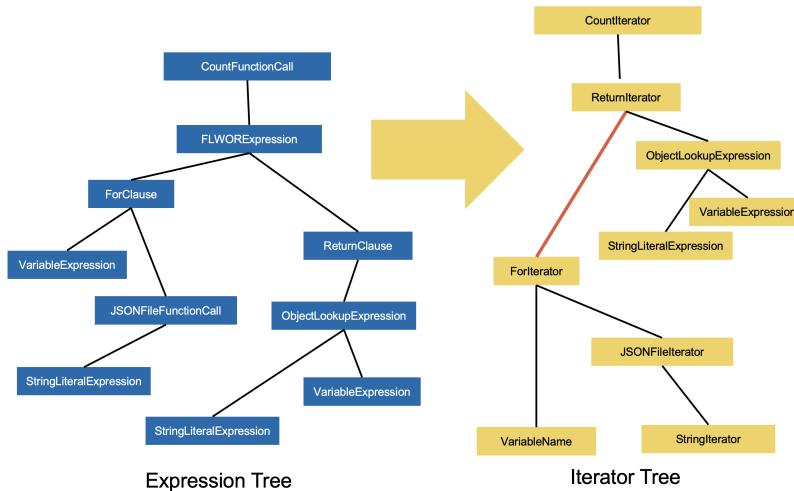
An Abstract Syntax Tree, even though it already has the structure of a tree, is tightly tied to the original syntax. Thus, it needs to be converted into a more abstract Intermediate Representation called an expression tree. Every node in this tree corresponds to either an expression or a clause in the JSONiq language, making the design modular.



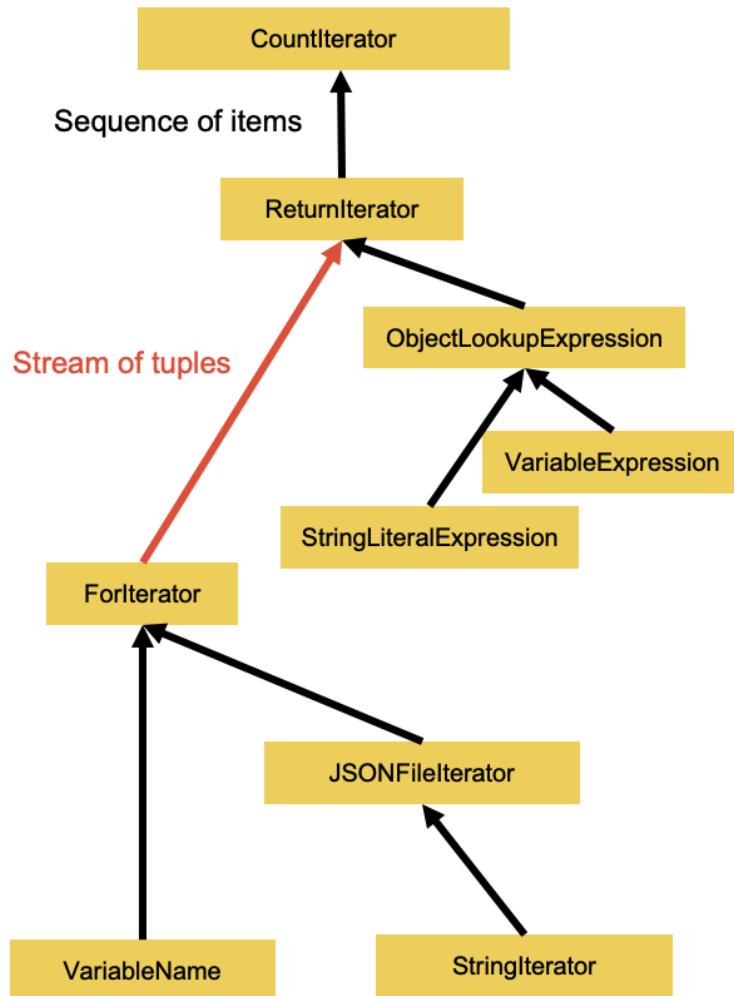
At this point, static typing takes place, meaning that the engine infers the static type of each expression, that is, the most specific type possible expected at runtime (but without actually running the program). User-specified types are also taken into account for this step. Inferring static types facilitates the optimization step.

Engines like RumbleDB perform their optimization round on this Intermediate Representation. Optimizations consist in changing the tree to another one that will evaluate faster, but without changing the semantics of the query (i.e., it should produce the same output). An example is that, if RumbleDB detects that both sides to a general comparison are single items, then the comparison is rewritten as a more efficient value comparison. Another example is that user-defined function calls are “inlined”, meaning that the body of the function is copied over instead of the function call, as if the user had written it manually there.

Once optimizations have been done, RumbleDB decides the mode with which each expression and clause will be evaluated (locally, sequentially, in parallel, in DataFrames, etc). The resulting expression tree is then converted to a runtime iterator tree; this is the query plan that will actually be evaluated by the engine.



Every node in a runtime iterator tree outputs either a sequence of items (if it corresponds to an expression) or a tuple stream (if it corresponds to a clause other than the return clause).



12.11.2 Dynamic phase

During the dynamic phase, the root of the tree is asked to produce a sequence of items, which is to be the final output of the query as a whole.

Then, recursively, each node in the tree will ask its children to produce sequences of items (or tuple streams). Each node then combines the sequences of items (or tuple streams) it receives from its children in

order to produce its own sequence of items according to its semantics, and pass it to its parent. That way, the data flows all the way from the bottom of the tree to its root, and the final results are obtained and presented to the user or written to persistent storage (drive or data lake).

There are many different ways for a runtime iterator to produce an output sequence of items (or tuple stream) and pass it to its parent runtime iterator in the tree:

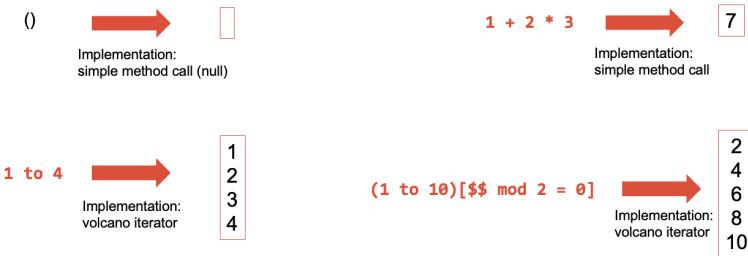
- By materializing sequences of items (or tuple streams) completely in local computer memory.
- By locally iterating over each item in a sequence, one after the other (or over each tuple in a tuple stream, one after the other).
- By working in parallel over the sequence of items, internally stored as a Spark RDD.
- By working in parallel over the sequence of items (or tuple stream), internally stored as a Spark DataFrame.
- By natively converting the semantics of the iterator to native Spark SQL.

Materialization

When a sequence of items is materialized, it means that an actual List (or Array, or Vector), native to the language of implementation (in this case Java) is stored in local memory, filled with the items. This is, of course, only possible if the sequence is small enough that it fits.

The parent runtime iterator then directly processes this List in place, in order to produce its output.

A special case is when an expression is statically known to return either zero or one item (e.g., an addition, or a logical expression), but not more. Then no List structure is needed, and a single Item can be returned via a simple method call in the language of implementation (Java).



Streaming

With larger sequences of items, it becomes impracticable to materialize because the footprint in memory becomes too large, and the size of the sequences that can be manipulated is strictly limited by the total memory available.

Thus, another technique is used instead: streaming. When a sequence of items (or tuple stream) is produced and consumed in a streaming fashion, it means that the items (or tuples) are produced and consumed one by one, iteratively. But the whole sequence of items (or tuple stream) is never stored anywhere.

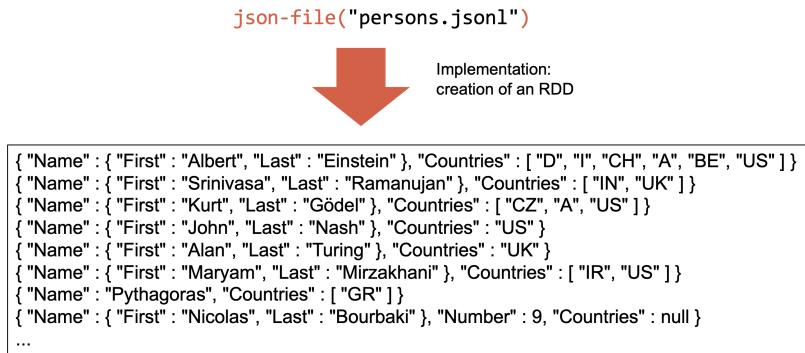
The classical pattern for doing so is known as the Volcano iterator architecture. It consists in first calling a method called `open()` to initialize the iterator, then `hasNext()` to check if there exists a next item (or tuple), and if so, then `next()` to consume it; and then `hasNext()` and `next()` are called again and repeatedly as long as `hasNext()` does not return false. When it finally does, `close()` is called to clean up the iterator.

With this technique, it is possible to process sequences that are much larger than memory, because the actual sequence is never fully stored. However, there are two problems with this: first, it can take a lot of time to go through the entire sequence (imagine doing so with billions or trillions of items). Second, there are expressions or clauses that are not compatible with streaming (consider, for example, the group by or order by clause, which cannot be implemented without materializing their full input).

Parallel execution (with RDDs)

When a sequence becomes unreasonably large, RumbleDB switches to a parallel execution, leveraging Spark capabilities: the sequences of items are passed and processed as RDDs of Item objects. Each runtime

iterator then calls Spark transformations on these RDDs to produce an output RDD, or in some cases (e.g., `count()`) calls a Spark action to produce a single, local, materialized Item with an action.



A Spark transformation or action often needs to be supplied with an additional function (e.g., a `map` function, a `filter` function), called a Spark UDF (for “User-Defined Function”). What RumbleDB then does is that it squeezes an entire runtime iterator subtree into a UDF, so that this subtree can be recursively evaluated on each node of the cluster, as a local execution (materialized or streaming).

For example, imagine a filter expression, with a specific predicate, on a sequence of a billion items. If the input sequence is physically available as an RDD, RumbleDB squeezes the predicate’s runtime iterator tree into a UDF, and invokes the `filter()` transformation with this UDF, resulting in a smaller RDD that contains the filtered sequence of items. Physically, the predicate’s runtime iterator tree will be evaluated on items, in parallel, across thousands of machines in the cluster; relative to each one of these machines, this is a local execution (local to each machine), where the predicate iterator streams over each batch.

```
json-file("persons.jsonl").Countries
```



Implementation:
transformation (flatMap) + UDF

```
[ "D", "I", "CH", "A", "BE", "US" ]  
[ "IN", "UK" ]  
[ "CZ", "A", "US" ]  
"US"  
"UK"  
[ "IR", "US" ]  
[ "GR" ]  
null  
...
```

```
json-file("persons.jsonl").Countries[]
```



Implementation:
transformation (flatMap) + UDF

```
"D"  
"I"  
"CH"  
"A"  
"BE"  
"US"  
"IN"  
"UK"  
"CZ"  
"A"  
"US"  
"IR"  
"US"  
"GR"  
...
```

The use of RDDs is specific to sequences of items and does not exist for tuple streams.

Parallel execution (with DataFrames)

The RDD implementation supports heterogeneous sequences by leveraging the polymorphism of Item objects. However, this is not efficient in the case that Items in the same sequence happen to have a regular structure.

Thus, if the Items in a sequence are valid against a specific schema, or even against an array type or an atomic type, the underlying physical storage in memory relies on Spark DataFrames instead of RDDs. Homogeneous sequences of arrays or of atomics (e.g., a sequence of integers) are physical implemented as a one-column DataFrame with the corresponding type.

Thus, there exists a mapping from JSONiq types to Spark SQL types. In the case that there is no corresponding Spark SQL type, the implementation falls back to RDDs.

`parquet-file("persons.parquet")`



Implementation:
creation of a DataFrame

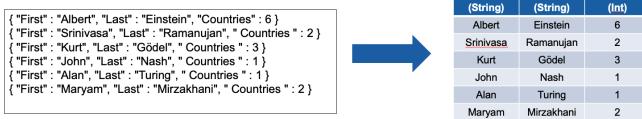
First (String)	Last (String)	Countries (Int)
Albert	Einstein	6
Srinivasa	Ramanujan	2
Kurt	Gödel	3
John	Nash	1
Alan	Turing	1
Maryam	Mirzakhani	2

To summarize, homogeneous sequences of the most common types are stored in DataFrames, and RDDs are used in all other cases.

Heterogeneous



Homogeneous

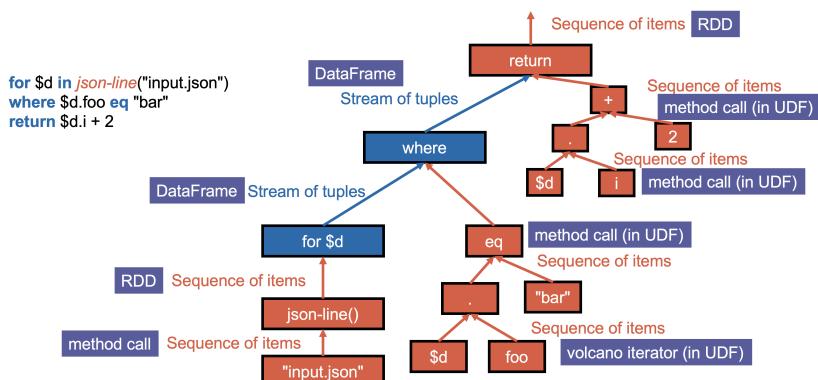


DataFrames are also consistently used for storing tuple streams and parallelizing the execution of FLWOR clauses. In FLWOR DataFrames, every column corresponds to one FLWOR variable, which is similar to the visuals provided earlier for FLWOR expressions in this chapter. The column type can either be native if the variable type can be mapped seamlessly to a Spark SQL type. Otherwise, the column type will be binary and Items are serialized to sequences of types and deserialized back on demand.

Parallel execution (with Native SQL)

In some cases (more in every release), RumbleDB is able to evaluate the query using only Spark SQL, compiling JSONiq to SQL directly instead of packing Java runtime iterators in UDFs. This leads to faster execution, because UDFs are slower than a native execution in SQL. This is because, to a SQL optimizer, UDFs are opaque and prevent automatic optimizations.

RumbleDB switches seamless between all execution modes, even within the same query, as shown on the following diagram.



12.12 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain why a language such as JSONiq provides, in the context of denormalized data, a similar functionality as SQL in a relational database? Do you understand how it generalizes querying to nested, heterogenous data models, and is thus more powerful than SQL?
- b. Can you name and describe the first-class citizen of the JSONiq Data Model: a sequences of item?
- c. Can you name various kinds of items in the JDM?
- d. Can you name a few query languages in the XML/JSON ecosystem?
- e. Do you understand how to navigate nested structures in JSONiq (object lookup, array lookup, array unboxing, filtering predicates)?
- f. Are you able, in JSONiq, to construct items (atomic items, elements, etc.)?
- g. Are you able, in JSONiq, to perform logical operations? Do you understand what the Effective Boolean Value of a sequence is, and how it fits in the context of logical operations?
- h. Are you able, in JSONiq, to perform arithmetic operations (addition, etc.)? Do you understand the constraints on the input sequences of such operations? Can you explain the behavior of these operations on empty sequences? Can you explain what happens if one of the two operands is a node and not an atomic item?
- i. Are you able, in JSONiq, to perform comparisons (lt, ge, etc.)? Do you understand the constraints on the input sequences of such operations? Can you explain the behavior of these operations on empty sequences? Can you explain what happens if one of the two operands is a node and not an atomic item?
- j. Do you understand how general comparisons ($<$, \geq , etc.) work on sequences with more than one item, and implicitly use an existential quantifier? Can you explain why, in the special case of single items on both sides, they are equivalent to value comparisons?
- k. Do you understand how FLWOR expressions work and describe what they return? (for clause, let clause, where clause, order by clause, etc.)

1. Are you able to use further expressions (if-then-else, switch, ...)?
- m. Do you understand how to dynamically build JSON content with object and array constructors?
- n. Do you understand that expressions can be combined at will, as any expression takes and returns sequences of items? Do you know how to use parentheses to make precedence clear, like you did in primary school with addition and multiplication?
- o. Do you know the JSONiq type syntax (atomic types taken from XML Schema, syntax for XML node types, as well as cardinality symbols), and how to use type checking (instance of, cast as, etc.)?
- p. Given a collection of JSON objects (for example JSON Lines on HDFS), are you able to write JSONiq queries (FLWOR) that do projection? selection? grouping? ordering? joins? sorting?
- q. Do you understand how, given a SQL query, you can write something equivalent in JSONiq?
- r. Do you understand that this is not true the other way round (rewriting JSONiq as an equivalent SQL query)? Can you characterize examples of when this is not true or very difficult (hint: denormalized data)?

12.13 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Müller, I., Fourny, G., Irimescu, S., Cikis, C., Alonso, G. (2021). *Rumble: Data Independence for Large, Messy Data Sets*. In: PVLDB 14(4). 10-minute presentation of RumbleDB (above paper) at VLDB 2021.

<https://www.youtube.com/watch?v=q3IxXnYZ8UM>

JSONiq language reference, and online sandbox

<https://www.jsoniq.org/>

RumbleDB engine, free and open source.

<https://www.rumbledb.org/>

Graur, D., Müller, I., Proffitt, M., Fourny, G., Watts, G., Alonso, G. *Evaluating Query Languages and Systems for High-Energy Physics Data*. Joint interdisciplinary work ETH Zurich - University of Washington. In: PVLDB 15(2).

Chapter 13

Graph databases

Most of the textbook so far was dedicated to storing and querying data in the shape of tables or trees, collections of trees being a denormalized form of tables.

We now turn to data in the shape of graphs.

13.1 Why graphs

Relational tables are excellent at querying highly structured, normalized datasets. Normal form avoid redundancies and anomalies by spreading the dataset over multiple tables, with each table storing “one thing” (customers, products, etc). These tables are related to each other by means of primary keys, and foreign keys pointing to them, which is in particular apparent in the entity-relationship model. At runtime, this translates into the use of joining queries.

Even though a single join can be highly optimized: with hash tables, it is even possible to do it in linear time, and in fact primary keys are commonly indexed (hash indices, etc). But this does not scale well, and there are query patterns that bring the relational algebra to its limits (traversal of a high number of relationships, reverse traversal requiring also indexing foreign keys, looking for patterns in the relationships, etc).

Now, we do know a way to avoid joins and studied it at length: denormalizing the data to (homogeneous) collections of trees is a way of “pre-computing” the joins statically, so that the data is already joined (via nesting) at runtime.

Now, why is it efficient? Because doing down a tree only necessitates following pointers in memory. But trees cannot have cycles. Graph databases provide a way of generalizing the use in-memory pointers to traverse data to the general case in which cycles are present: this is called “index-free adjacency.”

13.2 Kinds of graph databases

There are many different graph database system products on the market, and they can be classified along several dimensions.

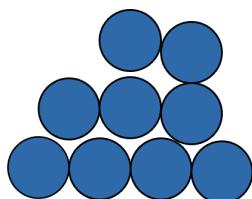
- Labeled property graph model vs. triple stores: there are two competing high-level data models for graphs. We will study them both in this chapter.
- Read-intensive vs. write-intensive: some graph database systems are more suited for large-scale analytics (read-intensive, OLAP), while others support efficient updates (OLTP).
- Local vs. distributed: some graph database systems work on a single machine, some others replicate the (same) data across multiple machines, and recent developments even allowed graph sharding, meaning that the data and the queries are partitioned and distributed over several machines.
- Native vs. non-native: some graph database systems are in fact a layer on top of another kind of database system in a different shape (for example, a relational database or a document store) or query engine (for example, Apache Spark). Other graph database systems are directly implemented from scratch to efficiently support graphs.

In fact, these dimensions are not specific to graph database systems, and also apply to other shapes (except, of course, that the competing data models will be other models than those two).

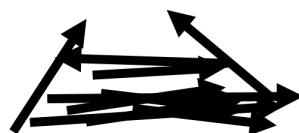
13.3 Graph data models

13.3.1 Labeled property graphs

The basic ingredients of a graph, from a mathematical perspective, are nodes and edges.

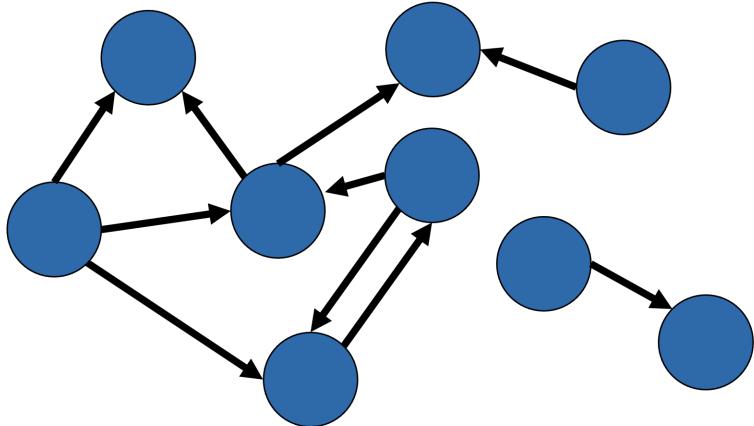


Nodes



Edges

Mathematically, a directed graph $G = (N, E)$ is given by a set N of nodes (also called vertices), and a set $E \subseteq N \times N$ of directed edges (also called arcs, or arrows). Given these, it is possible to visually represent the graph as follows (note that in this case, the graph is planar in the sense that edges do not cross, but this does not hold in general):

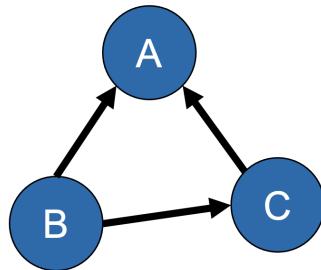


Graphs can also be undirected, in which case the arrows do not have a direction. Then formally $E \subseteq \{P \in \mathcal{P}(N) \mid \text{card}(P) = 2\}$. An alternative modeling is to keep $E \subseteq N \times N$, but with the additional constraint that all edges go in both directions:

$$\forall n, m \in P, (n, m) \in E \Rightarrow (m, n) \in E$$

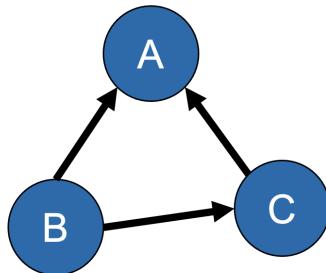
Such is the math at the logical level, and it is already powerful enough to overfill happy Bachelor's students with joy over many years of algorithms and problem solving (graph coloring, constraints on planar graphs, max-flow/min-cut, Dijkstra's algorithm, etc). In fact, one of the benefits of graph database systems (or at least some of them) is that they are well suited for implementing and running such algorithms.

Computer scientists need to go one step further and also design how to store graphs physically. One way of doing so is to create an associative array mapping each node to the list of nodes that it connects to via an edge (adjacency lists):



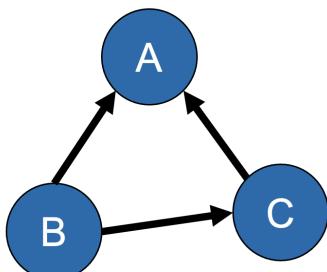
Node	Edges
A	[]
B	[A, C]
C	[A]

Another storage form is with an adjacency matrix: each row and each column represent a node, and a 0 or a 1 indicate the absence or presence of an edge between the row node and the column node:



	A	B	C
A	0	1	1
B	0	0	0
C	0	1	0

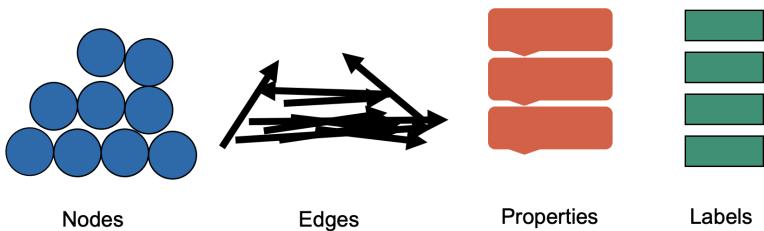
Yet another way is with nodes on the rows and edges on the columns, with a -1 (origin node) and 1 (destination node) in each column and 0s otherwise:



Nodes	Edges		
	1	2	3
A	1	1	0
B	-1	0	-1
C	0	-1	1

The latter way generalizes even to so-called hypergraphs, where each edge can have more than one origin node and more than one destination node.

Now, this does not quite work for us, because labeled property graphs enhance mathematical graphs with extra ingredients: properties, and labels:

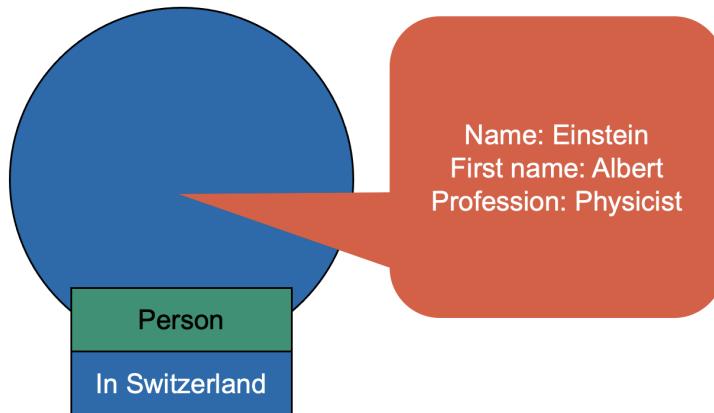


Labels are some sort of “tags”, in the form of a string, that can be attached to a node or an edge. For example, a node may be tagged as a “Person” and another as a “Course”, and the edge between them can be tagged as “Attends”. A small graph with these nodes and edge indicates that a person is attending a course.

Labels can be repeated, for example we could have a graph with 23,000 Person nodes and 5,000 Course nodes, and 50'000 Attends edges.

Now, how would we then store information about the persons and courses? Labels are surely not an option, as it would be make it cumbersome to know which label is what. Instead, we use properties. Each node and each edge can be associated with a map from strings to values, which represents its properties. You can think of the properties associated with each node or edge as a JSON object, or as a single record in a relational table or in a DataFrame (if nested).

This is a visual of a single node with some labels and its properties:



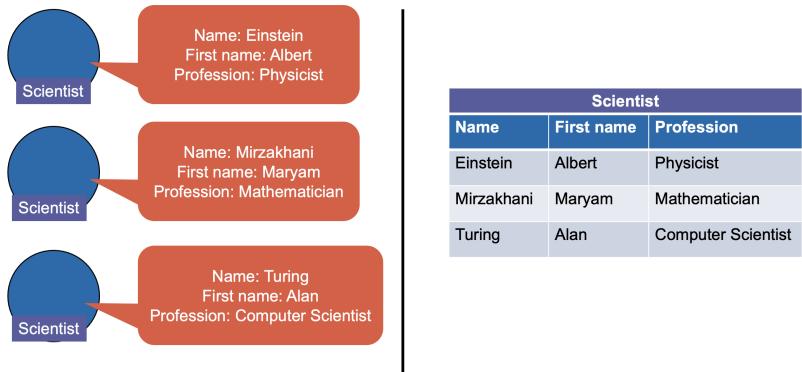
In the case of neo4j, many property types are supported and we list a few here together with their JSound/JSONiq/XML Schema equivalent:

Neo4j type	JSONiq/JSound/XML Schema type
STRING	string
BOOLEAN	boolean
INTEGER	long
FLOAT	double
NULL	null
DATE	date
TIME	time
DATETIME	dateTime
ZONED DATETIME	dateTimeStamp
ZONED TIME	time (time zone mandatory)
LOCAL DATETIME	dateTime (no time zone)
LOCAL TIME	time (no time zone)
DURATION	duration
LIST	array

Neo4j supports also an additional type (POINT) for geographic locations (GIS), which does not have a JSound/JSONiq equivalent. We will see that objects can appear in Neo4j's query language (the type is called MAP) but objects cannot be used as the properties of a node or edge: the only way to “break the first normal form” in properties is via arrays.

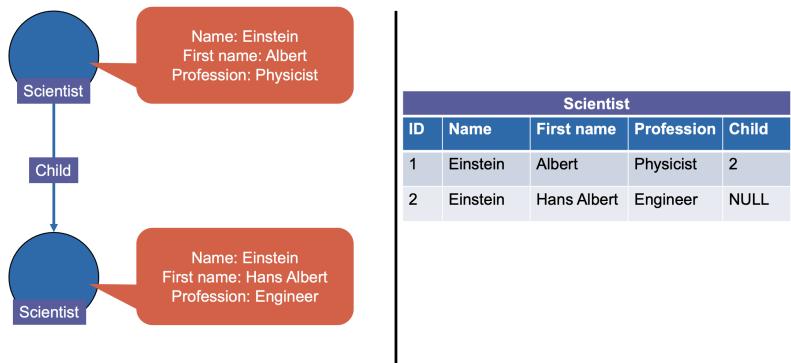
Maybe the reader, at this point, starts seeing some pattern, or might have even already figured out how to “convert” a relational table to a

labeled property graph: labels can be seen as table names, nodes as records, and properties as the attribute values for the records:



This shows that relational tables can be physically stored as labeled property graphs. Of course, this does not work the other way round: given a graph, it will often not be possible to convert it “back” to a table in this specific way.

If there are foreign keys pointing to primary keys, these can be represented with edges in the graph, like so:

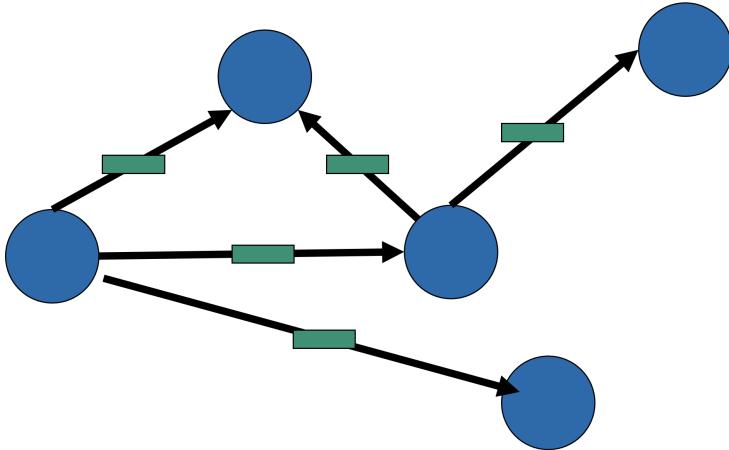


Here ID is the primary key and Child the foreign key, and the table points to itself to keep the example concise.

In the presence of multiple tables, the corresponding nodes are just “merged” into a bigger graph (no need to create any partitions, as the labels do that job), and edges are added for foreign keys from some table pointing to primary keys of some other table.

13.3.2 Triple stores

Triple stores are a different and simpler model. It views the graph as nodes and edges that all have labels, but without any properties.



The graph is then represented as a list of edges, where each edge is a triple with the label of the origin node (called the *subject*), the label of the edge (called the *property*), and the label of the destination node (called the *object*).

Labels can be:

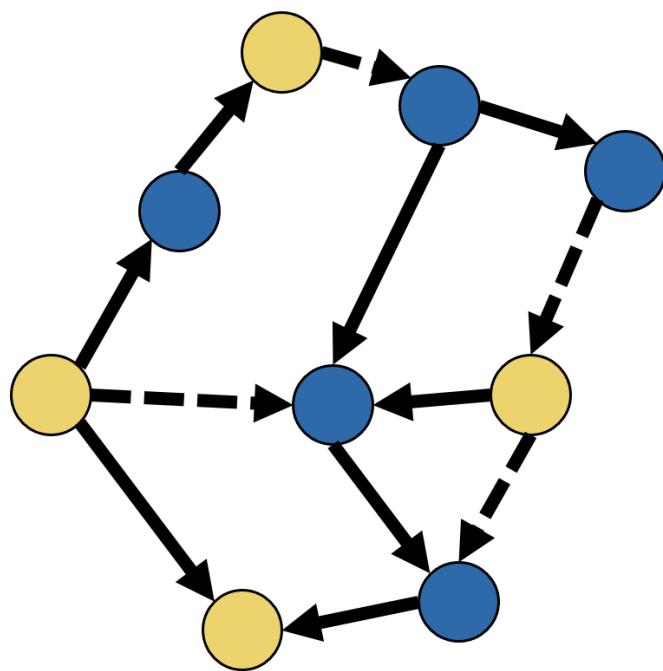
- URIs, which look just like the namespaces we have seen before in XML and like URLs in REST APIs.
- Literals, that is, atomic values: strings, integers, dates, etc (but not structured values such as objects or arrays). Literals are only allowed as objects.
- Absent, in which case the node is called a blank node. Blank nodes are only allowed as subjects or objects, but not as properties.

13.4 Querying graph data

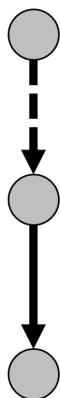
We will now have a look at query languages for querying graphs, with a focus on Cypher, which is neo4j's query language. Other languages include PGQL, SPARQL and Gremlin.

At the core of Cypher (and some of the other languages for graph data) lies graph patterns.

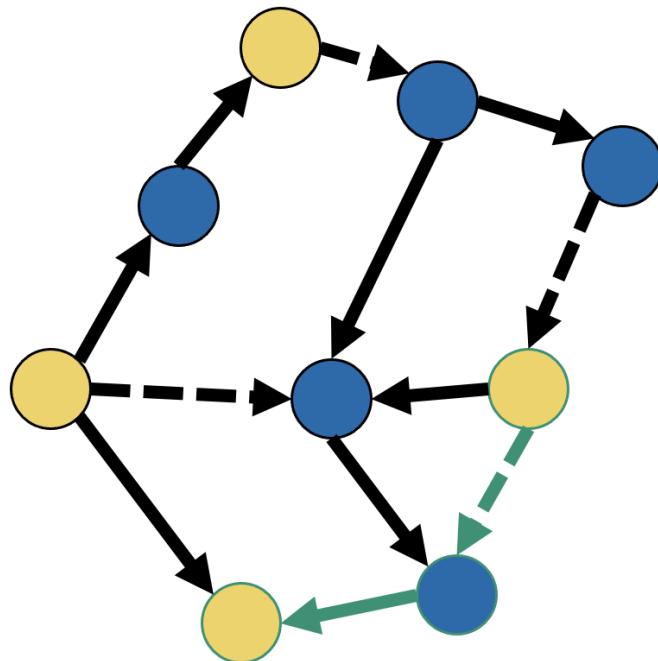
For example, let us consider the following graph:



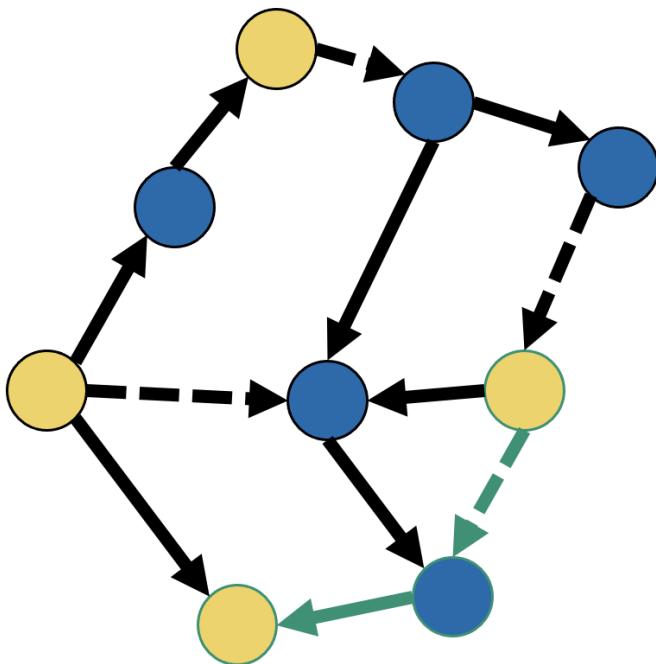
and the following pattern that we are looking for (a dashed edge followed by a full edge):



This is one instance of this pattern's being recognized in the previous graph, which is called a *match*:



and this is another match (and there are several further matches, which we leave for the reader to find):



Now, let us look at the syntax to express a pattern. Of course, edge labels are not dashes and full lines, but have a string names, let us assume A for a dashed edge and B for a solid-line edge. Then, the previous pattern can be expressed with this nice ASCII art:

`()-[:A]->()-[:B]->()`

where the label of an edge is given after the colon character inside square brackets, and nodes are denoted with regular parentheses.

Patterns expressed in this way are used in a MATCH clause, like so:

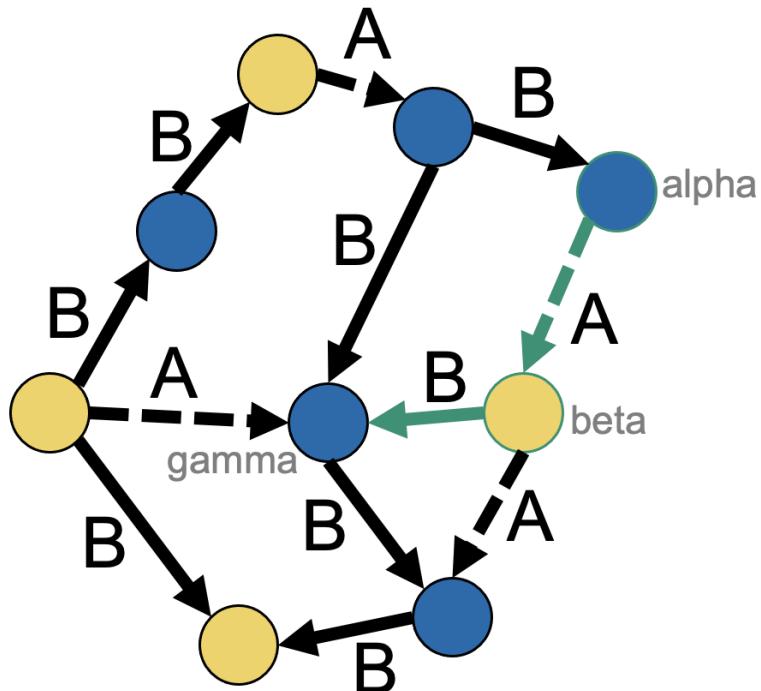
```

MATCH (alpha)-[:A]->(beta)-[:B]->(gamma)
RETURN alpha, beta, gamma
  
```

Inside the parentheses, we inserted variable names, which can be referred to in the RETURN clause. For each match, a binding for the three variables alpha, beta, and gamma is created, and the RETURN clause is evaluated for each one of this bindings (does it ring a bell with

JSONNiq's FLWOR expressions? This is indeed the same logic with the tuple streams as we will shortly see).

One example of binding is visually given here:



The query above returns an output with three columns (`alpha`, `beta` and `gamma`) which are of type “node”. In this respect, the syntax of the `RETURN` clause is closer to that of the `SELECT` clause in SQL than the return clause in JSONNiq.

The pattern can also filter nodes based on their labels, for example with the query below, the variable `beta` will only be bound to nodes that have the label “yellow” (leading to less matches):

```

MATCH (alpha)-[:A]->(beta:yellow)-[:B]->(gamma)
RETURN alpha, beta, gamma
  
```

It is also possible to filter on node or edge properties (in a way that will probably remind the reader of MongoDB selections in the JavaScript API):

```

MATCH (alpha)-[:A]->(beta { name: 'Einstein' })-[:B]->(gamma)
RETURN alpha, beta, gamma
  
```

It is also possible to filter on both a label and a property, like so:

```
MATCH (alpha)-[:A]->(beta)-[:B]->(gamma: blue { name : 'ETH'})  
RETURN alpha, beta, gamma
```

Reverse edges are also possible in the pattern syntax:

```
MATCH (alpha)-[:A]->(beta)-[:B]->(gamma)<-[:B]-(delta)  
RETURN alpha, beta, gamma, delta
```

It is also possible to reuse a variable in the pattern, for example to look for a cycle:

```
MATCH (alpha)-[:A]->(beta)-[:B]->(gamma)<-[:B]-(alpha)  
RETURN alpha, beta, gamma
```

Finally, it is also possible to match longer paths within a specified length interval:

```
MATCH (alpha)-[*1..4]->(beta)<-[:B]-(alpha)  
RETURN alpha, beta
```

13.5 Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you explain why joins in relational databases are slow, especially in the context of traversal?
- b. Can you explain why, while denormalizing the data partly addresses the issue of slow joins, this is still not satisfactory?
- c. Can you explain why reverse traversals are also tricky to support in a relational database?
- d. Can you explain what index-free adjacency is and why it solves the above problems?
- e. Can you define a graph mathematically? What about a directed/undirected graph?
- f. Can you give at least three examples of ways that graphs can be represented in memory?
- g. Can you explain what a labeled property graph is and what it is made of?
- h. Can you explain the difference between graph databases such as neo4j on the one hand, and RDF triple stores on the other hand?
- i. Can you explain the RDF data model (triples, subject, property, object, IRI, literal, ...)
- j. Can you give the usual constraints on RDF graphs, and explain how they can be lifted into generalized graphs?
- k. Can you sketch and parse the RDF/XML syntax of RDF?
- l. Can you sketch and parse the JSON-LD syntax of RDF?
- m. Can you sketch and parse the Turtle syntax of RDF?
- n. Can you explain how query-by-example is different from a classical query language?
- o. Can you name a few graph querying languages and relate them to products such as neo4j or RDF triple stores?
- p. Can you read and compose Cypher patterns, and explain how they work? Do you know how to use the main clauses (MATCH, CREATE, WHERE, ...)

- q. Can you read and create SPARQL queries, and explain how they work? Can you relate this syntax to SQL?
- r. Can you sketch the physical architecture of neo4j?
- s. Can you explain how properties and relationships are physical stored in neo4j?
- t. Can you explain why sharding graphs is a hard problem (even though it is now largely solved)?
- u. Can you explain why OWL is and briefly explain how it can leverage RDF to build ontologies and semantic reasonings (semantic Web)?
- v. Can you write simple Cypher queries?

13.6 Literature and recommended readings

The following is a list of recommended material for further reading and study.

Robinson, I. et al. (2015). *Graph Databases*. 2nd edition. O'Reilly. Chapters 1, 2, 3, 4, and 6.

Hong, S. et al. (2015). *PGX.D: a fast distributed graph processing engine*. In SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.

Neo4j Cypher Manual.

<https://neo4j.com/docs/cypher-manual/current/introduction/>

Chapter 14

Acknowledgements

The course could not have taken place without all the Teaching Assistants, who are doctoral students or Master students who worked very hard on preparing exercises and teaching exercise sessions, and on moderating the lectures. It could also not have taken place without the (now thousands of) students who took the course and participated actively, asking many smart questions and contributing valuable feedback.

Many thanks also to the people who gave direct feedback on this textbook, in particular Lazar Cvetković, Tristan Girard, Haobo Li, Abishek Ramdas, Lam Nguyen Thiet, Anna Smolarz, Matt Weingarten, Jakub Lucki, Javier Rando Ramirez, Manuel de Prada Corral, Francesc Martí Escofet, Matthias Otth, Nauryzbay Koishkenov, Dhruv Agrawal, Marcin Copik, Athina Xenou Gavrieli, Eren Cetin, Aishwarya Melatur, Amalie Hop, Florian van Dellen, Thomas Landeg, Michael Siebenmann, Mak Hei Yi, Benjamin Horner, Frederike Lohmann, Max Striebel, Oleh Kuzyk, James Wei, Nahin Khan, Hubert Dej, Lorenzo Asquini, Roman Fischer, Jingyuan Li, Güney Beste, Andrii Isakov, Katarína Osvaldová, Joel Fischer, Timur Eke, Andreas Spanopoulos, Hongyi Lan.

Chapter 15

Latest updates

These are the updates that were made after the first publication of the printed copy of the book.

- Fixed the examples and counterexamples for the first three normal forms to remove overlooked duplicates.
- Clarified in the HDFS chapter that the first blocks of a file being written might be visible to other clients before the write has been completed, but with no guarantee.
- Fixed incorrect LATERAL VIEW queries in Spark SQL by moving the FROM clause higher
- Expanded Chapter 12 on JSONiq with FLWOR coverage
- Expanded SQL query in Chapter 2 to also have an ORDER BY, LIMIT and OFFSET clause.
- Added learning objects as well as literature to all chapters.
- (August 23) In Chapter 7, the property [owner element] of attribute information items was incorrectly called [parent].
- (August 23) In Chapter 6, there is one HLog per RegionServer. It was incorrectly saying one per store.
- (August 23) In Chapter 7, pages 196/197, added missing foo element declarations in two schemas assigning the complex type.
- (August 23) In Chapter 7, page 178, the explanation of duration literals (e.g., 1 year and 3 months) did not match the actual literals (e.g., P2Y3M).
- (August 23) In Chapter 5, page 200, “like” changed to “unlike”.

- (September 28) Added content on key-value stores (Dynamo) in Chapter 2.
- (October 4) On page 74, eventual consistency in Dynamo was incorrectly labeled CA instead of AP. This is now fixed.
- (October 5) On page 185, non-positive integers were incorrectly given as equivalent to unsigned integers, but it should be non-negative integers.
- (October 5) On page 120, removed “(we will get to it quite shortly)”.
- (October 9) On page 74, another sentence read CA instead of AP, this is also fixed.
- (October 10) A few nonmaterial typos were fixed in Chapters 3 and 4, pages 66/80/82/89 (four aspects → three aspects, Second → Third, send → sent, semi-colon →dot).
- (October 31) A typo (deducted → deduced) was fixed on page 157.
- (October 31) A typo (on this chapter → in this chapter) was fixed on page 118.
- (October 31) A repeated sentence was shortened on page 120.
- (October 31) A typo (much complex → more complex) was fixed on page 121.
- (October 31) The year (2022 → 2023) was updated on page 126.
- (October 31) A typo (of not → if not) was fixed on page 58.
- (November 2) A typo (store → stores) was fixed on page 86.
- (November 2) An artifact was removed from the image on page 153.
- (November 2) “Such as the picture above” was confusing and was removed, and the sentence was reformulated on page 153.
- (November 2) A typo (consists in → consists of) was fixed on page 156.
- (November 2) A typo (wid → wide) was fixed on page 171.
- (November 13) A typo (if needs be → if need be) was fixed on page 273.

- (November 13) A redundant paragraph was shortened on page 276.
- (November 13) The correct image for the count action now appears on page 286.
- (November 13) The chapter on Graph Databases was started. It shifts the numbers of the last chapters.
- (November 20) A typo (stores → store) was fixed on page 323.
- (November 20) A typo (shall → shell) was fixed on page 321.
- (November 20) A typo (a top-level keys → top-level keys) was fixed on page 328.
- (November 20) A typo (structured → structure) was fixed on page 403.
- (November 20) A section header (7.8 Indices) was missing on page 332 and was now added. This renames the subsections.
- (November 20) the picture on page 334 has an easier-to-read yellow color.
- (November 20) The two vertical bars for string concatenation were fixed in the query on page 368.
- (November 20) An extraneous] was removed on page 371.
- (November 20) A missing } was added on page 379.
- (December 8) The results of two queries were corrected on pages 386 and 388.
- (December 8) The results of a query on page 351 is 3, not true.
- (December 8) On page 369, the value comparison for greater-or-equal is ge, not gt.
- (December 11) On page 384, \$product.country should be \$store.country.
- (December 11) On page 387, some line formatting at the start of a new paragraph was fixed.
- (December 11) On page 388, the query was rewritten to include a group by, which was the original intent. It now returns the expected results.
- (December 11) On page 347, cosmetic improvement putting (instead) in parentheses.

- (December 11) On page 359 and 360, separated paragraph into two sentences for improved readability.
- (December 11) On page 362, typo fixed (access with a function → accessed with a function).
- (December 11) On page 362, dot replaced with a comma.
- (December 11) On page 362, or one → of one.
- (December 11) On page 362, is thrown is → is thrown if.
- Various cosmetic improvements.
- (December 11) On page 373, removed extraneous space.
- (December 11) On page 388, its values of within → its values within.
- (December 11) On page 390, removed extra character at the end of the query.
- (December 11) On page 392, swapped the two examples (*integer* and *boolean?*).
- (December 11) Extended Chapter 13 on graph databases.
- (January 3) Fine-tuned the wording (KeyValue, cell, cell value) in Chapter 6 on wide column stores to clarify that cells are versioned, that KeyValues correspond to cell values, and that the same cell can thus contain KeyValues corresponding to different versions of it.
- (January 15) Fixed “in order words” typo to “in other words” in Chapter 3.
- (January 15) Fixed “four important properties” matho to “three important properties” on page 179.

