Complementos de Bases de Dados

UA.DETI.CBD

Carlos Costa

Unidade Curricular CBD

- Área científica
  - Sistemas de Informação
- Cursos:
  - Licenciatura em Engenharia Informática
- Escolaridade semanal:
  - 2 horas de aulas teórico-práticas; 2 horas de aulas práticas
- Créditos ECTS: 6
- Código: 40385

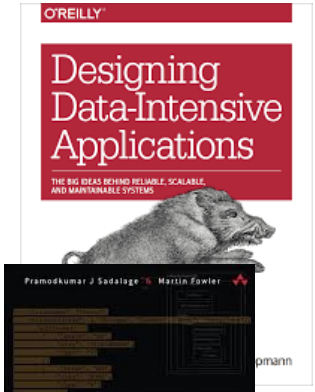
Objectivos

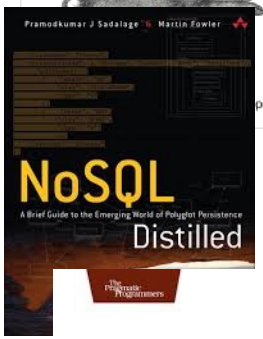
- **Compreender as técnicas** de modelação de dados orientadas a documentos, chave/valor, colunas e grafos, e selecionar soluções adequadas tendo em atenção o tipo de informação e os requisitos funcionais dos sistemas a desenvolver;
- **Desenhar modelos lógicos** e modelos físicos de dados para sistemas baseados em dados estruturados, semiestruturados e documentos;
- **Reconhecer e apresentar soluções** para problemas associados à gestão e ao processamento de dados distribuídos;
- **Desenvolver ferramentas e aplicações** eficientes para o processamento de grandes volumes de dados.

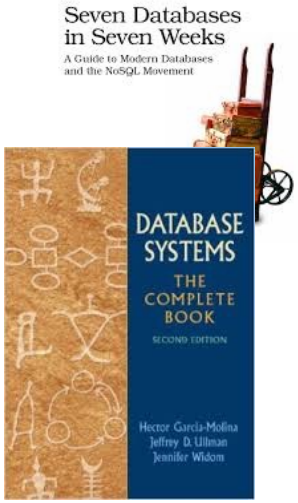
Programa resumido

- Dados e modelos de armazenamento
- Estruturas de dados e métodos de acesso
- Modelos de dados NoSQL
  - orientados a chave/valor, documentos, colunas e grafos
- Arquiteturas de bases de dados
  - distribuídas e paralelas
- Replicação e Partição
- Transações, consistência e integridade dos dados
- Processamento distribuídos de dados
  - modelo de programação MapReduce
  - plataformas tipo Hadoop

Bibliografia principal







- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.

Avaliação

- A avaliação da disciplina será discreta com as seguintes componentes:
  - (T) Avaliação Teórico-Prática Intercalar [**ATP1: 20%**]
    - Data: 31/10/2024
  - (T) Avaliação Contínua [**AC: 5%**]
    - Avaliação discreta nas aulas TP
  - (T) Avaliação Teórico-Prática Final [**ATP2: 25%**]
    - Data: 12/12/2024
  - (P) Avaliação Prática [**AP: 50%**]
    - Desempenho na realização dos trabalhos individuais
- A nota mínima para cada uma das componentes (T e P) é de 7 valores.

Avaliação (cont.)

- Não haverá registo de faltas nas aulas TP.
- Em regime ordinário, **as aulas práticas são de frequência obrigatória**.
  - O aluno que faltar a mais de 20% das práticas ficará automaticamente reprovado,
    - não podendo apresentar-se a qualquer exame da disciplina, durante o ano letivo em curso.
- Modelo de funcionamento das aulas práticas
  - Nas aulas terão de usar um **portátil pessoal** com o software necessário para cada módulo.
  - É importante a **assiduidade**, a **preparação** prévia, a discussão durante a aula, a **entrega** de todos os guiões.
  - **Cumprir o prazo** para submissão dos trabalhos.

ECTS

- Escolaridade (T/TP/P): 0/2/2 - ECTS: 6

- O número de créditos ECTS indica o número de horas espectável que devem estudar para esta disciplina.
  - 1 ECTS = 25-30 horas de estudo.
  - 6 ECTS = 150-180 horas de estudo.
- Num semestre com 15 semanas devem estudar pelo menos 10 horas por semana.
- Estas horas incluem: aulas presenciais, leitura de livros, resolução de exercícios, estudo para testes e exames, etc.

Recursos

- elearning.ua.pt
  - Slides TP
  - Guiões Práticos
  - Fóruns
  - Informações e resultados
  - Entregas dos trabalhos

- Links
  - Indicados em cada TP e guiões
  - … mas "*search yourself!!*"

Docentes e atendimento

- Carlos Costa – carlos.costa@ua.pt
- Tiago Godinho - tmgodinho@ua.pt

- Atendimento geral – IEETA, online

- As OTs funcionarão por marcação.
  - Por favor envie email para o docente até às 12h do dia anterior à OT que pretende agendar.

Bons estudos e bom semestre!

Database Systems Evolution

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Outline

- Why do we need storage system
- How they evolved along the time
- Milestone solutions
- Current landscape

Thinking about Data Systems

- Many applications today are **data-intensive**, as opposed to **compute-intensive**.
- Raw CPU power is rarely a limiting factor for these applications
  - bigger problems are usually the **amount** of data, the **complexity** of data, and the **speed** at which it is changing.

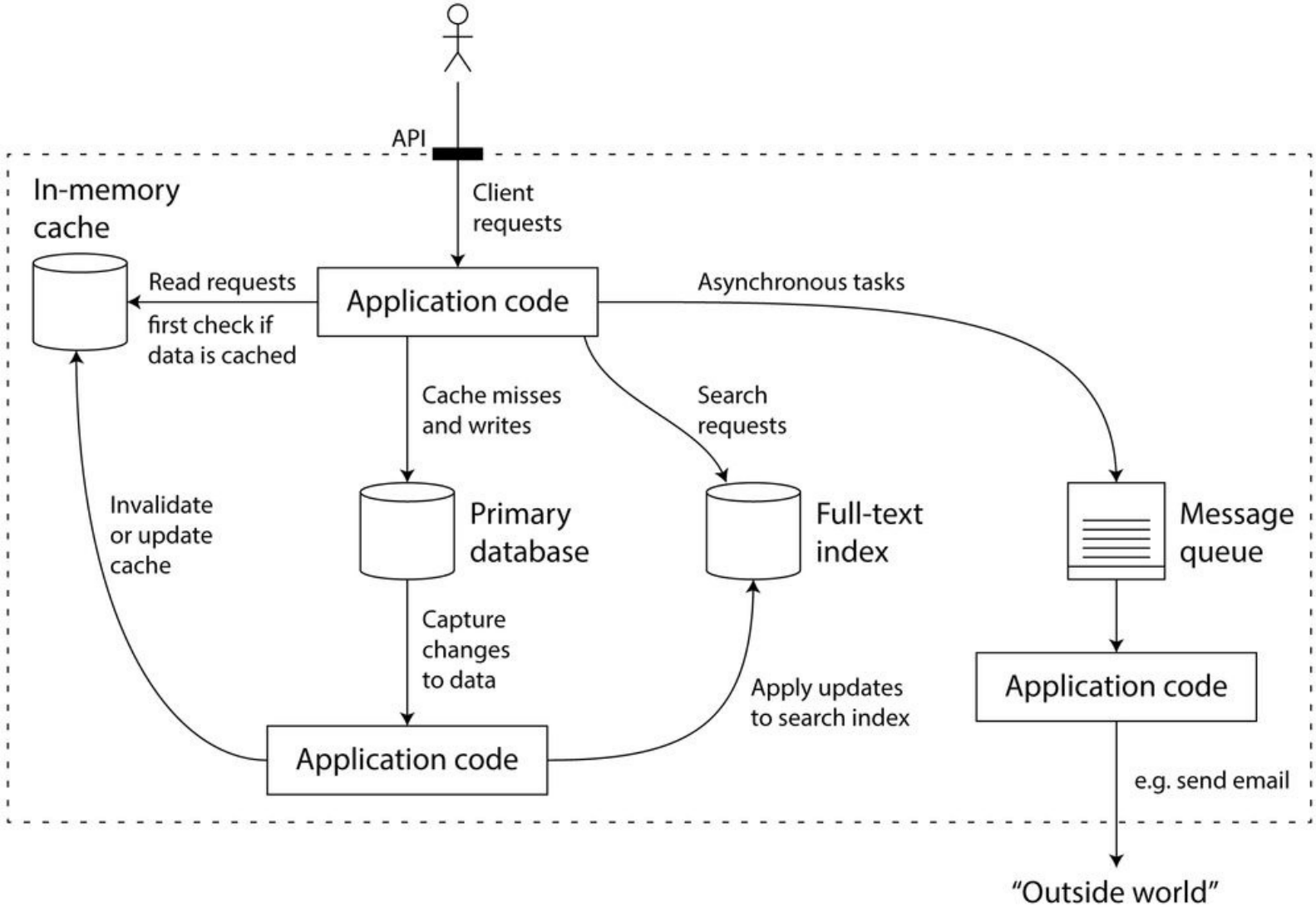
Data systems typically needs to

- Store data so that they, or another application, can find it again later (**databases**).
- Remember the result of an expensive operation, to speed up reads (**caches**).
- Allow users to search data by keyword or filter it in various ways (**search indexes**).
- Send a message to another process, to be handled asynchronously (**message queues**).
- Observe what is happening, and act on events as they occur (**stream processing**).
- Periodically crunch a large amount of accumulated data (**batch processing**).

Thinking about Data Systems

- Increasingly, many applications have wide-ranging requirements
  - Many times, a single tool can no longer meet all of its data processing and storage needs.
- Instead, the work is broken down into tasks that can be performed efficiently on a single tool,
  - the different tools are stitched together using application code.
- For example, we may have an application with:
  - a caching layer (e.g. memcached or similar),
  - a full-text search server (e.g. Elasticsearch or Solr),
  - separated from the main database (e.g. MySQL).

Thinking about Data Systems



Data Systems – some challenges

- How do you ensure that the data remains correct and complete,
  - even when things go wrong internally?
- How do you provide consistently good performance to clients,
  - even when parts of your system are degraded?
- How do you scale to handle an increase in load?
- What does a good API for the service look like?

Data Systems – some requirements

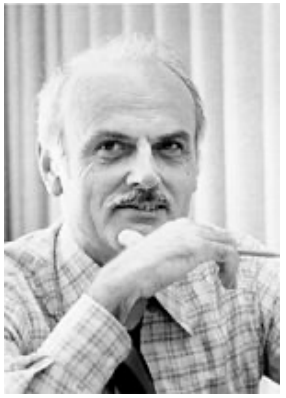
- **Reliability**: The system should continue performing the correct function at the desired performance,
  - even in the face of adversity (hardware or software faults, and even human error).
- **Scalability**: As the system grows (in data volume, traffic volume or complexity), there should be reasonable ways of dealing with that growth.
- **Maintainability**: Over time, many different people should all be able to work on it productively,
  - Engineering and operations, both maintaining current behavior and adapting the system to new use cases.

Database Systems

- A "database" is normally referred as a **set of related data** and its **organization**.
- A "database management system" (**DBMS**) controls the access to this data.
  - Providing functions that allow writing, searching, updating, retrieving, and removing large quantities of information.

Brief History of Database Systems

- Pre-relational era (1970's)
  - Hierarchical (IMS), Network (Codasyl)
  - Many database systems
    - Complex data structures and low-level query language
    - Incompatible, exposing many implementation details
- **Relational DBMSs** (1980s)
  - Edgar F. Codd's relational model in 1970
  - Powerful high-level query language
  - A few major DB systems dominated the market



- Object-Oriented DBMSs (1990s)
  - Motivated by "mismatch" between RDBMS and OO PL
  - Persistent types in C++, Java or Small Talk
  - Issues: Lack of high level QL, no standards, performance

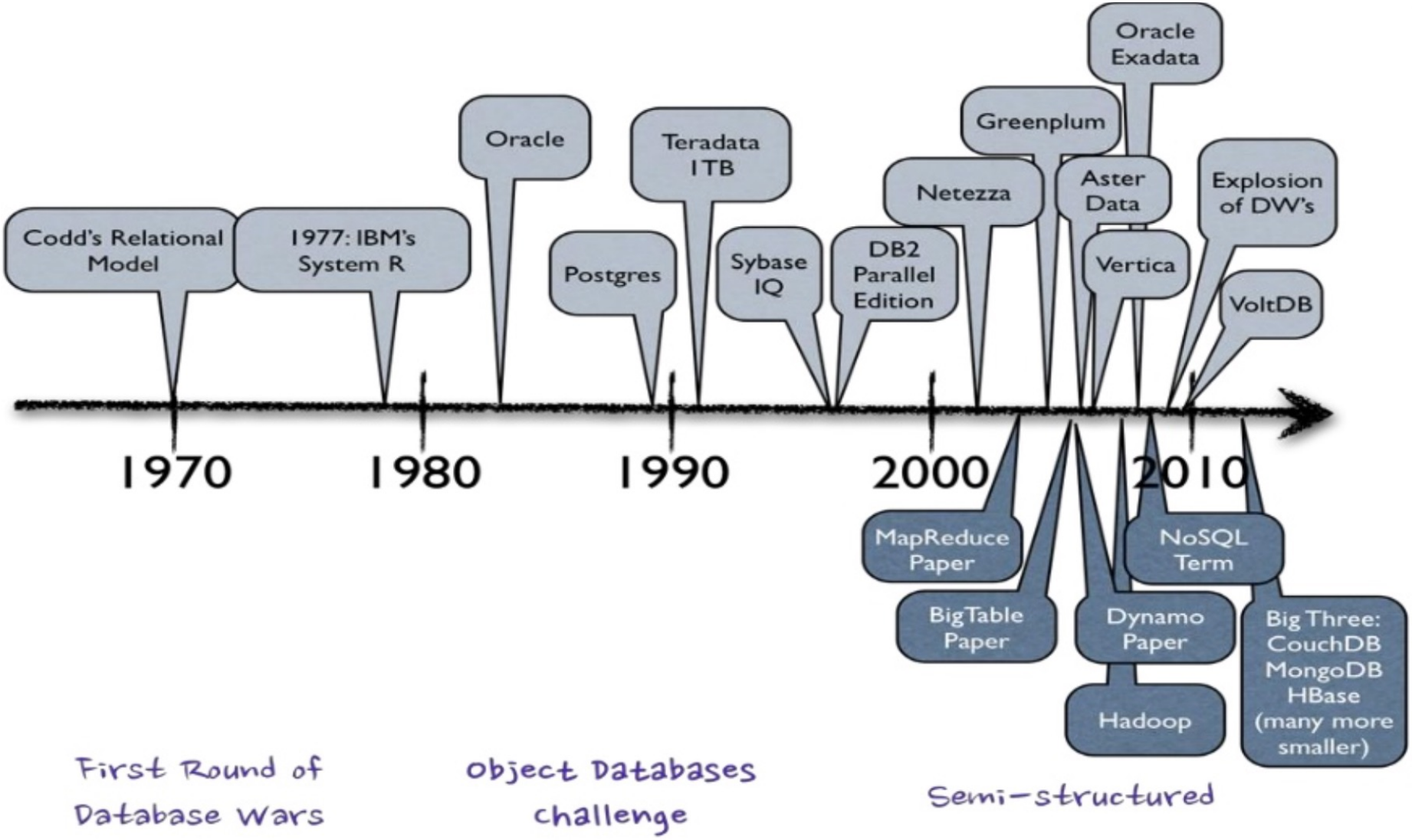
Brief History of Database Systems

- Object-relational DBMS (OR-DBMS) (1990s)
  - Relational DBMS vendors' answer to OO
  - User-defined types, functions (spatial, multimedia) Nested tables
  - SQL: 1999 (2003) standards. Plus performance.
- XML/DBMS (2000s)
  - Web and XML are merging
  - Native support of XML through ORDBMS extension or native XML DBMS
- Data analytics system (DSS) (2000s)
  - **Data warehousing and OLAP**

Brief History of Database Systems

- Data stream management systems (2000s)
  - Continuous query against data streams
- The era of big data (mid 2000-now):
  - **Big data**: datasets that grow so large (terabytes to petabytes) that they become awkward to work with traditional DBMS
  - Parallel DBMSs continue to push the scale of data
  - **MapReduce** dominates on Web data analysis
  - **NoSQL** (not only SQL) is fast growing

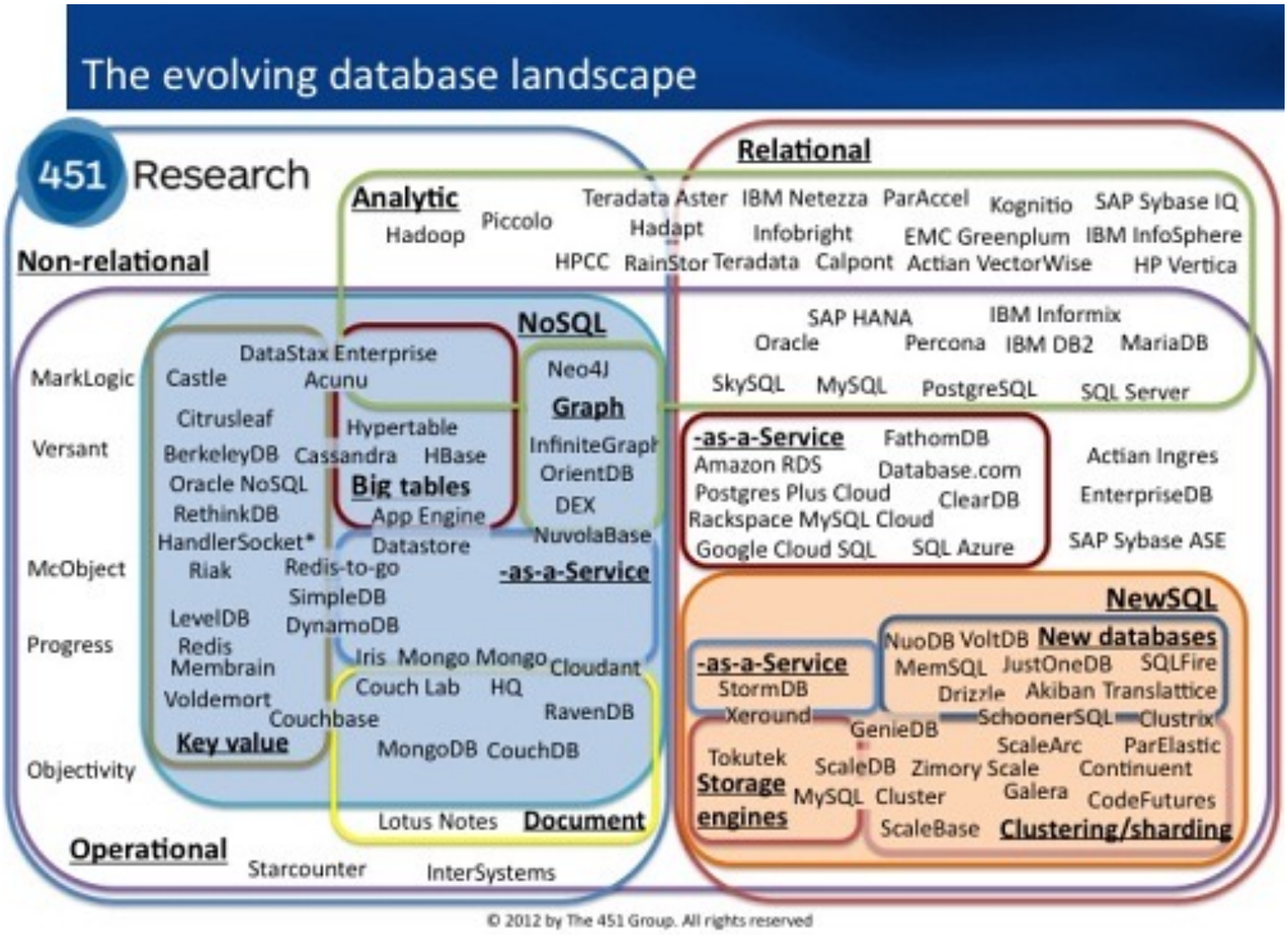
Database Evolution Timeline

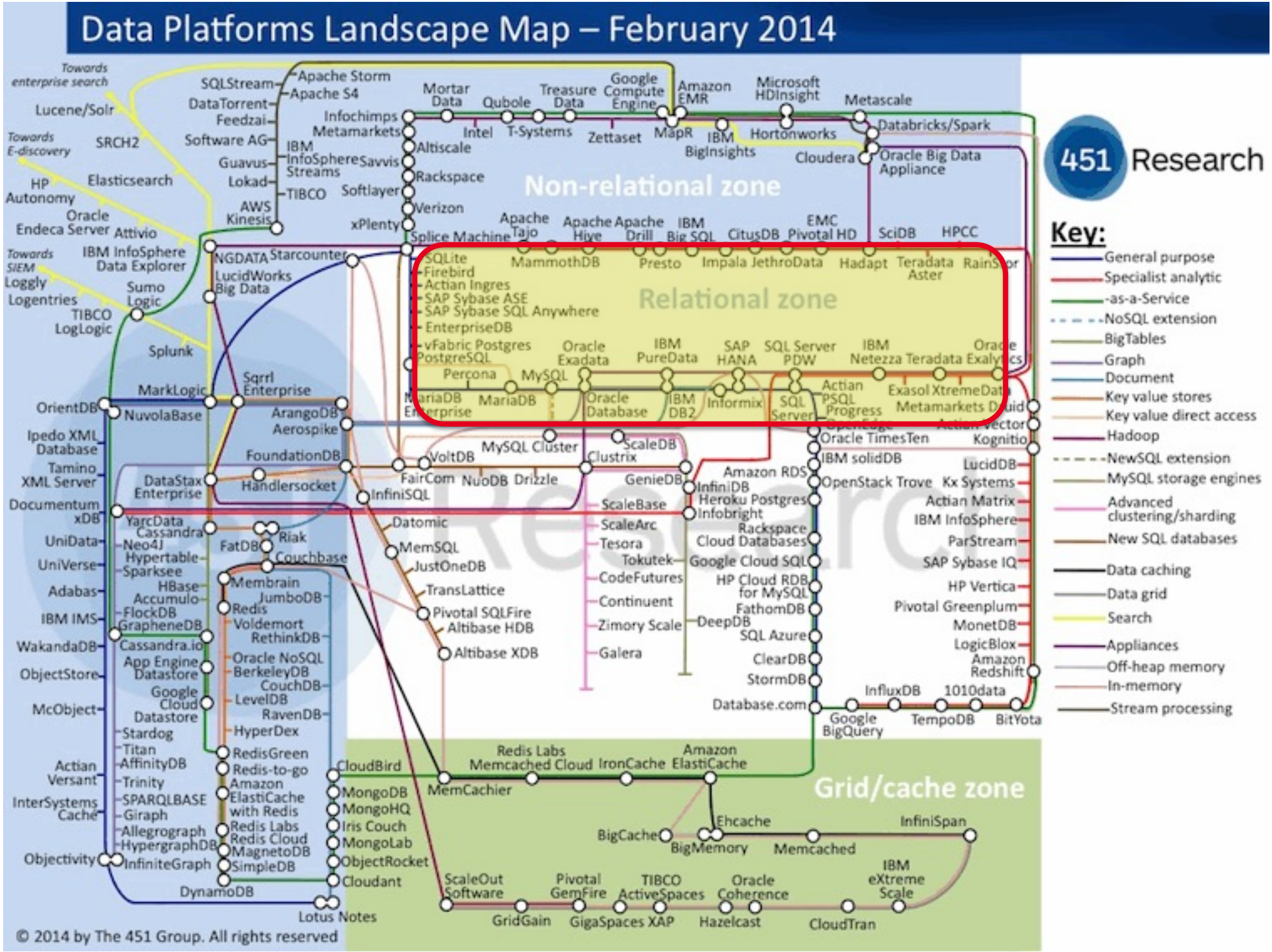


"Big data – an introduction", Peter Morgan

Database Systems Landscape

Database Systems Landscape




Data Platforms Landscape Map – February 2014
451 Research
Towards enterprise search
Towards E-discovery
Towards SIEM
Lucene/Solr
SRCH2
Elasticsearch
HP Autonomy
Oracle Endeca Server
Attivio
IBM InfoSphere Data Explorer
Loggly
Logentries
Sumo Logic
TIBCO LogLogic
Splunk
SQLStream
DataTorrent
Feedzai
Software AG
Guavus
Lokad
AWS Kinesis
Apache Storm
Apache S4
IBM InfoSphere Streams
TIBCO
Infochimps
Metamarkets
Savvis
Softlayer
xPlenty
Mortar Data
Qubole
Treasure Data
Google Compute Engine
Amazon EMR
Microsoft HDInsight
Metascale
Intel
T-Systems
Zettaset
MapR
IBM BigInsights
Hortonworks
Cloudera
Databricks/Spark
Oracle Big Data Appliance
Altiscale
Rackspace
Verizon
Non-relational zone
Apache Tajo
Apache Hive
Apache Drill
IBM Big SQL
CitusDB
EMC Pivotal HD
SciDB
HPCC
Splice Machine
NGDATA
LucidWorks Big Data
Starcounter
SQLite
Firebird
Actian Ingres
SAP Sybase ASE
SAP Sybase SQL Anywhere
EnterpriseDB
vFabric Postgres
PostgreSQL
Relational zone
MammothDB
Presto
Impala
JethroData
Hadapt
Teradata Aster
Oracle Exadata
IBM PureData
SAP HANA
SQL Server PDW
IBM Netezza
Teradata
Percona
MySQL
MariaDB Enterprise
MariaDB
Oracle Database
IBM DB2
Informix
SQL Server
Actian PSQL
Progress
Exasol
Metamarkets
Kognitio
MarkLogic
Sqrrl Enterprise
OrientDB
NuvolaBase
ArangoDB
Aerospike
FoundationDB
Handlersocket
InfiniSQL
VoltDB
FairCom
NuoDB
Drizzle
MySQL Cluster
Clustrix
ScaleDB
GenieDB
ScaleBase
ScaleArc
Tesora
Tokutek
CodeFutures
Continuent
Zimory Scale
Galera
Oracle TimesTen
IBM solidDB
InfiniDB
Heroku Postgres
Infobright
Amazon RDS
Rackspace Cloud Databases
Google Cloud SQL
HP Cloud RDB for MySQL
FathomDB
DeepDB
SQL Azure
ClearDB
StormDB
Database.com
OpenStack Trove
LucidDB
Kx Systems
Actian Matrix
IBM InfoSphere
ParStream
SAP Sybase IQ
HP Vertica
Pivotal Greenplum
MonetDB
LogicBlox
Amazon Redshift
InfluxDB
1010data
Google BigQuery
TempoDB
BitYota
Ipedo XML Database
Tamino XML Server
DataStax Enterprise
Documentum xDB
UniData
UniVerse
Adabas
IBM IMS
WakandaDB
ObjectStore
McObject
Actian Versant
InterSystems Caché
Objectivity
YarcData
Cassandra
Neo4J
Hypertable
Sparksee
HBase
Accumulo
FlockDB
GrapheneDB
Cassandra.io
App Engine Datastore
Google Cloud Datastore
Stardog
Titan
AffinityDB
Trinity
SPARQLBASE
Giraph
Allegrograph
HypergraphDB
InfiniteGraph
Riak
FatDB
Couchbase
Membrain
JumboDB
Redis
Voldemort
RethinkDB
Oracle NoSQL
BerkeleyDB
CouchDB
LevelDB
RavenDB
HyperDex
RedisGreen
Redis-to-go
Amazon ElastiCache with Redis
Redis Labs
Redis Cloud
MagnetoDB
SimpleDB
DynamoDB
Lotus Notes
Datomic
MemSQL
JustOneDB
TransLattice
Pivotal SQLFire
Altibase HDB
Altibase XDB
Redis Labs Memcached Cloud
IronCache
Amazon ElastiCache
CloudBird
MongoDB
MongoHQ
Iris Couch
MongoLab
ObjectRocket
Cloudant
MemCachier
Grid/cache zone
BigCache
Ehcache
BigMemory
Memcached
InfiniSpan
ScaleOut Software
Pivotal GemFire
TIBCO ActiveSpaces
Oracle Coherence
IBM eXtreme Scale
GridGain
GigaSpaces XAP
Hazelcast
CloudTran
Key:
General purpose
Specialist analytic
-as-a-Service
NoSQL extension
BigTables
Graph
Document
Key value stores
Key value direct access
Hadoop
NewSQL extension
MySQL storage engines
Advanced clustering/sharding
New SQL databases
Data caching
Data grid
Search
Appliances
Off-heap memory
In-memory
Stream processing
© 2014 by The 451 Group. All rights reserved

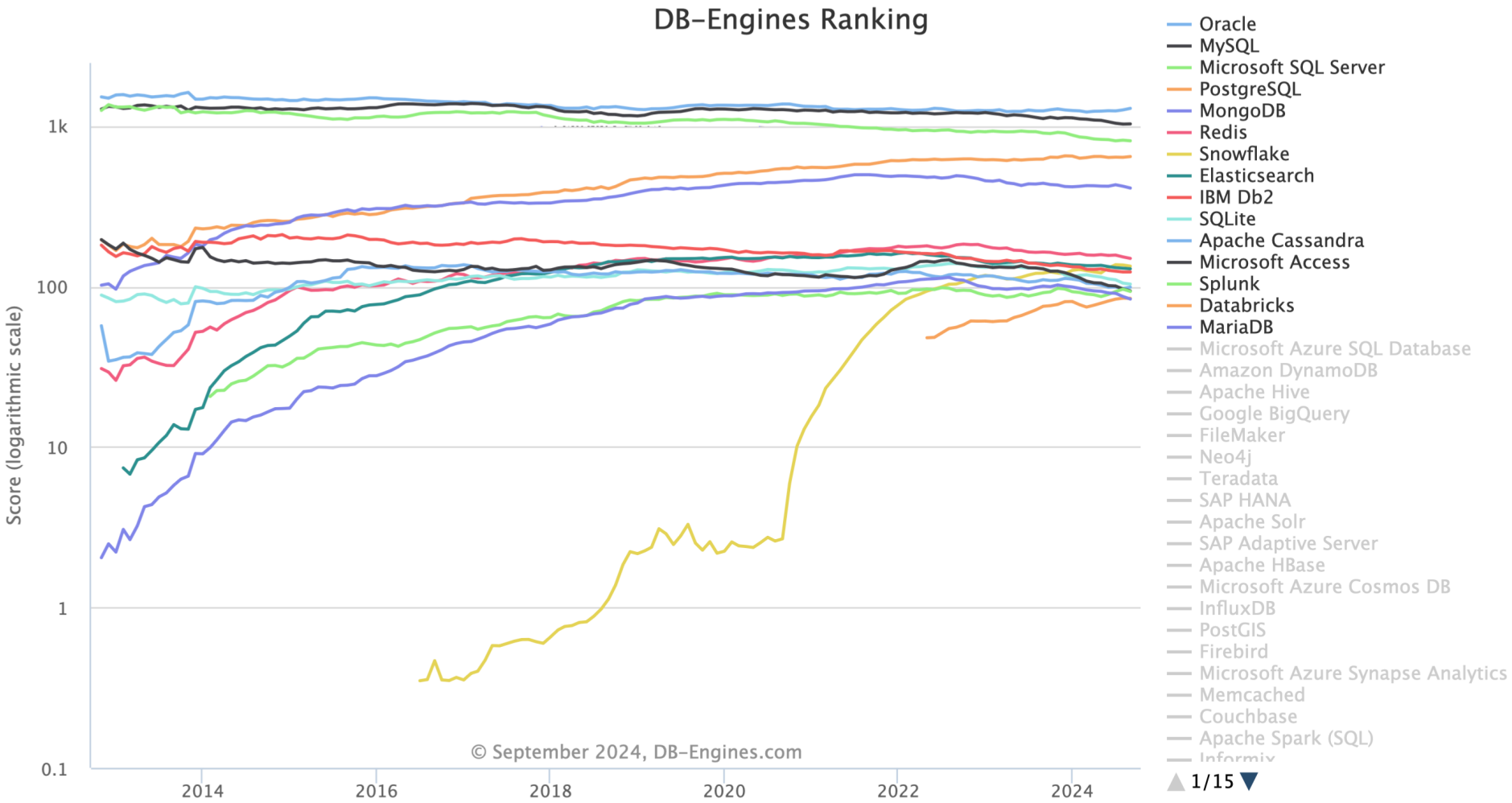
Database Systems Landscape

423 systems in ranking, September 2024

Rank Sep 2024	Rank Aug 2024	Rank Sep 2023	DBMS	Database Model	Score Sep 2024	Score Aug 2024	Score Sep 2023
1.	1.	1.	Oracle	Relational, Multi-model	1286.59	+28.11	+45.72
2.	2.	2.	MySQL	Relational, Multi-model	1029.49	+2.63	-82.00
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	807.76	-7.41	-94.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	644.36	+6.97	+23.61
5.	5.	5.	MongoDB	Document, Multi-model	410.24	-10.74	-29.18
6.	6.	6.	Redis	Key-value, Multi-model	149.43	-3.28	-14.26
7.	7.	↑ 11.	Snowflake	Relational	133.72	-2.25	+12.83
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	128.79	-1.04	-10.20
9.	9.	↓ 8.	IBM Db2	Relational, Multi-model	123.05	+0.04	-13.67
10.	10.	↓ 9.	SQLite	Relational	103.35	-1.44	-25.85
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	98.94	+1.94	-11.11
12.	12.	↓ 10.	Microsoft Access	Relational	93.76	-2.61	-34.81
13.	13.	↑ 14.	Splunk	Search engine	93.02	-3.08	+1.63
14.	↑ 15.	↑ 17.	Databricks	Multi-model	84.24	-0.22	+9.06
15.	↓ 14.	↓ 13.	MariaDB	Relational, Multi-model	83.44	-3.09	-17.01
16.	16.	↓ 15.	Microsoft Azure SQL Database	Relational, Multi-model	72.95	-2.08	-9.78
17.	17.	↓ 16.	Amazon DynamoDB	Multi-model	70.06	+1.15	-10.85
18.	↑ 19.	18.	Apache Hive	Relational	53.07	-2.17	-18.76
19.	↓ 18.	↑ 20.	Google BigQuery	Relational	52.67	-2.86	-3.80
20.	20.	↑ 21.	FileMaker	Relational	45.20	-1.47	-8.40
21.	21.	↑ 23.	Neo4j	Graph	42.68	-1.22	-7.71
22.	↑ 23.	↓ 19.	Teradata	Relational, Multi-model	41.47	-0.78	-18.86

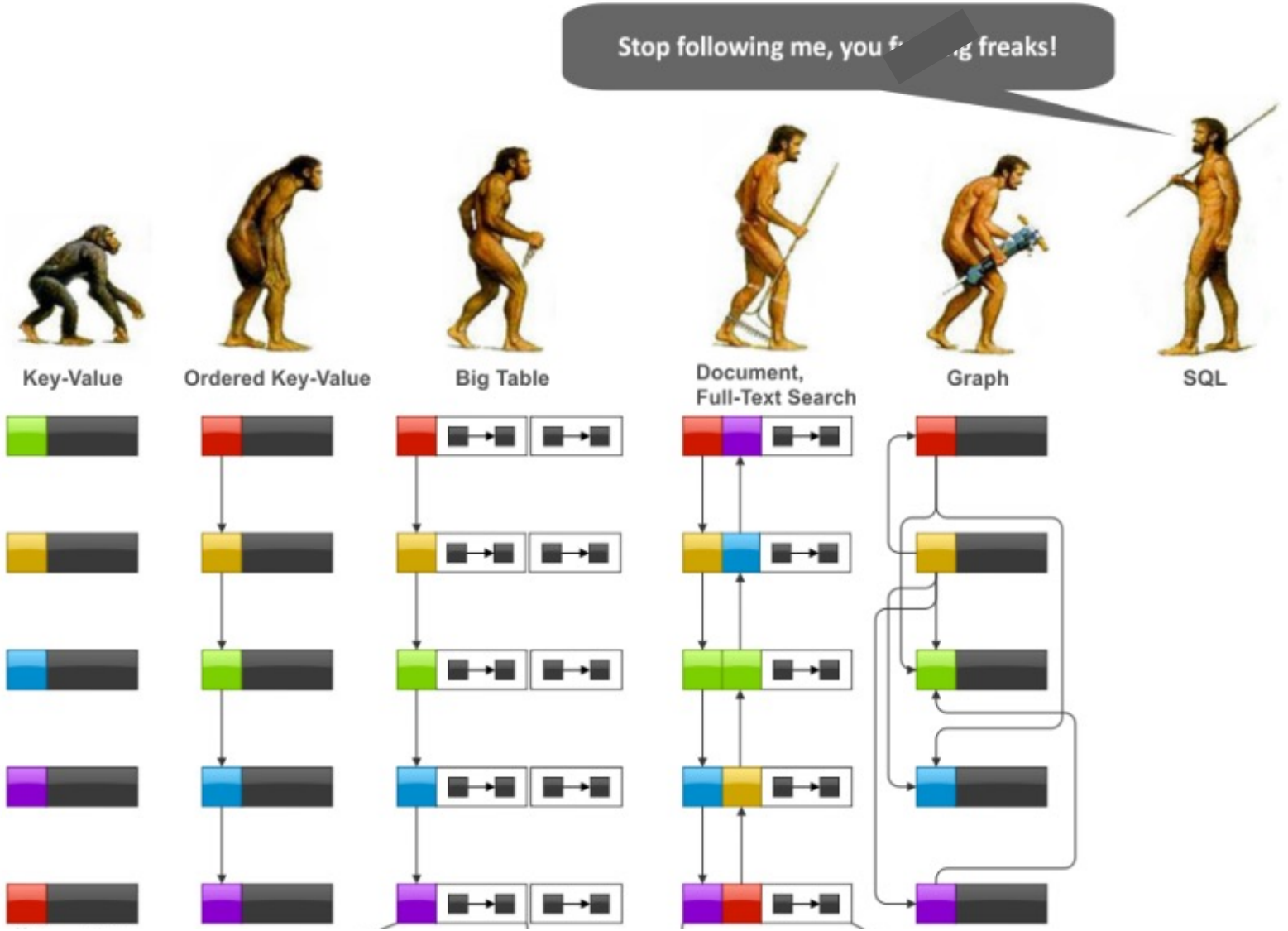
https://db-engines.com/en/ranking

Database Systems Landscape



https://db-engines.com/en/ranking_trend

Database Systems Landscape



https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.

Database Models: Beyond RDBMS

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

The Battle of the Data Models

- Data models are perhaps the most important part of developing software
- They have a profound effect on:
  - how the software is written
  - how we think about the problem that we are solving.
- There are many different kinds of data model
  - Each data model embodies assumptions about how it is going to be used.

- We will now look at a range of general-purpose data models for data storage and querying

When we have some data...

- **Relational Databases solve most data problems**

Why?

- **Persistence**
  - We can store data, and it will remain stored!
- **Integration**
  - We can integrate lots of different apps through a central DB
- **SQL**
  - Standard, well understood, very expressive
- **Transactions**
  - ACID transactions, strong consistency

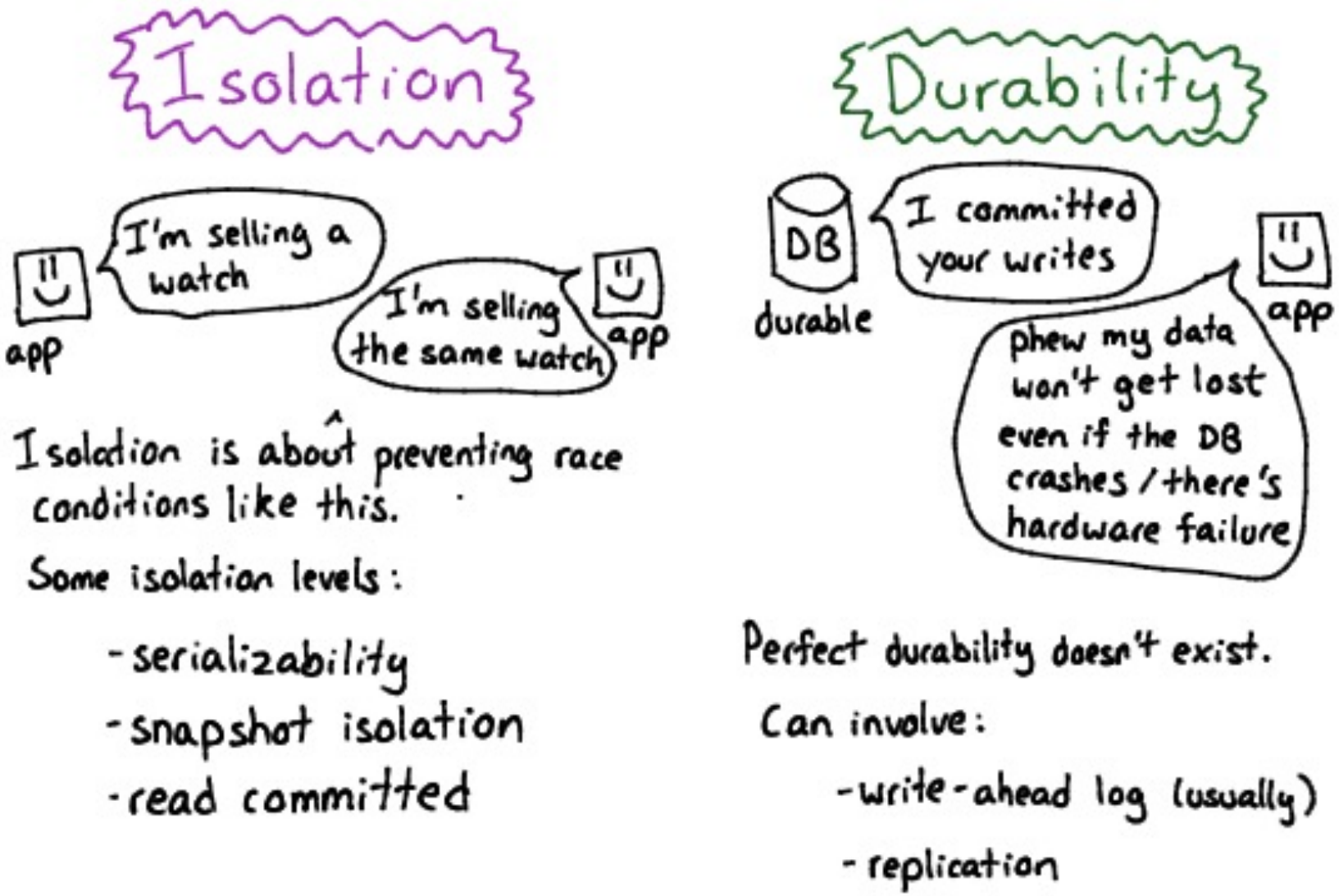
Transactions – ACID Properties

- **Atomic**
  - All of the work in a transaction completes (commit) or none of it completes
- **Consistent**
  - A transaction transforms the database from one consistent state to another consistent state.
- **Isolated**
  - The results of any changes made during a transaction are not visible until the transaction has committed. Concurrent interactions behave as though they occurred serially
- **Durable**
  - The results of a committed transaction survive failures

ACID

https://jvns.ca/blog/2017/06/11/log-structured-storage/

ACID



https://jvns.ca/blog/2017/06/11/log-structured-storage/

The Relational Model

- The relational model, proposed by Edgar Codd in 1970, is still the best-known data model today.
  - data is organized into relations (in SQL: tables), where each relation is an unordered collection of tuples (rows).
- The dominance of relational databases has been around for +40 years.
  - An "eternity" in computing history.
- Other databases at that time forced application developers to think a lot about the internal representation of the data in the database.
  - The goal of the relational model was to hide that implementation detail behind a cleaner interface.

Rivals of the Relational Model

- Over the years, there have been many competing approaches to data storage and querying.
  - Object databases came and went again in the late 1980s and early 1990s.
  - XML databases appeared in the early 2000s, but have only seen niche adoption.
- Much of what you see on the web today is still powered by relational databases
  - Online publishing, discussion, social networking, e-commerce, games, software-as-a-service productivity applications, or much more.
- Now, NoSQL is the most recent attempt to overthrow the relational model's dominance.

Current trends and Issues

- A few key trends and issues have motivated change in relational data storage technologies
  - …In use cases
  - …In technology

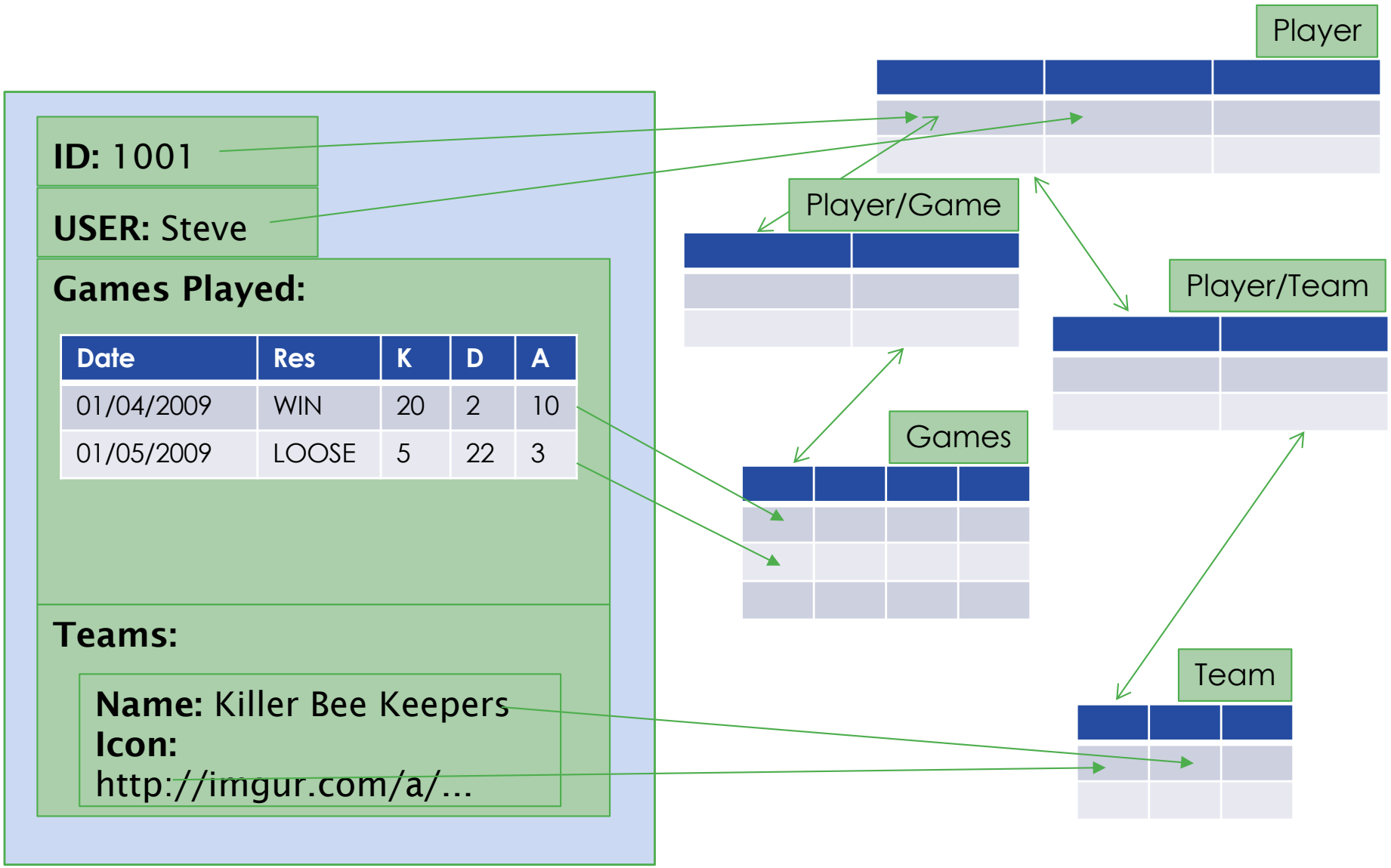
- Key trends include:
  - Increasing **volume of data and traffic**
  - More complex data connectedness
- Key Issues include:
  - The **impedance mismatch** problem

Impedance Mismatch

- **Object** Orientation
  - based on software engineering principles
- **Relational** Paradigms
  - based on mathematics and set theory
- Mapping from one world to the other has problems

- To store data persistently in modern programs a single logical structure must be split up
  - The nice word is **normalised**

Impedance Mismatch – example



Impedance Mismatch – example

http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

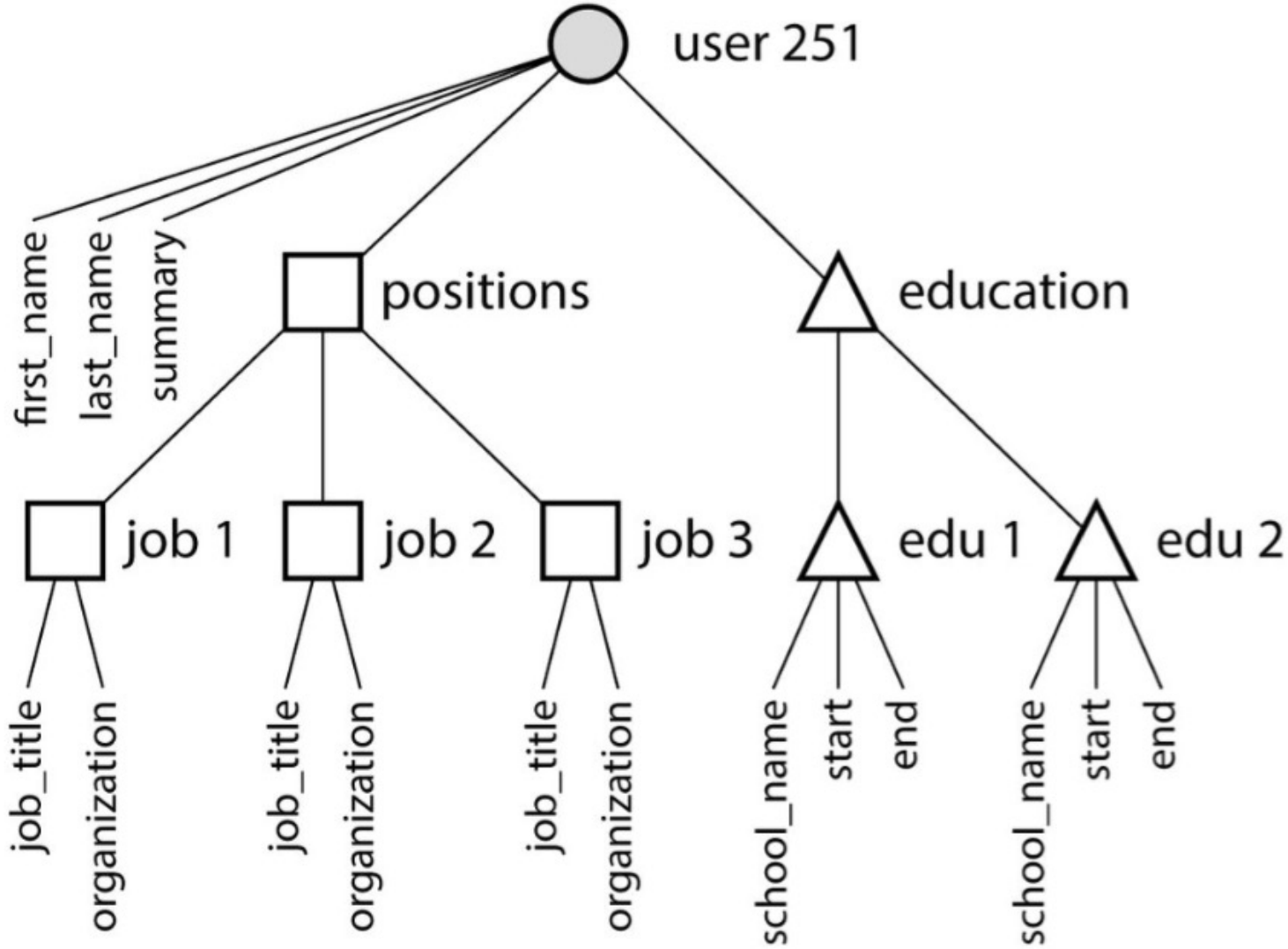
contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

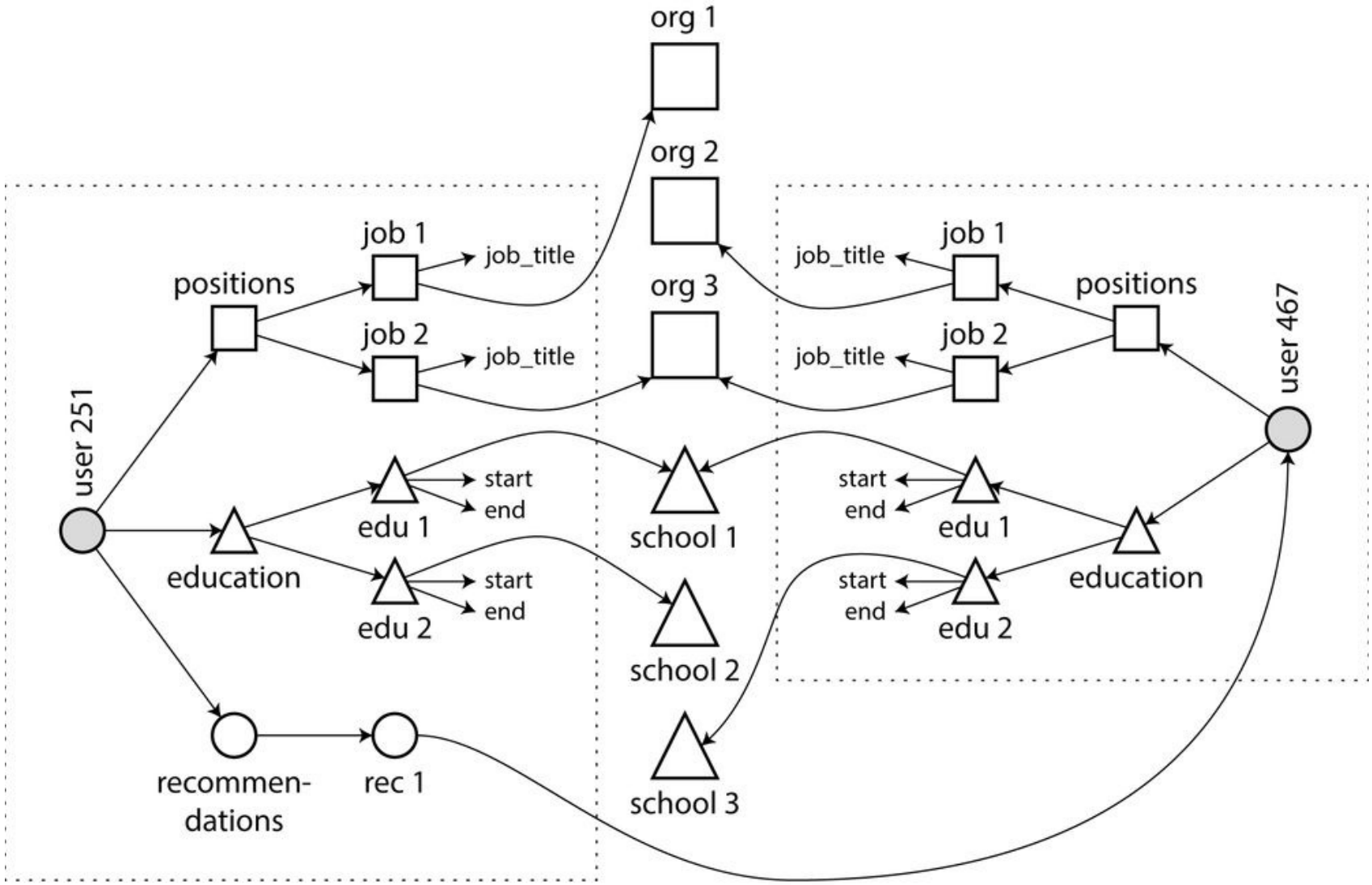
Normalization

- Why IDs (region_id, industry_id, ..) and not plain-text?
  - Consistent style and spelling across profiles,
  - Avoiding ambiguity, e.g. if there are several cities with the same name,
  - The name is stored only in one place, so it is easy to update,
  - Simplify translation into other languages,
- A database in which entities like region and industry are referred to by ID is called **normalized**.
- A database that duplicates the names and properties of entities on each document is **denormalized**.

One-to-Many relations



Many-to-Many relationships



Increased Data Volume

- We are creating, storing, processing more data than ever before!
  - *"From 2005 to 2020, the digital universe will grow by a factor of 300, from 130 exabytes to 40,000 exabytes, or 40 trillion gigabytes (more than 5,200 gigabytes for every man, woman, and child in 2020)".*
    - THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, Dec 2012, John Gantz and David Reinsel
  - *"IDC predicts that the collective sum of the world's data will grow from 33 zettabytes this year to a 175ZB by 2025."*
    - The Digitization of the World, From Edge to Core, Nov 2018
    - https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

EB = 10^{18} ZB = 10^{21}

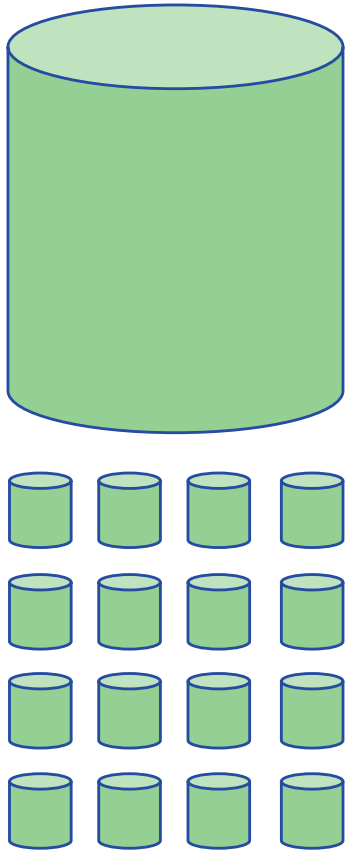
Increased Data Connectivity

- The data we're producing has fundamentally changed
  - from isolated Text Documents (early 1990s)
  - … to html pages with links (early web)
  - … to blogs with pingback, RSS feeds (web 2.0)
  - … to social networks (… add links between people)
  - … to massive linked open data sets (web 3.0… one of them anyway)

Dealing with data size Trends

- Two options when dealing with these trends:

- Build Bigger Database machines
  - This can be expensive
  - Fundamental limits to machine size



- Build Clusters of smaller machines
  - Lots of small machines (commodity machines)
  - Each machine is cheap, potentially unreliable
  - Needs a DBMS which understands clusters

RDBMS have fundamental issues

- In dealing with (horizontal) scale
  - Designed to work on single, large machines
  - Difficult to distribute effectively
- More subtle: An Impedance Mismatch
  - We create logical structures in memory
    - and then rip them apart to stick it in an RDBMS
  - The RDBMS data model often disjoint from its intended use
    - (Normalisation sucks sometimes)
  - Uncomfortable to program with (joins and ORM etc.)

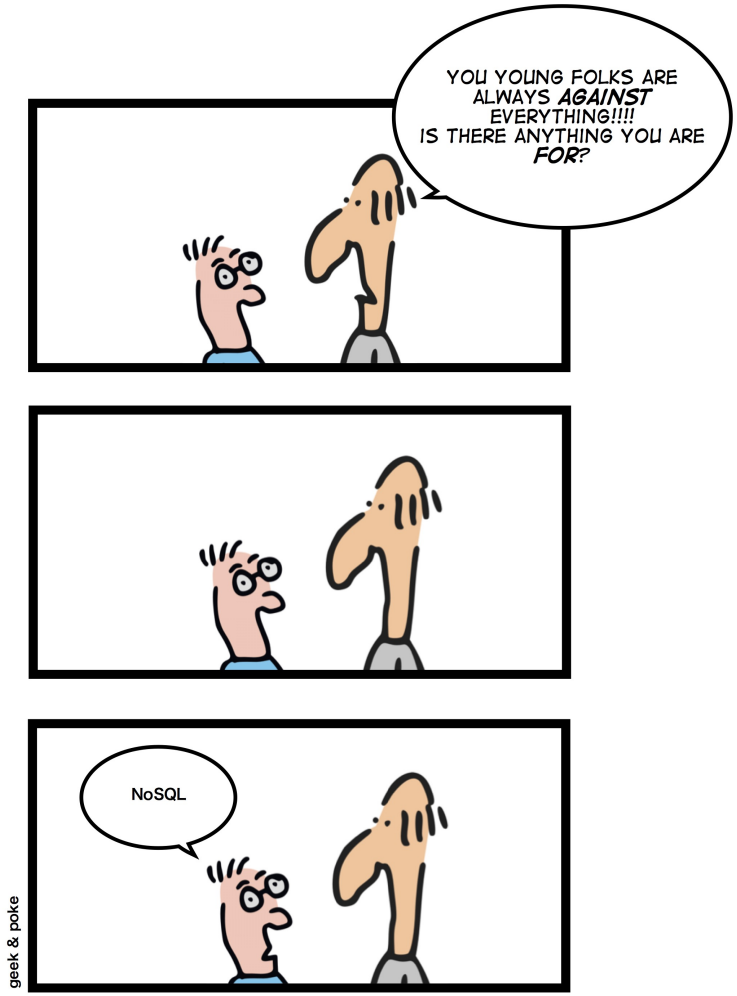
The NoSQL Movement

NoSQL

- The term NoSQL is unfortunate, since it doesn't refer to any technology
  - "Not only SQL"
- Nevertheless, the term struck a nerve, and quickly spread through the web startup community and beyond.
- Several interesting database systems are now associated with the #NoSQL hashtag.

The NoSQL movement

- Key attributes include:
  - **Non-Relational**
    - They can be, but aren't good at it
  - **Simple API**
    - No Join
  - **BASE** & **CAP** Theorem
    - No ACID requirements
  - **Schema-free**
    - Implicit schema, application side
  - **Inherently Distributed**
    - Some more so than others
  - **Open Source**
    - mostly



BASE Transactions

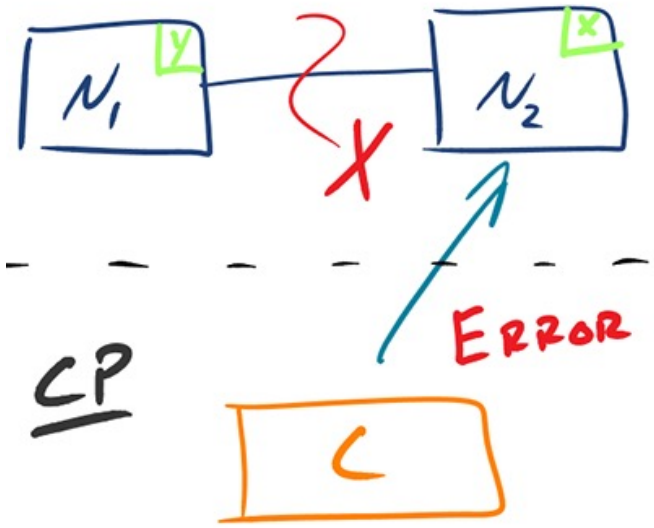
- Acronym contrived to be the opposite of ACID
  - Basic Availability
    - The database appears to work most of the time.
  - Soft-state
    - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
  - Eventual consistency
    - Stores exhibit consistency at some later point (e.g., lazily at read time).
- Characteristics
  - Optimistic
  - Simpler and faster
  - Availability first
  - Best effort
  - Approximate answers OK

Brewer's CAP Theorem

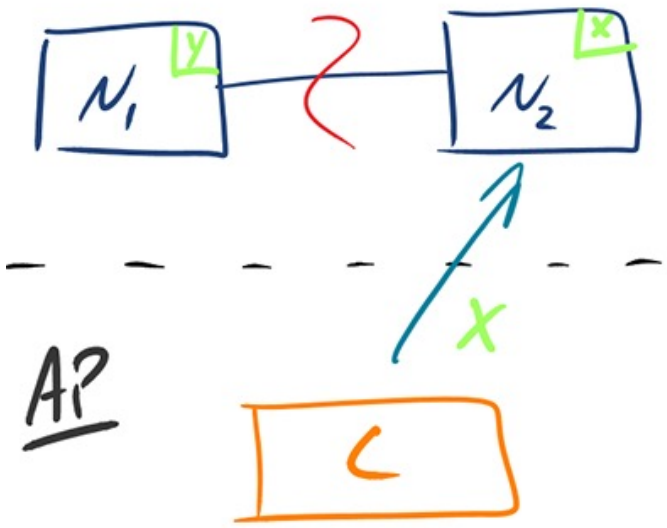
- A distributed system can support only two of the following characteristics:
  - **Consistent**
    - writes are atomic, all subsequent requests retrieve the new value
  - **Available**
    - The database will always return a value so long as the server is running
  - **Partition Tolerant**
    - The system will still function even if the cluster network is partitioned (i.e. the cluster loses contact with parts of itself)
- The overly stated well cited issue is:
  - We can only ever build an algorithm which satisfies 2 of 3.
    - But .. horizontal scaling strategy is based on data partitioning;
    - Therefore, designers are forced to decide between consistency and availability.

Brewer's CAP Theorem

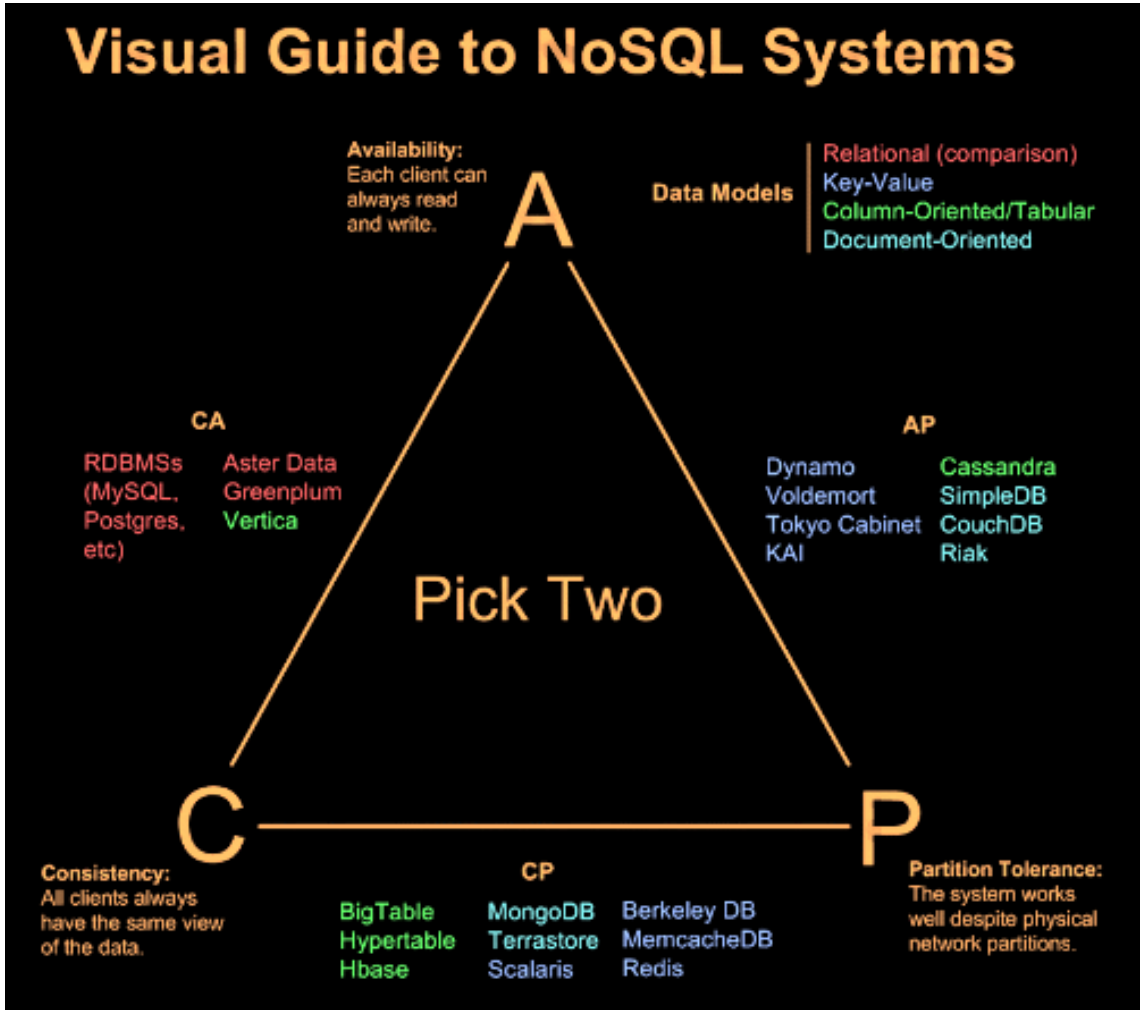
- **CP** - Consistency/Partition Tolerance



- **AP** - Availability/Partition Tolerance



CAP Theorem


Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

Types of NoSQL Databases

- Core types
  - **Key-value** stores
  - **Document** stores
  - **Column** stores
  - **Graph** databases

- Non-core types
  - **Object** databases
  - Native **XML** databases
  - **RDF** stores
  - …

Key-Value Databases – Basics

- Data model
  - The most simple NoSQL database type
  - Works as a simple hash table (mapping)
- Key-value pairs
  - **Key** (id, identifier, primary key) – usually a string.
  - **Value**: can be anything (text, structure, image, etc.) – a black box for the database system.
- Query patterns
  - Create, update or remove value for a given key
  - Get value for a given key
- Characteristics
  - great performance, easily scaled, …
  - not for complex queries nor complex data

Key-Value Databases – Basics

- Suitable use cases
  - Session data, user profiles, user preferences, shopping carts, …
    - I.e. when values are only accessed via keys
- When not to use
  - Relationships among entities
  - Queries requiring access to the content of the value part
- Examples
  - Redis, MemcachedDB, Riak KV, Amazon SimpleDB, Berkeley DB, Oracle NoSQL, LevelDB, Project Voldemort
  - *Multi-model:* OrientDB, ArangoDB

Document Databases – Basics

- Data model - Documents
  - Self-describing complex data structure
  - **Hierarchical tree structures** (JSON, XML, …)
    - Scalar values, maps, lists, sets, nested documents, …
  - Identified by a unique identifier (key, …)
- Document data stores understand their documents
  - Queries can run against values of document fields
  - Indexes can be constructed for document fields
- Query patterns
  - Create, update or remove a document
  - Retrieve documents according to complex queries
- Difference from Key-Value stores
  - Extended key-value stores. The value part is examinable!

Document Databases – Basics

```
{
"_id": "1",
  "name": "steve",
  "games_owned": [
    {"name":"Super Meat Boy"},
    {"name":"FTL"},
  ],
}
```

```
{
  "_id": "2",
  "name": "darren",
  "handle":"zerocool",
  "games_owned": [
    {"name":“FTL"},
    {"name":“Assassin’s Creed 3“, “dev”: “ubisoft”},
  ],
}
```

Document Databases – Basics

- Suitable use cases
  - Event logging, content management systems, blogs, web analytics, e-commerce applications, …
  - I.e. for **structured documents with similar schema**
- When not to use
  - Set operations involving multiple documents
  - Design of document structure is constantly changing
    - I.e. when the required level of granularity would outbalance the advantages of aggregates
- Examples
  - MongoDB, Couchbase, Amazon DynamoDB, CouchDB, RethinkDB, RavenDB, Terrastore
  - *Multi-model:* MarkLogic, OrientDB, OpenLink Virtuoso, ArangoDB

Column Databases – Basics

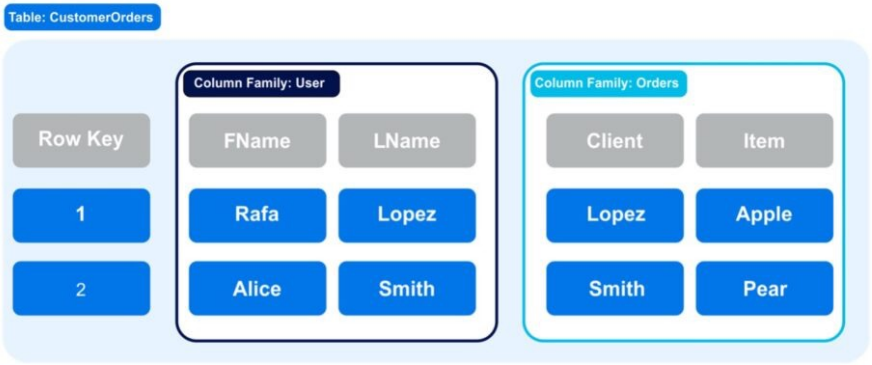
- Data model
  - Column family (table)
    - Table is a collection of similar rows (not necessarily identical)
  - Row
    - Row is a collection of columns - should encompass a group of data that is accessed together
    - Associated with a unique row key
  - Column
    - Column consists of a column name and column value (and possibly other metadata records)
    - Scalar values, but also flat sets, lists or maps may be allowed
- Query patterns
  - Create, update or remove a row within a given column family
  - Select rows according to a row key or simple conditions

Column Databases – Basics

- Suitable use cases
  - Event logging, content management systems, blogs, …
    - I.e. for structured flat data with similar schema
  - Batch processing via mapreduce
- When not to use
  - ACID transactions are required
  - Complex queries: aggregation (SUM, AVG, …), joining, …
  - Early prototypes: i.e. when database design may change
- Examples
  - Apache Cassandra, Apache HBase, Apache Accumulo, Hypertable, Google Bigtable

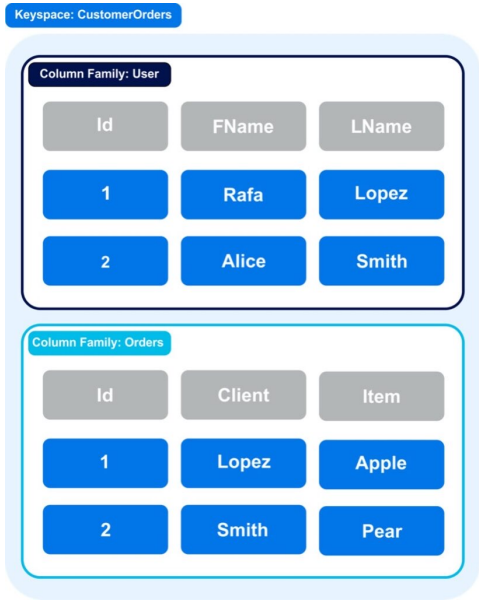
Column Databases – Approaches

- Apache HBase versus Apache Cassandra
  - **HBase**
    - data model is the column-oriented table
    - rows are divided into related columns of data called column families




```
> get 'CustomerOrders ', '1'
COLUMN                          CELL
User: FName timestamp = 1675123184293, value = Rafa
User: LName timestamp = 1675123184293, value = Lopez
Orders: Client timestamp = 1675123388213, value = Lopez
Orders: Client timestamp = 1675123388214, value = Apple
4 row(s) in 0.0260 seconds
```

  - **Cassandra**:
    - data model is best described as a partitioned row store
    - at top-level data model, keyspaces are column-families

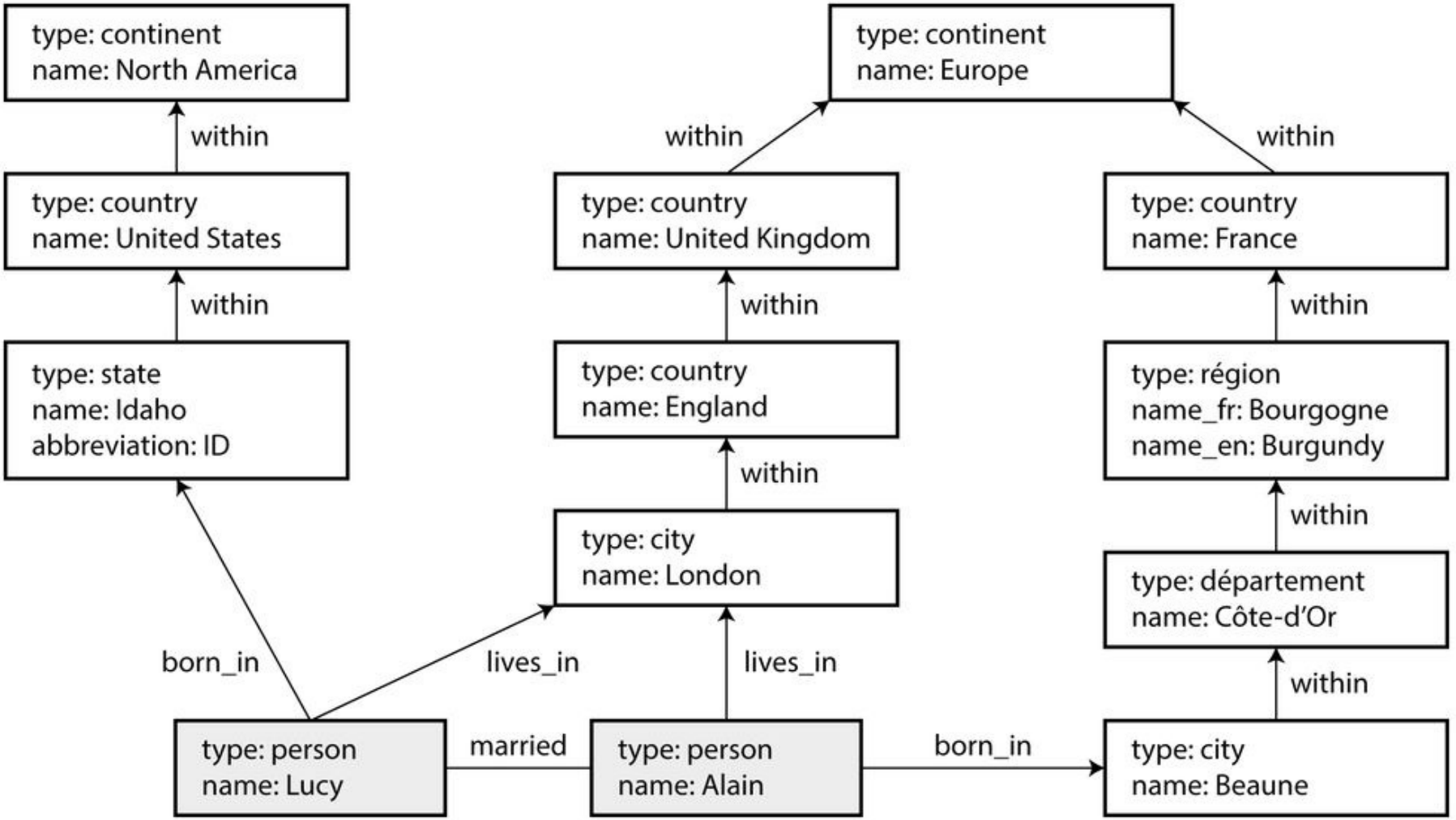



```
cqlsh> select * from User;

id |  fname | lname
--------+-----------+----------
      1 | Rafa   |   Lopez
      2 | Alice  |   Smith

(2 rows)
```

Graph Databases – Basics

- Data Model
  - Focus on modelling graphs' structure and properties
  - Directed / undirected graphs, i.e. collections of …
    - nodes (vertices) for real-world entities, and
    - relationships (edges) between these nodes
  - Both the nodes and relationships can have properties
- Query patterns
  - Create, update or remove a node / relationship in a graph
    - Graph algorithms (shortest paths, spanning trees, …)
    - General graph traversals
    - Sub-graph queries or super-graph queries
    - Similarity based queries (approximate matching)

Graph Databases – Basics



Graph Databases – Basics

- Suitable use cases
  - Social networks, routing, dispatch, and location-based services, recommendation engines, chemical compounds, biological pathways, linguistic trees, …
- When not to use
  - Extensive batch operations are required
    - Multiple nodes / relationships are to be affected
  - Too large graphs to be stored
    - Graph distribution is difficult or impossible at all
- Examples
  - Neo4j, Titan, Apache Giraph, InfiniteGraph, FlockDB
  - *Multi-model*: OrientDB, OpenLink Virtuoso, ArangoDB

Native XML Databases – Basics

- Data model
  - XML documents
  - **Tree structure** with nested elements, attributes, and text values (beside other less important constructs)
  - Documents are organized into collections
- Query languages
  - **XPath**: XML Path Language (navigation)
  - **XQuery**: XML Query Language (querying)
  - **XSLT**: XSL Transformations (transformation)
- Examples
  - Sedna, Tamino, BaseX, eXist-db
  - *Multi-model*: MarkLogic, OpenLink Virtuoso

RDF Databases – Basics

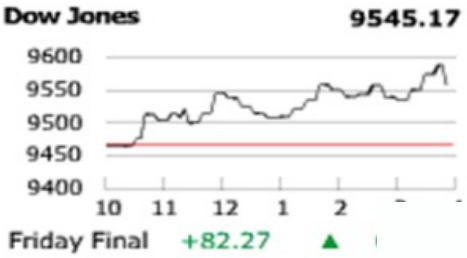
- Data model
  - RDF **triples**
    - Components: **subject**, **predicate**, and **object**
    - Each triple represents a statement about a real-world entity
  - Triples can be viewed as graphs
    - Vertices for subjects and objects
    - Edges directly correspond to individual statements
- Query language
  - **SPARQL**: SPARQL Protocol and RDF Query Language
- Examples
  - Apache Jena, rdf4j (Sesame), Algebraix
  - *Multi-model*: MarkLogic, OpenLink Virtuoso

Time Series Databases – basics

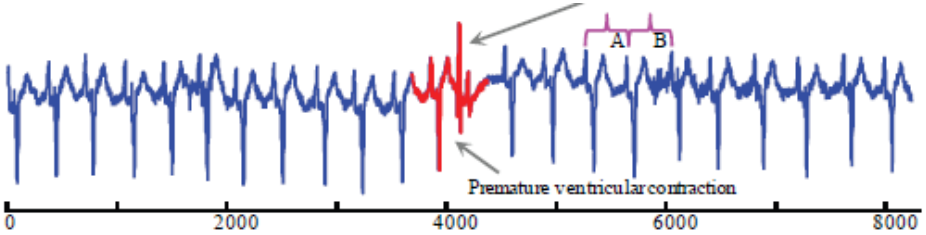
- Data model
  - Stores pairs "Time:Value"
- Query language
  - **Proprietary**: InfluxQL, …
  - **SQL:** some multi-model engines
- Usage
  - store profiles, curves, traces or trends
  - fewer relationships between data entries
  - long sets of data
  - data patterns are "appreciated"
    - compression algorithms to manage the data efficiently
- Examples
  - InfluxDB, Prometheus, Graphite
  - *Multi-model*: Kdb, TimescaleDB

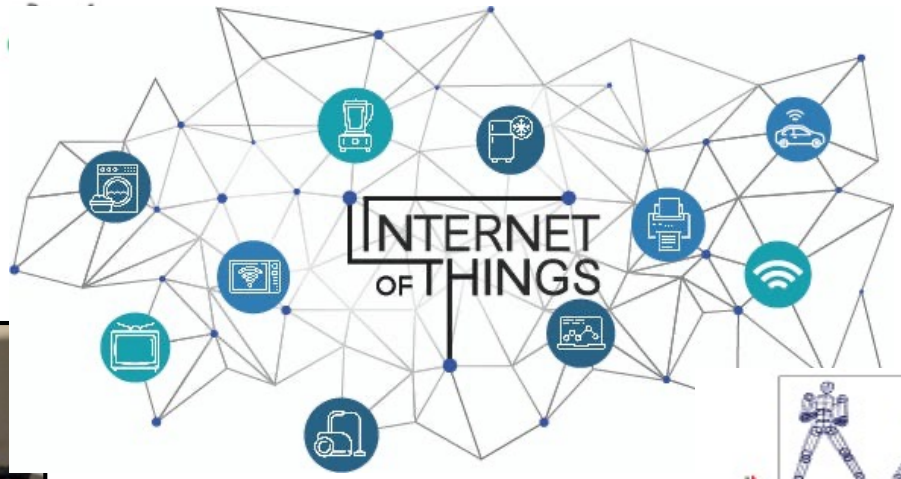
Time Series Examples

Stocks Data

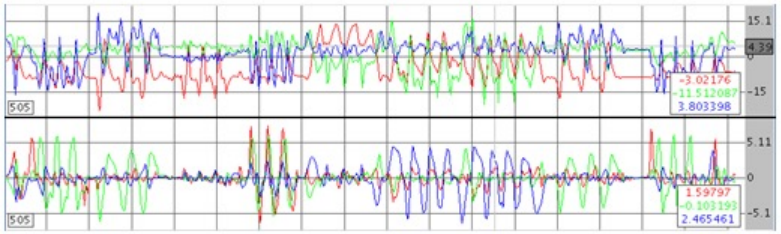


Vital Signals: ECG, EEG,...

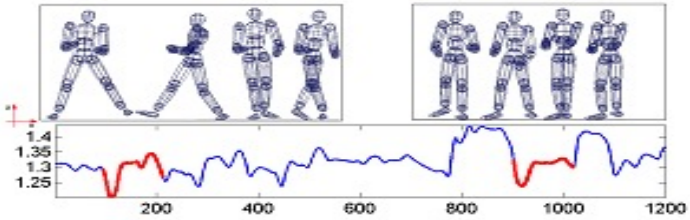


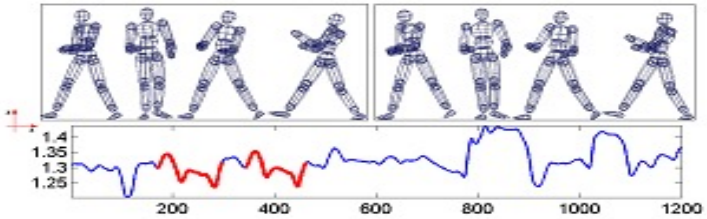


Sensors



Motion Data





NoSQL Databases

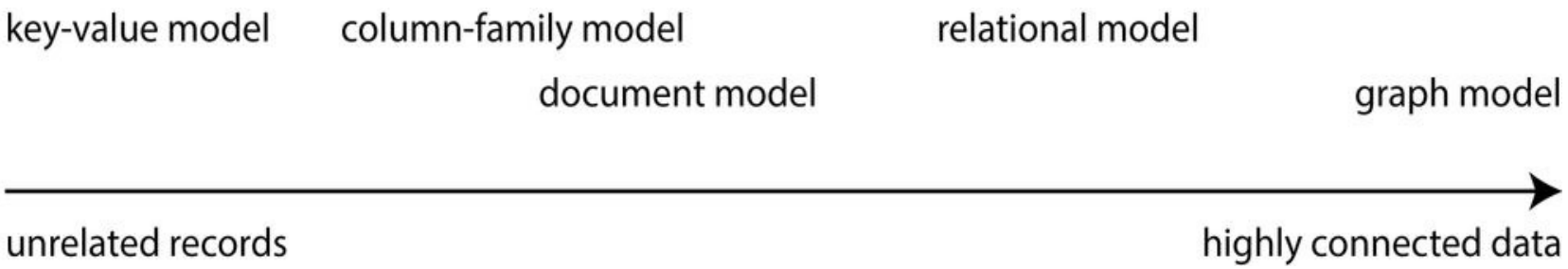
- The end of relational databases?

- **Certainly no!**
  - They are still suitable for most projects (90%)
  - Familiarity, stability, feature set, available support, …

- However, we should also consider different database models and systems
  - **Polyglot persistence** = usage of different data stores in different circumstances

Databases and data connectivity

- Relational model
- NoSQL models
  - Key-value stores
  - Document stores
  - Column stores
  - Graph databases



What next?

- Basic principles
  - Data formats: JSON, YAML, XML, RDF, …
  - Distribution, scaling, sharding, replication, consistency
  - Parallelism, transactions, visualization, processing of graphs
- NoSQL technologies: principles, models, interfaces, languages, …
  - Core databases: Redis, MongoDB, Cassandra, Neo4j
  - MapReduce: Apache Hadoop

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.

Storage and Retrieval

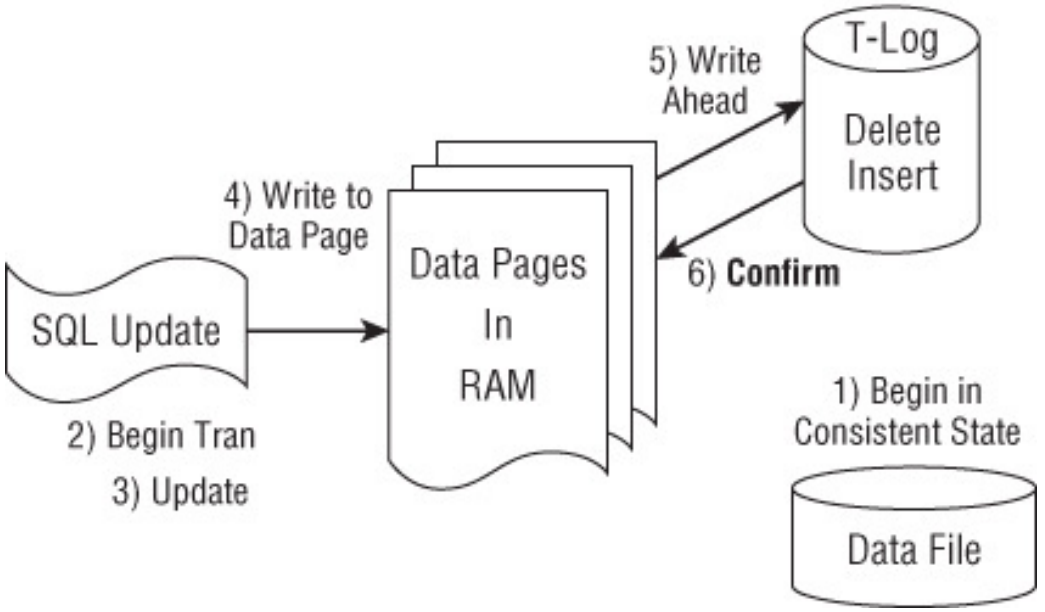
UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Objectives

- How we can store the data.
- How we can find it again.
- How to select a storage engine that is appropriate for an application, from the many that are available.
- Difference between storage engines that are optimized for transactional workloads and those that are optimized for analytics.

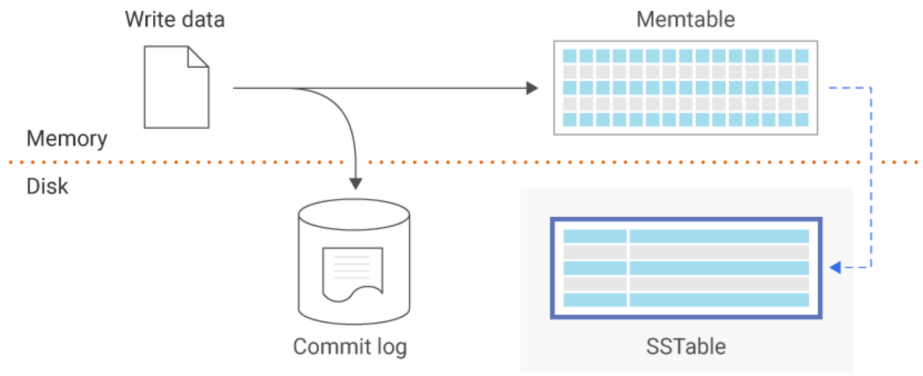
RDMS – Big Picture

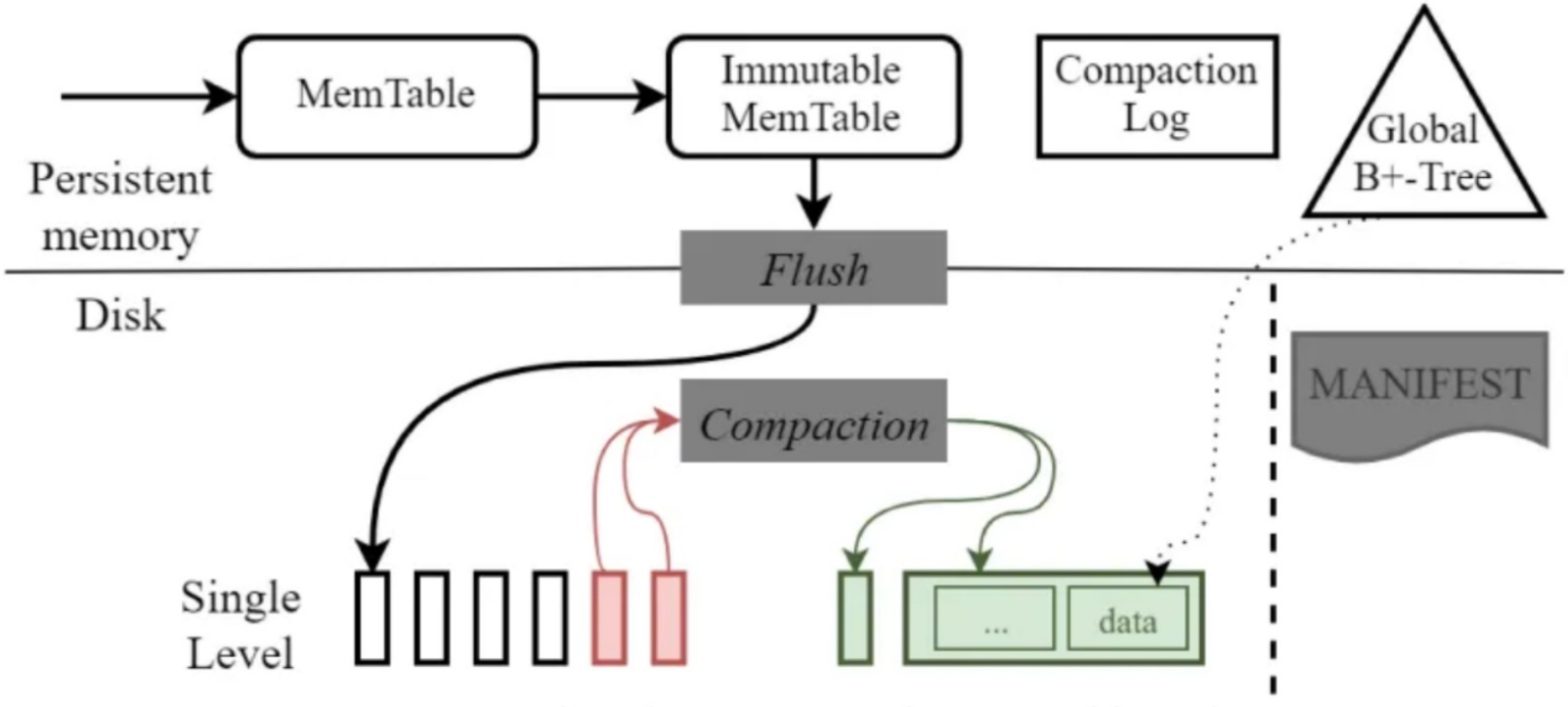
- Heap
  - Flat structure: File_ID | Page_ID | Row
    - Append rows to page
    - File contains pages
- B-Tree
  - Secondary Indexes
  - Clustered
- Transaction Log
  - Append-only log



NoSQL Engines – Big Picture

- Log-Structured Merge Tree
  - Commit-log (write-ahead log)
  - Memtable (sorted k-v@memory)
  - SSTable (sorted k-v@disk)
    - Compaction procedure
    - Indexing (e.g. B-tree)





Storage Data Structures

First example - Bash

- Consider a simple key-value store (text file):

```
#!/bin/bash

cbd_set () {
    echo "$1,$2" >> database
}

cbd_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- The key and value can be (almost) anything you like (e.g. the value could be a JSON document).
- Example:

```
$ cbd_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'
$ cbd_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ cbd_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Bash example

- This storage format is very simple
  - a text file where each line contains a key-value pair, separated by a comma
  - Every call to **cbd_set** appends to the end of the file
  - Old versions of the value are not overwritten

```
$ cbd_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ cbd_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Bash example – performance

- Write (cbd_set): O(1)
  - appending to a file is generally very efficient.
  - problem: huge file - system may need to expand or reallocate disk space. Solution: split n files
  - many databases use an append-only data file or logs.

- Read (cbd_get): *O(n)*
  - Improve efficiency: **index**.
    - additional structure that is derived from the primary data.
    - well-chosen indexes speed up read queries, but every index slows down writes.

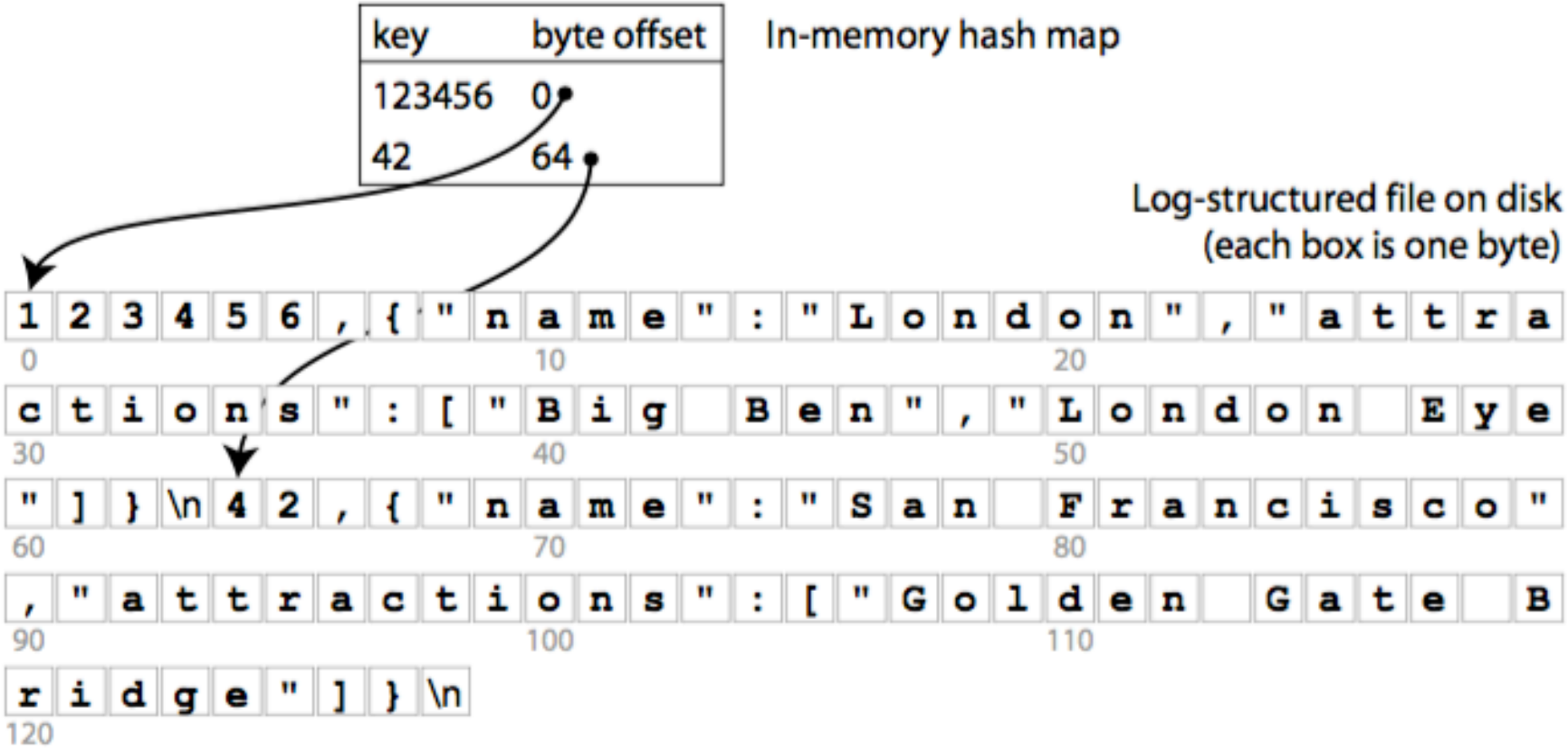
Hash indexes

- Key-value stores are like a *dictionary* which is usually implemented as a hash map.

- Simple indexing strategy:
  - in-memory hash map
  - every key is mapped to a byte offset in the data file.

- This is essentially what some key-value databases do (e.g. Bitcask/Riak)
  - they offer high-performance reads and writes, if the hash map is kept completely in memory.

Hash indexes

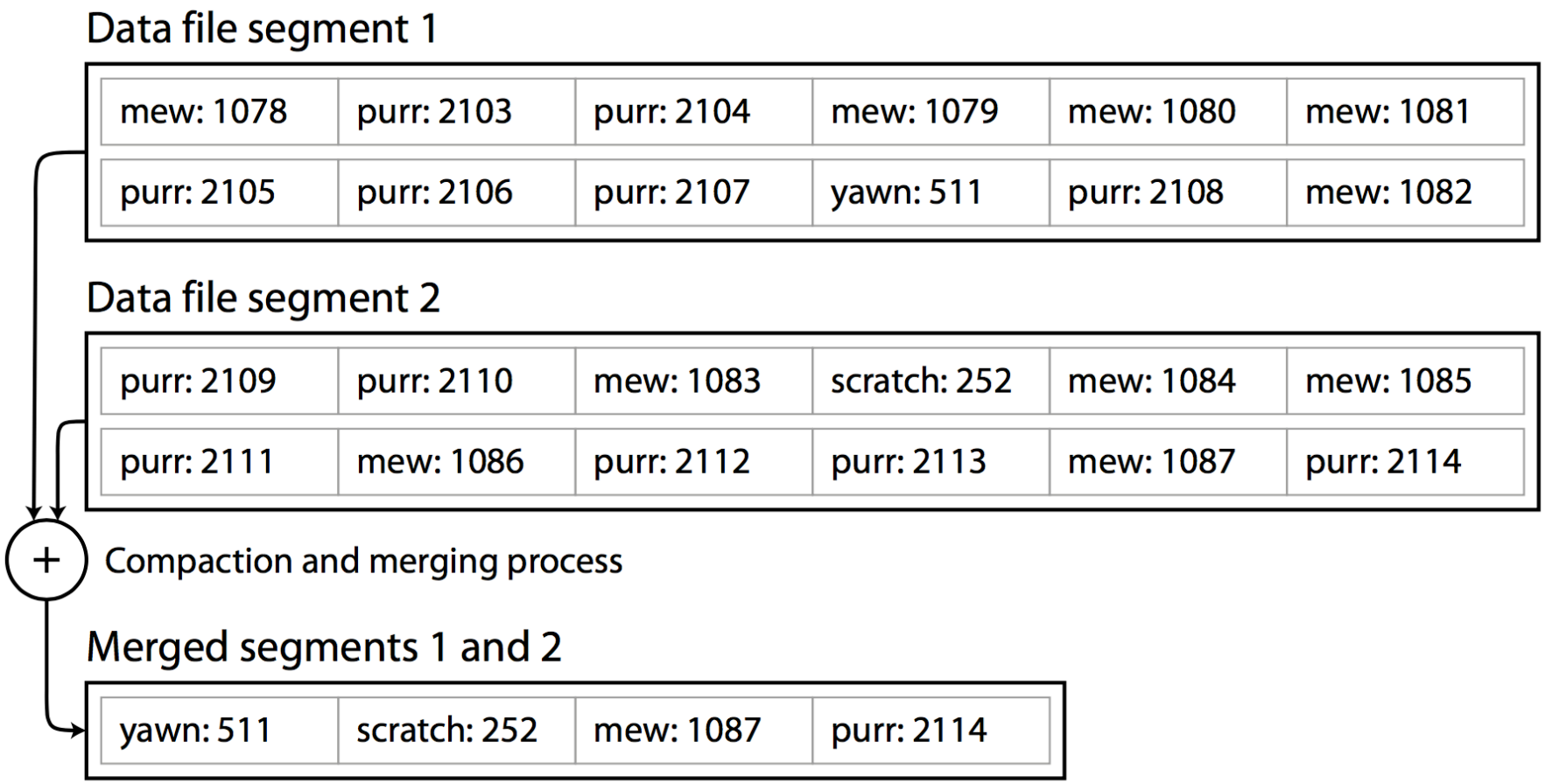


Managing disk space

- How do we avoid running out of disk space?
  - Creating segments (for better performance).
  - Each segment contains all the values written to the database during some period of time.
  - Performing compaction (throwing away duplicate keys in the log).

- The merging and compaction can be done in a background thread.
  - We can continue to serve read and write requests as normal, using the old segment files.

Compaction and merging



Other issues

- File format
  - CSV is not the best format for a log.
  - A binary format may be used here.
- Deleting records
  - To delete a key, we need to append a special deletion record to the data file (tombstone).
  - When log segments are merged, the process discards any previous values for the deleted key.
- Crash recovery
  - If the system is restarted, the in-memory hash maps are lost.
  - To speed up recovery, we may store a snapshot of each segment's hash map on disk.

1. user_id: 001, name: John Doe
2. user_id: 002, name: Jane Smith
3. user_id: 003, name: Alice Brown
4. user_id: 002, tombstone, timestamp: 2024-10-09 12:00:00

Other issues

- Partially written records
  - The database may crash at any time, including halfway through appending a record to the log.
  - Checksums may allow such corrupted parts of the log to be detected and ignored.

- Concurrency control
  - As writes are appended in sequential order, we may have only one writer thread.
  - Data file segments are append-only and immutable, so they can be concurrently read by multiple threads.

Append-only log

- Append-only design seems wasteful at first glance.
  - why don't you update the file in place, overwriting the old value with the new value?
- But, it turns out to be good for several reasons:
  - Appending and segment merging are sequential write operations, which are generally much faster than random writes.
  - Concurrency and crash recovery are much simpler if segment files are append-only or immutable.
- However, it also has limitations:
  - The hash table must fit in memory. It is difficult to make an on-disk hash map perform well.
  - Range queries are not efficient. For example, search all keys between A01 and A99.

Sorted String Table (SSTable)

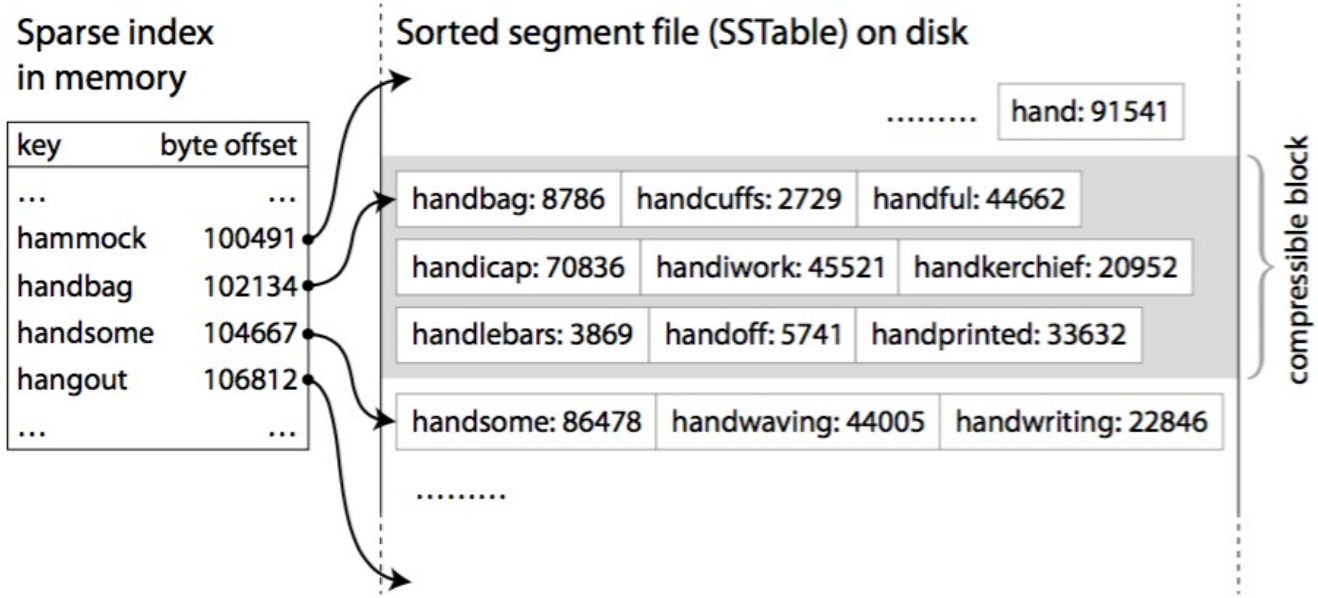
- Previous examples: key-value pairs appear in the order that they were written.

- Idea: sequence of key-value pairs ***sorted by key***.
  - at first glance, that requirement seems to break our ability to use sequential writes.

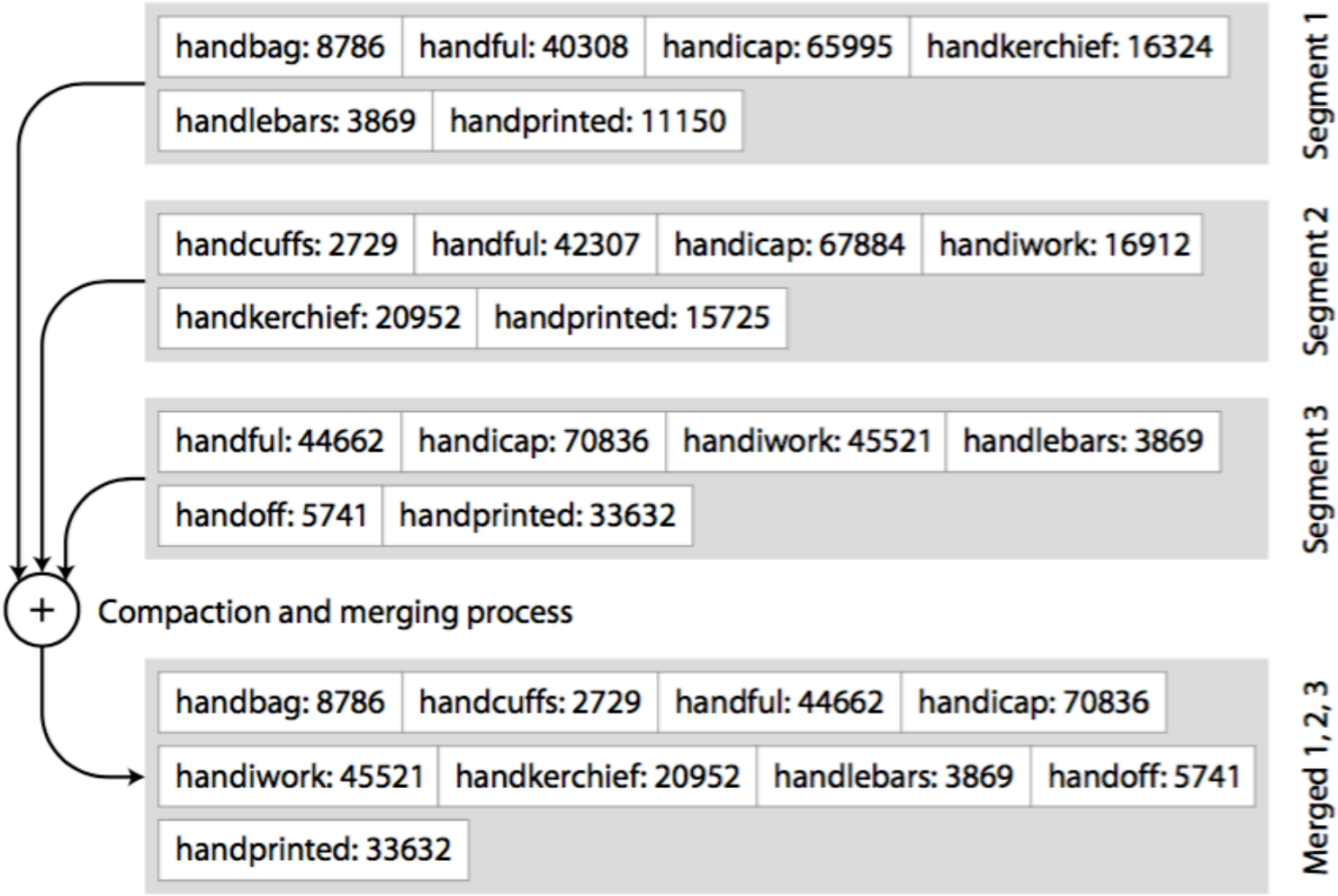
- *Sorted String Table (SSTable)*
  - each key only appears once within each merged segment file (the merging process already ensures that).

SSTable

- SSTables have several advantages over log segments with hash indexes:
  - Merging segments is simple and efficient, even if the files are bigger than the available memory (like the mergesort algorithm).
  - No need to keep an index of all the keys in memory



Merging SSTable segments



SSTable – sorting

- How do you sort the data by key?
  - Incoming writes can occur in any order.

- Solution: a sorted structure in memory.
  - Using tree data structures (B-trees, AVL, Red-Black trees, …)
    - insert keys in any order
    - read them back in sorted order
  - This in-memory tree is sometimes called a *memtable*.

SSTable – sorting

- When the memtable gets bigger than some threshold, write it out to disk as an SSTable file.
  - **efficient** since the tree already maintains the key-value pairs sorted by key.
  - new SSTable file is the most recent database segment.
  - memtable can be emptied after the SSTable file is written.

- Read process
  - first search the key in the memtable
  - then in the most recent on-disk segment, then in the next-older segment, etc.

- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

SSTable and Log

- Problem: database crashes
  - the most recent writes (in the memtable but not yet written out to disk) are lost.

- Solution:
  - keep a separate log on disk to which every write is immediately appended.
  - every time the memtable is written out to an SSTable, the corresponding log can be discarded.

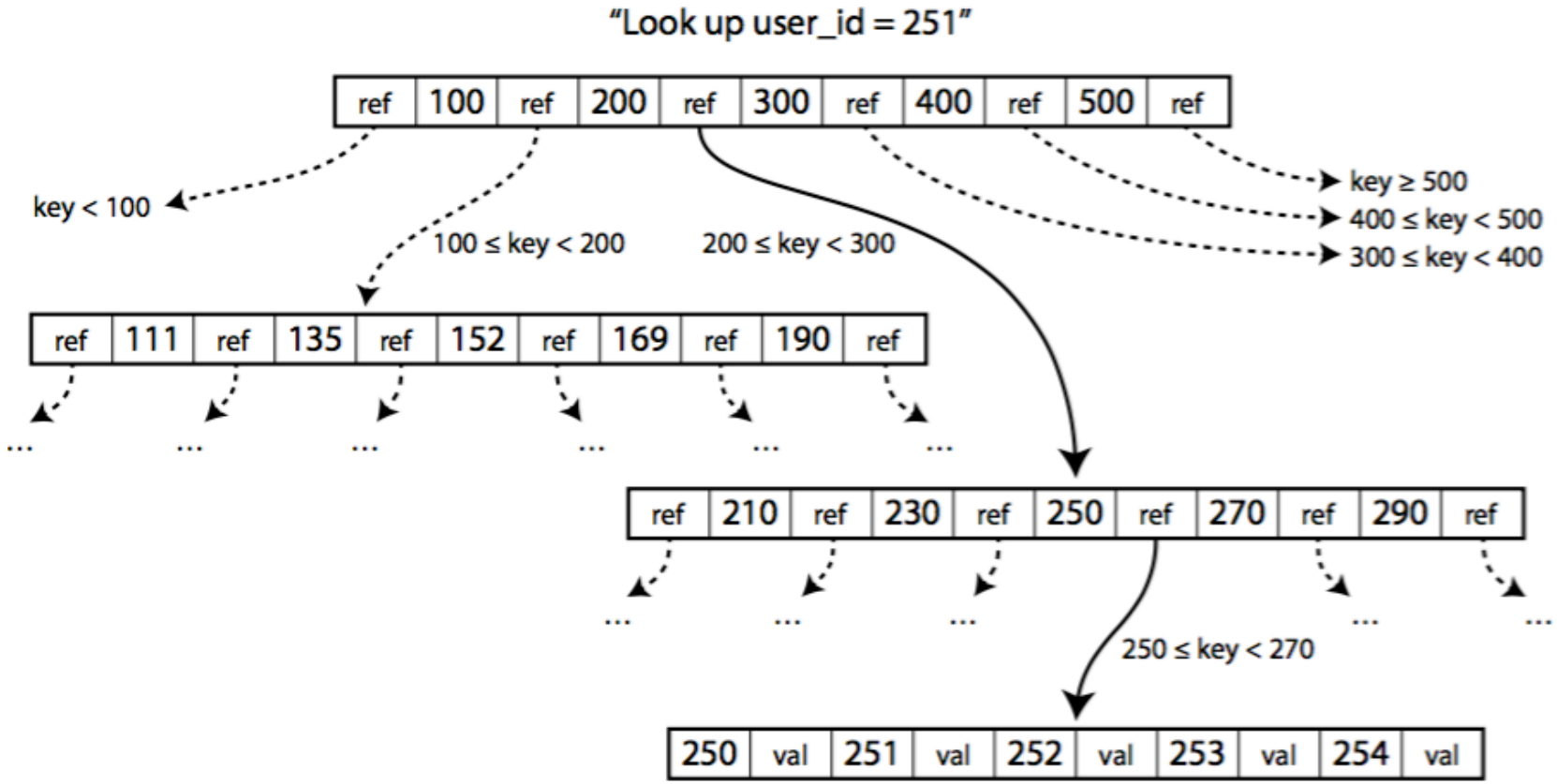
SSTable applications

- Google's Bigtable
  - which introduced the terms *SSTable* and *memtable*.
- LevelDB and RocksDB
  - key-value storage engine libraries that are designed to be embedded into other applications (e.g. Riak).
- Cassandra and HBase
  - both inspired by Google's Bigtable.
- Lucene
  - an indexing engine for full-text search used by Elasticsearch and Solr.

B-trees

- B-tree are the standard index implementation in almost all relational databases and many non-relational databases.
- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.
- B-trees break the database down into fixed-size blocks or pages, traditionally 4 kB in size, and read or write one page at a time.
  - closely to the underlying hardware as disks are also arranged in fixed-size blocks.

B-trees


"Look up user_id = 251"
ref 100 ref 200 ref 300 ref 400 ref 500 ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
ref 111 ref 135 ref 152 ref 169 ref 190 ref
ref 210 ref 230 ref 250 ref 270 ref 290 ref
250 ≤ key < 270
250 val 251 val 252 val 253 val 254 val

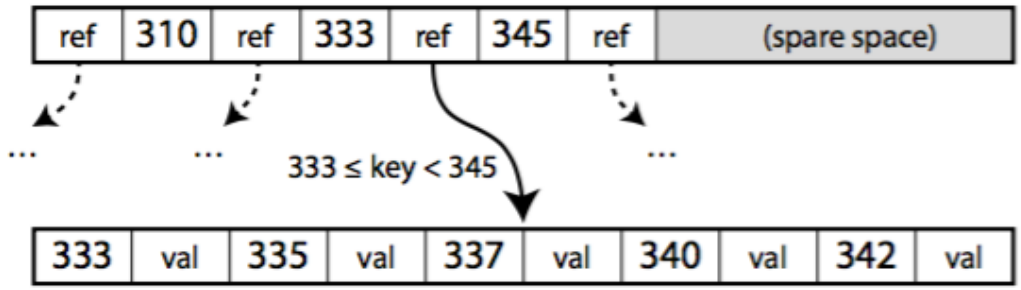
B-trees

- The structure starts at the root page.
- Each page contains *k* keys and *k + 1* references to child pages
  - *k* would typically be in the hundreds.
- Each child is responsible for a continuous range of keys, and the keys on the root page indicate where the boundaries between those ranges lie.

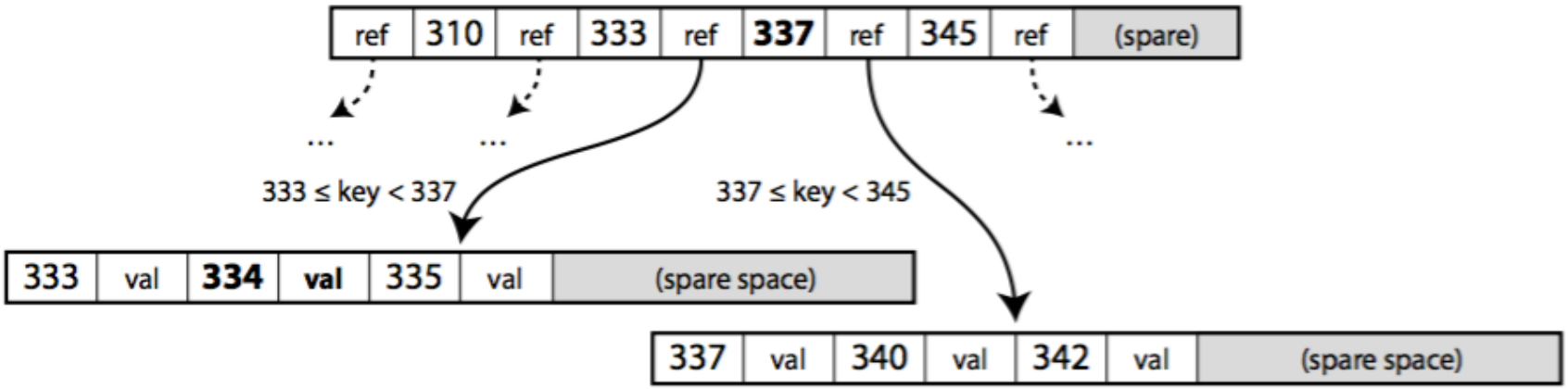
B-trees – operations

- Adding a new key:
  - find the page for the new key and add it.
  - no page free space?
    - page split into two half-full pages
    - parent page is updated to account for the new subdivision
- Updating the value for an existing key:
  - search for the leaf page containing that key
  - change the key value in the page
  - write the page back to disk
- The tree remains *balanced*
- Time complexity (Search, Insert, Delete)
  - $O(\log_k n)$ for a B-tree with k entries per block and n keys

B-tree – splitting a page



After adding key 334:



Update-in-place *vs* append-only log

- Log-structured index: only append to files but never modify files in place.
- B-tree index: write operation is to overwrite a page on disk with new data.
  - Some operations require several pages to be overwritten.
- B-tree Problem: crash after writing part of the pages
  - corrupted index: orphan page which is not a child of any parent.
- Solution: include a *write-ahead log* (WAL, or *redo log*)
  - An append-only file to which every B-tree modification must be written before it can be applied to the tree.

Update-in-place vs. append-only log

- More issues: updating pages in-place requires concurrency control.
  - avoid inconsistent states when multiple threads access the B-tree.

- This is typically done by protecting the tree's data structures with latches (lightweight locks).

- In this case, log-structured approaches are simpler
  - The merging and swapping occur in background without interfering with incoming queries.

B-tree optimizations

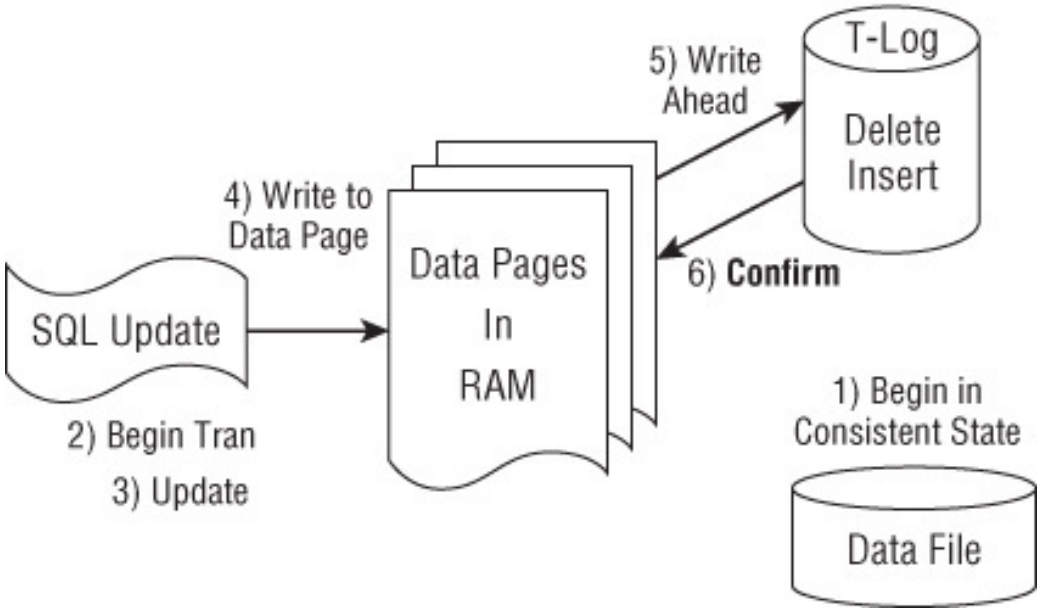
- Copy-on-write scheme.
  - To deal with crash recovery, a modified page is written to a different location, and a new version of parent pages in the tree is created, pointing at the new location.

- We can save space in pages by not storing the entire key, but abbreviating it.
  - Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.

- B-tree variants such as fractal trees borrow some log-structured ideas to reduce disk seeks.

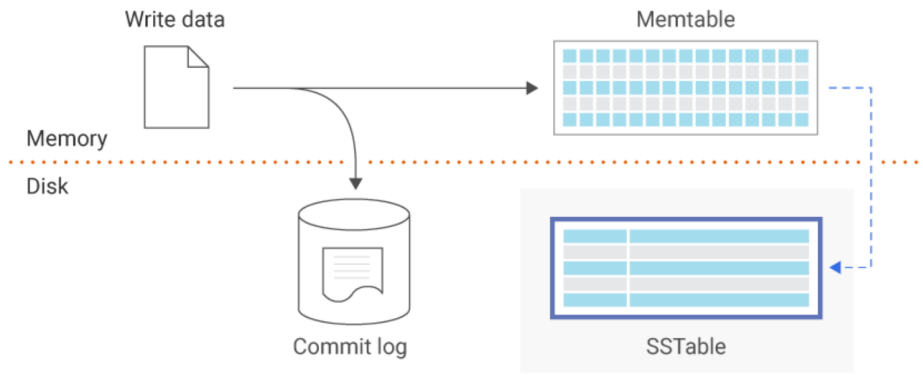
RDMS – Big Picture

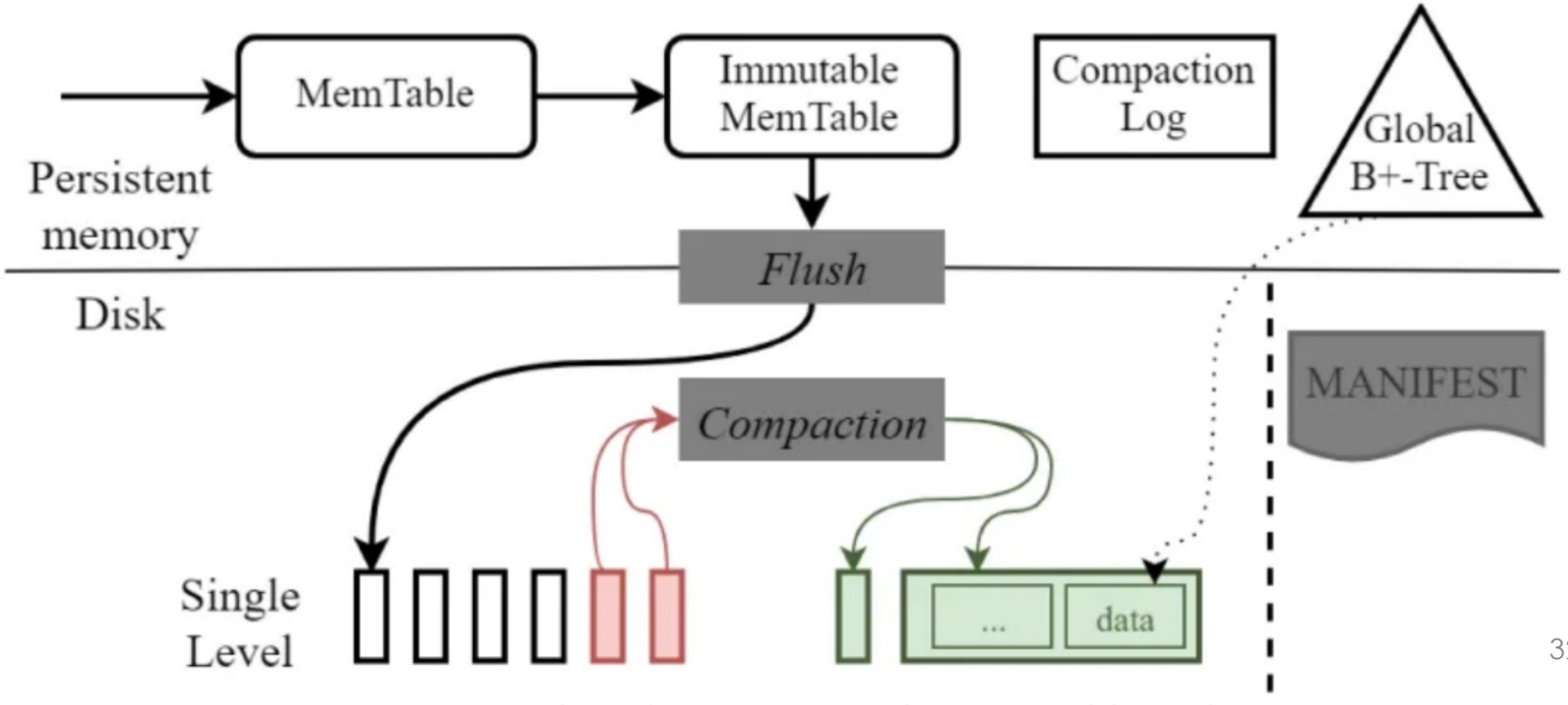
- Heap
  - Flat structure: File_ID | Page_ID | Row
    - Append rows to page
    - File contains pages
- B-Tree
  - Secondary Indexes
  - Clustered
- Transaction Log
  - Append-only log



NoSQL Engines – Big Picture

- Log-Structured Merge Tree
  - Commit-log (write-ahead log)
  - Memtable (sorted k-v@memory)
  - SSTable (sorted k-v@disk)
    - Compaction procedure
    - Indexing (e.g. B-tree)

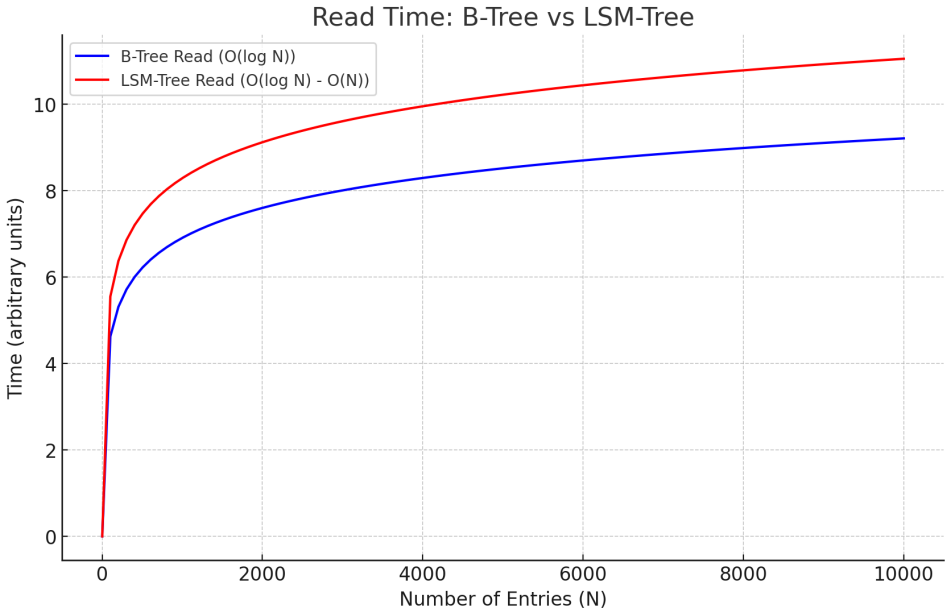


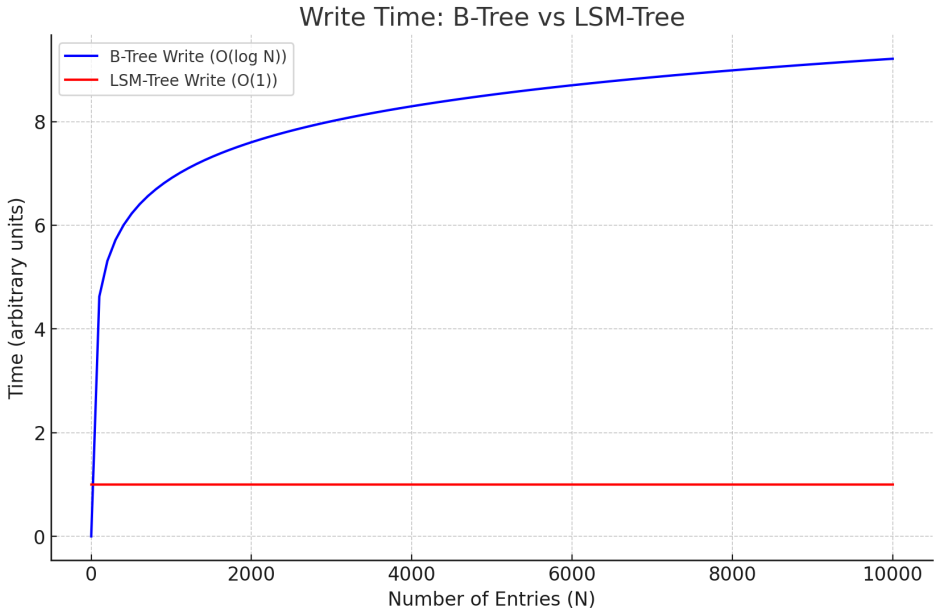


B-Tree versus LSM-Tree

- B-Tree
  - Read: O(log N)
    - Optimized for reading.
    - Data is ordered and allows direct access to disk pages.
  - Writing: O(log N)
    - Writes directly to disk
    - May involve several I/O operations - node rearrangements or splits.
- LSM-Tree
  - Read: Between O(log N) and O(N)
    - Data is spread across multiple levels. The read may need to combine multiple disk segments, requiring real-time merges.
    - Optimizations: e.g. Bloom filters are used to mitigate unnecessary reads
  - Write: O(1)
    - Highly efficient
    - Data is first stored in memory and only later written to disk.

B-Tree versus LSM-Tree





B-Tree versus LSM-Tree

- A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes.

- B-trees: each key exists in exactly one place in the index; Log-structured storage: multiple copies in different segments.
  - B-trees attractive in databases that want to offer strong transactional semantics.
  - In many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.

Other indexing structures

- Secondary indexes
  - the main difference is that keys are not unique,
  - i.e. there might be many rows (documents, vertices) with the same key.

- Implementation - two ways:
  - either by making each value in the index a list of matching row identifiers (like a posting list in a full-text index), or
  - by making each key unique by appending a row identifier to it.

- B-trees and log-structured indexes can be used as secondary indexes.

Storing values within the index

- **Key** is the thing that queries search
- **Value** could be one of two things:
  - the actual row (document, vertex) in question
  - a reference to the row stored place
- In the reference case, the place where rows are stored is known as a ***heap file***.
  - It avoids duplicating data when multiple secondary indexes are present.
- When updating a value, the heap file approach can be quite efficient.
- The record can be overwritten in-place, if the new value is not larger than the old value.

Storing values within the index

- ***Clustered index***
  - store the indexed row directly within an index.
  - primary key of a table can be a clustered index
  - secondary indexes refer to the primary key (rather than a heap file location).

- A compromise between a clustered and a nonclustered index is known as a ***covering index***.
  - it stores some table's columns within the index.

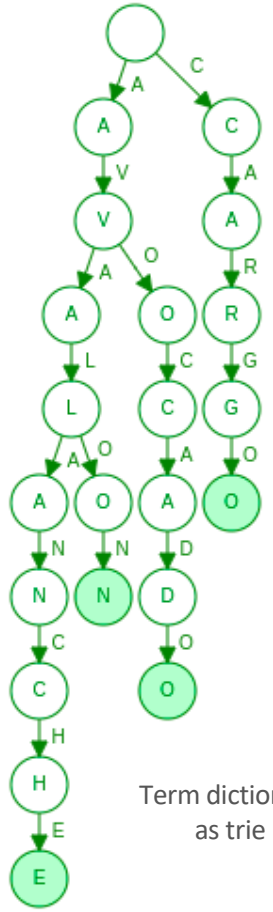
- These indexes can speed up reads, but
  - they require additional storage and overhead on writes.
  - databases also need additional effort to enforce transactional guarantees, because of the duplication.

Multi-column indexes

- The indexes discussed so far only map a single key to a value.
  - that is not sufficient if we need to query multiple columns of a table (or multiple fields in a document) simultaneously.
- The most common type of multi-column index is called a concatenated index.
  - Combines several fields into one key by appending one column to another.
- This is like a phone book, which provides an index from (*lastname, firstname*) to phone number.
  - This index can be used to find all the people with a particular last name, or all the people with a particular lastname-firstname combination.

Fuzzy indexes

- All the indexes discussed so far assume that we query for exact values of a key, or a range of values of a key with a sort order.
  - How to search for similar keys, such as misspelled words.
- Some data stores (e.g. Lucene) allow searching text within a certain edit distance.
  - Lucene uses a SSTable-like structure for its term dictionary.
  - This structure tells queries at which offset in the sorted file they need to look for a key.
  - This is finite state automaton over the characters in the keys, like a trie, which supports efficient search for words within a given edit distance.



Term dictionary as trie

Keeping everything in memory

- For many datasets, it is feasible to keep them entirely in memory.
  - Potentially distributed across several machines.
- This led the development of in-memory databases.
- Some in-memory key-value stores, such as Memcached, are intended for caching use only.
  - Acceptable for data to be lost if a machine is restarted.
- But others aim for durability.
  - by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.
  - Writing to disk also has operational advantages: files on disk can easily be backed up, inspected and analyzed by external utilities.

In-memory solutions

- VoltDB, MemSQL and Oracle TimesTen are in-memory databases with a relational model.
- RAMCloud is an open-source in-memory key-value store with durability (using a log-structured approach for the data in memory as well as the data on disk).
- Redis and Couchbase provide weak durability by writing to disk asynchronously.

In-memory solutions

- The performance advantage of in-memory databases is not due to the fact that they don't need to read from disk.
  - Even a disk-based storage engine may never need to read from disk if you have enough memory.
- Rather, they can be faster because they can avoid the **overheads of encoding in-memory** data structures in a form that can be written to disk.
  - Besides performance, they provide data models that are difficult to implement with disk-based indexes.
  - For example, Redis offers a database-like interface to various data structures such as priority queues and sets.
  - By keeping all data in memory, its implementation is comparatively simple.

Transaction Processing and Transaction Analytics

Online Transaction Processing (OLTP)

- In the early days, a write to the database typically corresponded to a commercial **transaction**:
  - making a sale, placing an order with a supplier, paying an employee's salary, etc.
- Despite databases expanding into many other areas, the term *transaction* nevertheless stuck.
  - ***Transaction processing*** means allowing clients to make low-latency reads and writes.
  - On the other hand, ***batch processing*** jobs run periodically, for example once per day.
- Applications are typically interactive (insert, update, search, etc.).
  - This access pattern became known as ***online transaction processing*** (**OLTP**).

Data analytics

- Databases also started being increasingly used for *data analytics*.
  - which has very different access patterns than OLTP.
- Usually, an analytic query needs to scan over a huge number of records and calculates aggregate statistics.
- Some examples:
  - What was the total revenue of each of our stores in January?
  - How much more bananas than usual did we sell during our latest promotion?
  - Which brand of baby food is most often purchased together with brand X diapers?

Online Analytic Processing (OLAP)

- Analytic queries are often written by business analysts.
  - and feed into reports that help a company's management make better decisions (business intelligence).
- To differentiate this pattern of using databases from transaction processing, this has been called

  online analytic processing (**OLAP**)

OLTP versus OLAP

Property	OLTP	OLAP
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

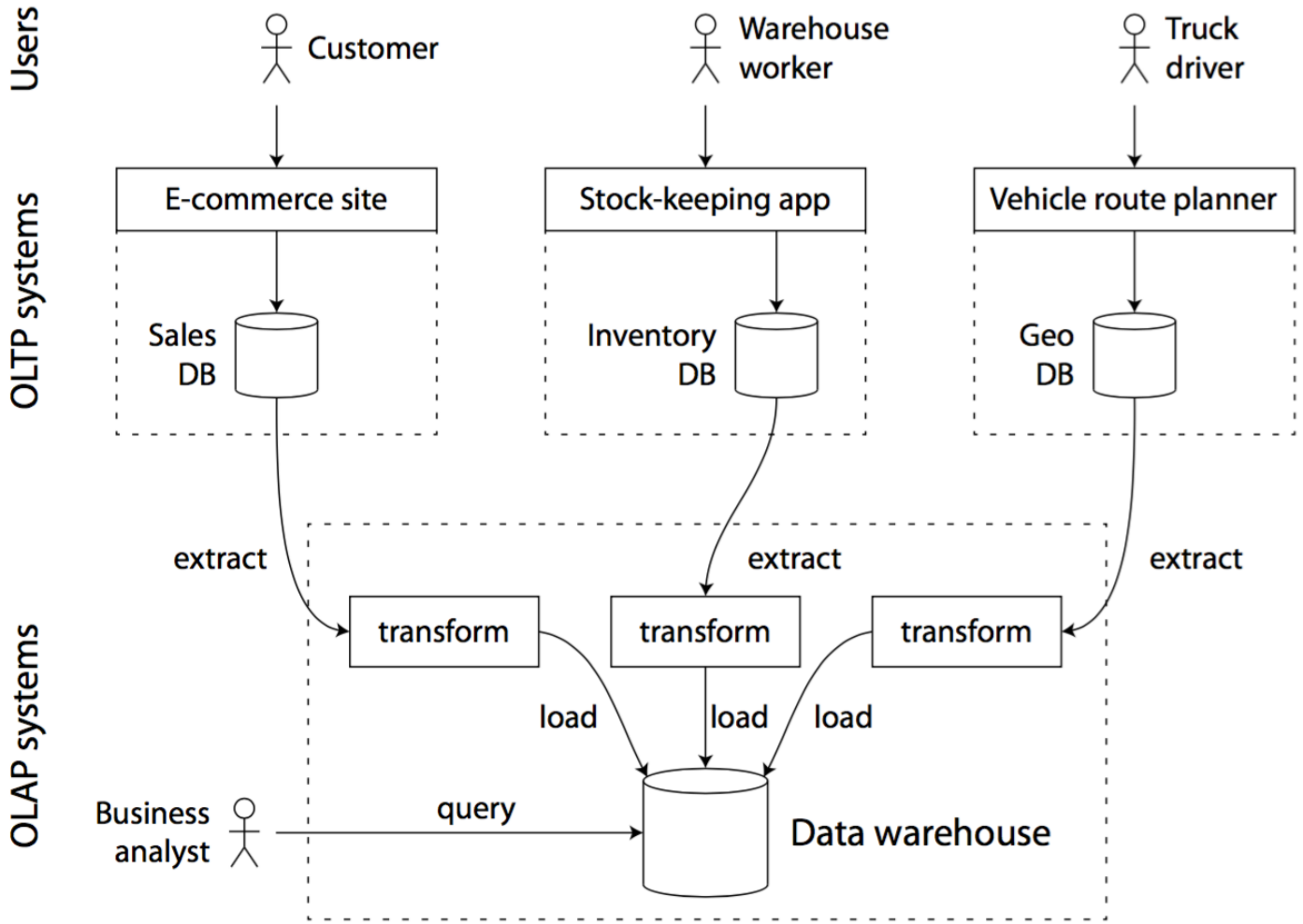
- RDBMS and SQL work well for OLTP-type queries as well as OLAP-type queries.
- However, OLAP normally uses a separate database from OLTP, i.e. a **data warehouse**.

Data warehousing

- **OLTP** systems are expected to be **highly available** and to process transactions with **low latency**.

- **Data warehouse** is a **separate database** that analysts can query without affecting OLTP operations.
  - It typically contains a **read-only copy** of the **data** in all the various OLTP systems in the company.
- Data is extracted from OLTP databases, transformed, cleaned, and loaded into the data warehouse.
  - This process of getting data into the warehouse is called ***Extract-Transform-Load*** (ETL).

Data warehousing – ETL



Why Separate Data Warehouse?

- Why combine OLTP and OLAP?
  - Data warehouses are commonly relational because SQL fits for analytic queries well.
  - Many OLTP tools (querying, visualization, etc.).
- Why separate?
  - **Different functions** and **different data**:
    - **missing data**: Decision support requires historical data which operational DBs do not typically maintain.
    - **data consolidation**: Decision support requires data consolidation (aggregation, summarization) from heterogeneous sources.
    - **data quality**: different sources typically use inconsistent data representations, codes, and formats which must be reconciled.
- Many vendors now support either transaction processing or analytics workloads.

Data Warehouses: some solutions

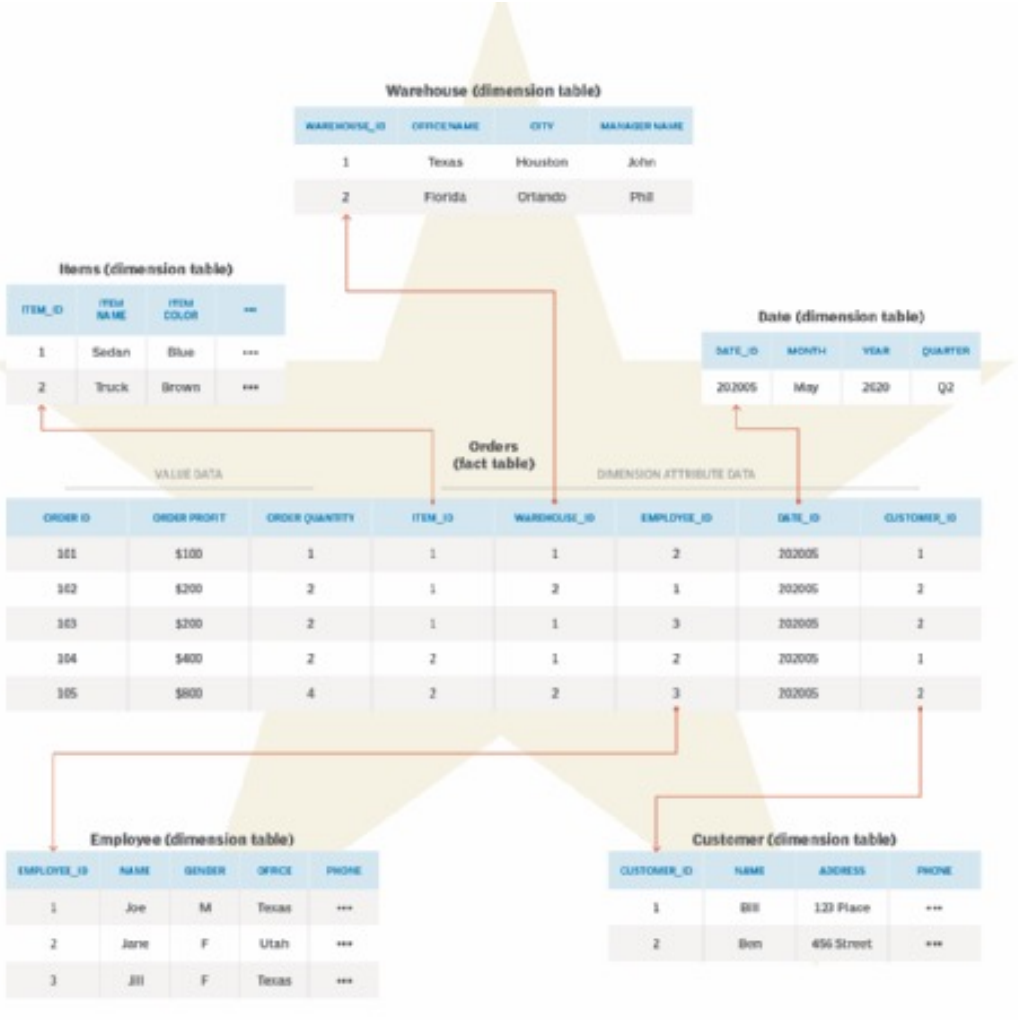
- Commercial
  - Amazon Redshift, IBM PureData, MS SQL Parallel Data Warehouse, Oracle Exadata, SAP Business Warehouse, Teradata, Vertica.
- Open source
  - Apache Hive, AMPLab's Shark, Cloudera Impala, Hortonworks Stinger, Facebook Presto, Apache Tajo, Apache Drill.

Schemas for analytics

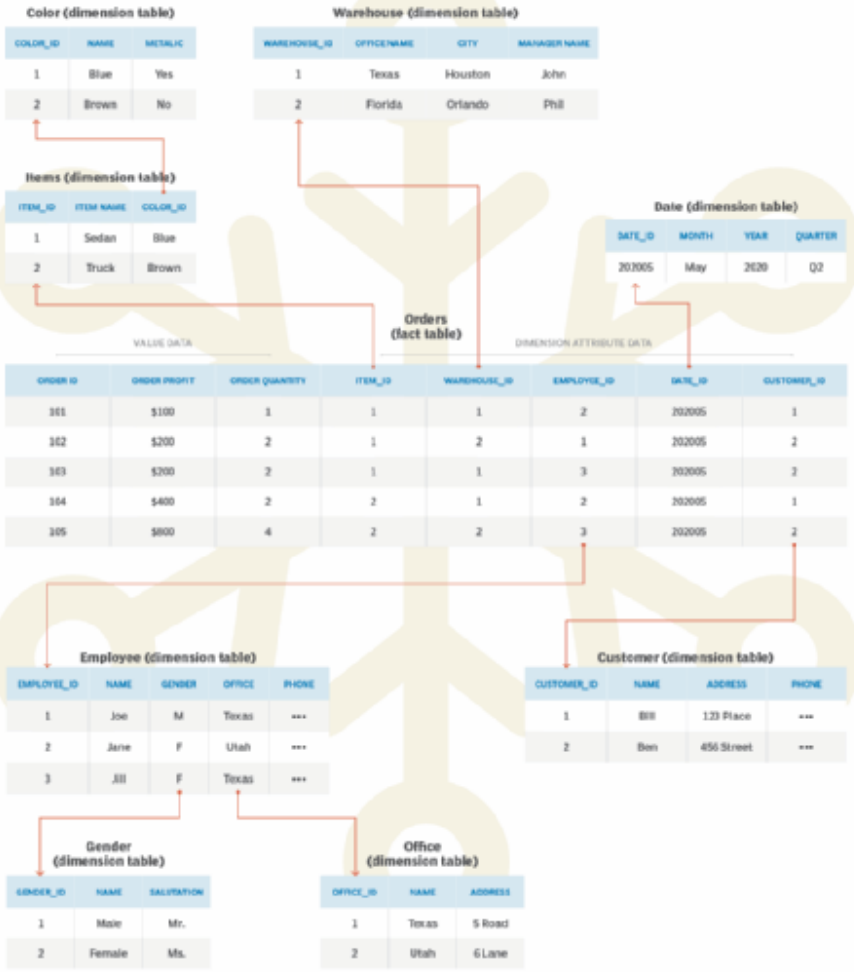
- Many data warehouses use a ***star schema*** (or dimensional modeling).
- The main entity is the ***fact table***.
  - Each fact table row represents an event at a particular time (a purchase, a page view, etc.).
  - Some columns are attributes (e.g. price). Others are references (foreign keys) to other tables (**dimension tables**).
- A variation of the star template is the ***snowflake schema***
  - The dimensions are broken down into sub-dimensions (more normalized but more complex to work with).

Star and Snowflake Schemas

Star Schema



Snowflake Schema



Schemas for analytics – Example

dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Star Schema

Querying problem

- A **fact table** is typically **huge** – can have trillions of rows and petabytes of data.
  - Storing and querying becomes a challenging problem.
- Fact tables are often over 100 columns wide.
- But… a typical data warehouse query only accesses a few of them at one time.

- Problem?

- Solution?

Querying problem

- Problem:
  - We need to read the entire table to process one single column.
  - OLTP databases are typically row-oriented: all the values from one row of a table are stored next to each other.

- Solution:
  - ***Column-oriented storage***

Column-oriented storage

- The idea:
  - don't store all the values from one row together, but store all the values from each column together instead.
  - If each column is stored in a separate file, a query only needs to read and parse those columns.

- The column-oriented storage layout relies on each column containing the rows in the same order.

- To reassemble an entire row *n*, we need to take all the *n* entries from each column file.

Column-oriented storage

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Column compression

- Column-oriented storage often lends itself very well to compression.
- The sequences of values for each column are often repetitive.
- Different compression techniques can be used depending on the data in the column.
  - Bitmap encoding: technique particularly effective in data warehouses
    - One binary vector per every term in the column
  - Run Length Encoding (RLE)
    - TSLA, TSLA, TSLA, TSLA, TSLA, SQ, SQ, SQ, APPL, AAPL **=> TSLA x 5, SQ x 3, AAPL x 2**

Column sorting

- Normally, columns are stored in the inserted order.
- However, we can choose another order, using one-column values.
  - It serves as an indexing mechanism.
- A second column can determine the sort order of any rows that have the same value in the first sorting column.
  - E.g., if **date_key** is the first sort key, and **product_sk** the second, all sales for the same product on the same day are grouped together.
- Another advantage of sorted order is that it can help with compression of columns.

Writing problem

- Column-oriented storage, compression and sorting all help to make OLAP queries faster.
- **What about writing?**
- **What's the best structure?**

Writing problem

- Column-oriented storage, compression and sorting all help to make OLAP queries faster.

- **What about writing?**
  - As rows are identified by their position within a column, the insertion has to update all columns consistently.
- **What's the best structure?**

Writing to column-oriented storage

- A good solution: LSM-trees (Log Structured Merge)

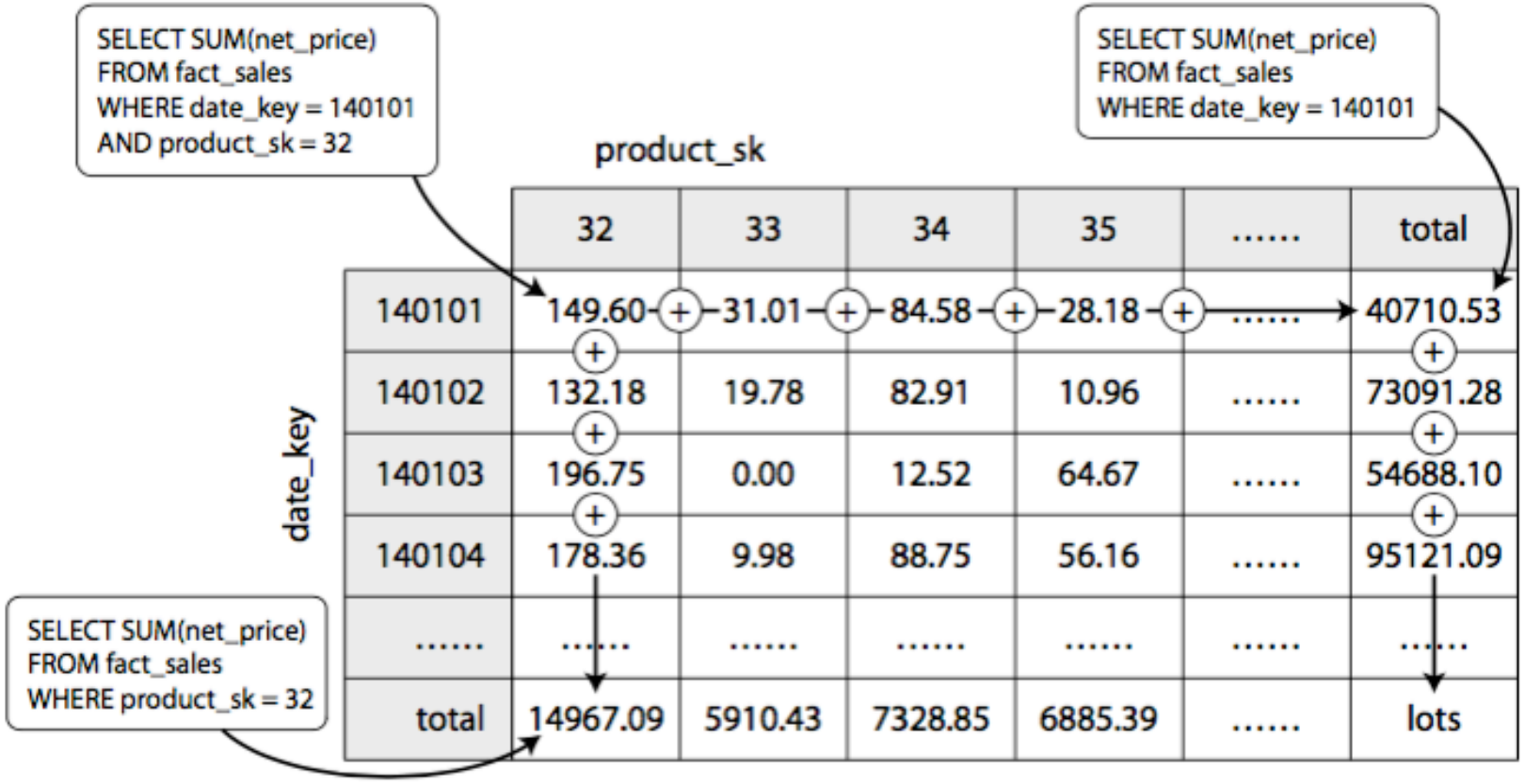
- All writes first go to an in-memory store, where they are added to a sorted structure, and prepared for writing to disk.
- When enough writes have accumulated, they are merged with the column files on disk, and written to new files in bulk.
- Queries need to examine both the column data on disk and in memory.

Materialized views

- Data warehouse queries often involve an aggregate function (COUNT, SUM, AVG, MIN, ...).
- Aggregates can be used by many different queries.
  - Repeating the data processing.
- Solution?
- Cache these aggregates in a ***materialized view***.
  - When the underlying data changes, a materialized view needs to be updated.
- A ***data cube*** or ***OLAP cube*** is a special case of a materialized view, which is a grid of aggregates grouped by different dimensions.

Data cube

- Example: Two-dimensional data cube, aggregating data by summing



	32	33	34	35	.....	total
140101	149.60	31.01	84.58	28.18	.....	40710.53
140102	132.18	19.78	82.91	10.96	.....	73091.28
140103	196.75	0.00	12.52	64.67	.....	54688.10
140104	178.36	9.98	88.75	56.16	.....	95121.09
.....	.....	.....	.....	.....	.....	.....
total	14967.09	5910.43	7328.85	6885.39	.....	lots

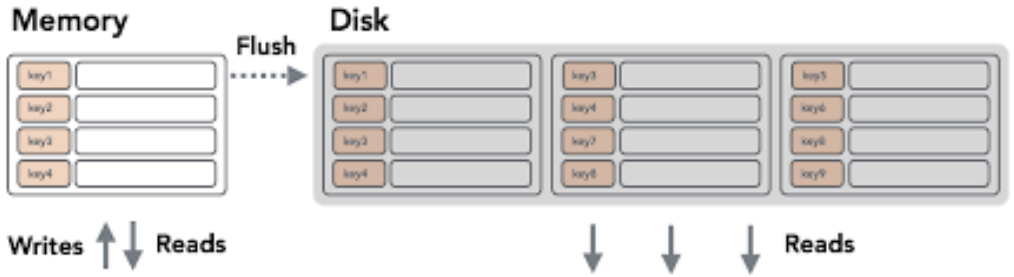
Summary

- Log-structured storage.
  - e.g. Append-only Log, Append-Log, Write-Ahead Log (WAL)

- Only allows appending to files and deleting obsolete files
  - Never updates a file that has been written.
- Enables higher write throughput.
- Read not .. so good.

Summary

- Log Structured **Merge Tree** storage (LSM-trees).
  - Buffered write, Multi-level storage
- They are immutable
  - SSTables are written on disk once and never updated.
  - Flushed tables can be accessed concurrently.
- They are write-optimized
  - writes are buffered and flushed on disk sequentially.
- Reads might require accessing multiple sources
- Compaction is required, as buffered writes are flushed on disk.



Summary

- Update-in-place storage
  - Indexing structures and databases that are optimized for keeping all data in memory.
  - Treats disk as fixed-size pages which can be overwritten.
- B-trees are the biggest example of this philosophy.
  - They are mutable, and do not require complete file rewrites or multisource merges.
  - They are read-optimized, i.e., do not require reading and merging from multiple sources.
  - Writes might trigger a cascade of node splits, making some write operations more expensive.
  - Concurrent access requires reader/writer isolation and involves chains of locks and latches.

Summary

- Storage engines fall into two broad categories:
  - optimized for transaction processing (OLTP).
  - optimized for analytics (OLAP).

- OLAP queries processing.
  - They require sequentially scanning of a large number of rows.
  - It is important to encode data very compactly, to minimize the amount of data that the query needs to read from disk.
- Column-oriented storage helps achieve this goal.

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.
- Petrov, Alex. "Algorithms Behind Modern Storage Systems." *Queue* 16, no. 2 (2018): 30.

Data Formats

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Outline

- We will look into several formats for **encoding data**

- **CSV** – Comma-Separated Values
- **XML** – Extensible Markup Language
- **JSON** – JavaScript Object Notation
- **BSON** – Binary JSON
- **RDF** – Resource Description Framework
- **Protocol Buffers**

Data encoding

- Software applications inevitably **change** over time.
  - In most cases, this also requires a change to data.
  - Old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time.
- For the system to continue running smoothly, we need to maintain compatibility in both directions:
  - **Backward compatibility** - newer code can read data that was written by older code.
  - **Forward compatibility** - older code can read data that was written by newer code. It requires older code to ignore additions made by a newer version of the code.

Data encoding

- Programs usually work with data …
  - In memory, data is kept in objects, structs, lists, arrays, hash tables, trees and so on.
  - Out of memory, to write data to a file, or send it over the network (i.e., a different sequence of bytes).
- The translation from the in-memory representation to a byte sequence is called **encoding** (also known as **serialization** or marshalling),
- The reverse is called **decoding** (parsing, **deserialization**, unmarshalling).

Language-specific formats

- Many programming languages come with built-in support for encoding in-memory objects into byte sequences.
  - Java (*Serializable*), Ruby (*Marshal*), Python (*pickle*), ...
- These encoding libraries are very convenient, but…
- … reading the data in another language is very difficult.
  - using such kind of encoding commits to the current programming language.
- So, it is a bad idea to use these built-in encoding for anything other than **transient purposes**.

Textual Formats

- Main advantage: human-readable
  - Examples: CSV, JSON, XML and RDF
- But they bring some **issues**:
- **Ambiguity** between a **number** and a **string**
  - JSON handles this, but not integers # floating-point, i.e., lacks to specify precision.
- CSV does **not** have any **schema**
  - It is up to the application to define the meaning of each row and column.

- Despite some flaws, JSON, XML and CSV are good enough for many purposes.

Binary Encoding

- Binary encoding
- More compact, faster to parse.
  - For a small dataset, the gains are negligible, but once you get into the terabytes, the choice of data format can have a big impact.
- Some binary encodings for JSON
  - MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, …
- But none of them is as widely adopted as the textual versions of JSON and XML.

CSV

- **CSV – Comma-Separated Values**
- XML – Extensible Markup Language
- JSON – JavaScript Object Notation
- BSON – Binary JSON
- RDF – Resource Description Framework
- Protocol Buffers

CSV – Comma-Separated Values

- Unfortunately, not fully standardized
  - Different field separators (commas, semicolons)
  - Different escaping sequences
  - No encoding information
- File extension: *.csv
- RFC 4180, RFC 7111
  - URI Fragment Identifiers for the text/csv Media Type
- Media type (MIME)
  - Content type: text/csv

Example

- Document
  - A header line (optional) + records

```
firstname,lastname,year
Ana,Katrina,1974
Paul,Machado,1956
Luis,Morais,1974
Sofia,Silvasky,1986
Maria,Marinova,1976
```

XML

- CSV – Comma-Separated Values
- **XML – Extensible Markup Language**
- JSON – JavaScript Object Notation
- BSON – Binary JSON
- RDF – Resource Description Framework
- Protocol Buffers

XML – Extensible Markup Language

- Representation of semi-structured data
  - + a family of related technologies, languages, specifications, …
- Derived from SGML, developed by W3C, since 1996
- Design goals
  - Simplicity, generality and usability across the Internet
- File extension: *.xml, content type: text/xml
- Versions: 1.0 and 1.1
- W3C recommendation
  - http://www.w3.org/TR/xml11/
- XML formats = particular languages
  - **XSD**, XSLT, XHTML, DocBook, ePUB, SVG, RSS, SOAP, …

Example

```
<?xml version="1.1" encoding="UTF-8"?>
<movie year="2007">
    <title>The Great Marnoto</title>
    <actors>
        <actor>
            <firstname>Jakim</firstname>
            <lastname>Dalmeida</lastname>
        </actor>
        <actor>
            <firstname>Sofia</firstname>
            <lastname>Ravara</lastname>
        </actor>
        </actors>
    <director>
        <firstname>Paulo</firstname>
        <lastname>Castanho</lastname>
    </director>
</movie>
```

Constructs – Element

- Marked using <opening> and </closing> tags
  - … or an abbreviated tag in case of empty <elements/>
- Each element can have a set of attributes
- Well-formedness is required
- Types of content
  - Empty content
  - Text content
  - Element content
    - Sequence of nested elements
  - Mixed content
    - Elements arbitrarily interleaved with text

JSON

- CSV – Comma-Separated Values
- XML – Extensible Markup Language
- **JSON – JavaScript Object Notation**
- BSON – Binary JSON
- RDF – Resource Description Framework
- Protocol Buffers

JSON – JavaScript Object Notation

- Open standard for data interchange
- Design goals
  - Simplicity: text-based, easy to read and write
  - Universality: object and array data structures
    - Supported by majority of modern programming languages
- Derived from JavaScript (but language independent)
- Started in 2002
- File extension: *.json
- Content type: application/json
- http://www.json.org/

JSON structure

- JSON is built on two structures:

- A **collection of name/value pairs**.
  - In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An **ordered list** of values.
  - In most languages, this is realized as an *array*, vector, list, or sequence.

Example

```
{
   "title":"The Great Marnoto",
   "year":2007,
   "actors":[
      {
         "firstname":"Jakim",
         "lastname":"Dalmeida"
      },
      {
         "firstname":"Sofia",
         "lastname":"Ravara"
      }
   ],
   "director":{
      "firstname":"Paulo",
      "lastname":"Castanho"
   }
}
```

Data Structure – Object

- Unordered collection of name-value pairs (properties)
  - Correspond to structures such as objects, records, structs, dictionaries, hash tables, keyed lists, associative arrays, …

- Example

```
{ "name" : ”Manuel Sliav", "year" : 2000 }
{ }
```

Data Structure – Array

- Ordered collection of values
  - Correspond to structures such as arrays, vectors, lists, sequences, …
- Values can be of different types, duplicate values are allowed
- Example

```
[ 2, 7, 7, 5 ]
[ "Some person", 1979, 77 ]
[ ]
```

Data Structure – Value

- Unicode string
  - Enclosed with double quotes
  - Backslash escaping sequences
    - Example: `"a \n b \" c \\ d"`
- Number
  - Decimal integers or floats
    - Examples: `1`, `-0.5`, `1.5e3`
- Nested object
- Nested array
- Boolean value: `true`, `false`
- Missing information: `null`

```
{
  "stuff": {
     "onetype": [
        {"id":1,"name":"John"},
        {"id":2,"name":"Don"}
     ],
     "othertype":
        {"id":2,"company":"ACME"}
  },
  "otherstuff": {
     "thing": [[1,42],[2,2]]
  }
}
```

BSON

- CSV – Comma-Separated Values
- XML – Extensible Markup Language
- JSON – JavaScript Object Notation
- **BSON – Binary JSON**
- RDF – Resource Description Framework
- Protocol Buffers

BSON – Binary JSON

- Binary-encoded serialization of JSON documents
  - Design characteristics: lightweight, **traversable**, efficient
  - **convenient storage of binary information**:
    - better suitable for exchanging **images** and **attachments**
  - designed for **fast in-memory manipulation**
  - **extra data types (then JSON)**:
    - double, date, byte array, JavaScript code, …
- Used by MongoDB
  - Document NoSQL database for JSON documents
  - Data storage and network transfer format
- File extension: *.bson
- http://bsonspec.org/

Example

- JSON

```
{
  "title" : "Marnoto",
  "year" : 2007
}
```

- BSON

```
              t  i  t  l  e                 M
2200 0000 0274 6974 6c65 0008 0000 004d
a  r  n  o  t  o        y  e  a  r     2007       // = 0x07d7
6172 6e6f 746f 0010 7965 6172 00d7 0700
0000
```

Document Structure

- **Document**
  - serialization of one JSON object or array
- JSON object is serialized directly
- JSON array is first transformed to a JSON object
  - Property names correspond to position numbers , e.g.
    `[ "Some", "Another" ] → { "0" : "Some", "1" : "Another" }`
- Structure
  - Document size (total number of bytes)
  - Sequence of elements
  - Terminating hexadecimal 00 byte

```
            t  i t  l e             M
2200 0000 0274 6974 6c65 0008 0000 004d
a r  n o  t o      y e  a r   2007  // = 0x07d7
6172 6e6f 746f 0010 7965 6172 00d7 0700
0000
```

Document Structure

- **Element**
  - serialization of one JSON property
- Structure
  - Type selector
    - 02 (string), 03 (object), 04 (array)
    - 01 (double), 10 (32-bit integer), 12 (64-bit integer)
    - 08 (boolean), 09 (datetime), 11 (timestamp)
    - 0A (null)
    - …
  - Property name
    - Unicode string terminated by 00
  - Property value

```
                  t  i  t  l  e                 M
2200 0000 0274 6974 6c65 0008 0000 004d
a r  n o  t o       y e  a r    2007  // = 0x07d7
6172 6e6f 746f 0010 7965 6172 00d7 0700
0000
```

RDF

- CSV – Comma-Separated Values
- XML – Extensible Markup Language
- JSON – JavaScript Object Notation
- BSON – Binary JSON
- **RDF – Resource Description Framework**
- Protocol Buffers

RDF – Resource Description Framework

- Language for representing information about resources in the World Wide Web
  - + a family of technologies, languages, specifications, …
  - Used in graph databases and in the context of the Semantic Web, Linked Data, …
- Developed by W3C
  - Started in 1997
- Versions: 1.0 and 1.1
- W3C recommendations
  - https://www.w3.org/TR/rdf11-concepts/
    - Concepts and Abstract Syntax
  - https://www.w3.org/TR/rdf11-mt/
    - Semantics

Statements

- RDF is based on the concept that every **resources** can have different **properties** which have **values**.
- Resource - Any real-world entity
  - **Referents** = resources identified by IRI (Internationalized Resource Identifier)
    - E.g. physical things, documents, abstract concepts, …

    `http://db.pt/movies/Marnoto`
    `http://db.pt/terms#actor`
    `mailto:somegirl@nowhere.com`
    `urn:issn:0167-6423`

  - **Values** = resources for literals
    - E.g. numbers, strings, …

Statements

- Example of a **statement** about a web page:

  `http://www.example.org/index.html` has an `author` whose name is `Pete Maravich`.

- A RDF statement is a **triple** that contains a:
  - **Resource**, the **subject** of a statement
  - **Property**, the **predicate** of a statement
  - **Value**, the **object** of a statement

- Several properties for this web page could be:

  `http://www.example.org/index.html` has an `author` whose name is `Pete Maravich`.

  `http://www.example.org/index.html` has a `language` which is `English`.

  `http://www.example.org/index.html` has a `title` which is `Example_Title`.

RDF example

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

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:edu="http://example.org/education#">

  <!- Resource Descriprion: João -->
  <rdf:Description rdf:about="http://example.org/people/Joao">
    <foaf:name>João Silva</foaf:name>
    <foaf:age>30</foaf:age>
    <foaf:mbox rdf:resource="mailto:joao.silva@example.org"/>
    <foaf:knows rdf:resource="http://example.org/people/Maria"/>
    <edu:studiedAt rdf:resource="http://example.org/university/ABC"/>
  </rdf:Description>

  <!-- Resource Descriprion: Universidade -->
  <rdf:Description rdf:about="http://example.org/university/ABC">
    <foaf:name>Universidade ABC</foaf:name>
    <foaf:location>Lisboa, Portugal</foaf:location>
  </rdf:Description>
</rdf:RDF>
```

Resource

Value

Property

Resource

Value

Serialization approaches

- RDF/XML notation
  - XML syntax for RDF (.rdf, .rdfs, .owl, .xml)
  - https://www.w3.org/TR/rdf-syntax-grammar/
- Turtle notation (Terse RDF Triple Language)
  - .ttl extension
  - https://www.w3.org/TR/turtle/
- N-Triples notation
  - .nt extension
  - https://www.w3.org/TR/n-triples/
- JSON-LD notation
  - JSON-based serialization for Linked Data
  - https://www.w3.org/TR/json-ld/

Protocol Buffers

- CSV – Comma-Separated Values
- XML – Extensible Markup Language
- JSON – JavaScript Object Notation
- BSON – Binary JSON
- RDF – Resource Description Framework
- **Protocol Buffers**

Protocol Buffers

- Protocol Buffers is a **binary** encoding library that require a **schema** for any data that is encoded.
- Extensible mechanism for serializing structured data
  - Used in communication protocols, data storage, …
- Developed (and widely used) by Google
  - Mostly for server-side communication
- Design goals
  - Language-neutral, platform-neutral
  - **Small**, **fast**, **simple**
- File extension: *.proto
- https://developers.google.com/protocol-buffers/

Protocol Buffers

- Intended usage
  - Schema creation
    → automatic source code generation
      → sending messages between applications
- Components
  - Interface description language
  - Source code generator (`protoc` compiler)
  - Supported languages
    - Official: C++, C#, Java, Python, Ruby …
    - 3rd party: Perl, PHP, Scala, …
  - Binary serialization format
  - Compact, not self-describing

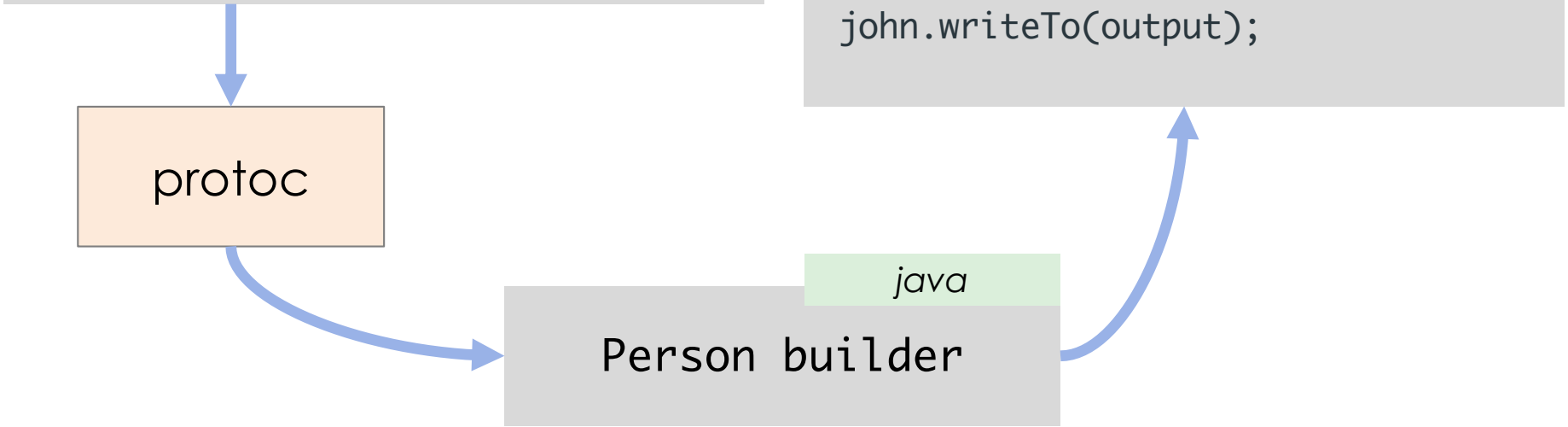
Schema: encoding examples

```
message Person {
    required string  user_name = 1;
    optional int64   favorite_number = 2;
    repeated string  interests = 3;
}
```

```
syntax = "proto3";
message Actor {
    string firstname = 1;
    string lastname = 2;
}
message Movie {
    string title = 1;
    int32 year = 16;
    repeated Actor actors = 17;
    enum Genre {
        UNKNOWN = 0;
        COMEDY = 1;
    }
    repeated Genre genres = 2048;
}
```

Example: ProtoBuf to Java

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```



java

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new
FileOutputStream(args[0]);
john.writeTo(output);
```

Summary

Data encoding formats:

- **Programming-language-specific**
  - restricted to a single programming language.
- **Textual** formats
  - widespread, and its compatibility depends on the use.
  - Somewhat vague about datatypes, namely numbers and binary strings.
- **Binary** schema-driven formats
  - More compact and efficient encoding, with clearly defined forward and backward compatibility semantics.
  - The schema can be useful for documentation and code generation in statically typed languages.

Summary

Data formats

- **Relational**: CSV
- **Tree**: XML, JSON
- **Graph**: RDF
- **Binary**: BSON, Protocol Buffers

Other binary formats

- **Avro**
  - Apache
- **Thrift**
  - Facebook + Apache
- MessagePack

… and many others

- A good comparison is available at:
  - https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.

Key-Value Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Outline

- Key-value stores
  - General principles
- Riak, Redis
  - Characteristics
  - Main Features
  - Use Cases

Key-Value Databases

- Key value stores are the simplest of NOSQL types
  - consisting only of a unique key and a bucket containing any data you wish to store there.
- Key-value pairs
  - Key (id, identifier, primary key) – usually a string.
  - Value: can be anything (text, structure, image, etc.) – a black box for the database system.
- The content of the bucket can be literally anything
  - But unstructured or semi-structured data are the most common.
- The buckets can hold quite large entries including BLOBs (Basic Large Objects).
- KVs are row based systems designed for efficiency.

Key-Value Databases – Advantages

- Highly fault tolerant – always available.
- Schema-less offers easier upgrade path for changing data requirements
  - (Document stores provide even greater flexibility).
- Efficient at retrieving information about a particular object (bucket) with a minimum of disc operations.
- Very simple data model. Very fast to set up and deploy.
- Great at scaling horizontally across hundreds or thousands of servers.

Key-Value Databases – Advantages

- No requirement for SQL queries, indexes, triggers, stored procedures, temporary tables, forms, views, or the other technical overheads of RDBMS.
- Very high data ingest rates.
  - Favors write once, read many applications.
- Powerful offline reporting with very large data sets.
- Some vendors are offering advanced forms of KVs that approach the capabilities of document stores or column oriented stores.

Key-Value Databases – Disadvantages

- Not suitable for complex applications.
- Not efficient at updating records where only a portion of a bucket is to be updated.
- Not efficient at retrieving limited data from specific records.
  - For example, in an employee database returning only records of employees making between $40K and $60K.
- As the volume of data increases maintaining unique values as keys becomes more difficult
  - Some more complexity in generating character strings that will remain unique over a large set of keys.
- Generally needs to read all the records in a bucket or you may need to construct secondary indexes.

Key-Value Databases

- **Suitable** use cases
  - Session data, user profiles, user preferences, shopping carts, …
  - Create ever-growing datasets that are rarely accessed but grow over time. (Caching)
  - Where write performance is your highest priority.
- **When not** to use
  - Relationships among entities
  - Queries requiring access to the content of the value part
  - Set operations involving multiple key-value pairs

Key-Value Databases

Key Management

- How the keys should actually be designed?

- **Manually assigned keys**
  - Real-world natural identifiers
  - E.g. e-mail addresses, login names, …
- **Automatically generated keys**
  - Auto-increment integers
    - Not suitable in peer-to-peer architectures!
  - More complex keys generated by algorithms
    - Keys composed from multiple components such as time stamps, cluster node identifiers, …
    - Used in practice

Query Patterns

- Basic **CRUD operations**
  - Only when a key is provided
  - The knowledge of the keys is essential
  - It might even be difficult for a particular database system to provide a list of all the available keys!
- **No searching by value**
  - But we could instruct the database how to parse the values
  - … so that we can fetch the intended search criteria
  - … and store the references within index structures
- **Batch / sequential processing**
  - MapReduce

Other Functionality

- **Expiration** of key-value pairs
  - After a certain interval of time key-value pairs are automatically removed from the database
  - Useful for user sessions, shopping carts etc.
- **Collections** of values
  - We can store not only ordinary values, but also their collections such as ordered lists, unordered sets etc.
- **Links** between key-value pairs
  - Values can mutually be interconnected via links
  - These links can be traversed when querying

- Particular functionality depends on the store.

Riak Key-Value Store

RiakKV

- Developed by Basho Technologies
  - http://basho.com/products/riak-kv/
  - Implemented in Erlang
  - Initial release in 2009
  - Operating system: Linux, Mac OS X, … (not Windows)

- Open source, incremental scalability, high availability, operational simplicity, decentralized design, automatic data distribution, advanced replication, fault tolerance, …
- General-purpose, concurrent, garbage-collected programming language and runtime system

Data Model

- Instance (→ bucket types) → buckets → objects

- **Bucket** = collection of objects (logical, not physical collection)
  - Each object must have a unique key
  - Various properties are set at the level of buckets
    - E.g. default replication factor, read / write quora, …
- **Object** = key-value pair
  - Key is a Unicode string
  - Value can be anything (text, binary object, image, …)
    Each object is also associated with metadata
    - E.g. its content type (text/plain, image/jpeg, …),
    - and other internal metadata as well

Data Model

- How buckets, keys and values should be designed?

- Complex objects containing various kinds of data
  - E.g. one key-value pair holding information about all the actors and movies at the same time
- Buckets with different kinds of objects
  - E.g. distinct objects for actors and movies, but all in one bucket
  - Structured naming convention for keys might help
    - E.g. actor_trojan, movie_medvidek
- Separate buckets for different kinds of objects
  - E.g. one bucket for actors, one for movies

Riak Operations

- Basic CRUD operations
  - Create: POST or PUT methods
    - Inserts a key-value pair into a given bucket
    - Key is specified manually, or will be generated automatically
  - Read: GET method
    - Retrieves a key-value pair from a given bucket
  - Update: PUT method
    - Updates a key-value pair in a given bucket
  - Delete: DELETE method
    - Removes a key-value pair from a given bucket
- Extended functionality
  - Links – relationships between objects and their traversal
  - Search 2.0 – full-text queries accessing values of objects
  - MapReduce

Riak Usage: API

- HTTP API
  - All the user requests are submitted as HTTP requests with an appropriately selected method and specifically constructed URL, headers, and data.
  - Example
    - GET /types/<type>/buckets/<bucket>/keys/<key>
- Protocol Buffers API
- Erlang API
- Client libraries for a variety of programming languages
  - Official: Java, Ruby, Python, C#, PHP, …
  - Community: C, C++, Haskell, Perl, Python, Scala, …

Redis
(REmote DIctionary Service)

Redis Overview

- Redis
  - **In-memory** key-value store
  - Open source, master-slave replication architecture, sharding, high availability, various persistence levels, …
- Developed by Redis Labs
- Implemented in C
- First release in 2009
- Available at http://redis.io/

Redis Overview

- Functionality
  - Standard key-value store
  - Support for structured values (e.g. lists, sets, …)
  - Time-to-live
  - Transactions

- Redis is not just a plain key-value store, but a data structures server, supporting different kind of values.

- Real-world users
  - Twitter, GitHub, Pinterest, StackOverflow, Flicker, …

Data Model

- Structure
  - Instance → databases → objects

- **Database** = collection of objects
  - Databases do not have names, but integer identifiers [0-15]
- **Object** = key-value pair
  - Key is a string (i.e. any binary data)
  - Values can be…
    - Atomic: string
    - Structured: list, set, ordered set, hash

Data Types

- **String**
  - The only atomic data type
  - May contain any binary data
    (e.g. string, integer counter, PNG image, …)
  - Maximal allowed size is 512 MB
- **List**
  - Ordered collection of strings
  - Elements should preferably be read / written at the head / tail

Data Types

- **Set**
  - Unordered collection of strings
  - Duplicate values are not allowed
- **Sorted set**
  - Ordered collection of strings
  - The order is given by a score (floating number value) associated with each element (from the smallest to the greatest score)
- **Hash**
  - Associative map between string fields and string values
  - Field names have to be mutually distinct

Interface

- **Command line client**
  - redis-cli
- Two modes are available…
- Basic
  - Commands are passed as standard command line arguments
    - E.g. redis-cli PING
  - Batch processing is possible as well
    - E.g. cat script.txt | redis-cli
- Interactive
  - Users type database commands at the prompt redis-cli
- **RESP** (REdis Serialization Protocol)

Basic Commands

- **SELECT [0-15]**
  - Select a database (default is 0)
- **SET** key value
  - inserts / replaces a given string
- **GET** key
  - returns a given string
- **MOVE [key] [db]**
  - move key to another database
- **DBSIZE**
- **HELP** command
  - Provides basic information about Redis commands

Basic Commands

- **FLUSHDB**
  - Deletes all the keys of the currently selected database
- **FLUSHALL**
  - delete all the keys in all the databases
- **SAVE / BGSAVE**
  - Saves the current dataset directly / on background
- **MONITOR**
  - what's going on against your redis datastore (check also redis-stat)

Strings Operations

- **STRLEN** key
  - returns a string length
- **APPEND** key value
  - appends a value at the end of a string
- **GETRANGE** key start end
  - returns a substring Both the boundaries are considered to be inclusive
  - Positions start at 0;
  - Negative offsets for positions starting at the end
- **SETRANGE** key offset value
  - replaces a substring
  - Binary 0 are padded when the original string is not long enough

Counter Operations

- **INCR** key
- **DECR** key
  - Increments / decrements a value by 1
- **INCRBY** key increment
- **DECRBY** key increment
  - Increments / decrements a value by a given amount

Handling Keys

- **EXISTS** key
  - determines whether a key exists
- **KEYS** pattern
  - finds all the keys matching a pattern (*, ?, …)
  - E.g. KEYS *
- **DEL** key …
  - removes a given object / objects
- **RENAME** key newkey
  - changes the key of a given object
- **TYPE** key – determines the type of a given object
  - Types: string, list, set, zset and hash

Volatile Keys

- Keys with limited time to live
  - When a specified timeout elapses, a given object is removed
  - Works with any data type

- **EXPIRE** key seconds
  - Sets a timeout for a given object, i.e. makes the object volatile
  - Can be called repeatedly to change the timeout
- **TTL** key
  - Returns the remaining time to live for a key
- **PERSIST** key
  - Removes the existing timeout

Complex Datatypes

- Redis' popularity comes mostly by supporting:
  - lists, hashes, sets, and sorted sets
- These collection can contain up to 2^32 elements (more than 4 billion) per key.
- Commands follow a good pattern.
  - Set commands begin with S,
  - Hashes with H
  - Sorted sets with Z.
  - List commands generally start with either an L (for left) or an R (for right),
    - depending on the direction of the operation (such as LPUSH).

Lists

- **LPUSH** key value
- **RPUSH** key value
  - Adds a new element to the head / tail (Left / Right)

- **LINSERT** key BEFORE | AFTER pivot value
  - Inserts an element before / after another one

- **LPOP** key
- **RPOP** key
  - Removes and returns the first / last element (Left / Right)

Lists

- **LINDEX** key index
  - gets an element by its index
    - The first item is at position 0;
- **LRANGE** key start stop
  - gets a range of elements
- **LREM** key count value
  - Removes a "count" number of elements equals to value
  - count:
    - Positive / negative = moving from head to tail / tail to head
    - 0 = all the items equals to value are removed
- **LLEN** key
  - gets the length of a list

Sets

- **SADD** key value …
  - Adds an element / elements into a set
- **SREM** key value …
  - Removes an element / elements from a set
- **SISMEMBER** key value
  - Determines whether a set contains a given element
- **SMEMBERS** key
  - gets all the elements of a set
- **SCARD** key
  - gets the number of elements in a set
- **SUNION / SINTER / SDIFF** key …
  - Calculates and returns a set union / intersection / difference of two or more sets

Hashes

- **HSET** key field value
  - sets the value of a hash field
- **HGET** key field
  - gets the value of a hash field

Batch alternatives

- **HMSET** key field value … …
  - Sets values of multiple fields of a given hash
- **HMGET** key field …
  - Gets values of multiple fields of a given hash

Hashes

- **HEXISTS** key field
  - determines whether a given field exists
- **HGETALL** key
  - gets all the fields and values
- **HKEYS** key
  - gets all the fields in a given hash
- **HVALS** key
  - gets all the values in a given hash
- **HDEL** key field
  - Removes a given field / fields from a hash
- **HLEN** key
  - returns the number of fields in a given hash

Sorted Sets

Basic operations

- **ZADD** key score value … …
  - Inserts one element / multiple elements into a sorted set
- **ZREM** key value …
  - Removes one element / multiple elements from sorted set

Working with score

- **ZSCORE** key value
  - Gets the score associated with a given element
- **ZINCRBY** key increment value
  - Increments the score of a given element

Sorted Sets

Retrieval of elements

- **ZRANGE** key start stop
  - Returns all the elements within a given range based on positions
- **ZRANGEBYSCORE** key min max
  - Returns the elements within a given range based on scores

Other operations

- **ZCARD** key
  - Gets the overall number of all elements
- **ZCOUNT** key min max
  - Counts elements within a given range based on score

Geospatial field operations

- **GEOADD** key longitude latitude member …
  - Adds the specified geospatial items (latitude, longitude, name) to the specified key.
- **GEODIST** key member1 member2 …
  - Return the distance between two members.
- **GEOHASH** key member …
  - Return Geohash string (compatible with geohash.org)
- **GEOPOS** key member …
  - Return the positions (longitude,latitude) of all the specified members.
- **GEORADIUS** key longitude latitude radius …
  - Return the members which are within the radius of the location.

RDBMS to Redis: Data Modeling

- Employees: Table

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

- In general, any RDBMS table can be represented in a key-value schema as follows:

$table_name:$primary_key_value:$attribute_name = $value

RDBMS to Redis: Data Modeling

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

employee:1:first_name = "John"
employee:1:last_name = "Doe"
employee:1:address = "New York"

employee:2:first_name = "Benjamin"
employee:2:last_name = "Button"
employee:2:address = "Chicago"

employee:3:first_name = "Mycroft"
employee:3:last_name = "Holmes"
employee:3:address = "London"

References

- Commands
  - http://redis.io/commands
- Documentation
  - http://redis.io/documentation
- Data types
  - http://redis.io/topics/data-types

Document Databases

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Outline

- Document databases
  - General introduction
  - Relational versus Document stores
- MongoDB
  - Data model
  - CRUD operations
    - Insert, Update, Remove
    - Find: projection, selection, modifiers
  - Index structures
- Java driver

Storage example: *Linkedin* in RDMS

http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

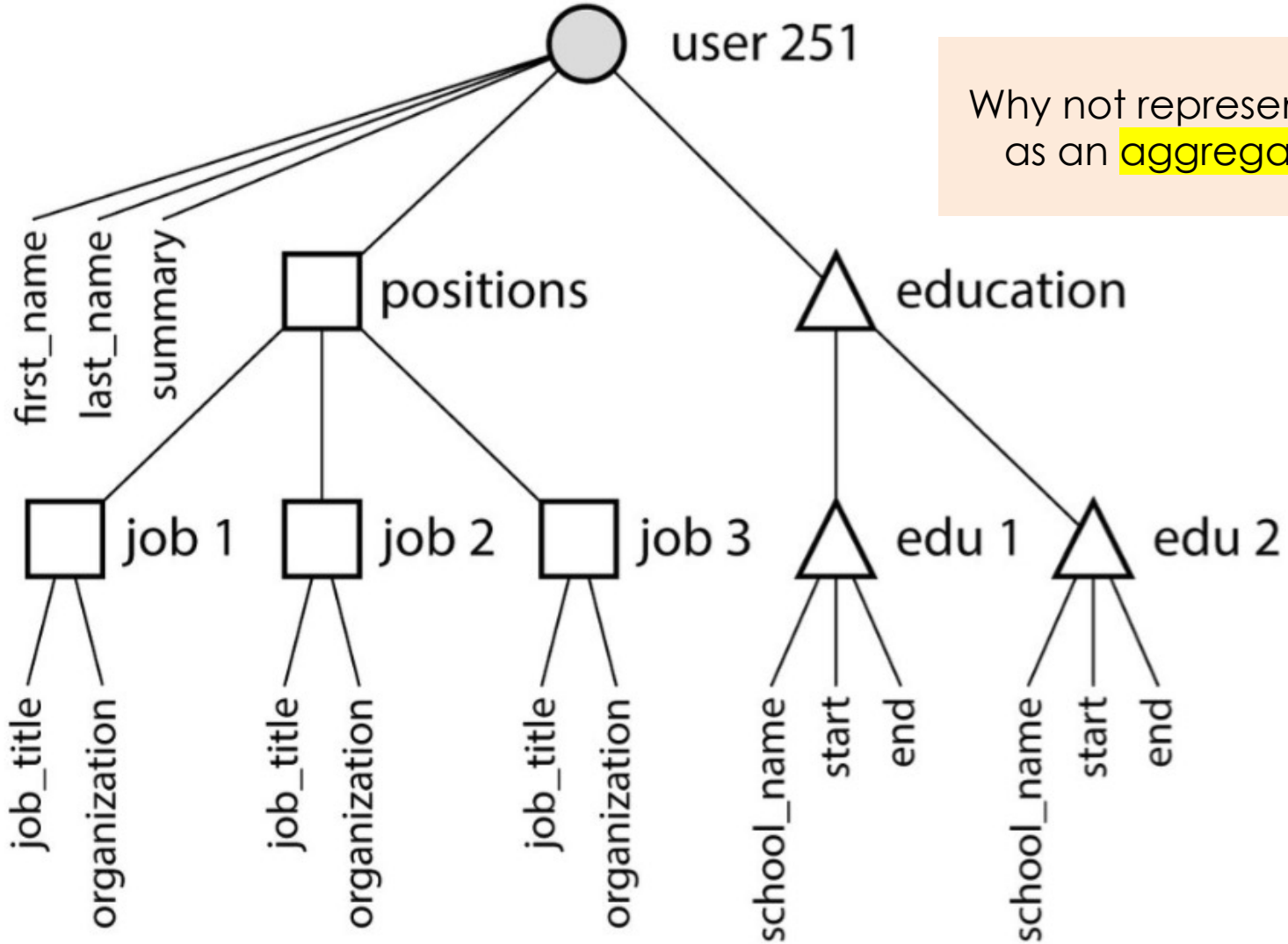
education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

One-to-Many relations



Why not represent this as an aggregate?

JSON representation


```
{
    "user_id": 251,
    "first_name": "Bill",
    "last_name": "Gates",
    "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
    "region_id": "us:91",
    "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
    "positions": [
        {"job_title": "Co-chair", "organization": "B&M Gates Foundation"},
        {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
    ],
    "education": [
        {"school_name": "Harvard University", "start": 1973, "end": 1975},
        {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
    ]
}
```

Document vs. Relational databases

- The main arguments in favour of the document data model are:
  - simpler application code, schema flexibility, and better performance due to locality.

- May be used for data with a document-like structure,
  - i.e. a tree of one-to-many relationships, where typically the entire tree is loaded at once.
  - When splitting a document-like structure into multiple tables can lead to unnecessarily complicated application code.
    - Event logging, content management systems, blogs, web analytics, e-commerce applications, …

Documents for schema flexibility

- The *schemaless* approach is **advantageous if the data is heterogeneous**
  - i.e. the items in the collection don't all have the same structure.
- For example, because:
  - there are many different types of objects, and it is not practical to put each type of object in its own table, or
  - Data structure determined by external systems, over which we have no control, and which may change at any time.
- In situations like these, a schema may hurt more

Documents for schema flexibility

- A document is usually stored as a **single continuous string**, encoded as JSON, XML or a binary variant thereof (such as MongoDB's BSON).
  - If one needs to access the entire document, there is a performance advantage to this storage locality.
- The **locality advantage** only applies if you need large parts of the document at the same time.
  - The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents.
  - On updates to a document, the entire document usually needs to be re-written.
- Recommended to keep documents small.

Document vs. Relational databases

- When not to use Document?
  - Set operations involving multiple documents
  - Design of document structure is constantly changing
    - i.e. when the required level of granularity would outbalance the advantages of aggregates
- If the application does use **many-to-many relationships**, the document model becomes less appealing (no joins).
  - We may denormalize the database, or joins can be emulated in application code by making multiple requests to the database.
  - But… the problems of managing denormalization and joins may be greater than the problem of object-relational mismatch.

Convergence of document and relational databases

- Most **relational** database systems also started supporting XML and JSON
  - i.e. functions to create/update documents, and the ability to index and query inside documents.
  - This allows applications to use data models similar to document databases.

- On the **document** database side..
  - Several solutions have also evolving to provide SQL-like experience in document databases (MongoDB Atlas, RethinkDB, Knomi, ObjectRocket, …)

Document Stores

- Data model
  - **Documents**
    - Self-describing
    - Hierarchical tree structures (JSON, XML, …) – Scalar values, maps, lists, sets, nested documents, …
    - Identified by a unique identifier (key, …)
  - **Collections** – a set of documents
- Query patterns (CRUD)
  - **C**reate, **U**pdate or **D**elete a document
  - **R**ead/retrieve documents according to complex query conditions
  - Extended key-value stores where the value part is examinable

Document Stores

- Document
  - MongoDB, Couchbase, CouchDB,
  - RethinkDB, RavenDB,
  - Google Cloud Firestore

- Multi-model
  - MarkLogic, OrientDB, ArangoDB
  - Amazon DynamoDB,
  - Microsoft Azure Cosmos DB,

  - *… many others*

DB-Engines Ranking of Document Stores

☐ include secondary database models

58 systems in ranking, October 2024

Rank Oct 2024	Rank Sep 2024	Rank Oct 2023	DBMS	Database Model	Score Oct 2024	Score Sep 2024	Score Oct 2023
1.	1.	1.	MongoDB	Document, Multi-model	405.21	-5.02	-26.21
2.	2.	↑ 3.	Databricks	Multi-model	85.60	+1.35	+9.78
3.	3.	↓ 2.	Amazon DynamoDB	Multi-model	71.85	+1.78	-9.07
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	24.50	-0.47	-9.80
5.	5.	5.	Couchbase	Document, Multi-model	17.10	+0.36	-5.31
6.	6.	6.	Firebase Realtime Database	Document	13.59	-0.01	-3.89
7.	7.	7.	CouchDB	Document, Multi-model	7.53	+0.07	-5.91
8.	8.	↑ 9.	Realm	Document	7.05	-0.13	-1.16
9.	9.	↓ 8.	Google Cloud Firestore	Document	6.46	-0.17	-3.86
10.	10.	↑ 11.	Aerospike	Multi-model	5.57	+0.41	-0.86
11.	11.	↓ 10.	MarkLogic	Multi-model	4.38	+0.22	-3.82
12.	12.	12.	Google Cloud Datastore	Document	4.04	-0.09	-1.55
13.	13.	13.	Virtuoso	Multi-model	3.91	-0.08	-1.51
14.	14.	↑ 15.	ArangoDB	Multi-model	3.44	+0.13	-0.83
15.	15.	↑ 16.	Oracle NoSQL	Multi-model	3.27	+0.20	-0.67

https://db-engines.com/en/ranking/document+store

MongoDB Document Database

MongoDB

- JSON document database
  - https://www.mongodb.com/
- Features
  - Open source, high availability, eventual consistency, automatic sharding, master-slave replication, automatic failover, secondary indexes, …
- Developed by MongoDB, Inc.
- Implemented in C++, C, and JavaScript
- Operating systems: Windows, Linux, Mac OS X, …
- Initial release in 2009

Data Model

- Structure
  - Instance → databases → collections → documents

- **Database**
  - Set of Collections
- **Collection**
  - Set of Documents, usually of a similar structure
- **Document**
  - MongoDB document = one JSON object
  - Internally stored as BSON
  - Each document…
    - belongs to exactly one collection
    - has a unique identifier `_id`

```
{
  name: "martin",
  age: 22,
  interests: [ sports, CBD ]
}
```

Example

Collection redwine

```
{
  _id: "1",
  name: "Cartuxa",
  year: 2012
}
{
  _id: "2",
  name: "Evel",
  year: 2010
}
{
  _id: "3",
  name: ”EA",
  year: 2016
}
```

Query statement

Wines older than 2014 and later, sorted by these titles in descending order

```
db.redwine.find(
{ year: { $lt: 2014} },
{ _id: false, name: true } )
.sort({ name: -1 })
```

Query result

```
{ "name" : "Evel" }
{ "name" : "Cartuxa" }
```

Data Model – Primary Keys

- **_id** is reserved for a primary key
  - Unique within a collection
  - Immutable (cannot be changed once assigned)
  - Can be of any type other than an array
- Possible values
  - Natural identifier (e.g., a key)
    - Must be unique!
  - UUID (Universally unique identifier)
    - 16-byte number (ISO/IEC 11578:1996, RFC 4122)
  - **ObjectId**
    - Special 12-byte BSON type (default option)
    - Small, likely unique, fast to generate, ordered, based on a timestamp, machine id, process id, and a process-local counter

Data Model – Denormalized

- **Embedded documents**
  - Related data in a single structure with subdocuments
  - Suitable for one-to-one or one-to-many relationships
  - Brings ability to read / write related data in a single operation
    - i.e., better performance, less queries need to be issued

```
> db.redwine.insertOne( {
   winepack: "Dinner",
   bottles: [
      { name: "Cartuxa", year: 2012 },
      { name: "Evel", year: 2010 },
      { name: "EA", year: 2016 }
      ]
   })
```

Data Model – Normalized

References

- Directed links between documents, expressed via identifiers
  - Idea analogous to foreign keys in relational databases
  - Suitable for **many-to-many relationships**
  - Embedding in this case would result in data duplication
- References provide more flexibility than embedding
  - But follow up queries are needed

```
> db.redwine.insertOne( {
    winepack: "Dinner",
    bottles: [
        { "$id" : "1" },
        { "$id" : "3" }
        ]
    })
```

```
{ _id: "1",
   name: "Cartuxa",
   year: 2012 }
{ _id: "2",
   name: "Evel",
   year: 2010 }
{ _id: "3",
   name: ”EA",
   year: 2016 }
```

  - The $id field contains the value of the _id field in the referenced document.

Tools

- MongoDB Atlas – cloud server
  - https://www.mongodb.com/cloud/atlas
- Local installation
  - https://www.mongodb.com/try/download/community
  - Local server

```
$ mongod --dbpath <path to data directory>
```

- Mongo Shell (client)
  - https://www.mongodb.com/try/download/shell
  - interactive JavaScript interface to MongoDB.

```
$ mongosh
```

- Mongo CLI Database Tools
  - https://www.mongodb.com/try/download/database-tools
  - bsondump, dump, mongodump, mongoexport, mongofiles, mongoimport, mongooplog, mongoperf, mongoreplay, mongorestore, mongos, mongostat, mongotop

Query Language

- JavaScript commands
  - Each individual command is evaluated over exactly one collection
  - Queries return a cursor
    - Allows us to iterate over all the selected documents

- Query patterns
  - Basic CRUD operations
    - Accessing documents via identifiers or conditions on fields
  - Aggregations: MapReduce, pipelines, grouping

CRUD Operations

- Create
  - db.collection.**insertOne**()
  - db.collection.**insertMany**()
- Read
  - db.collection.**find**()
    - Finds documents based on filtering/projection/sorting conditions
- Update
  - db.collection.**updateOne**()
  - db.collection.**updateMany**()
- Delete
  - db.collection.**deleteOne**()
  - db.collection.**deleteMany**()

https://docs.mongodb.com/manual/crud/

Create – insert examples

```
> db.invoice.insertOne({ _id: 901, inv_no: "I001", inv_date: "20171010"
})
{ "acknowledged" : true, "insertedId" : 901 }

> db.orders.insertMany(
...     [
...       { _id: 15, ord_no: 2001, qty: 200, unit: "doz" },
...       { ord_no: 2005, qty: 320 },
...       { ord_no: 2008, qty: 250, rate:85 }
...     ]
... );
{
    "acknowledged" : true,
    "insertedIds" : [
        15,
        ObjectId("59b1a6d6935c2a0ca72c432a"),
        ObjectId("59b1a6d6935c2a0ca72c432b")
    ]
}
```

Read/query operation

```
> db.inventory.insertMany([
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);

> db.inventory.find( {} )  // SELECT * FROM inventory
{ "_id" : ObjectId("59b1b730935c2a0ca72c432c"), "item" : "journal", "qty" : 25,
"size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432d"), "item" : "notebook", "qty" : 50,
"size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432e"), "item" : "paper", "qty" : 100,
"size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432f"), "item" : "planner", "qty" : 75,
"size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c4330"), "item" : "postcard", "qty" : 45,
"size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

Selection

```
> db.inventory.find( { status: "D" } )
                    // SELECT * FROM inventory WHERE status = "D"

> db.inventory.find( { status: { $in: [ "A", "D" ] } } )
                    // SELECT * FROM inventory WHERE status in ("A", "D")

> db.inventory.find( { status: "A", qty: { $lt: 30 } } )
                    // SELECT * FROM inventory WHERE status = "A" AND qty < 30

> db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
                    // SELECT * FROM inventory WHERE status = "A" OR qty < 30

> db.inventory.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )     // SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Selection operators

- Comparison
  - **`$eq, $ne`**
    - Tests the actual field value for equality / inequality
  - **`$lt, $lte, $gte, $gt`**
    - Less than / less than or equal / greater than or equal / greater
  - **`$in`**
    - Equal to at least one of the provided values
  - **`$nin`**
    - Negation of `$in`
- Logical
  - **`$and, $or`**
  - **`$nor`**
    - returns all documents that fail to match both clauses.
  - **`$not`**

Selection operators

- Element operators
  - **`$exists`**
    - tests whether a given field exists / not exists
  - **`$type`**
    - selects documents if a field is of the specified type.

- Evaluation operators
  - **`$regex`**
    - tests whether the field value matches a regular expression (PCRE)
  - **`$text`**
    - performs text search (text index must exists)

Selection operators

- Array query operators
  - `$all`
    - Matches arrays that contain all elements specified in the query.
  - `$elemMatch`
    - Selects documents if an element in the array field matches all the specified $elemMatch conditions.
  - `$size`
    - Selects documents if the array field is a specified size.

Projection

```
// SELECT _id, item, status FROM inventory
> db.inventory.find( { } , { item: 1, status: 1 } )
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b2"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b3"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b4"), "item" : "paper", "status" : "D" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b5"), "item" : "planner", "status" : "D" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b6"), "item" : "postcard", "status" : "A" }

// SELECT item, status FROM inventory
> db.inventory.find( { } , { _id: 0, item: 1, status: 1 } ) // true or 1 is included
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }

> db.inventory.find( {} , { _id: 0, qty: 0, size: 0 } ) // false or 0 is excluded
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }
```

Modifiers (sort, limit, skip)

```
// SELECT _id, item, status FROM inventory ORDER BY status ASC
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).sort({ status: 1 })
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "postcard", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }

> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).sort({ status: -1 })
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "postcard", "status" : "A" }
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).limit(3)
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).skip(3)
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }
```

CRUD Operations – Update

- Syntax

```
db.collection.updateOne(filter, update, options)
db.collection.updateMany(filter, update, options)

db.collection.updateOne(
     <filter>,   // = selectors in find()
     <update>,   // modification to apply
     {            // optional …
          upsert: <boolean>,         // if no doc -> insert
          writeConcern: <document>, // ack of num of replicas
          collation: <document>     // language/type-specific rules
          ..
     }
)
```

- Update operators

  $set, $unset, $rename

https://docs.mongodb.com/manual/reference/delete-methods/

Update

```
> db.inventory.find({"item":"journal"}, {_id:0, size:0})
{ "item" : "journal", "qty" : 25, "status" : "A" }

> db.inventory.updateOne({"item":"journal"}, {$set: {"status":"B"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.inventory.find({"item":"journal"}, {_id:0, size:0})
{ "item" : "journal", "qty" : 25, "status" : "B" }

> db.inventory.updateOne({"item":"computer"},
    {$set: {"status":"C", qty:30 } },
    {upsert:true})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0,
"upsertedId" : ObjectId("59b2524f92403315277cbd8f") }

> db.inventory.find( {"item":"computer"} )
{ "_id" : ObjectId("59b2524f92403315277cbd8f"), "item" : "computer",
"status" : "C", "qty" : 30 }
```

Update

```
> db.inventory.updateMany({}, {$unset: { size:""}})
{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }

> db.inventory.find()
{ "_id" : ObjectId("59b1b730935c2a0ca72c432c"), "item" : "journal", "qty" :
25, "status" : "A" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432d"), "item" : "notebook", "qty"
: 50, "status" : "A" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432e"), "item" : "paper", "qty" :
100, "status" : "D" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c432f"), "item" : "planner", "qty" :
75, "status" : "D" }
{ "_id" : ObjectId("59b1b730935c2a0ca72c4330"), "item" : "postcard", "qty"
: 45, "status" : "A" }
```

CRUD Operations – Delete

- Syntax

```
db.collection.deleteOne(filter, options)
db.collection.deleteMany(filter, options)

db.collection.deleteOne(
    <filter>,   // = selectors in find()
    {           // optional …
        writeConcern: <document>, // ack of num of replicas
        collation: <document>     // language-specific rules
        ..
    }
)
```

https://docs.mongodb.com/manual/reference/delete-methods/

Delete

```
> db.inventory.find( {} , { _id: 0, qty: 0, size: 0 } )
{ "item" : "journal", "status" : "B" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }
{ "item" : "computer", "status" : "C" }

> db.inventory.deleteOne({"item":"computer"})
{ "acknowledged" : true, "deletedCount" : 1 }

> db.inventory.find( {} , { _id: 0, qty: 0, size: 0 } )
{ "item" : "journal", "status" : "B" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }
```

Indexes

- **Motivation**
  - Full collection scan must be performed when searching for the documents, unless an appropriate index exists
- **Primary** index
  - MongoDB creates a unique index on the *_id* field during the creation of a collection
- **Secondary** indexes
  - Created manually for a given key field / fields
  - To create an index, use *db.collection.createIndex()* or a similar method from your driver.

```
db.<collection>.createIndex(keys, options)
```

  - MongoDB indexes use a B-tree data structure.

Index Types

- **Single** Field
  - Ascending/descending indexes on a single field.
- **Compound** Index
  - Indexes on multiple fields
    - The order of fields listed in a compound index has significance
    - e.g. { userid: 1, score: -1 }, sort by userid ASC and then, by score DESC.
- **Multikey** Index
  - To index a field that holds an array value.
- **Text** Indexes
- **Hashed** Indexes
- **Geospatial** Index

Index Types

- **1, -1** – standard ascending / descending value indexes

```
db.<collection>.createIndex( { field: -1 } )
```

- **hashed** – hash values of a single field are indexed

```
db.<collection>.createIndex( { _id: "hashed" } )
```

- **text** – basic full-text index

```
db.<collection>.createIndex( { comments: "text" } )
```

- **2d** – points in planar geometry

```
db.<collection>.createIndex( { <location field> : "2d" , <additional field> : <value> } , { <index-specification options> } )
```

- **2dsphere** – points in spherical geometry

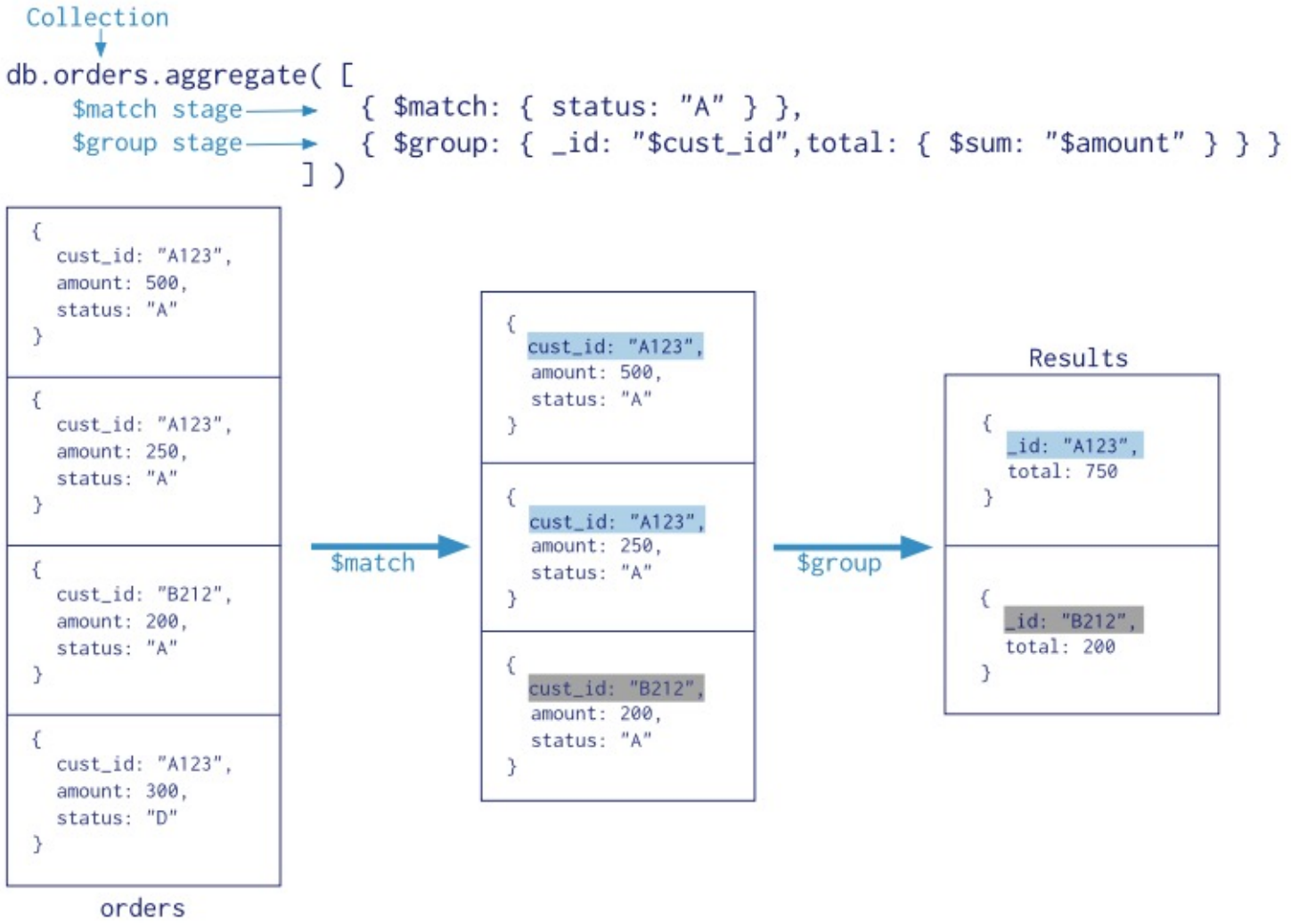
```
db.<collection>.createIndex( { <location field> : "2dsphere" } )
```

Indexes

```
// Full collection scan
> db.inventory.find( {qty: { "$gte" : 50 }}, {_id:0}).sort( {qty: -1})
{ "item" : "paper", "qty" : 100, "status" : "D" }
{ "item" : "planner", "qty" : 75, "status" : "D" }
{ "item" : "notebook", "qty" : 50, "status" : "A" }

> db.inventory.getIndexes()
[
   {"v": 2, "key": { "_id": 1 }, "name": "_id_", "ns": "test.inventory" }
]

> db.inventory.createIndex( { qty : 1 } )

> db.inventory.getIndexes()
[
   {"v": 2, "key": { "_id": 1 }, "name": "_id_", "ns": "test.inventory" }
   {"v": 2, "key": { "qty": 1 }, "name": "qty_1", "ns": "test.inventory" }
]
```

Aggregation pipeline

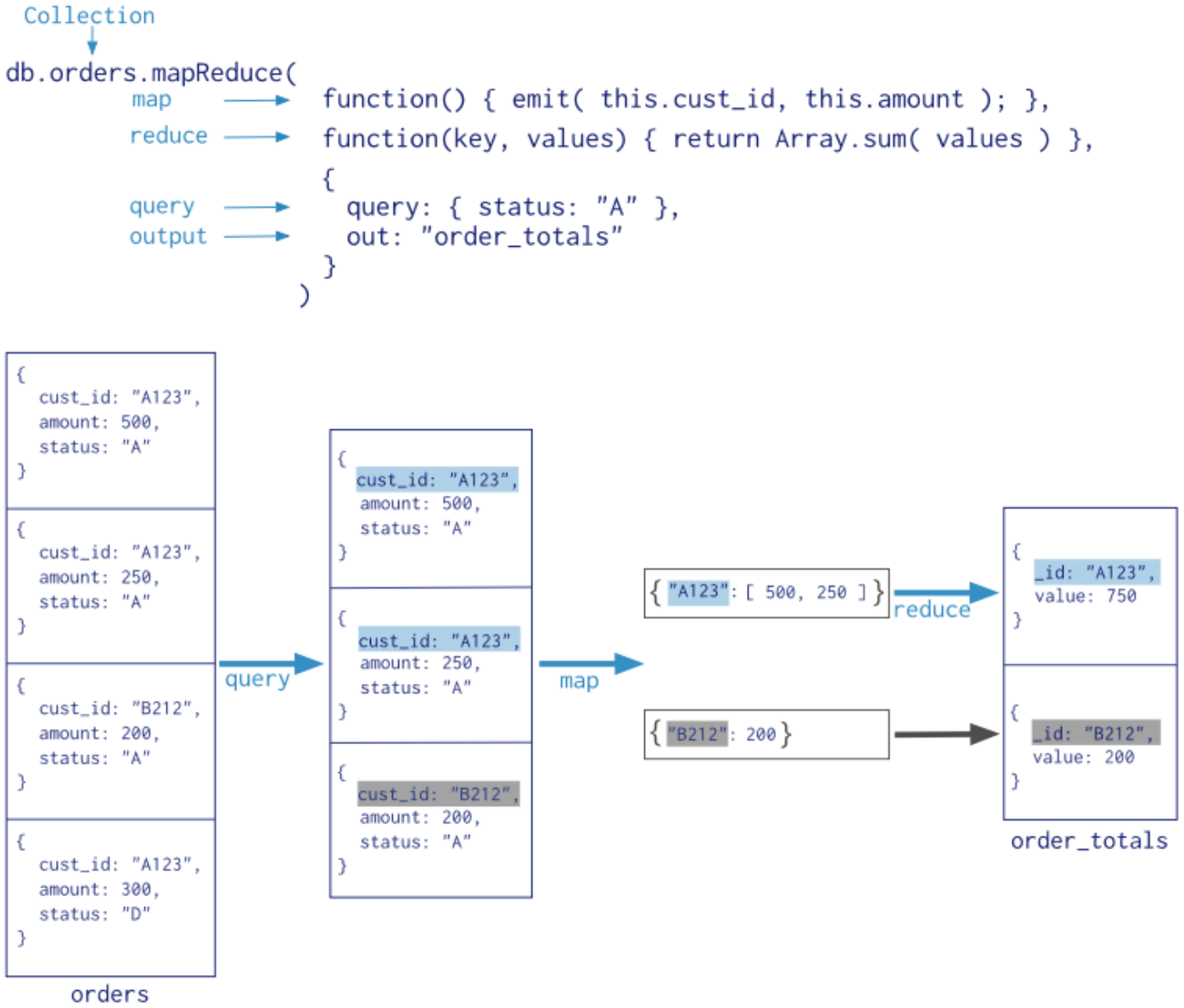
– Documents enter a multi-stage pipeline that transforms the documents into aggregated results



MapReduce

- Data processing paradigm for condensing large volumes of data into useful *aggregated* results.
- Both map and reduce functions are implemented as ordinary JavaScript functions
  - **Map** function: current document is accessible via this, emit(key, value) is used for emissions
  - **Reduce** function: key and array of values are provided as arguments, reduced value is published via return
- Beside others, **query**, **sort** or **limit** options are accepted
  - **out** option determines the output (e.g. a collection name)

MapReduce example



MongoDB Drivers

- The MongoDB Ecosystem contains documentation for the drivers, frameworks, tools, and platform services that work with MongoDB.
  - https://docs.mongodb.com/ecosystem/drivers/
- Drivers are available for many languages
  - C, C++, Java, Python, Ruby, …
- Java
  - http://mongodb.github.io/mongo-java-driver/
    - bson.jar
    - mongodb-driver-core.jar
    - mongodb-driver.jar

Java driver – example 1 (list)

```
public class Test {
    public static void main(String[] args) {
            // remove log in the console
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
                Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
            // os dados foram colocados manualmente no mongo
        MongoDatabase out = mongo.getDatabase("test");
        System.out.println("-- Colecções na BD " + "'"  + out.getName() + "'" );
        MongoIterable<String> x = out.listCollectionNames();
        for (String s : x)
            System.out.println(s);
        MongoCollection<Document> c = out.getCollection("inventory");
        System.out.println("-- Total de documentos em 'inventory': " +
c.count());
        FindIterable<Document> docs = c.find();
        for (Document doc : docs)
            System.out.println(doc.toJson());
        mongo.close();
    }
}
```

Java driver – example 1 output

```
--- Colecções na BD 'test'
invoice
inventory
collection
orders
--- Total de documentos em 'countries': 5
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432c" }, "item" : "journal",
"qty" : 25.0, "status" : "A" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432d" }, "item" : "notebook",
"qty" : 50.0, "status" : "A" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432e" }, "item" : "paper", "qty"
: 100.0, "status" : "D" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432f" }, "item" : "planner",
"qty" : 75.0, "status" : "D" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c4330" }, "item" : "postcard",
"qty" : 45.0, "status" : "A" }
```

Java driver – example 2 (insert)

```
public class Test2 {
    public static void main(String[] args) {
            // remove log in the console
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
                Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
        MongoCollection<Document> coll =
            mongo.getDatabase("test").getCollection("inventory");

        Document doc = new Document("item", "database")
            .append("qty", 1)
            .append("status","M");
        coll.insertOne(doc);
        FindIterable<Document> docs = coll.find();
        for (Document d : docs)
            System.out.println(d.toJson());
        mongo.close();
    }
}
```

Java driver – example 2 output

```
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432c" }, "item" : "journal",
"qty" : 25.0, "status" : "A" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432d" }, "item" : "notebook",
"qty" : 50.0, "status" : "A" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432e" }, "item" : "paper", "qty"
: 100.0, "status" : "D" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c432f" }, "item" : "planner",
"qty" : 75.0, "status" : "D" }
{ "_id" : { "$oid" : "59b1b730935c2a0ca72c4330" }, "item" : "postcard",
"qty" : 45.0, "status" : "A" }
{ "_id" : { "$oid" : "59b2a7e98cbca6f6497c7110" }, "item" : "database",
"qty" : 1, "status" : "M" }
```

Java driver – example 3 (multi-doc)

```
public class Test3 {
    public static void main(String[] args) {
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
                Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
        MongoCollection<Document> coll =
            mongo.getDatabase("test").getCollection("inventory");
        Document doc = new Document("item", "record")
            .append("size",
                new Document("h", 10).append("l", 20).append("w", 30))
            .append("qty", 1)
            .append("status","R");
        coll.insertOne(doc);
        FindIterable<Document> docs = coll.find(new Document("status", "R"));
        for (Document d : docs)
            System.out.println(d.toJson());
        mongo.close();
    }
}
```

```
{ "_id" : { "$oid" : "59b2a9eb8cbca6f6527068b2" }, "item" : "record",
"size" : { "h" : 10, "l" : 20, "w" : 30 }, "qty" : 1, "status" : "R" }
```

Summary

- Document Database
- MongoDB
  - JSON document database
  - Sharding with master-slave replication architecture
- Query functionality
  - CRUD operations
    - Insert, find, update, remove
  - Complex filtering conditions
  - Index structures
  - MapReduce
- Java driver

Resources

- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.

- MongoDB Docs, https://docs.mongodb.com
- Java Driver, http://mongodb.github.io/mongo-java-driver/