CC3201-1

Bases de Datos

Otoño 2025

Clase 12: Datos Semiestructurados: Árboles

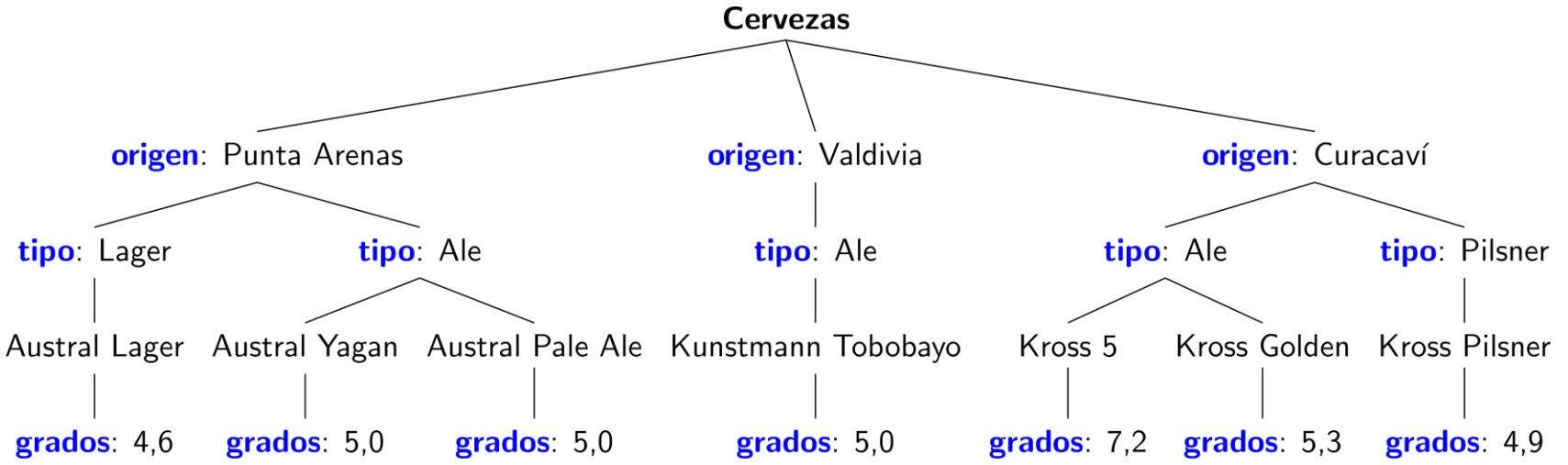
Aidan Hogan

aidhog@gmail.com

Modelos de datos

Sección 1.5 | Ramakrishnan / Gehrke

Modelo de datos (árbol/jerarquía)



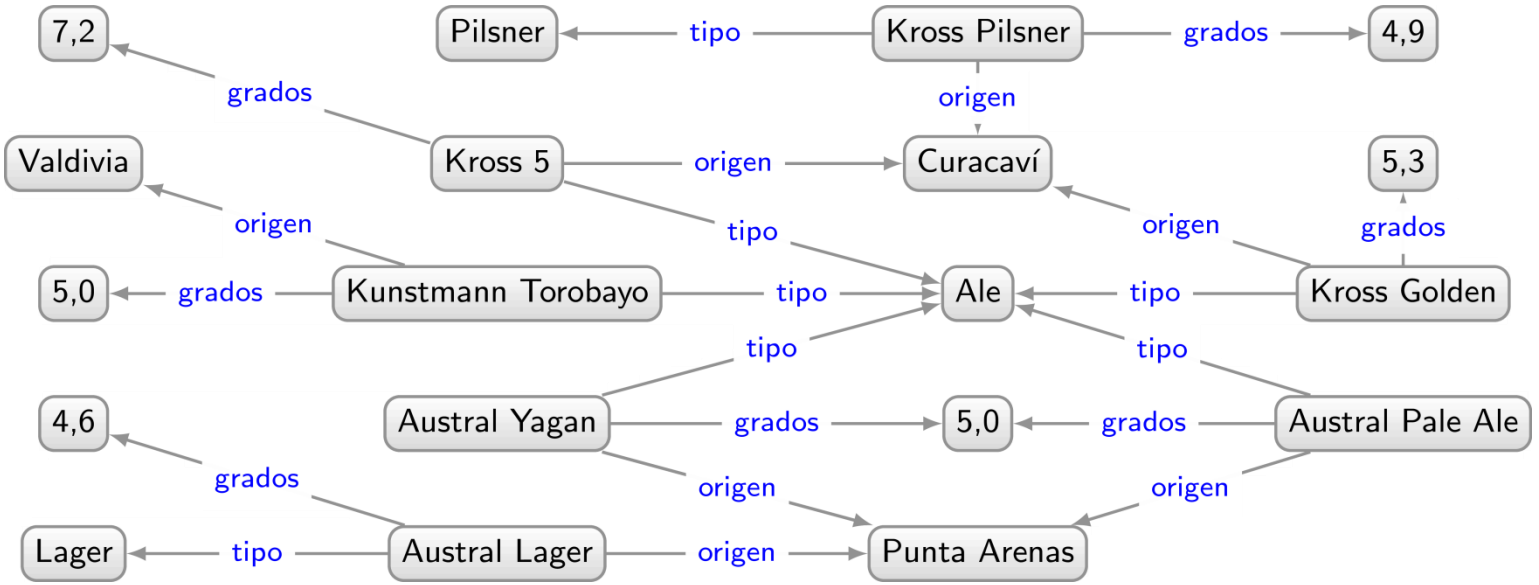
Modelo de datos (árbol/jerarquía)

✓ **Muy simple de entender**

✗ **Repeticiones**

✗ **¿Si no hay una jerarquía natural?**

Modelo de datos (grafo)



Modelo de datos (grafo)

✓ **Muy flexible / simple**

✗ **Difícil resumir los datos**
(no hay esquema obvio)

Modelo de datos (tabla)

Cervezas

nombre	tipo	grados	ciudad-origen
Austral Lager	Lager	4,6	Punta Arenas
Austral Yagan	Ale	5,0	Punta Arenas
Austral Pale Ale	Ale	5,0	Punta Arenas
Kuntsmann Torobayo	Ale	5,0	Valdivia
Kross 5	Ale	7,2	Curacaví
Kross Golden	Ale	5,3	Curacaví
Kross Pilsner	Pilsner	4,9	Curacaví

Modelo de datos (tabla)

Cervezas

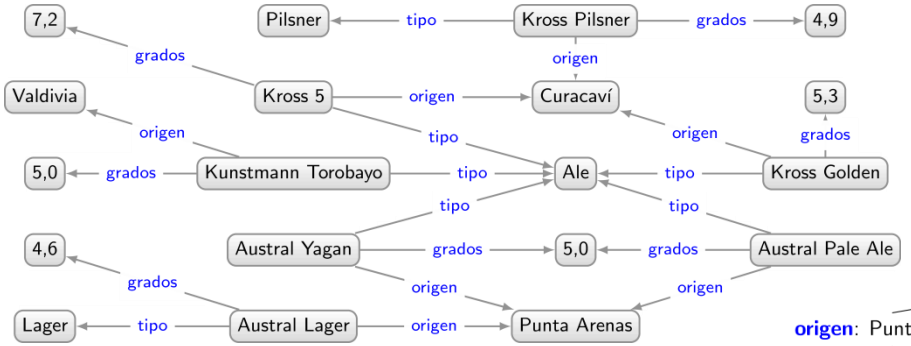
nombre	tipo	grados	ciudad-origen
Austral Lager	Lager	4,6	Punta Arenas
Austral Yagan	Ale	5,0	Punta Arenas
Austral Pale Ale	Ale	5,0	Punta Arenas
Kuntsmann Torobayo	Ale	5,0	Valdivia
Kross 5	Ale	7,2	Curacaví
Kross Golden	Ale	5,3	Curacaví
Kross Pilsner	Pilsner	4,9	Curacaví

✓ **Muy simple de entender**

✗ **¿Si queremos agregar un atributo nuevo?**

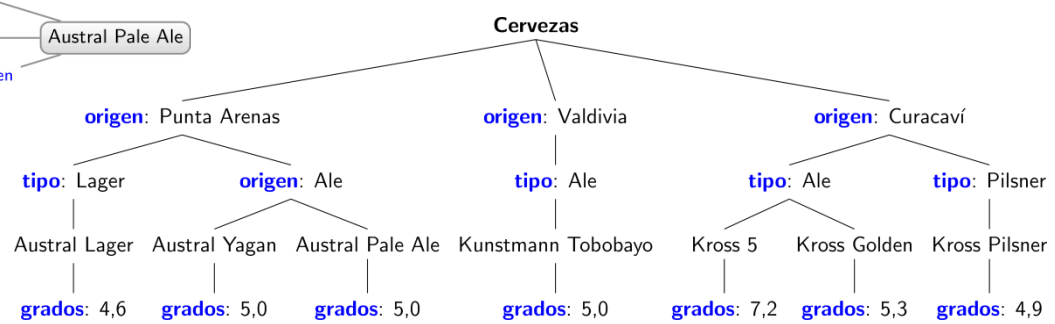
✗ **¿Si no sabemos los grados de algunas cervezas?**

Diferentes modelos de datos tienen diferentes fortalezas y debilidades …



Cervezas

nombre	tipo	grados	ciudad-origen
Austral Lager	Lager	4,6	Punta Arenas
Austral Yagan	Ale	5,0	Punta Arenas
Austral Pale Ale	Ale	5,0	Punta Arenas
Kuntsmann Torobayo	Ale	5,0	Valdivia
Kross 5	Ale	7,2	Curacaví
Kross Golden	Ale	5,3	Curacaví
Kross Pilsner	Pilsner	4,9	Curacaví



… como las cervezas.

Datos semiestructurados

Sección 1.5 | Ramakrishnan / Gehrke

El espectro de la estructura de datos

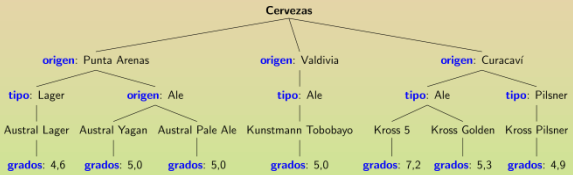
Texto Plano

Hay mucha variedad en las cervezas locales de Chile. La sede de la marca "Austral" se encuentra en Punta Arenas. Austral fabrica una amplia gama de cervezas, incluyendo Larger (4,6%), Yagan (un ale de 5%) y un Pale Ale (5%). La cerveza de marca "Kunstmann" también es popular, en particular su cerveza "Torobayo" (un ale de 5% elaborado en Valdiva). La marca Kross, basada en Curacaví, también tiene una gama de cervezas populares como, por ejemplo, Kross 5 (un ale fuerte de 7,2%) Kross Golden (un ale de 5,3%) y Kross Pilsner (4,9%).

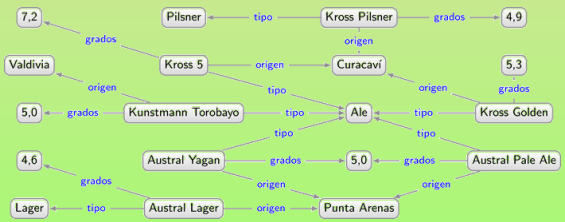
Texto Enriquecido
(HTML, Word, ...)

```
<ul>
  <li>Austral Lager [Lager] 4,6% de Punta Arenas</li>
  <li>Austral Yagan [Ale] 5% de Punta Arenas</li>
  <li>Austral Pale Ale [Ale] 5% de Punta Arenas</li>
  <li>Kunstsmann Torobayo [Ale] 5% de Valdivia</li>
  <li>Kross 5 [Ale] 7,2% de Curacaví</li>
  <li>Kross Golden [Ale] 5,3% de Curacaví</li>
  <li>Kross Pilsner [Pilsner] 4,9% de Curacaví</li>
</ul>
```

Árboles
(XML, JSON, ...)



Grafos
(RDF, Prop. Gs, ...)



Relacional
(SQL, CSV, ...)

Cervezas

nombre	tipo	grados	ciudad-origen
Austral Lager	Lager	4,6	Punta Arenas
Austral Yagan	Ale	5,0	Punta Arenas
Austral Pale Ale	Ale	5,0	Punta Arenas
Kuntsmann Torobayo	Ale	5,0	Valdivia
Kross 5	Ale	7,2	Curacaví
Kross Golden	Ale	5,3	Curacaví
Kross Pilsner	Pilsner	4,9	Curacaví

No estructurados

D A T O S

Semiestructurados

Estructurados

Datos semiestructurados: Árboles

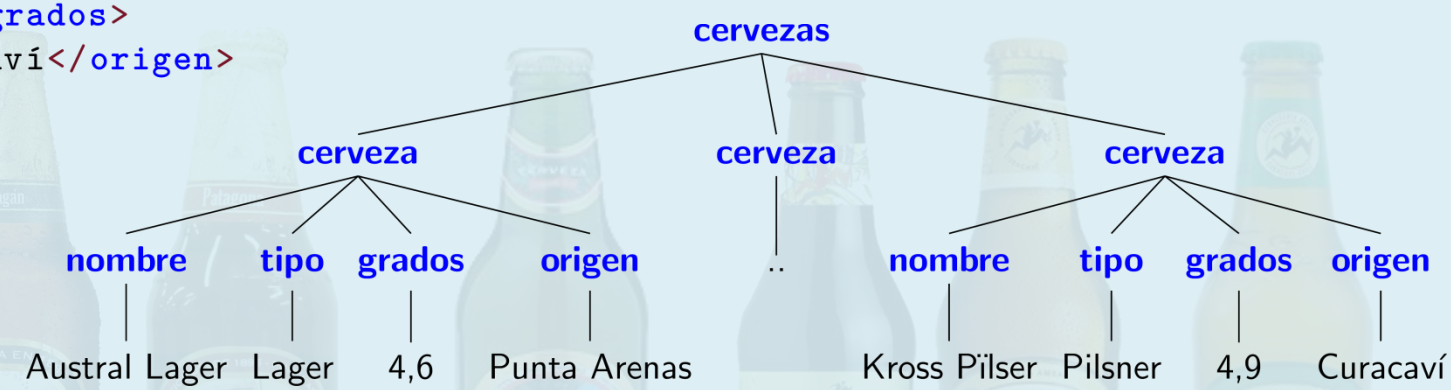
Capítulo 7,4 | Ramakrishnan / Gehrke

eXtensible Markup Language (XML)

```
<cervezas>
  <cerveza>
     <nombre>Austral Lager</nombre>
     <tipo>Lager</tipo>
     <grados>4.6</grados>
     <origen>Punta Arenas</origen>
  </cerveza>
  <cerveza>
     <nombre>Austral Yagan</nombre>
     <tipo>Ale</tipo>
     <grados>5</grados>
     <origen>Punta Arenas</origen>
  </cerveza>
  ...
  <cerveza>
     <nombre>Kross Pilsner</nombre>
     <tipo>Pilsner</tipo>
     <grados>4.9</grados>
     <origen>Curacaví</origen>
  </cerveza>
</cervezas>
```

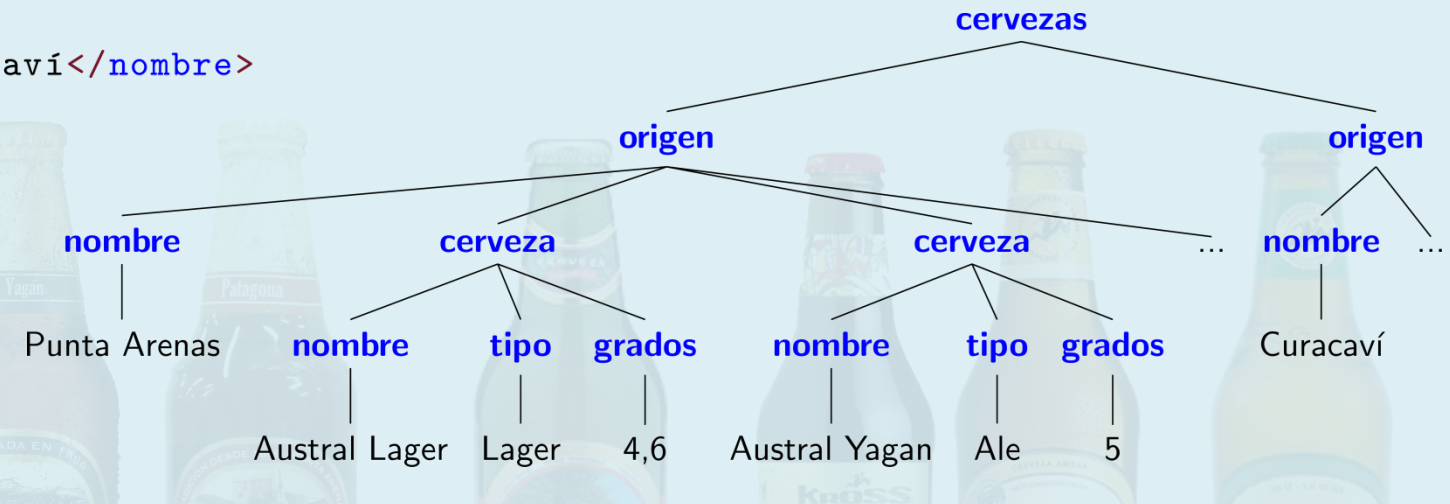
El cierre de etiquetas es obligatorio en XML

¿Hay otra forma de representar estos datos en XML?



eXtensible Markup Language (XML)

```
<cervezas>
  <origen>
    <nombre>Punta Arenas</nombre>
    <cerveza>
      <nombre>Austral Lager</nombre>
      <tipo>Lager</tipo>
      <grados>4.6</grados>
    </cerveza>
    <cerveza>
      <nombre>Austral Yagan</nombre>
      <tipo>Ale</tipo>
      <grados>5</grados>
    </cerveza>
    ...
  </origen>
  <origen>
    <nombre>Curacaví</nombre>
    ...
  </origen>
</cervezas>
```



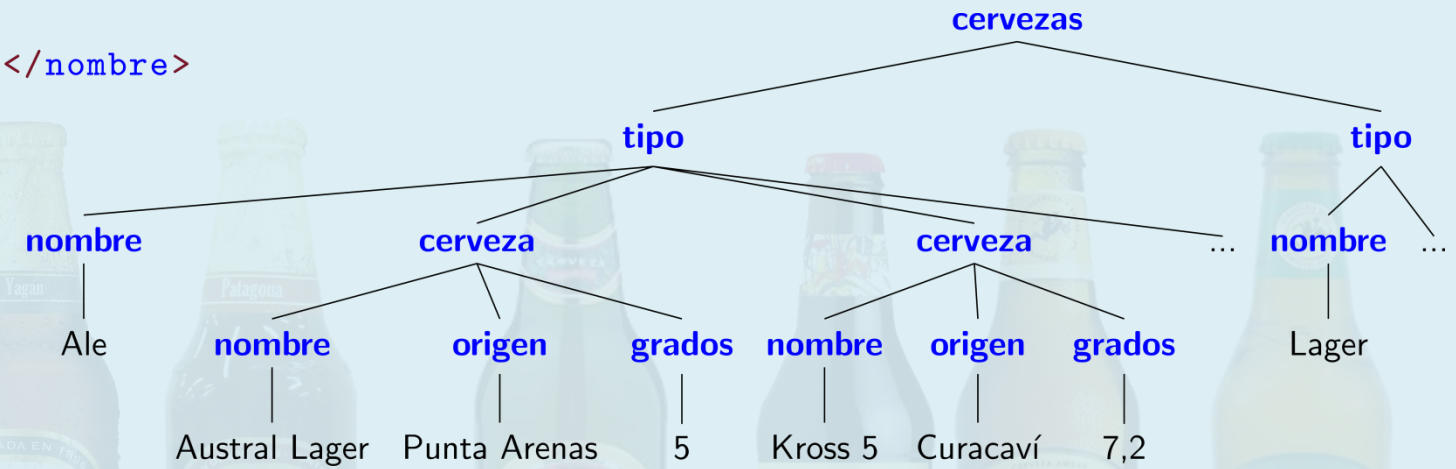
eXtensible Markup Language (XML)

```
<cervezas>
  <tipo>
    <nombre>Ale</nombre>
    <cerveza>
      <nombre>Austral Pale Ale</nombre>
      <origen>Punta Arenas</origen>
      <grados>5</grados>
    </cerveza>
    <cerveza>
      <nombre>Kross 5</nombre>
      <origen>Curacaví</origen>
      <grados>7.2</grados>
    </cerveza>
    ...
  </tipo>
  <tipo>
    <nombre>Lager</nombre>
    ...
  </tipo>
</cervezas>
```

✓ **Muy simple de entender**

✗ **Repeticiones**

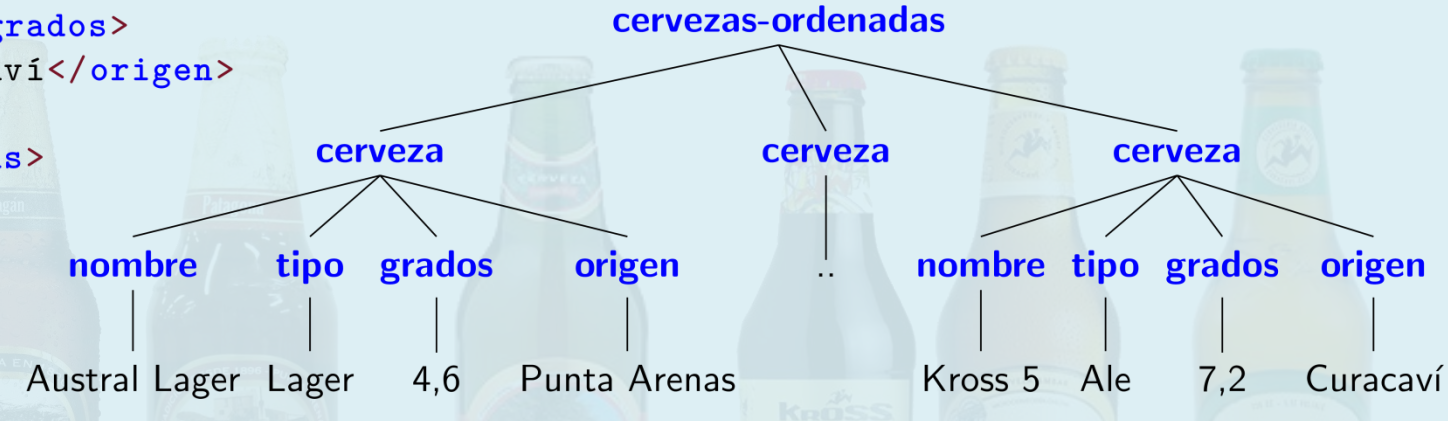
✗ **¿Si no hay una jerarquía natural?**



eXtensible Markup Language (XML)

```
<cervezas-ordenadas>
  <cerveza>
     <nombre>Austral Lager</nombre>
     <tipo>Lager</tipo>
     <grados>4.6</grados>
     <origen>Punta Arenas</origen>
  </cerveza>
  <cerveza>
     <nombre>Kross Pilsner</nombre>
     <tipo>Pilsner</tipo>
     <grados>4.9</grados>
     <origen>Curacaví</origen>
  </cerveza>
  ...
  <cerveza>
     <nombre>Kross 5</nombre>
     <tipo>Ale</tipo>
     <grados>7.2</grados>
     <origen>Curacaví</origen>
  </cerveza>
</cervezas-ordenadas>
```

Se preserva el orden

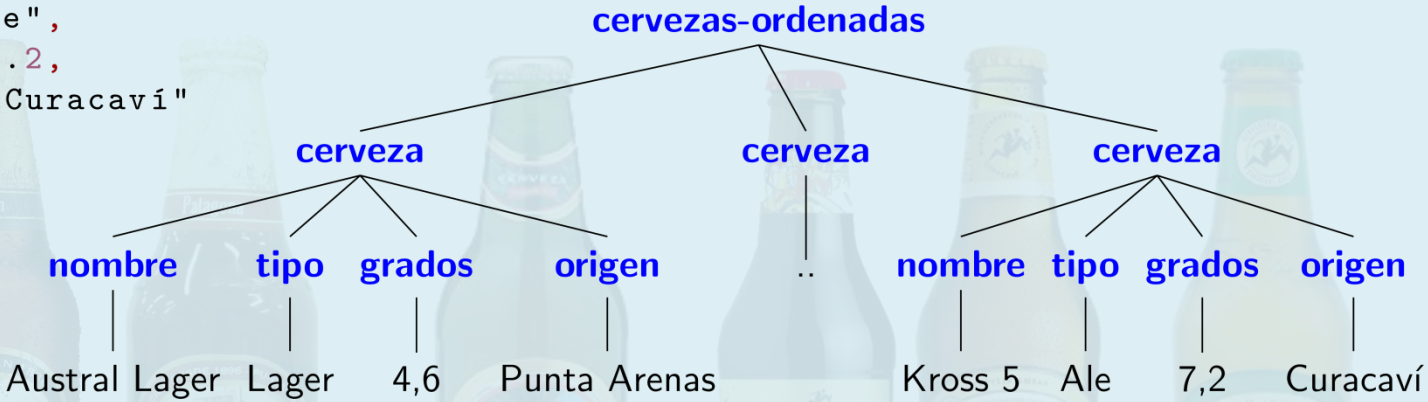


JavaScript Object Notation (JSON)

```
{ "cervezas-ordenadas" : [
    { "cerveza" :
      { "nombre" : "Austral Lager",
        "tipo" : "Lager",
        "grados" : 4.6,
        "origen" : "Punta Arenas"
      }
    },
    { "cerveza" :
      { "nombre" : "Kross Pilsner",
        "tipo" : "Pilsner",
        "grados" : 4.9,
        "origen" : "Curacaví"
      }
    },
    ...
    { "cerveza" :
      { "nombre" : "Kross 5",
        "tipo" : "Ale",
        "grados" : 7.2,
        "origen" : "Curacaví"
      }
    }
  ]
}
```

Sintaxis:

Arreglos:	[…]
Objetos:	{ … }
Strings:	"…"
Números:	1|2.9|-23|2.9e8
Booleanos:	true|false
Nulos:	null



Datos masivos:
Almacenar datos estructurados

Bases de datos relacionales

Bases de datos relacionales: ¿Talla única?

"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section


Spongecell
Hula
kiko
Trumba
eskobo
mayomi
Pageflakes
vimeo
skobee
shadows
grovee
YouTube
Zimbra
LookSmart FURL | Your Personal Web File
smugmug
newsgator
Blogniscient
TiNFiNGER
shutterfly
PodDater
Feedster
favoor
Planzo.com
ZAZZLE
Tailrank
TagWorld
nuvvo
dogear
yakalike
grouper
ODDPOST
QOOP
iNods
Lulu
BR
blish
flagr
FireAnt
simplyhired
Veoh
theadcloud
rbloc.com
pixagogo
gather
Agatra
browsr
oyogi
cafepress
Renkoo
standpoint
meebo
EXTRA TASTY!
last·fm
Jotspot
Frappr!
jeteye
dabble
YEDDA
Writeboard
SHOUTWIRE
iKarma
KaNOODLE
AirSet
tech memeorandum
CalendarHub
Suprglu
PIECING YOUR WEB TOGETHER
pando
zigtag
Findory
backfence
clipmarks
wayfaring
gOFFICE
AllPeers
orb
Rallypoint
Zoozio
blogbeat
Ziggs
zoto
vSocial
Boltfolio
wink
riya
Wordcast
Opinity
reddit
measuremap
gumshoo
bluepulse
imvu BETA
STREAMLOAD
Ta-da Lists
FeedSky
jellyBam.INC
nativetext
CONGOO
PODZINGER
RSS MAD
FeedTier
phanfare
Fruitcast
PubSub
dPolls
flickr
Ning
Ookles
Strongspace
zoominfo
CASTPOST
WIKIPEDIA
yubnub
AC
BLOOP
ProjectSpaces
FeedBurner
Bloglines
purevolume.com
FOTOLOG
ourmedia
Yub.com
Spot Runner
myspace
NewsAlloy
Allmydata.com
gabbacom
Gcast
blinkx
openomy
riffs
ajchat
chatsum
PANDORA
looklater
jambo
ROLLYO
ClipShack
squishr
PROTOPAGE
WebJay
PLAZES
Noodly
30 BETA
woofdir
diigo
box
Jots
Xdrive
vizu
digg
del.icio.us
omni drive
AlmondRocks
Tagyu
writely
Simpy
Gtalkr
TRUVEO
egoSurf
mozy
Quimble
pegasus
remember the milk
SQUIDOO
picturecloud
newsvine
Clipfire
Basecamp
Netvibes
goowy
My Videokaraoke
Lexxe alpha
powered by advanced natural language technology
LOOKSTER
PXN8
measuremap
facebook
43 Things
Technorati
Platial
Backpack
Gmail

Bases de datos relacionales: Aspectos costosos

- "Structured Query Language" (SQL):
  - Lenguaje declarativo
  - Muchas buenas características
  - **Pero ¡es difícil de optimizar!**
  - **¡Y el esquema es poco flexible!**

- "Atomicity, Consistency, Isolation, Durability" (ACID):
  - Asegura correctitud de la base de datos
  - **Pero ¡las transacciones incurren en altos gastos!**

- La distribución no es directa con bases de datos relacionales

¿Alternativas a bases de datos relacionales para gestionar datos masivos?

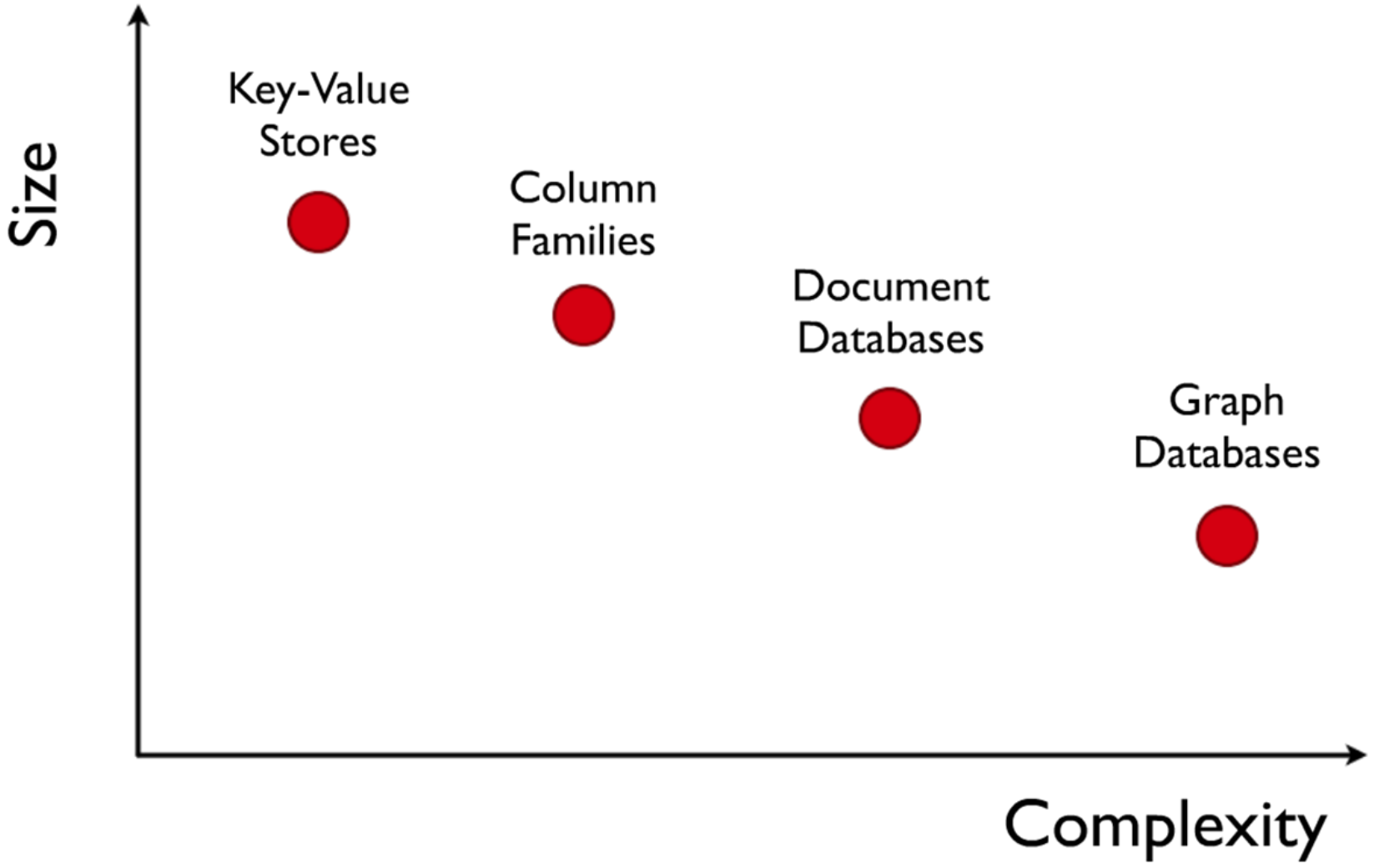
NoSQL

¿Qué saben ustedes de NoSQL?

NoSQL: No solo SQL ("Not only SQL")

- **Distribuido**
  - "Sharding": partición "horizontal" de datos
  - Replicación
  - Garantías diferentes a ACID
- A menudo **lenguajes más simples** que SQL
  - APIs no estandarizadas
  - Más trabajo para la aplicación
- **Sabores diferentes** (para escenarios diferentes)
  - Garantías diferentes
  - Perfiles diferentes de escalabilidad
  - Funcionalidades de consulta diferentes
  - Modelos de datos diferentes

NoSQL


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

424 systems in ranking, June 2025

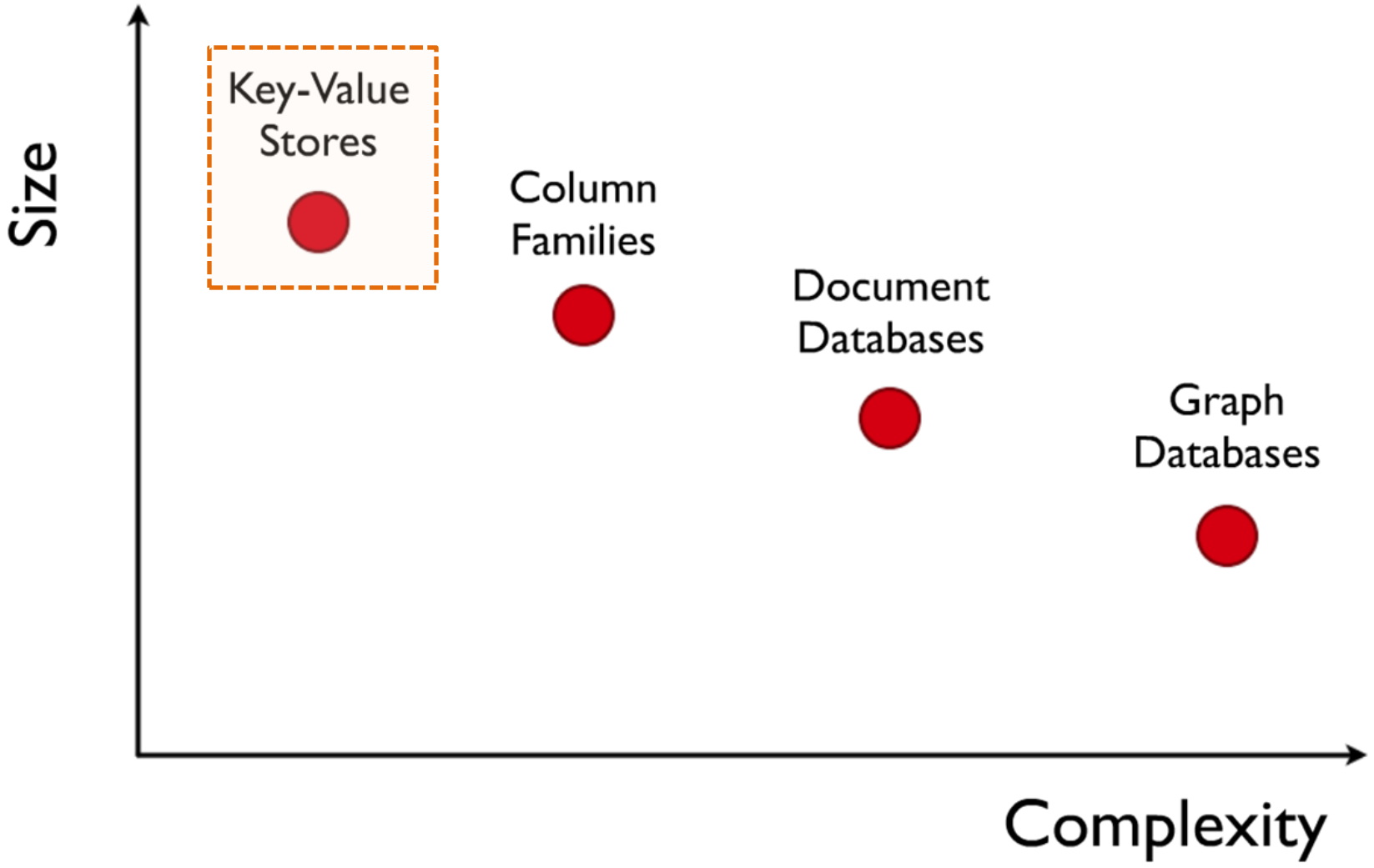
Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Oracle	Relational, Multi-model	1230.38	+3.82	-13.70
2.	2.	2.	MySQL	Relational, Multi-model	953.57	-11.41	-107.77
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	776.75	+1.86	-44.81
4.	4.	4.	PostgreSQL	Relational, Multi-model	680.65	+6.34	+44.41
5.	5.	5.	MongoDB	Document, Multi-model	402.85	+0.33	-18.23
6.	6.	↑ 8.	Snowflake	Relational	174.49	+2.48	+44.13
7.	7.	↓ 6.	Redis	Key-value, Multi-model	151.72	-0.47	-4.22
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model	125.13	-1.27	-0.77
9.	9.	↓ 7.	Elasticsearch	Multi-model	121.28	-2.53	-11.55
10.	10.	10.	SQLite	Relational	117.03	-0.74	+5.63
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	108.27	+0.22	+9.44
12.	12.	↑ 15.	Databricks	Multi-model	104.67	+2.02	+23.59
13.	↑ 14.	13.	MariaDB	Relational, Multi-model	94.54	+0.93	+3.50
14.	↓ 13.	↓ 11.	Microsoft Access	Relational	88.28	-7.63	-12.88
15.	15.	↑ 17.	Amazon DynamoDB	Multi-model	83.34	+4.04	+8.90
16.	16.	↑ 18.	Apache Hive	Relational	76.68	+0.91	+16.92
17.	17.	↓ 16.	Microsoft Azure SQL Database	Relational, Multi-model	75.31	-0.14	-1.47
18.	18.	↓ 14.	Splunk	Search engine	69.61	-1.78	-19.49
19.	19.	19.	Google BigQuery	Relational	64.54	+1.66	+6.44
20.	20.	↑ 21.	Neo4j	Graph	51.33	+3.42	+6.44

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

NoSQL:

Sistemas de Llave–Valor ("Key–Value")

NoSQL


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

Modelo de llave–valor

Es sólo un mapa ☺

- `put(key,value)`
- `get(key)`
- `delete(key)`

Key	Value
Afghanistan	Kabul
Albania	Tirana
Algeria	Algiers
Andorra la Vella	Andorra la Vella
Angola	Luanda
Antigua and Barbuda	St. John’s
…	…

Pero se puede hacer mucho con un mapa

Key	Value
Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
Kabul	country:Afghanistan,pop:3476000#2013
Tirana	country:Albania,pop:3011405#2013
…	…
user10239	basedIn@city:Tirana,post:{103,10430,201}
…	…

Sistemas populares del modelo llave-valor

redis

Memcached

RocksDB

Amazon
DynamoDB

riak

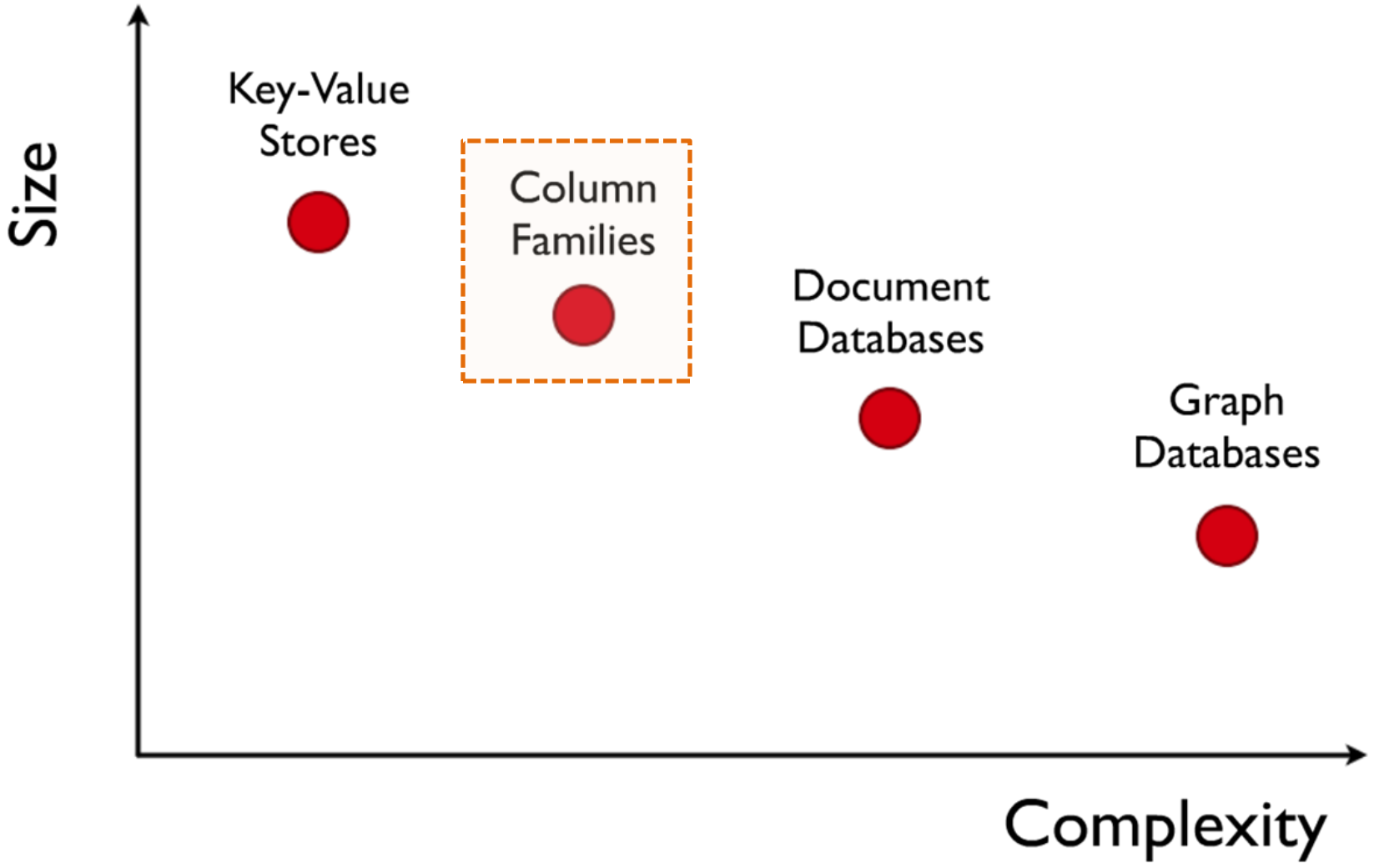
424 systems in ranking, June 2025

Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Oracle	Relational, Multi-model	1230.38	+3.82	-13.70
2.	2.	2.	MySQL	Relational, Multi-model	953.57	-11.41	-107.77
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	776.75	+1.86	-44.81
4.	4.	4.	PostgreSQL	Relational, Multi-model	680.65	+6.34	+44.41
5.	5.	5.	MongoDB	Document, Multi-model	402.85	+0.33	-18.23
6.	6.	↑ 8.	Snowflake	Relational	174.49	+2.48	+44.13
7.	7.	↓ 6.	Redis	Key-value, Multi-model	151.72	-0.47	-4.22
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model	125.13	-1.27	-0.77
9.	9.	↓ 7.	Elasticsearch	Multi-model	121.28	-2.53	-11.55
10.	10.	10.	SQLite	Relational	117.03	-0.74	+5.63
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	108.27	+0.22	+9.44
12.	12.	↑ 15.	Databricks	Multi-model	104.67	+2.02	+23.59
13.	↑ 14.	13.	MariaDB	Relational, Multi-model	94.54	+0.93	+3.50
14.	↓ 13.	↓ 11.	Microsoft Access	Relational	88.28	-7.63	-12.88
15.	15.	↑ 17.	Amazon DynamoDB	Multi-model	83.34	+4.04	+8.90
16.	16.	↑ 18.	Apache Hive	Relational	76.68	+0.91	+16.92
17.	17.	↓ 16.	Microsoft Azure SQL Database	Relational, Multi-model	75.31	-0.14	-1.47
18.	18.	↓ 14.	Splunk	Search engine	69.61	-1.78	-19.49
19.	19.	19.	Google BigQuery	Relational	64.54	+1.66	+6.44
20.	20.	↑ 21.	Neo4j	Graph	51.33	+3.42	+6.44

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

Modelo Tabular / Familia de Columnas

NoSQL


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

Llave–Valor = un mapa distribuido

Countries	
Primary Key	Value
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
Albania	capital:Tirana,continent:Europe,pop:3011405#2013
…	…

Tabular = un mapa multi-dimensional

Countries				
Primary Key	capital	continent	pop-value	pop-year
Afghanistan	Kabul	Asia	31108077	2011
Albania	Tirana	Europe	3011405	2013
…	…	…	…	…

Sistemas populares del modelo tabular

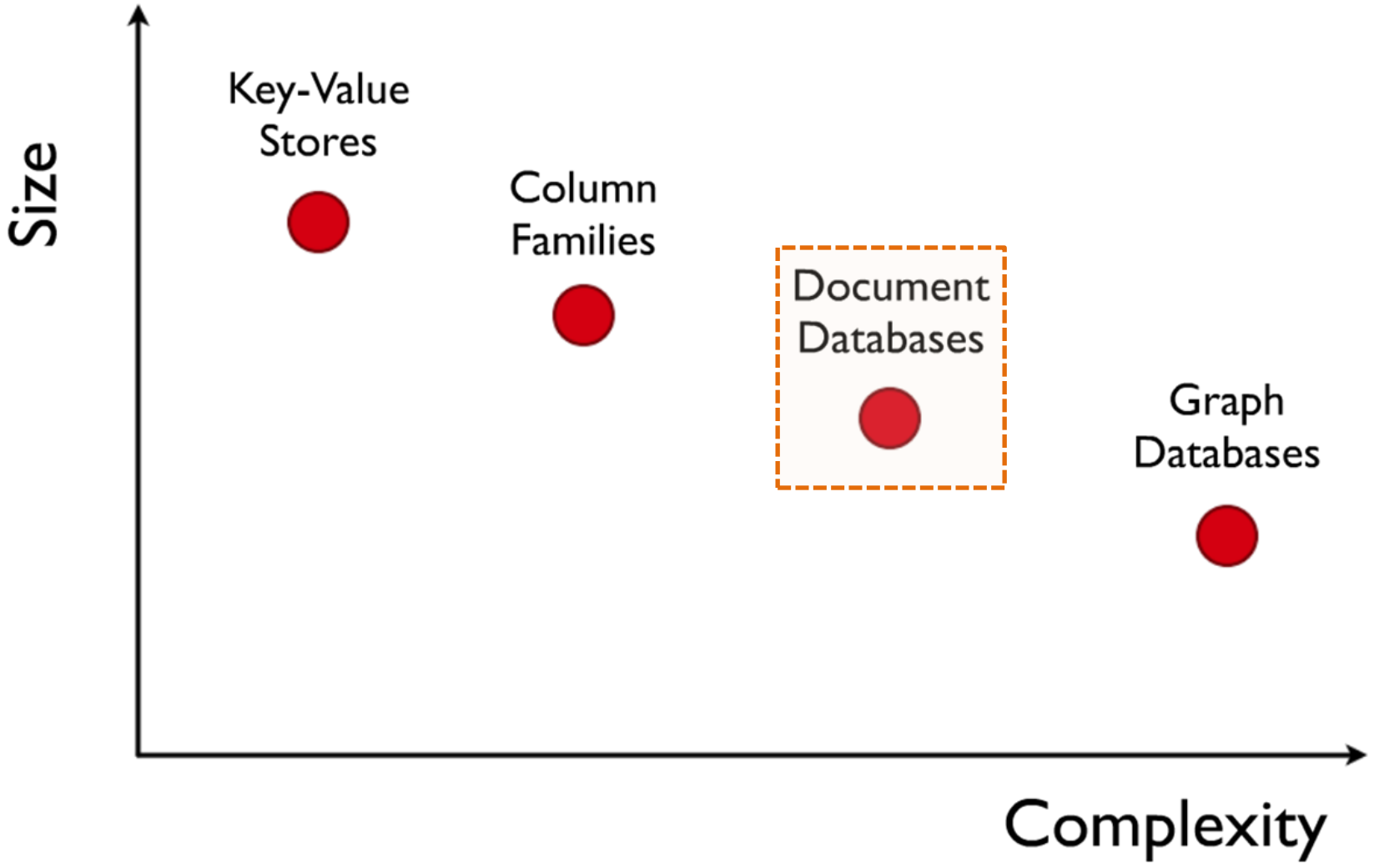
424 systems in ranking, June 2025

Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Oracle	Relational, Multi-model	1230.38	+3.82	-13.70
2.	2.	2.	MySQL	Relational, Multi-model	953.57	-11.41	-107.77
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	776.75	+1.86	-44.81
4.	4.	4.	PostgreSQL	Relational, Multi-model	680.65	+6.34	+44.41
5.	5.	5.	MongoDB	Document, Multi-model	402.85	+0.33	-18.23
6.	6.	↑ 8.	Snowflake	Relational	174.49	+2.48	+44.13
7.	7.	↓ 6.	Redis	Key-value, Multi-model	151.72	-0.47	-4.22
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model	125.13	-1.27	-0.77
9.	9.	↓ 7.	Elasticsearch	Multi-model	121.28	-2.53	-11.55
10.	10.	10.	SQLite	Relational	117.03	-0.74	+5.63
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	108.27	+0.22	+9.44
12.	12.	↑ 15.	Databricks	Multi-model	104.67	+2.02	+23.59
13.	↑ 14.	13.	MariaDB	Relational, Multi-model	94.54	+0.93	+3.50
14.	↓ 13.	↓ 11.	Microsoft Access	Relational	88.28	-7.63	-12.88
15.	15.	↑ 17.	Amazon DynamoDB	Multi-model	83.34	+4.04	+8.90
16.	16.	↑ 18.	Apache Hive	Relational	76.68	+0.91	+16.92
17.	17.	↓ 16.	Microsoft Azure SQL Database	Relational, Multi-model	75.31	-0.14	-1.47
18.	18.	↓ 14.	Splunk	Search engine	69.61	-1.78	-19.49
19.	19.	19.	Google BigQuery	Relational	64.54	+1.66	+6.44
20.	20.	↑ 21.	Neo4j	Graph	51.33	+3.42	+6.44

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

NoSQL:
Almacenamiento de documentos

NoSQL


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

Llave–Valor: un mapa

Countries	
Primary Key	Value
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
…	…

Tabular: un mapa multi-dimensional

Countries				
Primary Key	capital	continent	pop-value	pop-year
Afghanistan	Kabul	Asia	31108077	2011
…	…	…	…	…

Documentos: un mapa con valores de docs.

Countries	
Primary Key	Value
Afghanistan	{ cap: “Kabul”, con: “Asia”, pop: { val: 31108077, y: 2011 } }
…	…

Sistemas populares de documentos

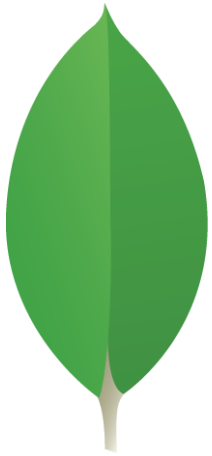
424 systems in ranking, June 2025

Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Oracle	Relational, Multi-model	1230.38	+3.82	-13.70
2.	2.	2.	MySQL	Relational, Multi-model	953.57	-11.41	-107.77
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	776.75	+1.86	-44.81
4.	4.	4.	PostgreSQL	Relational, Multi-model	680.65	+6.34	+44.41
5.	5.	5.	MongoDB	Document, Multi-model	402.85	+0.33	-18.23
6.	6.	↑ 8.	Snowflake	Relational	174.49	+2.48	+44.13
7.	7.	↓ 6.	Redis	Key-value, Multi-model	151.72	-0.47	-4.22
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model	125.13	-1.27	-0.77
9.	9.	↓ 7.	Elasticsearch	Multi-model	121.28	-2.53	-11.55
10.	10.	10.	SQLite	Relational	117.03	-0.74	+5.63
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	108.27	+0.22	+9.44
12.	12.	↑ 15.	Databricks	Multi-model	104.67	+2.02	+23.59
13.	↑ 14.	13.	MariaDB	Relational, Multi-model	94.54	+0.93	+3.50
14.	↓ 13.	↓ 11.	Microsoft Access	Relational	88.28	-7.63	-12.88
15.	15.	↑ 17.	Amazon DynamoDB	Multi-model	83.34	+4.04	+8.90
16.	16.	↑ 18.	Apache Hive	Relational	76.68	+0.91	+16.92
17.	17.	↓ 16.	Microsoft Azure SQL Database	Relational, Multi-model	75.31	-0.14	-1.47
18.	18.	↓ 14.	Splunk	Search engine	69.61	-1.78	-19.49
19.	19.	19.	Google BigQuery	Relational	64.54	+1.66	+6.44
20.	20.	↑ 21.	Neo4j	Graph	51.33	+3.42	+6.44

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

MongoDB:

Modelo de datos



mongoDB®

{JSON}

JavaScript Object Notation

"JavaScript Object Notation": JSON


```
{
  "id": 179,
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": "2002-06-02",
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```


JSON Binario: BSON

```
{
  "_id": ObjectId(99a88b77c66d),
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": ISODate("2002-06-02"),
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```

MongoDB: "Datatypes"

Boolean: `true`

String: `"sí"`

Array: `[]`

Object: {}

Double: `43.2`

{JSON}
JavaScript Object Notation

etc.

ObjectID: `ObjectId(0A0A0A0A0A0A)`

Date: `ISODate("2003-14-15T09:26:53.589Z")`

BSON{01010100 11101011 10101110 01010101}

etc.

MongoDB: Mapa de llaves a valores BSON

TVSeries	
Key	BSON Value

99a88b77c66d	<pre>{
	  "_id": ObjectId(99a88b77c66d),
	  "name": "The Wire",
	  "type": "Scripted",
	  "language": "English",
	  "genres": ["Drama", "Crime", "Thriller"],
	  "status": "Ended",
	  "runtime": 60,
	  "premiered": ISODate("2002-06-02"),
	  "schedule": {
	  "time": "21:00",
	  "days": [
	  "Sunday"
	  ]
	  },
	  "rating": {
	  "average": 9.4
	  }
	}</pre>

…	…

Colección MongoDB: Documentos relacionados

TVSeries	
Key	BSON Value
99a88b77c66d	{ "_id": ObjectId(99a88b77c66d), "name": "The Wire", "type": "Scripted", ... }
11f22e33d44c	{ "_id": ObjectId(11f22e33d44c), "name": "Rick and Morty", "type": "Animation", ... }

Base de datos MongoDB: Colecciones relacionadas

TVSeries

Key	BSON Value
...	...

TVEpisodes

Key	BSON Value
...	...

TVNetworks

Key	BSON Value
...	...

database: TV

Usar la database tvdb (la creará si no existe):

```
> use tvdb

switched to db tvdb
```

Ver todas las databases no vacías (tvdb es vacía aquí):

```
> show dbs

local     0.00001 GB
test      0.00231 GB
```

Ver la database actual:

```
> db

tvdb
```

Borrar la database actual:

```
> db.dropDatabase()

{ "dropped" : "tvdb", "ok" : 1 }
```

Crear la colección series:

```
> db.createCollection("series")

{ "ok" : 1 }
```

Ver las colecciones en la database actual:

```
> show collections

series
```

Borrar la colección `series`:

```
> db.series.drop()

true
```

Borrar los documentos de la colección `series`:

```
> db.series.remove( {} )

WriteResult({ "nRemoved" : 0 })
```

Crear colección acotada ("capped": guarda los n más recientes):

```
> db.createCollection("last100episodes",
   { capped: true, size: 12285600, max: 100 } )

{ "ok" : 1 }
```

Crear colección con índice (por defecto) sobre `_id`:

```
> db.createCollection("cast", { autoIndexId: true } )

{
    "note" : "the autoIndexId option is deprecated ...",
    "ok" : 1
}
```

MongoDB:

Insertar datos

Insertar un documento: sin _id

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60,
    "genres": [
      "Science-Fiction",
      "Thriller"
    ]
  })

WriteResult({ "nInserted" : 1 })
```

Insertar documento: con _id

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60,
    "genres": [
      "Science-Fiction",
      "Thriller"
    ]
  })

WriteResult({ "nInserted" : 1 })
```

... falla si el valor de _id ya existe

... usar update or save para actualizar un documento

Usar save y _id para sobrescribir

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60
  })

WriteResult({ "nInserted" : 1 })
```

```
> db.series.save(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror (Overwritten)"
  })

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

... sobrescribe el documento previo

MongoDB:

Consultas con selección

`find()`:

Devolver todos los documentos de la colección

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60
  })
```

```
> db.series.find()

{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",
"type" : "Scripted", "runtime" : 60 }
```

... usar `findOne()` para devolver un documento

pretty():

Imprimir con indentación

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60
  })
```

```
> db.series.find().pretty()

{
    "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
    "name" : "Black Mirror",
    "type" : "Scripted",
    "runtime" : 60
}
```

`find(σ)`:

Encontrar los documentos que satisfagan σ

```
> db.series.find(σ)
```

Selección σ: Igualdad

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "type": "Scripted",
    "runtime": 60
  })
```

Igualdad: `{ key: value }`

```
> db.series.find( { "type": "Scripted" } )

{ "name" : "Black Mirror", "type" : "Scripted", "runtime" : 60 }
```

Los resultados incluyen un valor de `_id` pero lo omitiremos aquí para mantener los ejemplos más concisos.

Selección σ: Llave anidada

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": 9.4 },
    "runtime": 60
  })
```

La llave puede referenciar a un valor anidado

```
> db.series.find( { "rating.avg": 9.4 } )

{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selección σ: Igualdad con nulos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": null },
    "runtime": 60
  })
```

Igualdad con un nulo anidado

```
> db.series.find( { "rating.avg": null } )

{ "name" : "Black Mirror", "rating": { "avg": null }, "runtime" : 60 }
```

... coincide cuando el valor sea nulo o ...

Selección σ: Igualdad con nulos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "val": 9.4 },
    "runtime": 60
  })
```

Igualdad con un nulo anidado

```
> db.series.find( { "rating.avg": null } )

{ "name" : "Black Mirror", "rating": { "val": 9.4 }, "runtime" : 60 }
```

... o cuando el key no exista.

Selección σ: Igualdad con un documento

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": 9.4 },
    "runtime": 60
  })
```

Un valor puede ser un documento

```
> db.series.find( { "rating": { "avg": 9.4 } } )

{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selección σ: Igualdad con un documento

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": 9.4, "votes": 9001 },
    "runtime": 60
  })
```

Tiene que coincidir completamente:

```
> db.series.find( { "rating": { "votes": 9001 } } )
```

... resultados vacíos: una coincidencia parcial

Selección σ: Igualdad con un documento

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": 9.4, "votes": 9001 },
    "runtime": 60
  })
```

El orden importa:

```
> db.series.find(
  { "rating": { "votes": 9001, "avg": 9.4 } }
)
```

... resultados vacíos: el orden no coincide

Selección σ: Igualdad con un arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Tiene que ser una coincidencia exacta:

```
> db.series.find( { "genres": [ "Science-Fiction",
                                "Thriller" ] } )

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Igualdad con un arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Tiene que ser una coincidencia exacta:

```
> db.series.find( { "genres": [ "Science-Fiction" ] } )
```

... resultados vacíos: una coincidencia parcial.

Selección σ: Igualdad con un arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

El orden importa ...

```
> db.series.find( { "genres": [ "Thriller",
                                "Science-Fiction" ] } )
```

... resultados vacíos: el orden no coincide

Selección σ: Pertenencia a un arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Un valor coincidirá con un elemento de un arreglo

```
> db.series.find( { "genres": "Thriller" } )

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Pertenencia a un arreglo

```
> db.series.insert( { "name": "A" , "val": [ 5, 6 ] } )
> db.series.insert( { "name": "B", "val": 5 } )
```

... incluso adentro y afuera del arreglo al mismo tiempo

```
> db.series.find( { "val": 5 } )

{ "name": "A" , "val": [ 5, 6 ] }
{ "name": "B" , "val": 5 }
```

Selección σ: Desigualdades

Menor a:	`{ key: { $lt: value } }`
Mayor a:	`{ key: { $gt: value } }`
Menor a o igual a:	`{ key: { $lte: value } }`
Mayor a o igual a:	`{ key: { $gte: value } }`
No igual a:	`{ key: { $ne: value } }`

Selección σ: Menor a

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Menor a: { key: { $lt: value } }

```
> db.series.find({ "runtime": { $lt: 70 } })

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Múltiples valores

Algún valor: `{ key: { $in: [ v1, …, vn ] } }`

Ningún valor: `{ key: { $nin: [ v1, …, vn ] } }`

… o coincidirá si la llave no existe

Selección σ: Múltiples valores

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Algún valor: `{ key: { $in: [ v1, …, vn ] } }`

```
> db.series.find({ "runtime": { $in: [30, 60] } })

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Múltiples valores

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Algún valor: `{ key: { $in: [ v1, …, vn ] } }`

```
> db.series.find({ "genres": { $in: ["Noir", "Thriller"] } })

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

… si la llave tiene un arreglo, cualquier valor del arreglo debería coincidir con cualquier valor de `$in`

Selección σ: Conectivos booleanos

Y: `{ $and: [ σ , σ′ ] }`

O: `{ $or: [ σ , σ′ ] }`

No: `{ $not: [ σ ] }`

No-O: `{ $nor: [ σ , σ′ ] }`

… se pueden anidar estas condiciones

Selección σ: Y

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Y: $\{ \$and: [\sigma , \sigma'] \}$

```
> db.series.find({ $and: [
    { "runtime": { $in: [30, 60] } } ,
    { "name": { $ne: "Lost" } } ] }
)

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Atributo (no) existe

Existe: `{ key: { $exists : true } }`

No Existe: `{ key: { $exists : false } }`

Selección σ: Atributo existe

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Existe: `{ key: { $exists : true } }`

```
> db.series.find({ "name": { $exists : true } })

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

... verifica que la llave `key` exista
(incluso si el valor es `NULL`)

Selección σ: Atributo no existe

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

No existe: `{ key: { $exists : false } }`

```
> db.series.find({ "name": { $exists : false } })
```

... verifica que la llave key no exista (resultados vacíos)

Selección σ: Arreglos

Todos: `{ key: { $all : [v1, …, vn] } }`

Cualquiera: `{ key: { $elemMatch : { σ1, …, σn } } }`

Largo: `{ key: { $size : int } }`

Selección σ: Arreglo tiene (al menos) cada elemento

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Sci-Fi", "Thriller", "Comedy" ],
    "runtime": 60
  })
```

Todo: `{ key: { $all : [v1, …, vn] } }`

```
> db.series.find(
  { "genres": { $all : [ "Comedy", "Sci-Fi" ] } })

{ "name" : "Black Mirror", "genres": [ "Sci-Fi", "Thriller", "Comedy" ],
"runtime" : 60 }
```

… cada valor está en el arreglo

Selección σ: Cada condición está satisfecha

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Cualquiera: `{ key: { $elemMatch : {σ1, …, σn } } }`

```
> db.series.find(
  { "series": { $elemMatch : { $gt: 1, $lt: 3 } } })

{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

… para cada criterio, existe un elemento que lo satisfaga

Selección σ: Arreglo con largo exacto

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Largo: `{ key: { $size : int } }`

```
> db.series.find( { "series": { $size : 3 } })

{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

... no hay rangos de largo (solo un valor exacto)

Selección σ: Tipo de un valor

Tipo: `{ key: { $type: typename } }`

"timestamp"

"decimal"

"string"

"double"

"binData"

"date"

"object"

"int"

"array"

"long"

"bool"

"undefined"

"objectId"

"null"

"regex"

"dbPointer"

"array"

"javascript"

"number"

…

Selección σ: Tipo de un valor

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Tipo: `{ key: { $type: typename } }`

```
> db.series.find({ "runtime": { $type : "number" } })

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],
"runtime" : 60 }
```

Selección σ: Encontrar valores que son arreglos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

Tipo: `{ key: { $type: typename } }`

```
> db.series.find({ "genres": { $type : "array" } })
```

... resultados vacíos:
coincidirá si algún valor del arreglo es de ese tipo

Selección σ: Encontrar valores que son arreglos

Arrays

When applied to arrays, `$type` matches any **inner** element that is of the specified BSON type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an `array`.

https://docs.mongodb.com/manual/reference/operator/query/type/


```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [ "Science-Fiction", "Thriller" ],
    "runtime": 60
  })
```

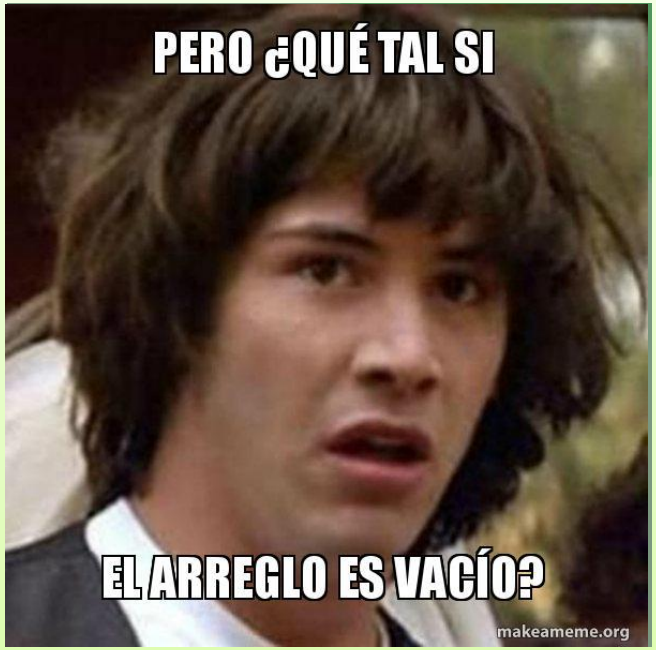
```
> db.series.find(
 { "genres": { $elemMatch: { $exists : true } } }
 )

{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller"
], "runtime" : 60 }
```

Selección σ: Encontrar valores que son arreglos

Arrays

When applied to arrays, `$type` matches any **inner** element that is of the specified BSON type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an `array`.

https://docs.mongodb.com/manual/reference/operator/query/type/




```
> db.series.insert(
  {
    "name": "Black Mirror",
    "genres": [],
    "runtime": 60
  })
```

```
> db.series.find(
  { $or: [
   { "genres": { $elemMatch: { $exists: true } } },
   { "genres": [] }
  ] } )

{ "name" : "Black Mirror", "genres": [], "runtime" : 60 }
```

Selección σ: Otros operadores

Mod: `{ key: { $mod [ div, rem ] } }`

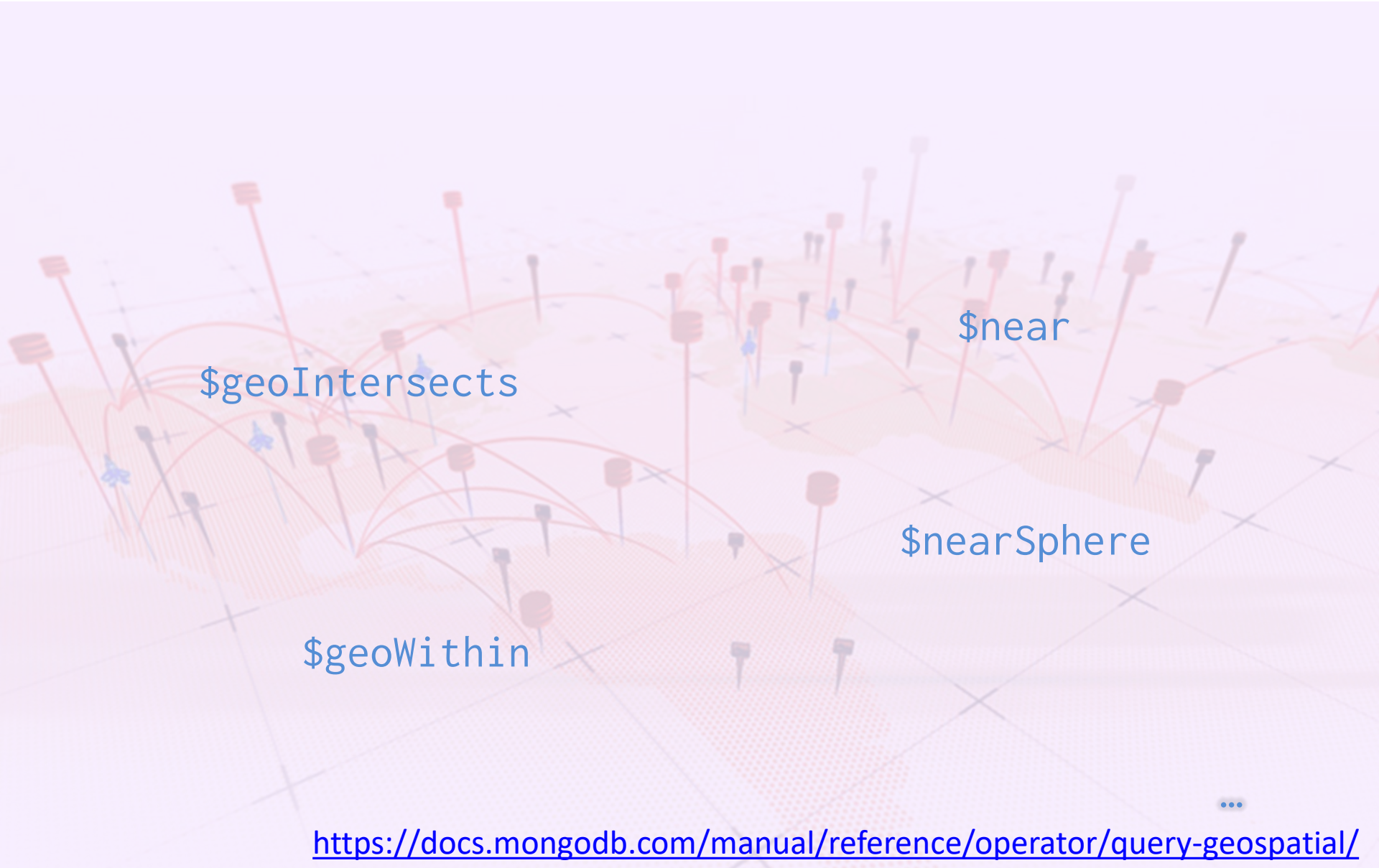
Regex: `{ key: { $regex: pattern } }`

Palabras claves: `{ $text: { $search: terms } }`

Donde (JS): `{ $where: javascript_code }`

... se ejecuta `where` para todos los documentos
(y debería ser evitado cuando sea posible)

Selección σ: Características geográficas



https://docs.mongodb.com/manual/reference/operator/query-geospatial/

Selección σ: Operadores al nivel de bits

$bitsAllClear

$bitsAllSet

$bitsAnyClear

$bitsAnySet

...

https://docs.mongodb.com/manual/reference/operator/query-bitwise/

MongoDB:

Proyección de valores de salida

Proyección π: Elegir valores de salida

`key: 1`:	Emitir el campo con `key`
`key: 0`:	Suprimir el campo con `key`
`array.$: 1`	Proyectar la 1.ª coincidencia del arreglo
`$elemMatch`:	Proyectar la 1.ª coincidencia del arreglo
`$slice`:	Emitir un sub-arreglo de valores

https://docs.mongodb.com/manual/reference/operator/Proyección/

Proyección π: Emitir algunos campos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Emitir algunos campos: { k1: 1, ..., kn: 1 }

```
> db.series.find(
  { "runtime": { $gt: 30 } },
  { "name": 1 , "runtime": 1, "network": 1 })

{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",
"runtime" : 60 }
```

... emite lo que está disponible (incluso `_id` por defecto)

Proyección π: Emitir campos anidados

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "rating": { "avg": 9.4 , "votes": 9001 },
    "runtime": 60
  })
```

Emitir campos (anidados): `{ k.k': 1 }`

```
> db.series.find(
  { "runtime": { $gt: 30 } },
  { "name": 1 , "rating.avg": 1 })

{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",
"rating" : { "avg": 9.4 } }
```

... el campo se mantiene anidado en la salida

Proyección π: Emitir valores de un arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "reviews": [
       { "user": "jack" , "score": 9.1 },
       { "user": "jill" , "score": 8.3 }
    ],
    "runtime": 60
  })
```

Emitir valores de un arreglo: `{ k.k': 1 }`

```
> db.series.find(
  { "runtime": { $gt: 30 } },
  { "name": 1 , "reviews.score": 1 })

{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",
"reviews" : [ { "score": 9.1 } , { "score": 8.3 } ] }
```

... proyecta valores del arreglo

Proyección π: Suprimir algunos campos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Emitir todos menos ...: `{ k1: 0, ..., kn: 0 }`

```
> db.series.find(
  { "runtime": { $gt: 30 } },
  { "series": 0 })

{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",
"runtime" : 60 }
```

... no se puede combinar 0 y 1 salvo ...

Proyección π: Suprimir algunos campos

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Suprimir ID: `{ _id: 0, k1: 1, ..., kn: 1 }`

```
> db.series.find(
  { "runtime": { $gt: 30 } },
  { "_id": 0, "name": 1 , "series": 1 })

{ "name" : "Black Mirror", "series" : [ 1, 2, 3 ] }
```

... se puede suprimir `_id` cuando se emiten otros valores

Proyección π: Emitir primera coincidencia

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Proyectar primera coincidencia del arreglo: `$elemMatch`

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $elemMatch: { $lt: 3 } } } )

{ "_id": ..., "series": [ 1 ] }
```

... separa las condiciones de selección y proyección

Proyección π: Emitir primera coincidencia

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Proyectar primera coincidencia del arreglo: `$elemMatch`

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $elemMatch: { $gt: 3 } } } )

{ "_id": ... }
```

... suprime el arreglo si no existe ninguna coincidencia

Proyección π: Emitir primera coincidencia

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "reviews": [
       { "user": "jack" , "score": 9.1 },
       { "user": "jill" , "score": 8.3 }
    ]
  })
```

Proyectar primera coincidencia del arreglo: `$elemMatch`

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "reviews": { $elemMatch: { "score":{ $gt: 8 } } } } )

{ "_id": ..., "reviews": [ { "user": "jack" , "score": 9.1 } ] }
```

... funciona con arreglos de documentos

Proyección π: Emitir un sub-arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Devolver los primeros n elementos: `$slice: n` (n > 0)

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $slice: 2 } } )

{ "_id": ..., "name": "Black Mirror", "series": [ 1, 2 ], "runtime": 60 }
```

Proyección π: Emitir un sub-arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Devolver los últimos n elementos: $slice: n (n < 0)

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $slice: -2 } } )

{ "_id": ..., "name": "Black Mirror", "series": [ 2, 3 ], "runtime": 60 }
```

Proyección π: Emitir un sub-arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

Saltar n y devolver m: `$slice: [n,m]` (n, m > 0)

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $slice: [ 2, 1 ] } } )

{ "_id": ..., "name": "Black Mirror", "series": [ 3 ], "runtime": 60 }
```

Proyección π: Emitir un sub-arreglo

```
> db.series.insert(
  {
    "name": "Black Mirror",
    "series": [ 1, 2, 3 ],
    "runtime": 60
  })
```

De los últimos n, devolver m: `$slice: [n,m]` (n < 0, m > 0)

```
> db.series.find(
  { "series": { $elemMatch: { $gt: 1 } } },
  { "series": { $slice: [ -2, 1 ] } } )

{ "_id": ..., "name": "Black Mirror", "series": [ 2 ], "runtime": 60 }
```

MongoDB:
Otras características

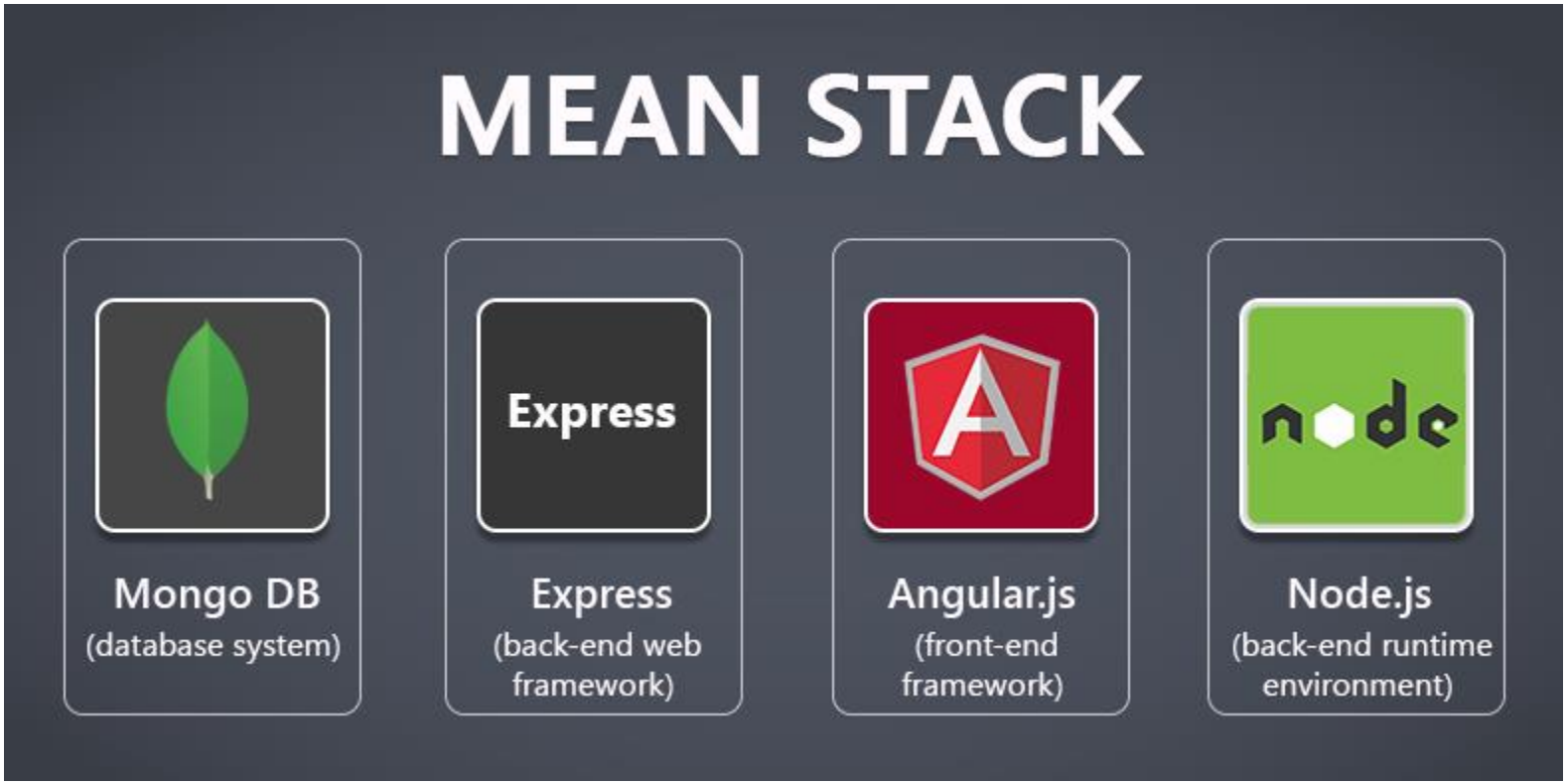
MongoDB: Otras cáracteristicas

- Actualizaciones
- Agregación / “Pipelines”
- Indexación
- Búsqueda sobre texto
- Consultas geográficas
- Procesamiento distribuido
- Y mucho más

Más detalles en el curso CC5212.

MEAN Stack

- MongoDB, Express.js, Angular(JS), Node.js



https://medium.com/@jeremyvsjeremy/what-is-the-mean-stack-9d11ae2cd384

Preguntas?