NOSQL Overview

Databases and Web Applications Laboratory (LBAW)
Bachelor in Informatics Engineering and Computation (L.EIC)

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Agenda

➔ Motivation and Context

➔ Principles of NOSQL Systems

➔ Key-Value Databases

➔ Document Stores

➔ Column-Family Databases

➔ Graph Databases

Context and Motivation

Relational Databases

- Relational databases are the default choice for data storage, a *de facto* standard.
- Recall the value of relational databases:
  - Persistence — keeping large amounts of data store for quick and easy access;
  - Concurrency — allow and control simultaneous accesses to the same data through transactions;
  - Integration — share a single data store that allows for integration at application level;
  - Standard — an adopted standard for modeling and manipulating data that shares the same core concepts despite differences in implementation.

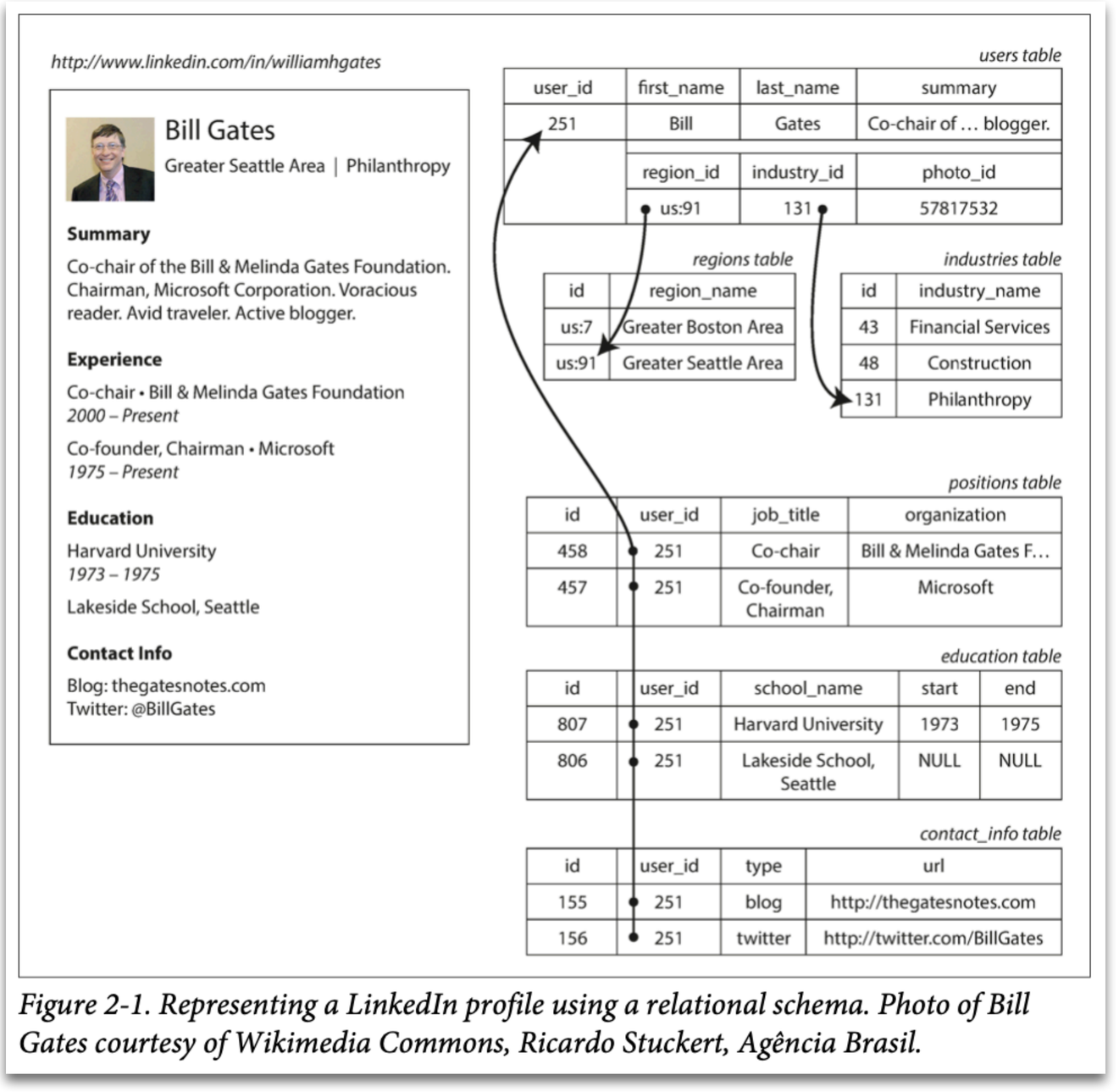


Figure 2-1. Representing a LinkedIn profile using a relational schema. Photo of Bill Gates courtesy of Wikimedia Commons, Ricardo Stuckert, Agência Brasil.

Image from Designing Data-Intensive Applications, Kleppmann (2017)

Three Database Revolutions

- The first database revolution was driven by the electronic computer.
  - Pillars: database management systems; record at a time processing, flat files.
- The second revolution is associated with the relational database paradigm.
  - Pillars: relational model; ACID transactions; and the SQL language.
- The third (current) revolution is associated with NoSQL solutions.
  - Pillars: non-relational; large scale; distributed and global scope.

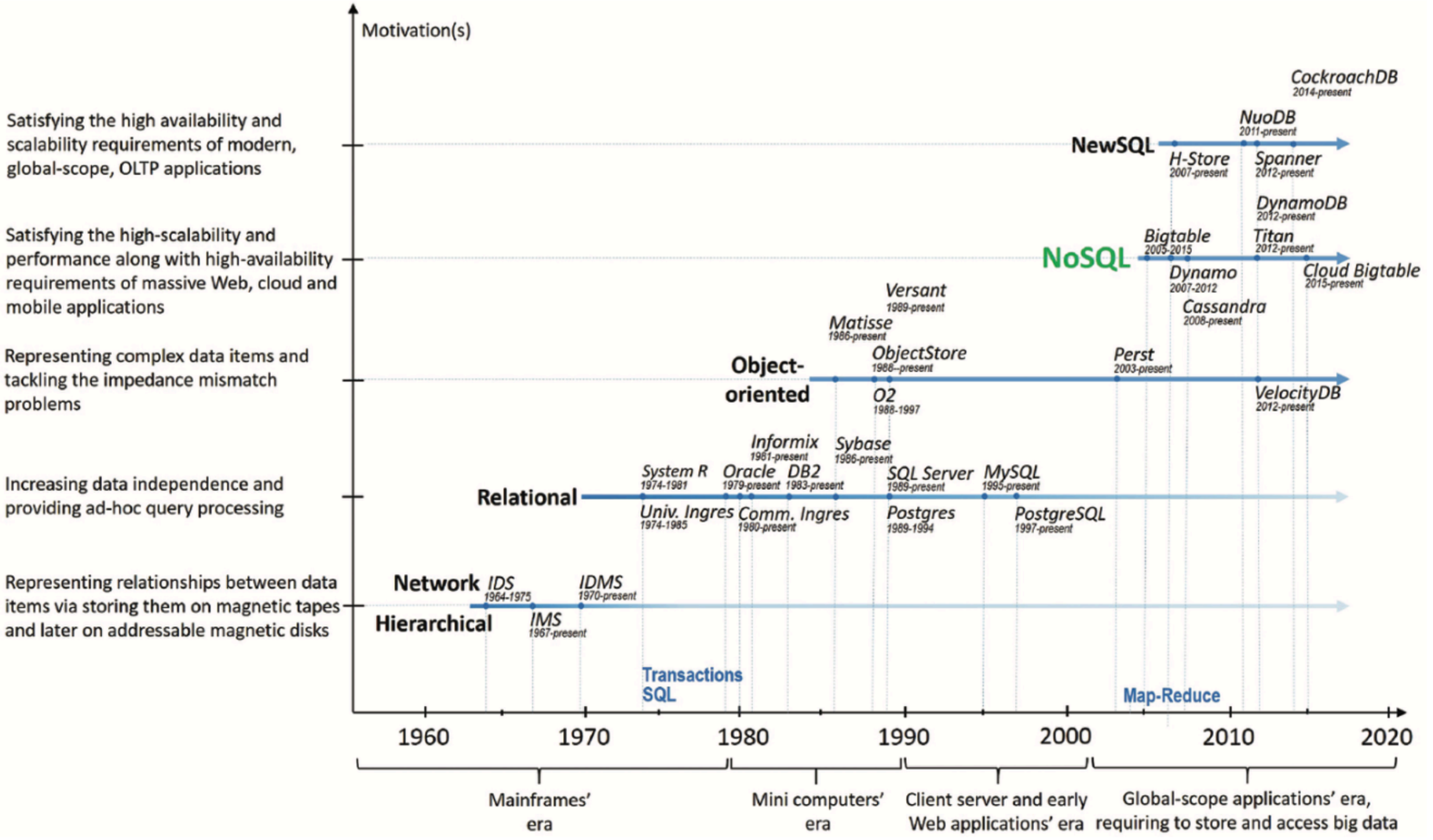


Fig. 1. The continuous development of major database technologies and some corresponding database systems.

Image from "A Survey on NoSQL Stores", Davodian et al. (2018)

Why NoSQL?

Impedance Mismatch

- "Impedance mismatch", is an expression from electronics referring to the difference between the electrical resistance of two circuits, resulting in inefficient power transfer.
- In the context of information systems, refers to the difference between distinct data representations, most notably between the relational model and in-memory data structures.
- In the relational model, data is organized and manipulated using relations, i.e. sets of tuples (name-value pair), which only allow for simple structures, e.g. they cannot contain nested elements.
- This limitation is not true for in-memory data structures, e.g. arrays, lists.
- As a result, to use richer data structures a translation is necessary between to and from a relational representation.

Impedance Mismatch

- Impedance mismatch exists when two different representations require translation.

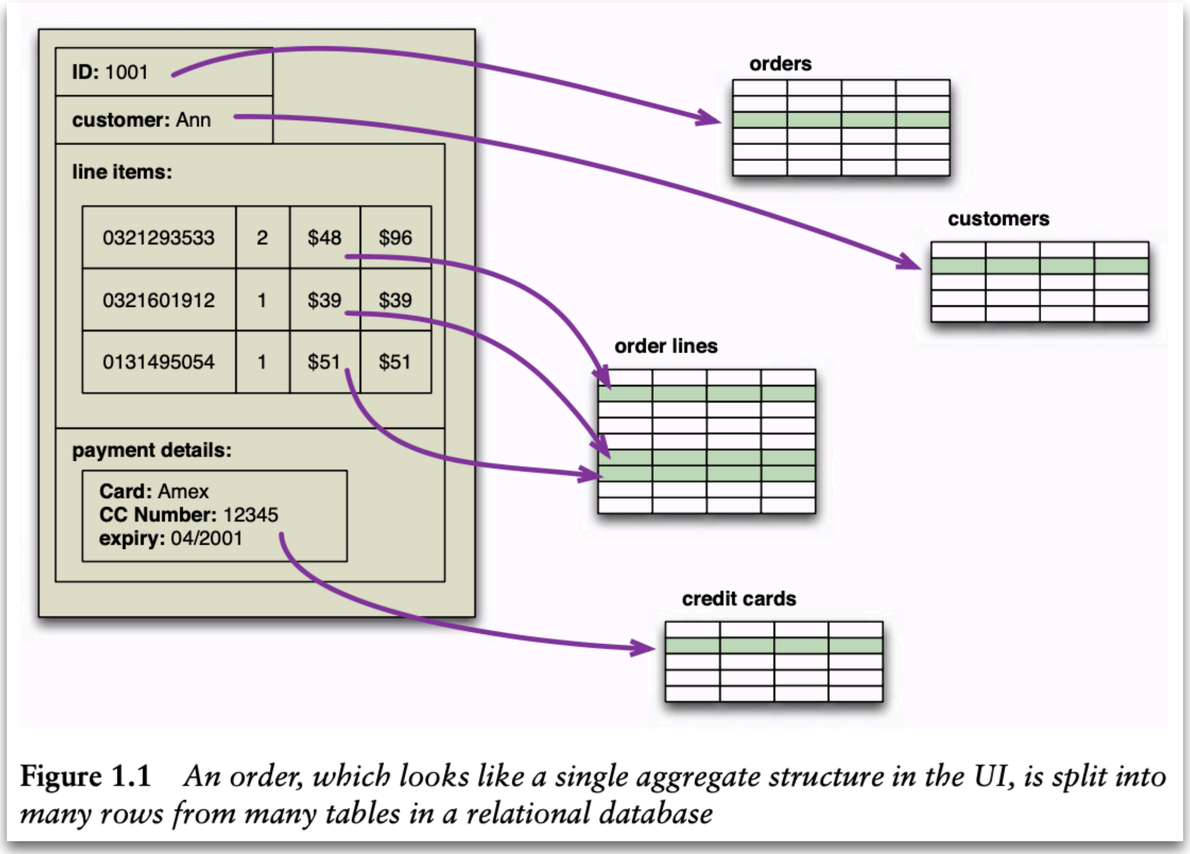


Figure 1.1 *An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Impedance Mismatch

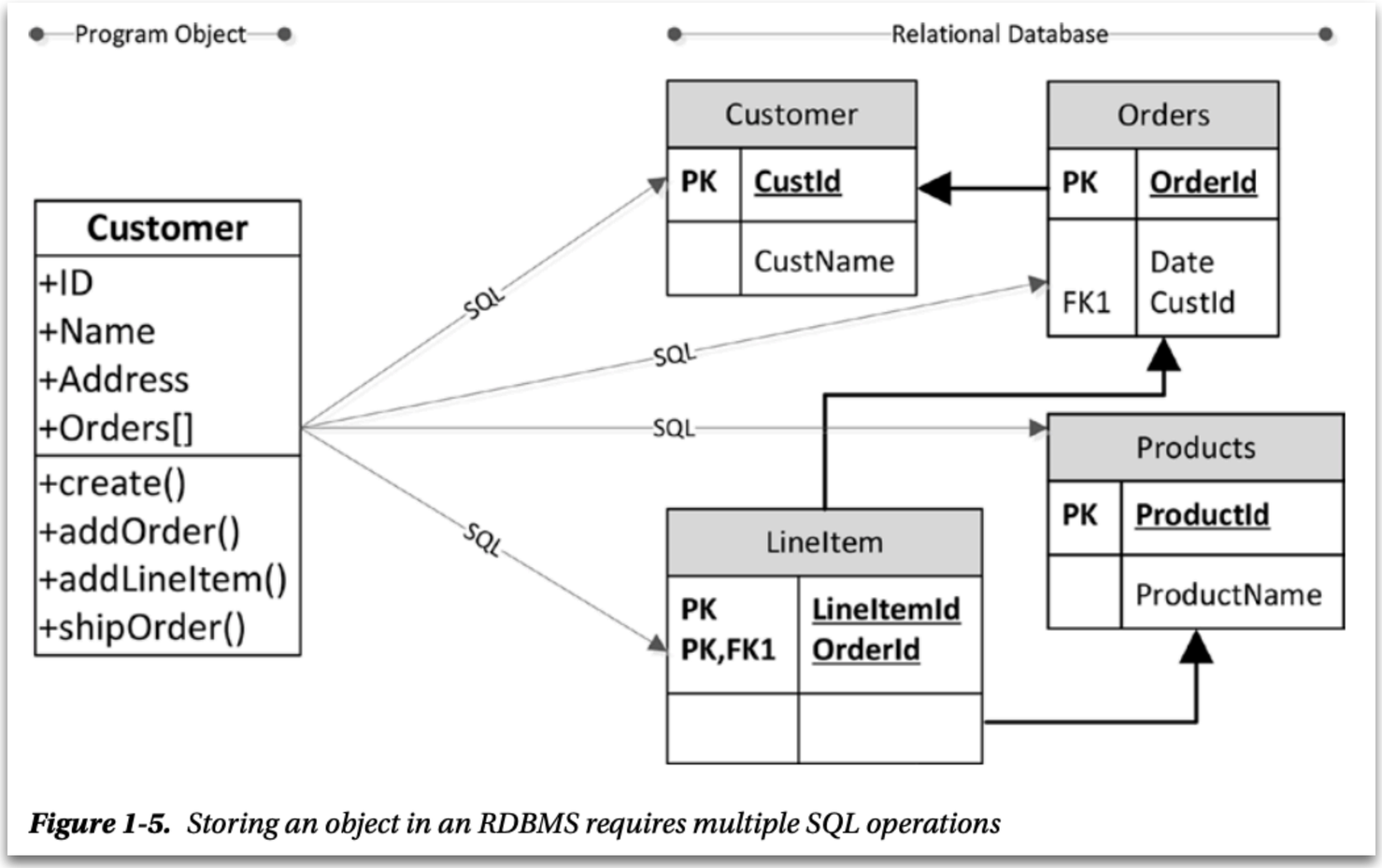


Figure 1-5. *Storing an object in an RDBMS requires multiple SQL operations*

Image from Next Generation Databases, Harrison (2016)

Big Data

- Since the 2000s, we witness to an era of consolidation in web platforms (big tech).
- Results in network effects, i.e. big platforms are more attractive to users.
- This large scale impacts many dimensions — data collection, heterogeneity, storage.
- To handle this growth, computational infrastructures can scale up (i.e. bigger machines to handle more load) or scale out (i.e. more machines to spread the load).
- As infrastructures moved towards clusters, relational database solutions revealed scaling problems adapting as they rely on shared disk subsystems. Solutions exist to overcome these problems (e.g. sharding) but they are "unnatural".
- This mismatch between relational databases and clusters let to the development of alternative routes for data storage solutions.

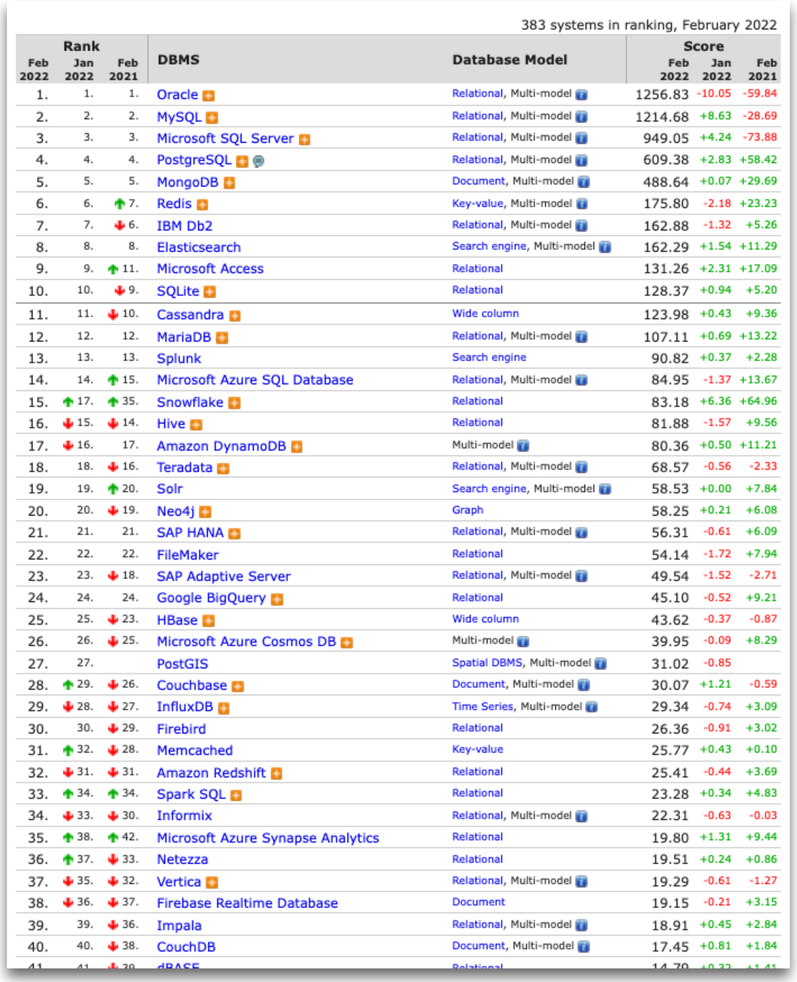
NoSQL Solutions

- Google and Amazon have greatly influenced development in this area, leading early large-scale implementations of data storage solutions based on distributed low-cost components.
- Google BigTable [1] and Amazon Dynamo [2] landmark papers inspired projects and experiments to develop alternative data storage solutions.
- The name "NoSQL" was a convenient handle to group a diverse set of technologies, although a clear and coherent definition still lacks.
- The central characteristics commonly found in NoSQL solutions are: do not use the non-relational paradigm, have a relaxed consistency model, and adopt a schemaless data model.

[1] Bigtable: A Distributed Storage System for Structured Data
[2] Dynamo: Amazon’s Highly Available Key-value Store

Summary

- Relational databases were, and still are, very successful technologies to provide data persistence, concurrency control, and an integration mechanism.
- Limitations of the relational model include the object-relational mismatch and the difficulty in scaling up or out.
- "NoSQL is an accidental neologism (i.e. new word)", whose characteristics include: non-relational, weaker consistency models (scale out is easier), and schemaless data models.
- One of the most important results is polyglot persistence, i.e. diverse data storage ecosystems — one size does not fit all.



383 systems in ranking, February 2022

Rank Feb 2022	Rank Jan 2022	Rank Feb 2021	DBMS	Database Model	Score Feb 2022	Score Jan 2022	Score Feb 2021
1.	1.	1.	Oracle	Relational, Multi-model	1256.83	-10.05	-59.84
2.	2.	2.	MySQL	Relational, Multi-model	1214.68	+8.63	-28.69
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	949.05	+4.24	-73.88
4.	4.	4.	PostgreSQL	Relational, Multi-model	609.38	+2.83	+58.42
5.	5.	5.	MongoDB	Document, Multi-model	488.64	+0.07	+29.69
6.	6.	7.	Redis	Key-value, Multi-model	175.80	-2.18	+23.23
7.	7.	6.	IBM Db2	Relational, Multi-model	162.88	-1.32	+5.26
8.	8.	8.	Elasticsearch	Search engine, Multi-model	162.29	+1.54	+11.29
9.	9.	11.	Microsoft Access	Relational	131.26	+2.31	+17.09
10.	10.	9.	SQLite	Relational	128.37	+0.94	+5.20
11.	11.	10.	Cassandra	Wide column	123.98	+0.43	+9.36
12.	12.	12.	MariaDB	Relational, Multi-model	107.11	+0.69	+13.22
13.	13.	13.	Splunk	Search engine	90.82	+0.37	+2.28
14.	14.	15.	Microsoft Azure SQL Database	Relational, Multi-model	84.95	-1.37	+13.67
15.	17.	35.	Snowflake	Relational	83.18	+6.36	+64.96
16.	15.	14.	Hive	Relational	81.88	-1.57	+9.56
17.	16.	17.	Amazon DynamoDB	Multi-model	80.36	+0.50	+11.21
18.	18.	16.	Teradata	Relational, Multi-model	68.57	-0.56	-2.33
19.	19.	20.	Solr	Search engine, Multi-model	58.53	+0.00	+7.84
20.	20.	19.	Neo4j	Graph	58.25	+0.21	+6.08
21.	21.	21.	SAP HANA	Relational, Multi-model	56.31	-0.61	+6.09
22.	22.	22.	FileMaker	Relational	54.14	-1.72	+7.94
23.	23.	18.	SAP Adaptive Server	Relational, Multi-model	49.54	-1.52	-2.71
24.	24.	24.	Google BigQuery	Relational	45.10	-0.52	+9.21
25.	25.	23.	HBase	Wide column	43.62	-0.37	-0.87
26.	26.	25.	Microsoft Azure Cosmos DB	Multi-model	39.95	-0.09	+8.29
27.	27.		PostGIS	Spatial DBMS, Multi-model	31.02	-0.85	
28.	29.	26.	Couchbase	Document, Multi-model	30.07	+1.21	-0.59
29.	28.	27.	InfluxDB	Time Series, Multi-model	29.34	-0.74	+3.09
30.	30.	29.	Firebird	Relational	26.36	-0.91	+3.02
31.	32.	28.	Memcached	Key-value	25.77	+0.43	+0.10
32.	31.	31.	Amazon Redshift	Relational	25.41	-0.44	+3.69
33.	34.	34.	Spark SQL	Relational	23.28	+0.34	+4.83
34.	33.	30.	Informix	Relational, Multi-model	22.31	-0.63	-0.03
35.	38.	42.	Microsoft Azure Synapse Analytics	Relational	19.80	+1.31	+9.44
36.	37.	33.	Netezza	Relational	19.51	+0.24	+0.86
37.	35.	32.	Vertica	Relational, Multi-model	19.29	-0.61	-1.27
38.	36.	37.	Firebase Realtime Database	Document	19.15	-0.21	+3.15
39.	39.	36.	Impala	Relational, Multi-model	18.91	+0.45	+2.84
40.	40.	38.	CouchDB	Document, Multi-model	17.45	+0.81	+1.84
41.	41.	39.	dBASE	Relational	14.70	+0.33	+1.41
42.	43.	40.	Presto	Relational	14.28	+0.91	+1.96
43.	42.	41.	Greenplum	Relational, Multi-model	13.52	-0.44	+1.31
44.	44.	54.	ClickHouse	Relational, Multi-model	12.82	+0.41	+5.16
45.	45.	46.	Amazon Aurora	Relational, Multi-model	12.23	+0.49	+3.26
46.	46.	48.	etcd	Key-value	11.71	+0.16	+3.07
47.	48.	47.	Datastax Enterprise	Wide column, Multi-model	10.75	+0.86	+1.95
48.	47.	44.	H2	Relational, Multi-model	10.11	+0.12	+0.80
49.	50.	50.	Hazelcast	Key-value, Multi-model	9.50	-0.03	+1.08
50.	51.	43.	MarkLogic	Multi-model	9.46	+0.27	+0.00
51.	49.	45.	Realm	Document	9.42	-0.17	+0.27
52.	53.	52.	Kdb+	Time Series, Multi-model	9.11	+0.34	+1.33
53.	52.	51.	Google Cloud Firestore	Document	9.06	+0.14	+1.01
54.	54.	53.	Algolia	Search engine	8.92	+0.55	+1.18
55.	57.	49.	Oracle Essbase	Relational	8.41	+0.99	-0.20
56.	56.	60.	Microsoft Azure Search	Search engine	7.90	+0.38	+1.63
57.	55.	57.	Sphinx	Search engine	7.63	-0.40	+1.07
58.	58.	56.	CockroachDB	Relational	7.47	+0.50	+0.87
59.	59.	65.	SingleStore	Relational, Multi-model	7.33	+0.67	+2.12
60.	64.	72.	Riak KV	Key-value	6.92	+0.83	+1.93
61.	60.	74.	Microsoft Azure Data Explorer	Relational, Multi-model	6.79	+0.20	+2.05
62.	61.	71.	Jackrabbit	Content	6.54	+0.09	+1.55
63.	66.	75.	Ignite	Multi-model	6.50	+0.48	+1.77
64.	63.	59.	Interbase	Relational	6.47	+0.34	+0.12
65.	68.	58.	Ingres	Relational	6.44	+0.67	-0.05
66.	62.	62.	Prometheus	Time Series	6.39	+0.12	+0.63
67.	65.	55.	Ehcache	Key-value	6.38	+0.29	-0.50
68.	69.	61.	SAP SQL Anywhere	Relational	5.76	+0.10	-0.19
69.	71.	66.	HyperSQL	Relational	5.74	+0.21	+0.56
70.	72.	73.	Microsoft Azure Table Storage	Wide column	5.64	+0.22	+0.70
71.	67.	63.	Aerospike	Key-value, Multi-model	5.61	-0.33	-0.06
72.	70.	79.	Graphite	Time Series	5.58	+0.00	+0.96
73.	78.	69.	ArangoDB	Multi-model	5.40	+0.67	+0.33
74.	73.	111.	Virtuoso	Multi-model	5.39	+0.02	+3.02
75.	76.	70.	Google Cloud Datastore	Document	5.25	+0.41	+0.23
76.	74.	64.	Derby	Relational	5.22	+0.25	-0.19
77.	75.	78.	SAP IQ	Relational	5.11	+0.26	+0.49
78.	77.	76.	Adabas	Multivalue	5.06	+0.32	+0.40
79.	80.	68.	OrientDB	Multi-model	5.03	+0.47	-0.10
80.	81.	80.	Oracle NoSQL	Multi-model	4.84	+0.43	+0.33
81.	79.	67.	OpenEdge	Relational	4.67	+0.01	-0.47
82.	82.	100.	TimescaleDB	Time Series, Multi-model	4.37	+0.15	+1.51
83.	83.	77.	MaxDB	Relational	4.23	+0.18	-0.41
84.	89.	85.	Google Cloud Bigtable	Wide column	4.17	+0.54	+0.46
85.	87.	84.	IBM Cloudant	Document	3.94	+0.18	+0.14
86.	86.	82.	Accumulo	Wide column	3.93	+0.05	-0.20
87.	84.	86.	SAP Advantage Database Server	Relational	3.92	-0.08	+0.33
88.	90.	81.	UniData,UniVerse	Multivalue	3.89	+0.27	-0.26
89.	88.	94.	RocksDB	Key-value	3.89	+0.19	+0.69
90.	85.	113.	ScyllaDB	Multi-model	3.88	-0.03	+1.73
91.	96.	89.	RavenDB	Document, Multi-model	3.82	+0.57	+0.47
92.	92.	110.	Google Cloud Spanner	Relational	3.69	+0.23	+1.22
93.	91.	96.	EXASOL	Relational	3.60	+0.07	+0.47
94.	95.	115.	TiDB	Relational, Multi-model	3.51	+0.25	+1.38
95.	99.	88.	PouchDB	Document	3.46	+0.48	+0.08
96.	93.	104.	Apache Druid	Multi-model	3.40	-0.04	+0.74
97.	94.	90.	Apache Phoenix	Relational	3.32	+0.06	+0.04
98.	102.	93.	InterSystems Caché	Multi-model	3.26	+0.36	+0.03
99.	98.	92.	LevelDB	Key-value	3.25	+0.17	+0.01
100.	101.	91.	Infinispan	Key-value	3.23	+0.29	-0.03
101.	107.	87.	Oracle Berkeley DB	Multi-model	3.11	+0.43	-0.29
102.	97.	83.	RethinkDB	Document, Multi-model	3.08	-0.06	-0.98
103.	100.	109.	4D	Relational	3.07	+0.11	+0.59
104.	105.	103.	Apache Drill	Multi-model	3.03	+0.30	+0.31
105.	109.	98.	IMS	Navigational	3.01	+0.38	+0.02
106.	108.	118.	Amazon Neptune	Multi-model	2.99	+0.36	+0.92
107.	103.	114.	GraphDB	Multi-model	2.93	+0.07	+0.79
108.	104.	95.	Apache Jena - TDB	RDF	2.90	+0.06	-0.26
109.	106.	166.	Trino	Relational	2.88	+0.19	+1.87
110.	110.	101.	Oracle Coherence	Key-value	2.67	+0.12	-0.15
111.	111.	102.	Percona Server for MySQL	Relational	2.54	+0.03	-0.21
112.	115.	99.	LMDB	Key-value	2.53	+0.29	-0.34
113.	113.	107.	CloudKit	Document	2.41	+0.12	-0.10
114.	118.	97.	RRDtool	Time Series	2.40	+0.32	-0.60
115.	112.	105.	JanusGraph	Graph	2.36	-0.03	-0.17
116.	114.	119.	EDB Postgres	Relational, Multi-model	2.36	+0.09	+0.31
117.	121.	163.	YugabyteDB	Relational, Multi-model	2.34	+0.39	+1.30
118.	116.	108.	Amazon CloudSearch	Search engine	2.31	+0.10	-0.18
119.	120.	147.	TigerGraph	Graph	2.24	+0.22	+0.91
120.	117.	112.	Amazon SimpleDB	Key-value	2.18	0.00	-0.03
121.	119.	124.	Tibero	Relational	2.04	+0.01	+0.16
122.	124.	137.	Stardog	Multi-model	1.98	+0.09	+0.52
123.	122.		SpatiaLite	Spatial DBMS, Multi-model	1.98	+0.04	
124.	123.	125.	IBM Db2 warehouse	Relational	1.94	+0.03	+0.07
125.	138.	117.	GridGain	Multi-model	1.94	+0.39	-0.14
126.	136.	122.	jBASE	Multivalue	1.92	+0.30	-0.05
127.	125.	120.	MonetDB	Relational, Multi-model	1.86	-0.02	-0.18

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

Principles of NoSQL Databases

Data Models

- A data model is the model through which we perceive and manipulate our data.
- The data model describes how a database user interacts with the data.
- This is distinct from a storage model, which describes how data is stored internally.
- Also distinct from the conceptual data model, which describes the high-level concepts and relationships used to describe a domain.
- The relational data model is the dominant data model, which includes the concepts of tables (relations), columns (attributed) and rows (tuples).
- The NoSQL ecosystem is dominated by four main categories of data models: key-value, document, column / tabular, and graph.

Aggregates

- The relational model divides the information into tuples (rows).
- A tuple is a limited data structure:
  - It captures a set of literal values;
  - It does not support nested tuples (records);
  - It is not possible to use lists as tuple values.
- Aggregate orientation takes a different approach, recognizing that is often necessary to operate on data in units that have a more complex structure than a set of tuples.
  - Example: a complex data record that allow list and other data record structures to be nested inside it.
- The term "aggregates" refers to these complex records, i.e. a collection of related objects that are treated as a unit. It corresponds to a unit for data manipulation and management of consistency.
- Typically, "aggregate structures" are easier for programmers to work with (e.g. retrieve, update).

Example of Relations and Aggregates

- Consider an e-commerce website, where it is necessary to store information about the product catalog, orders, shipping and billing addresses, and payment data.

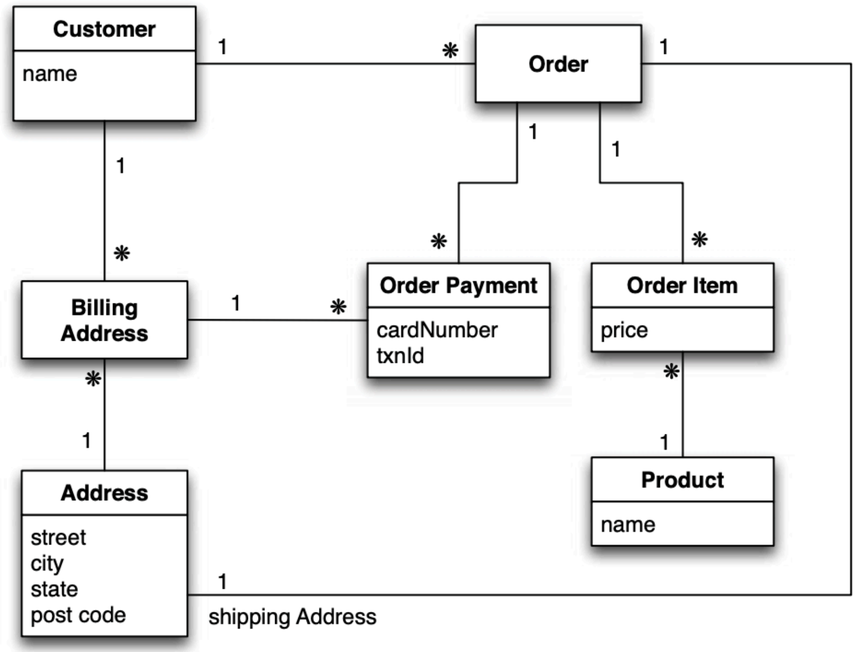


Figure 2.1 *Data model oriented around a relational database (using UML notation [Fowler UML])*

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figure 2.2 *Typical data using RDBMS data model*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Example Aggregate Data Model

- The same model in more aggregate-oriented terms. UML compositions (black-diamond marker) are used to show that data fits into the aggregation structure.

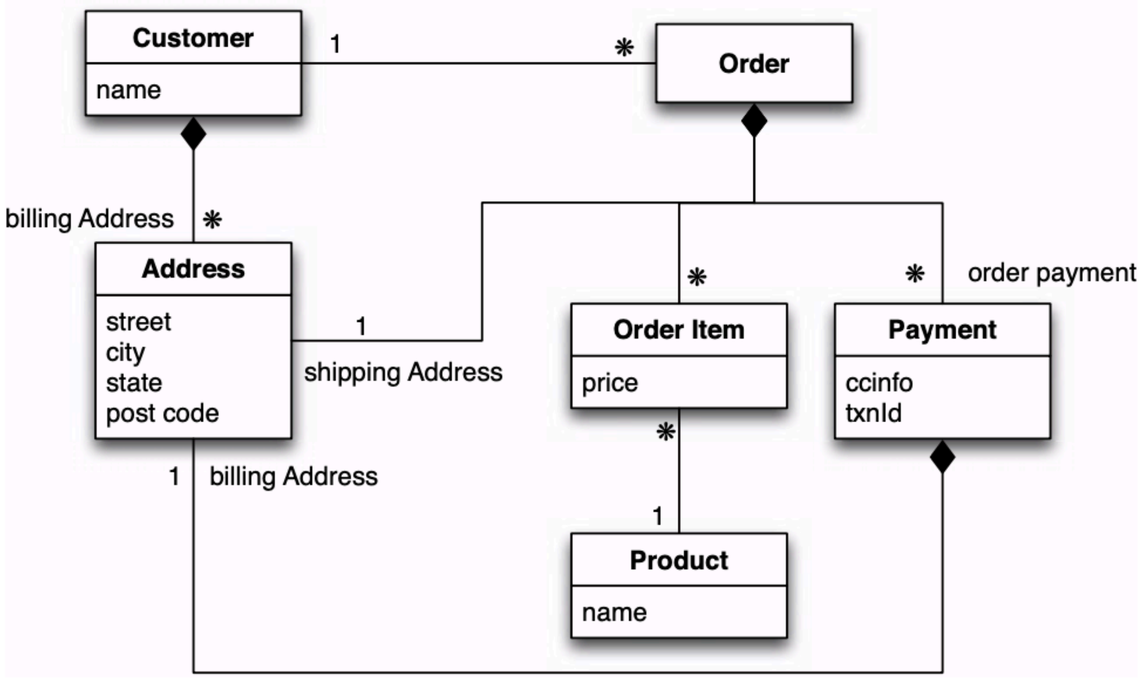


Figure 2.3 *An aggregate data model*

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Example Aggregate Data Model (2)

- A single logical address record appears three times in the example data, but instead of using IDs it is treated as a value and copied each time.
- With aggregates, we can copy the whole address structure into the aggregate as we need to.
- The product name is included as part of the order item here — this kind of denormalization is similar to the tradeoffs with relational databases, but is more common with aggregates because the goal is to minimize the number of aggregates accessed during a data interaction.
- The important thing to notice isn't the particular way the aggregate boundary are drawn, so much as the fact of how data is accessed — and make that part of the processes when developing the application data model.

Example Alternative Aggregate Data Model

- Aggregate boundaries could be drawn differently, putting all the orders for a customer into the customer aggregate.

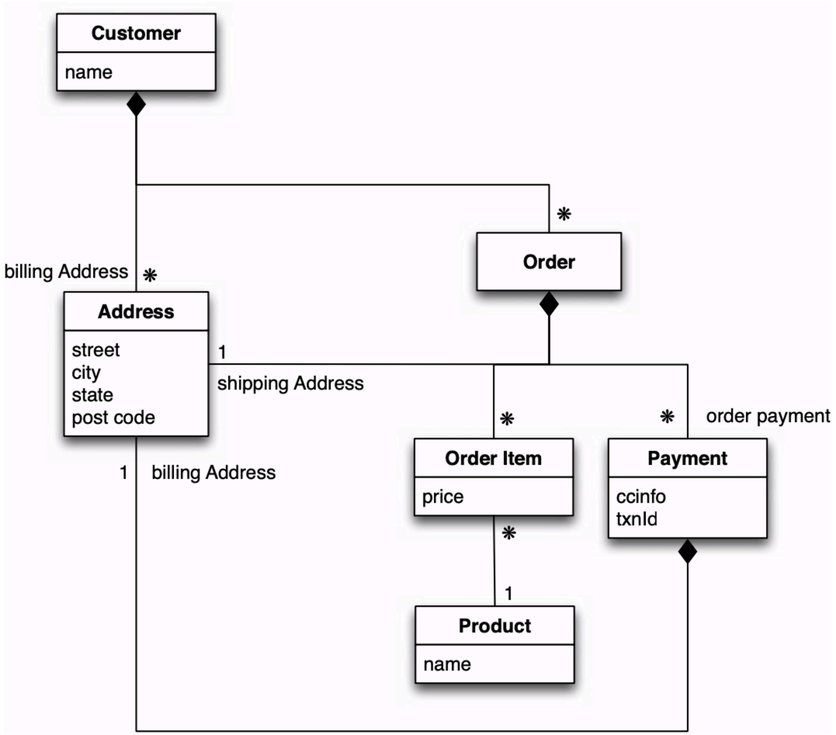


Figure 2.4 *Embed all the objects for customer and the customer's orders*

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]

  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Impact of Aggregate-Oriented Models

- The relational mapping captures the various data elements and their relationships reasonably well, but it does so without any notion of an aggregate entity.
- In the relational model, relationships can be expressed in terms of foreign key relationships, but there is nothing to distinguish relationships that represent aggregations from those that don’t.
- Thus, the database can’t use knowledge of aggregate structure to help it store and distribute the data.
- When working with aggregate-oriented databases, there is a clearer semantics to consider by focusing on the unit of interaction with the data storage. It is, however, not a logical data property: it’s all about how the data is being used by applications—a concern that is often outside the bounds of data modeling.
- Relational databases have no concept of aggregate within their data model, they're labeled as aggregate-ignorant. In the NoSQL world, graph databases are also aggregate-ignorant.

Aggregate-Ignorant Models

- Being aggregate-ignorant is a characteristic, not a problem.
- It's often difficult to draw aggregate boundaries well, particularly if the same data is used in many different contexts.
  - An order makes a good aggregate when a customer is making and reviewing orders, and when the retailer is processing orders.
  - However, if a retailer wants to analyze its product sales over the last few months, then an order aggregate becomes a trouble. To get to product sales history, you'll have to dig into every aggregate in the database.
- An aggregate structure may help with some data interactions but be an obstacle for others.
- An aggregate-ignorant model allows you to easily look at the data in different ways, so it is a better choice when you don't have a primary structure for manipulating your data.

Transactions in Aggregate-Oriented Models

- A strong reason for aggregate orientation is that it helps greatly with running on a cluster (one of the main arguments for the rise of NoSQL).
- When running on a cluster, there is a need to minimize the number of nodes queried to gather data. By explicitly including aggregates, the database is given important information about which bits of data will be manipulated together, and thus should live on the same node.
- Aggregates have an important impact for transactions. With atomicity, rows spawning many tables are updated as a single operation that either succeed or fail it its entirety.
- In general, aggregate-oriented databases don't have ACID transactions that span multiple aggregates. Instead, they support atomic manipulation of a single aggregate at a time.
- Handling multiple aggregates in an atomic way needs to be managed in the application code.

Key-Value and Document Data Models

- Key-value and document database are strongly aggregate-oriented, i.e. they are primarily constructed through aggregates. Both database types consists of many aggregates with each aggregate having an access key or ID.
- The difference between the two models is that
  - in key-value model, the aggregate is opaque to the database;
  - while in the document model, the database sees the structure in the aggregate.
- With a key-value store, an aggregate can only be accessed by lookup based on its key.
- With a document database, queries can use the fields in the aggregate for lookup; parts of the aggregate can be retrieved rather than the whole thing; and indexes can be created based on the contents of the aggregate.
- In practice, database systems combine features from both models.

Column-Family Databases

- **Data is stored in sets of columns** (column families), accessed using a unique key.
  - Each data record is a **key-value pair**, where the value is a set of column families.
  - **Column families** are groups of related data (columns) that is often accessed together.
- **Two-level aggregate structure**:
  - First level access is identical to a key-value store, where keys are used to select aggregates.
  - Second level access allows direct selection of individual columns or column aggregates.

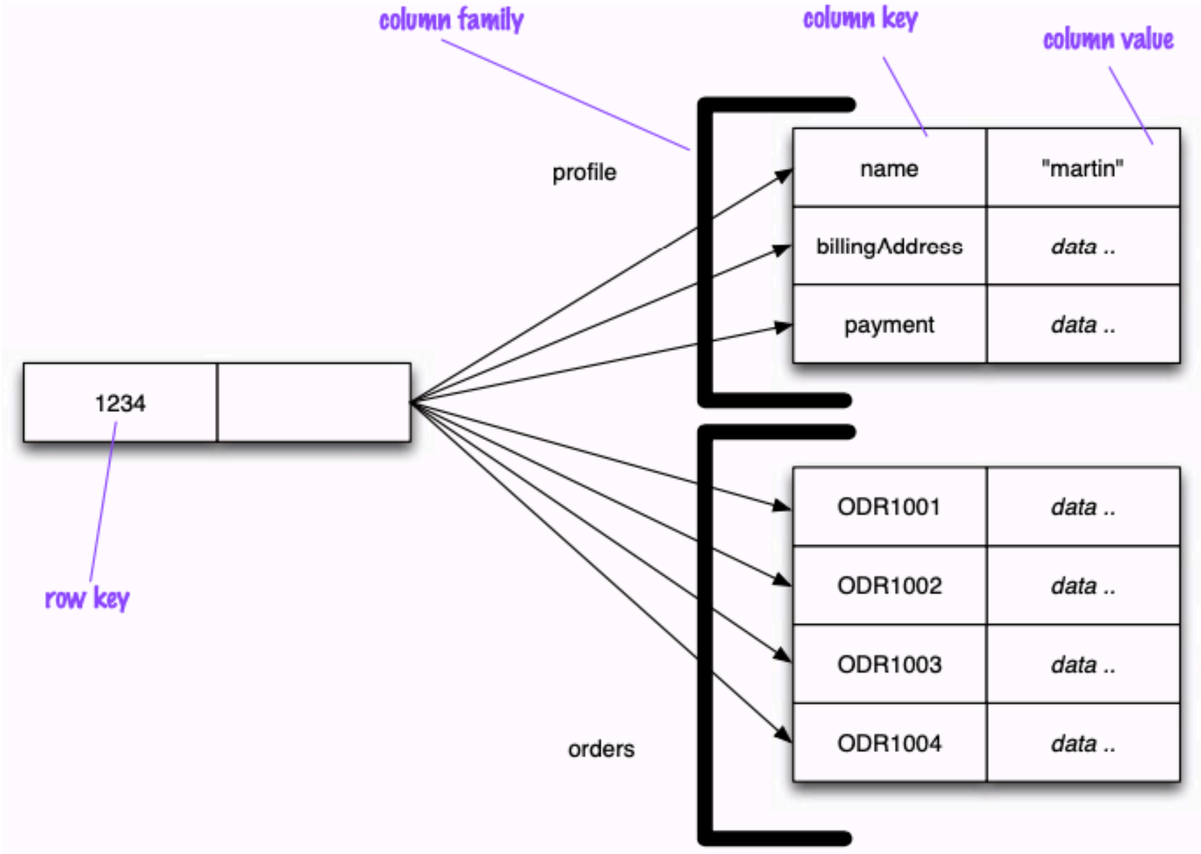


Figure 2.5 *Representing customer information in a column-family structure*

Image from "NoSQL Distilled", Sadalage and Fowler (2012).

Column-Family Databases

- Having rows as a unit of storage helps write performance. In scenarios where writes are rare, but read a few columns of many rows at once, it's better to store groups of columns for all rows as the basic storage unit.
- The column acts as the unit for access, with the assumption that data for a particular column family will be usually accessed together.
- Row-oriented: each row is an aggregate (e.g. customer with ID 123) with column families representing data (e.g. profile, order history) within that aggregate.
- Column-oriented: each column family defines a record type (e.g. customer profiles) with rows for each of the individual records. Column families can contain a single column (skinny) or many columns (wide).
- Since the database knows about this grouping of data, it can use this information to define storage and access behavior.

Summary of Aggregate-Oriented Models

- Aggregate-oriented models share the notion of an aggregate (complex unit of data) indexed by a key that can be used for lookup.
- This aggregate is central to running a distributed system, as the database will ensure that all the data for an aggregate is stored together on one node.
- The key-value model treats the aggregate as an opaque whole, only key-based lookups are possible.
- The document model makes the aggregate transparent to the database, allowing partial retrievals and lookups based on aggregate values.
- The column-family model divides the aggregate into column families, allowing the database to treat them as units of data within the row aggregate.

Summary of Aggregate Data Models

- An aggregate is a collection of data that is manipulated as a unit.
- Aggregates form the boundaries for ACID operations with the database.
- Key-value, document, and column-family database are forms of aggregate-oriented databases.
- Aggregates make it easier for the database to manage data storage over clusters.
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate.
- Aggregate-ignorant databases are better when interactions use data organized in many different formations.

Types of NoSQL Databases

NoSQL Decision Guide

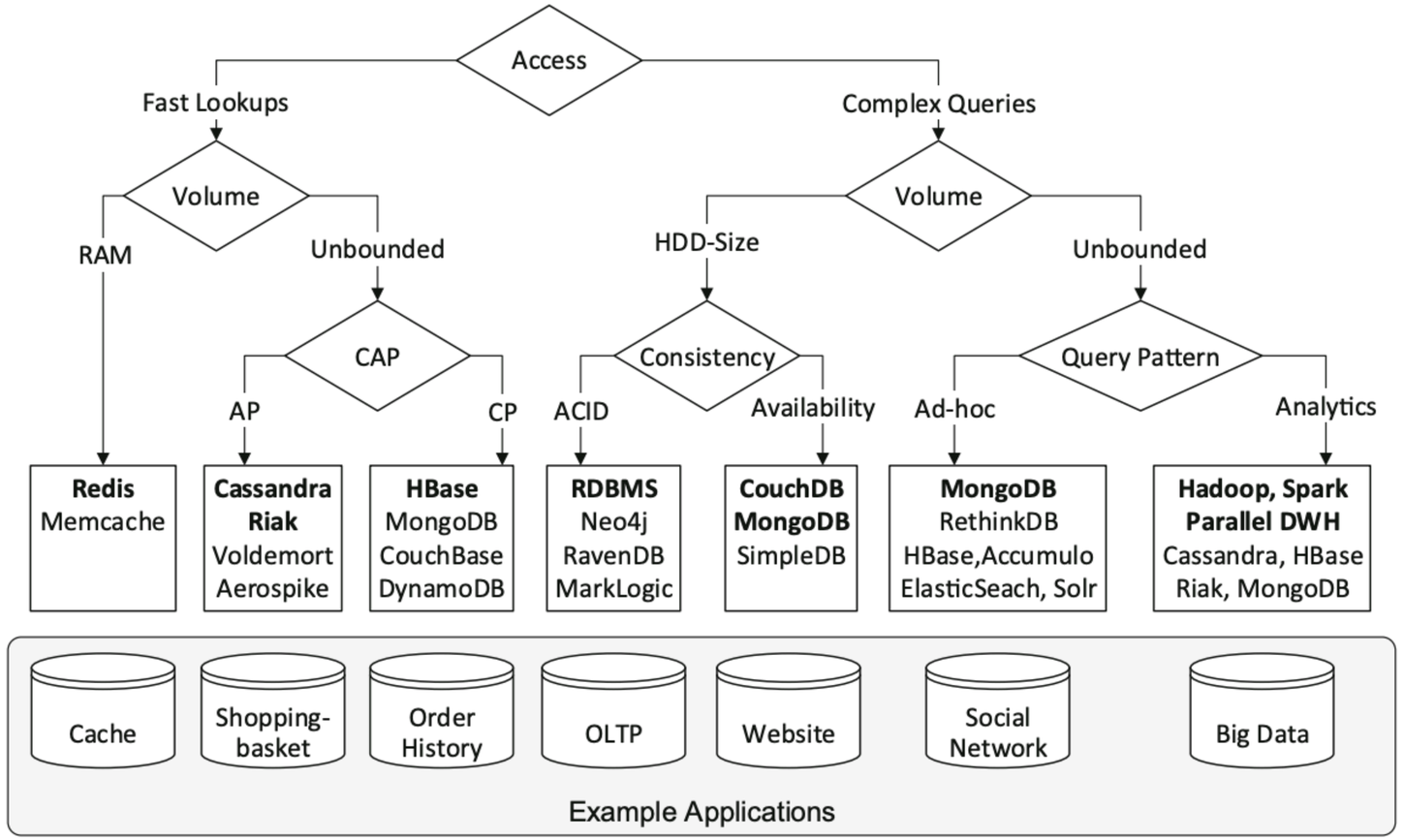


Fig. 6 A decision tree for mapping requirements to (NoSQL) database systems

Image from "NoSQL database systems: a river and decision guide", Gessert et al. (2016)

Key-Value Stores

Key-Value Store

- Key-value store are part of the simplest NoSQL solutions from an API perspective.
- The client can [set , get] the value for a key, or [delete] a key.
- The value is opaque to the data store (i.e. it is not exposed).
  - Note that some key-value stores offer complex data structures (e.g. Redis).
- The application is responsible to understand and manipulate the data stored.
- Given the simplicity of the model (key-based access pattern), the result is:
  - very high performance and
  - easy scalability.

Key-Value Store

(a). Using a basic key-value data model for a health management system

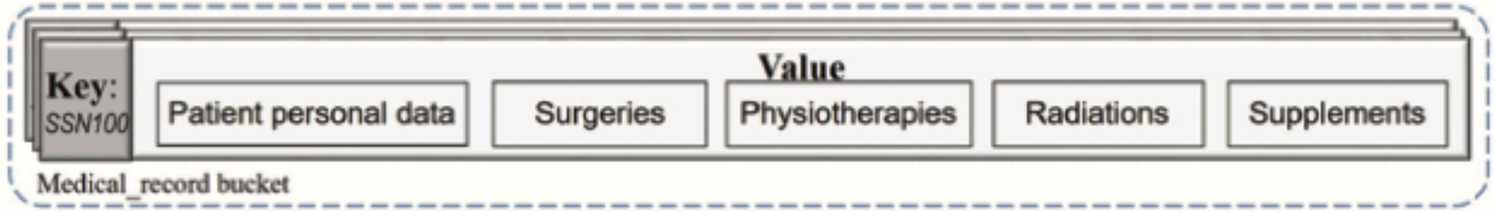


Image from "A Survey on NoSQL Stores", Davodian et al. (2018)

Key-Value Concepts

- Each data record corresponds to a **key-value pair**.
- **Keys** are unique and provide access to each pair.
  - Keys can be application defined or generated by the DBMS.
  - Examples include: session ids, hashes, URIs, filenames, timestamps.
- **Values** represent data with arbitrary data types, structure, and size.
  - Serialization/Deserialization of data is the responsibility of the client application.
- In practice, implementations exhibit additional features over this core set.

Suitable Use Cases

- High performance requirements (in-memory, fast).
- Mostly reads workload over a single object.
- Examples: caching, storing session information, user data, messaging middleware, etc.
- See "How to take advantage of Redis just adding it to your stack", Salvatore Sanfilippo (2011), http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html

Key-Value Solutions

- Oracle Berkeley DB, www.oracle.com/database/berkeley-db (1994) [GNU] (embedded)
- Memcached, memcached.org (2003) [BSD]
- Redis, redis.io (2009) [BSD]
- Riak, riak.com (2009) [Apache License]

- Amazon DynamoDB, aws.amazon.com/dynamodb (2012) [Proprietary]
- Microsoft Zure Cosmos DB, azure.microsoft.com/en-us/services/cosmos-db (2017) [Proprietary]

- More: db-engines.com/en/ranking/key-value+store

Postgres Support

Key-Value Store Support in Postgres

- Postgres supports key-value storage with the hstore data type.
- Keys and values are text strings.
- Keys are unique and the order of the pairs ir not significant.

- Model abstraction is the main advantage.
- Scalability is still dependent on Postgres scalability features.
- hstore is an additional data type, Postgres features (transactions, ACID, etc) are the same.

- Documentation: www.postgresql.org/docs/current/hstore.html

Key-Value Store Support in Postgres

```
CREATE EXTENSION HSTORE;

CREATE TABLE mytable (h hstore);

INSERT INTO mytable VALUES ('a=>b, c=>d');

SELECT h['a'] FROM mytable;
 h
---
 b
(1 row)


UPDATE mytable SET h['c'] = 'new';
SELECT h FROM mytable;
           h
----------------------
 "a"=>"b", "c"=>"new"
(1 row)
```

Document Databases

Document Database

- Documents are the main concept in document databases.
- The DBMS stores and retrieves documents.
- Documents are self-describing objects (e.g. JSON, BSON, XML).
- Documents stored are similar to each other but do not have to be.
- Similar to key-value stores but where the value-part is **visible** to the DBMS and can be used for access and manipulated.
- The schema of the data can differ across documents in the same collection.

Document Database

(c). Using the document data model for the McGraw-Hill Education

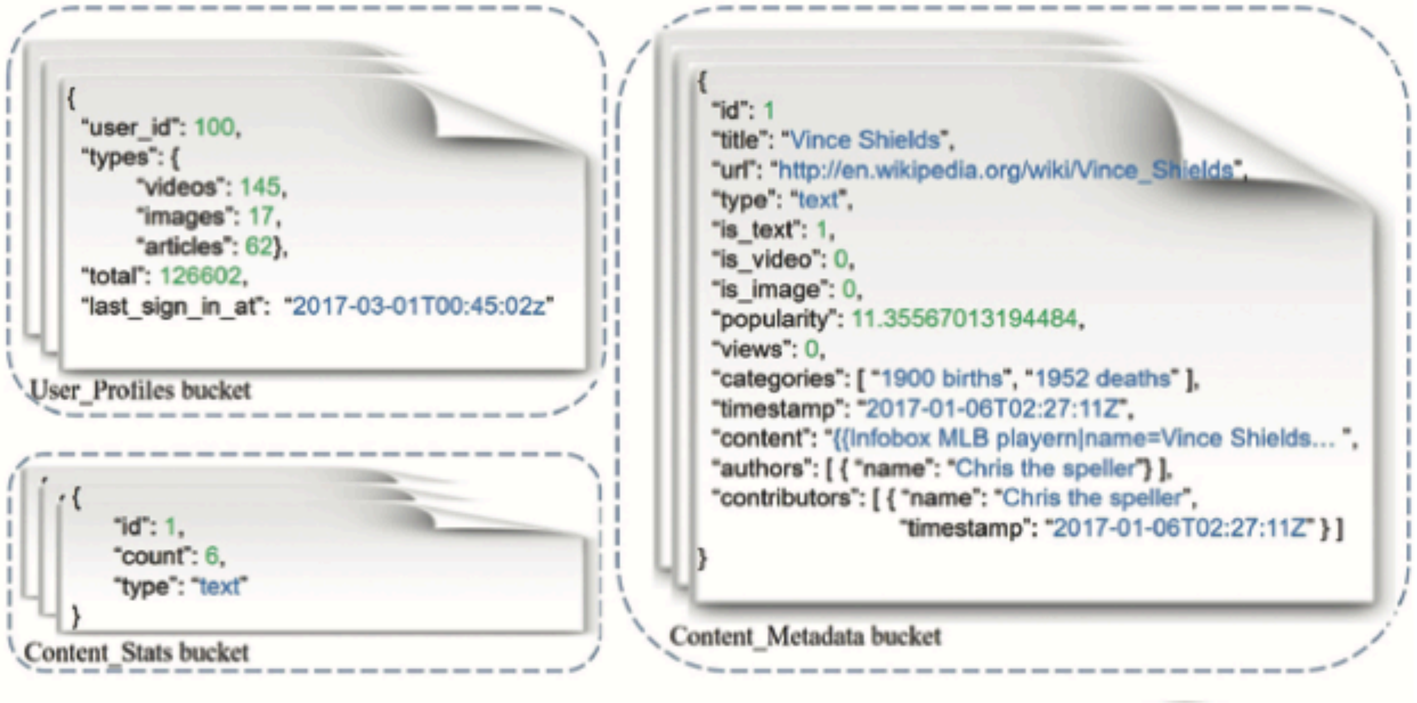


Image from "A Survey on NoSQL Stores", Davodian et al. (2018)

Document Database Concepts

- Each data record corresponds to a **document**.
- Document access can be made:
  - Using unique keys;
  - Document attributes.
- **Documents** are:
  - Self-describing data objects;
  - There are no empty attributes, a missing value is assumed to be not set or not relevant;
  - New attributes can be created in individual documents without the need to change the existing documents;
  - Schema can be different among documents in the same collection.
- Documents are organized in **collections**.
- The DBMS has visibility over the document, which can be used for access or manipulation.

Example Document

```
{ "firstname": "Martin",
  "likes": [ "Biking",
             "Photography" ],
  "lastcity": "Boston",
  "lastVisited":
}
```

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
  ],
  "lastcity": "Chicago"
}
```

Image from "NoSQL Distilled", Sadalage and Fowler (2012).

Suitable Use Cases

- High availability requirements (replication and scaling).
- Flexible schema data, e.g. multiple entity variants, evolving schemas.
- Examples: event logging, content management systems, analytics, e-commerce.

Document Database Solutions

- MongoDB, mongodb.com (2009) [GNU]
- CouchDB, couchdb.apache.org (2005) [Apache]
- Couchbase, couchbase.com (2011) [Apache]

- Google Firebase Realtime Database, firebase.google.com/products/realtime-database (2012) [Proprietary]
- Google Cloud Firestore, firebase.google.com/products/firestore (2017) [Proprietary]
- Amazon DocumentDB, aws.amazon.com/documentdb (2019) [Proprietary]

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

Postgres Support

Document Store Support in Postgres

- Postgres supports document storage with the **json and jsonb data types**.
- An additional datatype **jsonpath** exists to support efficient querying.

- Model abstraction is the main advantage:
  - E.g. reduce impedance mismatch, deal with variable schemas, build aggregates.
- Scalability is still dependent on Postgres scalability features.
- json and jsonb are additional data types, Postgres features (transactions, ACID, etc) are the same.

- Documentation: www.postgresql.org/docs/current/datatype-json.html

Document Store Support in Postgres

```
CREATE TABLE docs (
  id SERIAL PRIMARY KEY,
  info jsonb NOT NULL
);

INSERT INTO docs (info)
VALUES
('{"student": "Alice",
  "courses": {"name": "Databases", "grade": 18}}'),
('{"student": "Max",
  "courses": [
    {"name": "Web", "grade": 14},
    {"name": "Databases", "grade": 17}]}'),
('{"student": "Rita",
  "courses": {"name": "Databases", "grade": 18}}');

SELECT
  info -> 'student',
  info -> 'student' -> 'courses'
FROM docs;

SELECT *
FROM docs
WHERE info -> 'courses' ->> 'name' = 'Databases';
```

Column-Family Databases

Column-Family Database

- **Data is stored in sets of columns** (column families), accessed using a unique key.
- Each data record is a **key-value pair**, where the value is a set of column families.
- **Column families** are groups of related data (columns) that is often accessed together.

- **Two-level aggregate structure**:
  - First level access is identical to a key-value store, where keys are used to select aggregates.
  - Second level access allows direct selection of individual columns or column aggregates.

- Access is possible at **various aggregate levels** — record, column-family, column.

Column-Family Database

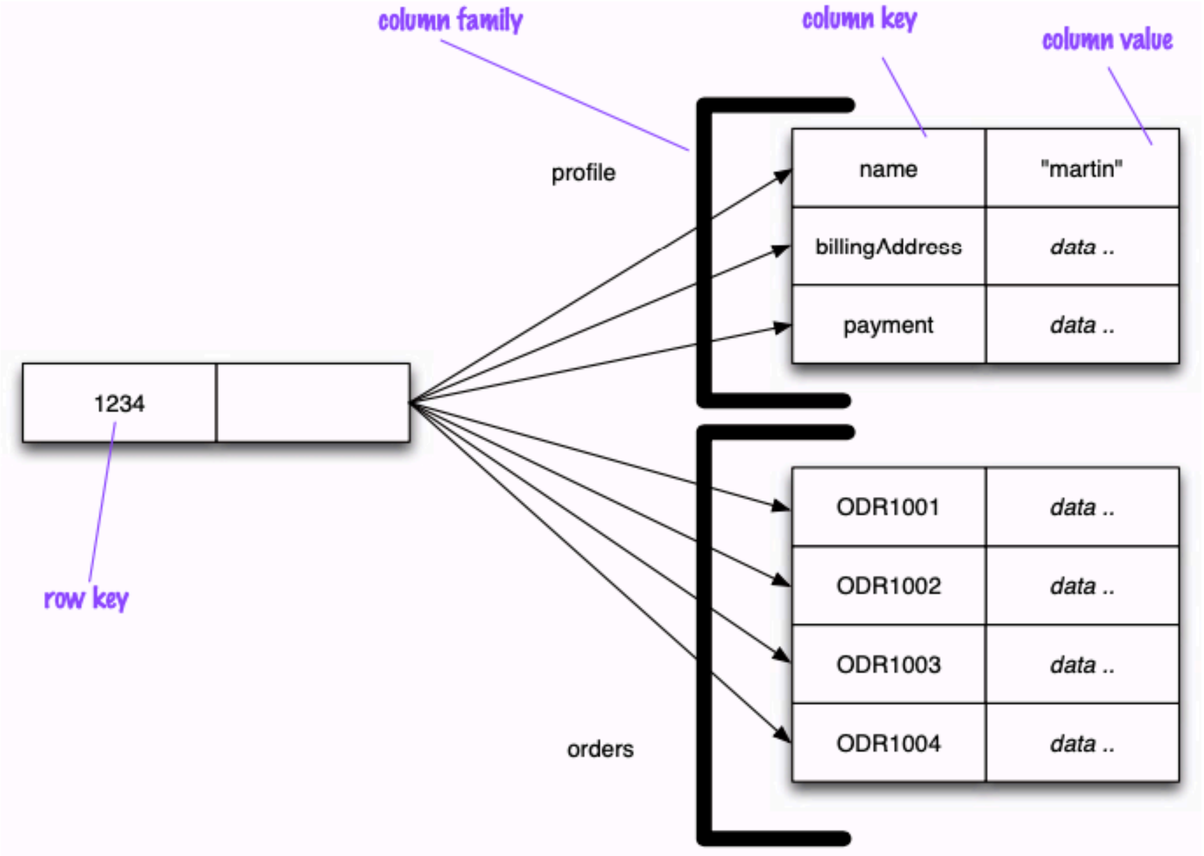


Figure 2.5 *Representing customer information in a column-family structure*

Image from "NoSQL Distilled", Sadalage and Fowler (2012).

Column-Family Database Concepts

- A **row** corresponds to a complete data record, and is composed by:
  - a unique identifying **key** (at the top-level data structure, the keyspace);
  - a collection of **column families**.
- Each **column family** is composed by:
  - a **column-family key**;
  - a set of **columns**, representing related data.
- The **column** is the basic data item, and is composed by:
  - a **column key**;
  - a **value**;
  - a **timestamp**, which are used to expire data, resolve write conflicts, etc.
- Some systems have the concept of **super columns-families**, i.e. column-families containing other column-families.

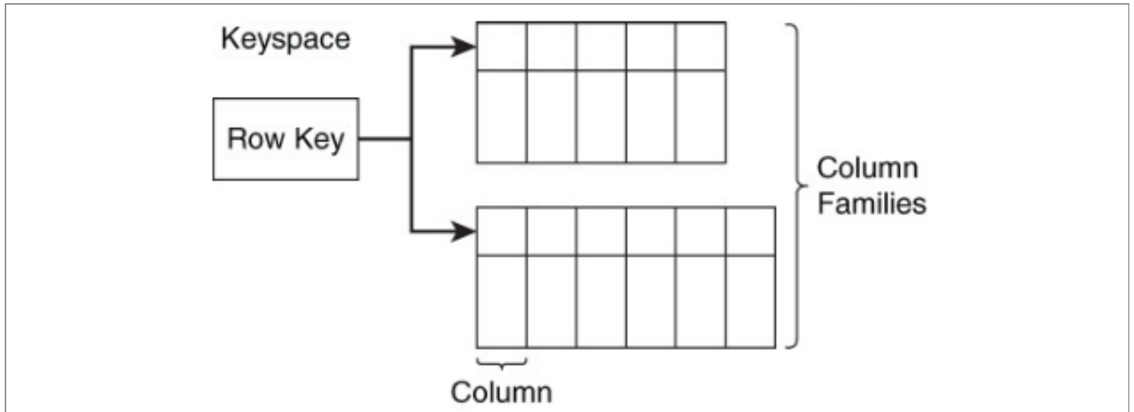


Figure 10.1 *A keyspace is a top-level container that logically holds column families, row keys, and related data structures. Typically, there is one keyspace per application.*

Image from "NoSQL for Mere Mortals", Sullivan (2015).

Contrast with Key-Value and Document Paradigms

- Comparison with **key-value databases**:
  - Both paradigms have key-based access to an aggregate.
  - In key-value databases, the aggregate is opaque to the database.
  - In column-family databases, the aggregate has structure that is visible to the database, and can be used to improve distribution and access to partial value data (instead of always accessing the complete value).

- Comparison **with document databases**:
  - Both paradigms have key-based access to an aggregate.
  - In document databases, the document is a single unit that is access as a whole.
  - In column-family databases, a second dimension is available to inform the database, allowing for partial readings of the (first-level) aggregate, and distribution.

Terminology

- Terminology, and definitions, are less established in the column-family paradigm.

- "Column-family" databases are **also known as "wide-column"** databases.
  - "Column-family" emphasizes the column grouping features.
  - "Wide-column" emphasizes the support for a large number of columns.

- **Not to confuse with** "column-oriented DBMS" or "columnar DBMS".
  - Databases where data is physically stored in columns instead of rows.
  - Otherwise can be used just as standard relational databases (e.g. SQL).
  - Are more efficient when querying a subset of columns (no need to read not needed columns).
  - MonetDB is an example of a relational column-oriented open-source system.

Origins - Google Bigtable

- Paper "Bigtable: A Distributed Storage System for Structured Data" (2006) [pdf]
  - *Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.*
  - *Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability.*
  - *Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage*
  - *Data is indexed using row and column names that can be arbitrary strings.*
  - *A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. (row:string, column:string, time:int64) → string*

Google Bigtable

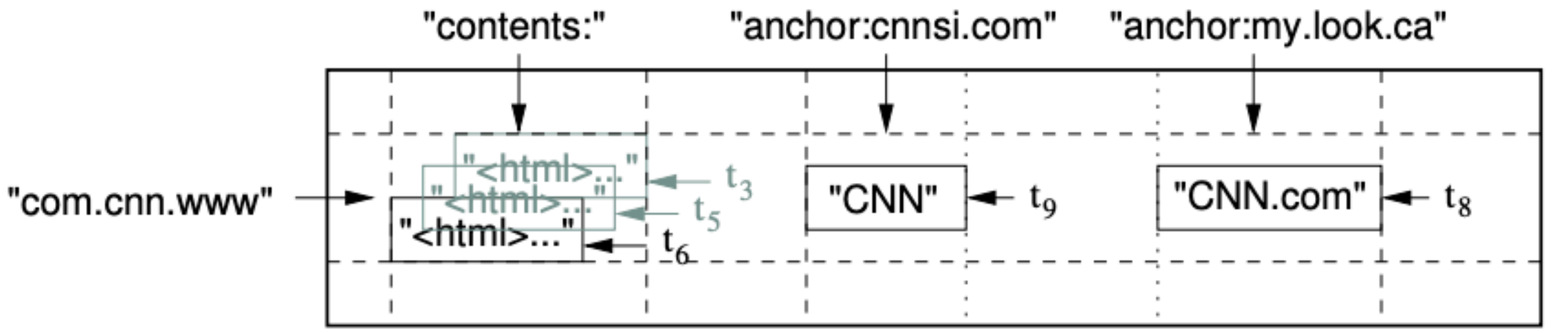
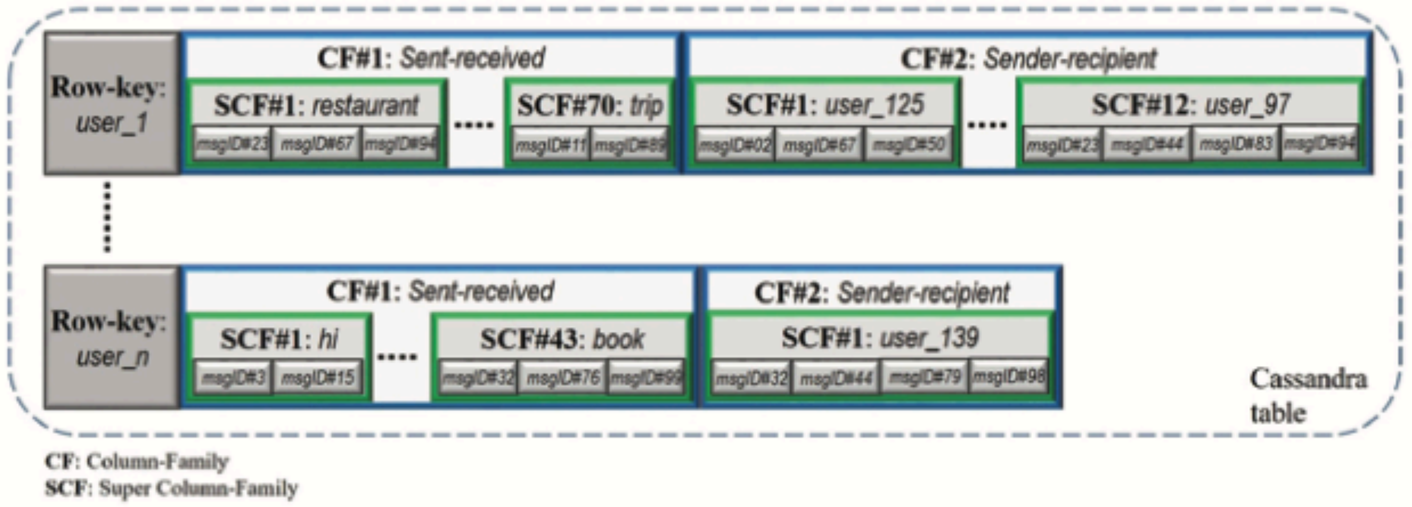


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

Image from "Bigtable: A Distributed Storage System for Structured Data", Chang et al. (2006).

Example Application - Facebook Inbox Search

- Example of how Facebook used a column-family data model for the Inbox Search service.
- The query requirements are to enable search through a user's messages based on:
  - a keyword (term queries);
  - the name of a sender or receiver (interaction queries).



Example and image from "A Survey on NoSQL Stores", Davoudian et al. (2018)

Example Application - Facebook Inbox Search

- The previous query requirements are facilitated by creating specific aggregates.
- For both term and interaction queries, the **user-ID is the row-key**.
- **Two column-families**, Sent-received and Sender-recipient, represent two different aggregates (with regard to the same user) that satisfy the requirements of term and interaction searches, respectively.
  - For the **Sent-received column-family**, the keywords that make up the user's messages become super column-families. For each super column-family, the individual message-IDs become the columns, which, in turn, minimizes redundancy.
  - For the **Sender-recipient column-family**, the user-IDs belonging to all senders/recipients of the user's messages become super column-families. For each super column-family, the individual message-IDs become the columns.

Example Application - Facebook Inbox Search

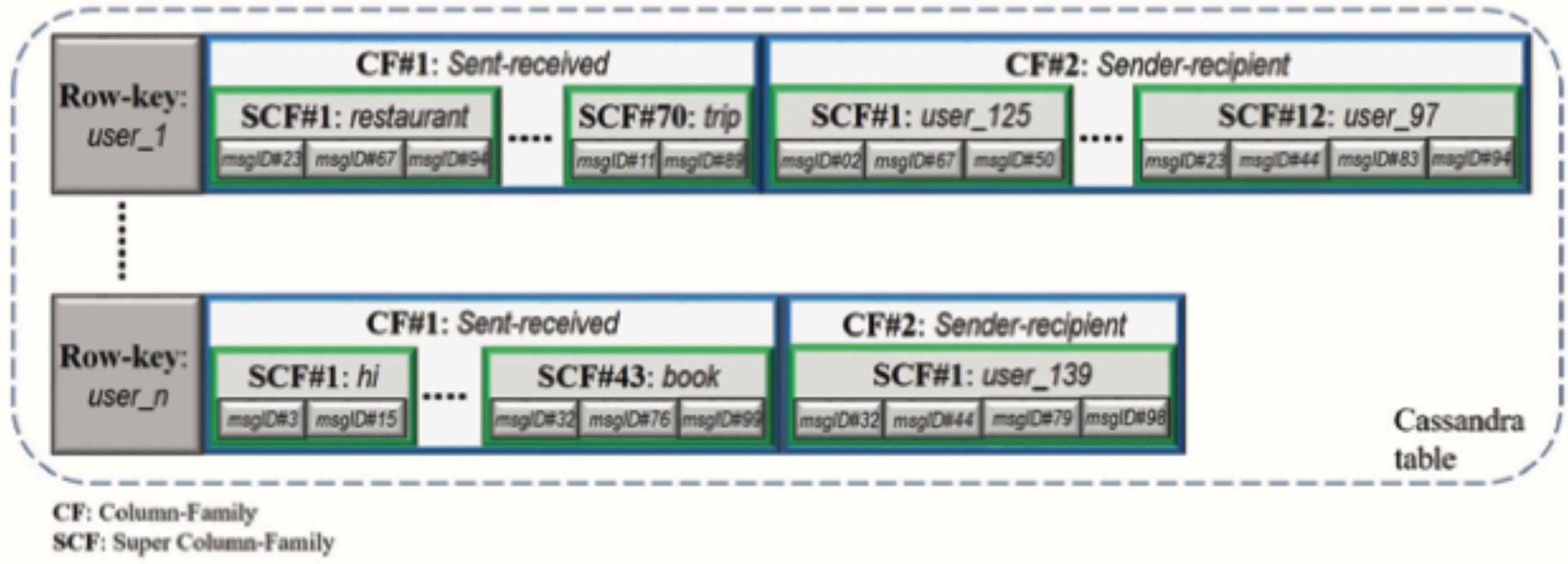


Image from "A Survey on NoSQL Stores", Davoudian et al. (2018)

Suitable Use Cases

- Scenarios with large volumes of data (petabyte scale).
  - Since data can be easily partitioned both horizontally (by rows) and vertically (by column families).
- High scalability and high availability requirements in both reads and writes (but no transactions).
  - Column-family databases support both read and write distribution.
- Disjoint data access patterns.
  - The partition options with various aggregate levels makes data distribution easier.
- Geographical distribution.
  - Can be replicated across multiple data centers, e.g. distributed user base.
- Examples: event logging, counters / analytics, big data processing (with MapReduce).

Column-Family Database Solutions

- Cassandra, cassandra.apache.org (2008) [Apache]
- HBase, hbase.apache.org (2007) [Apache]
- Accumulo, accumulo.apache.org (2008) [Apache]

- Microsoft Azure Table Storage, azure.microsoft.com/en-us/services/storage/tables [Proprietary]
- Google Cloud Bigtable, cloud.google.com/bigtable [Proprietary]

- More: db-engines.com/en/ranking/wide+column+store

Graph Databases

Graph Databases

- Graph databases **store entities and relationships between these entities**.
- **Vertices** represent the entities and; **Edges** represent the relationships.
- Graph databases **are relationship-oriented, not aggregate-oriented**.

- A query to the graph is also known as **traversing** the graph (PT: "travessia").
- Traversing requirements can be changed without changing the existing data or model (i.e. graphs can be traversed in any way).
- Offers **high speed in traversing relationships**, since they are persisted (i.e. not calculated at query time).

- Motivated by the increase in "networked data" and need for graph transversals (e.g. semantic web, bioinformatics).
- Supported by a strong graph theoretical foundation. Also called "Graph Stores".

Graph Structure

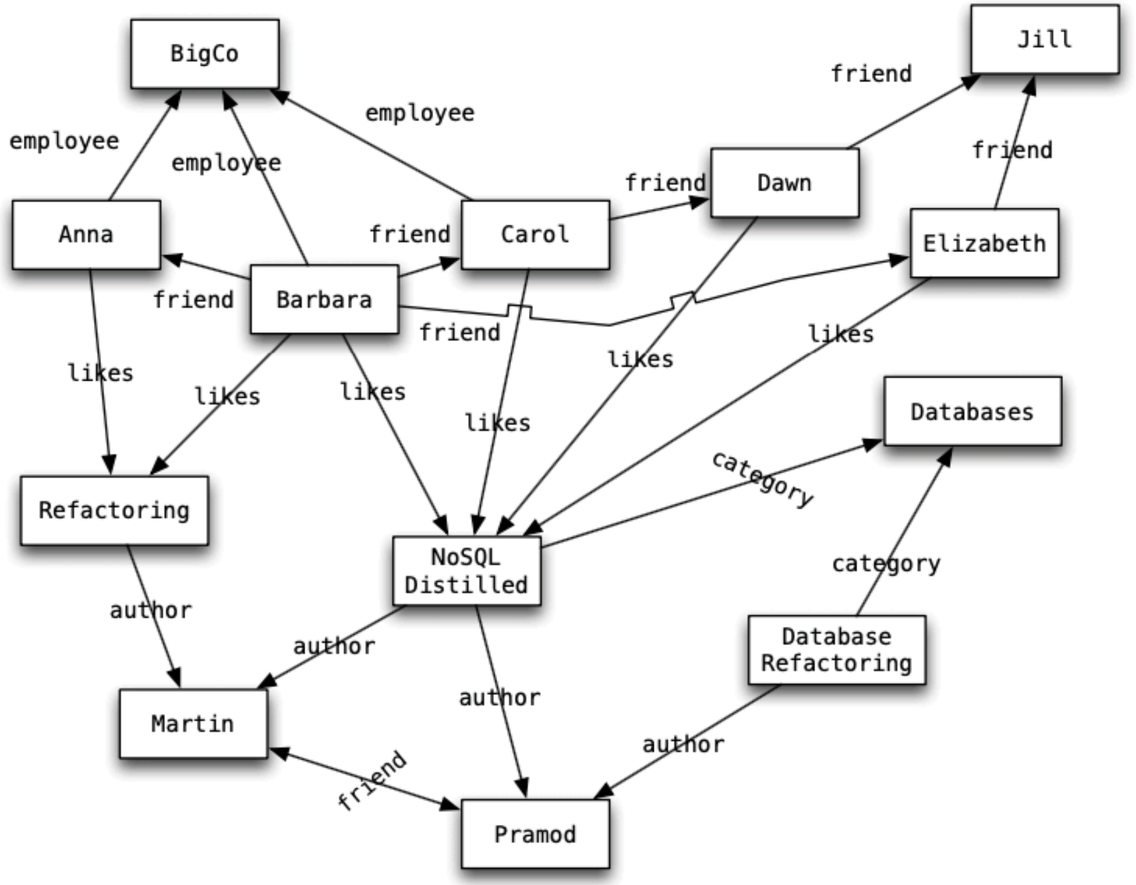


Figure 11.1 *An example graph structure*

Image from "NoSQL Distilled", Sadalage and Fowler (2012).

Graph Database Concepts

- **Nodes represent entities**.
- Nodes know about **incoming and outgoing relationships**.
  - Which can be traversed in both ways.
- **Relationships are first-class citizens** in graph databases.
  - Most of the value in graph databases is derived from the relationships;
  - Significant design work is required to model the relationships in the problem domain;
  - Relationships have a type, a start node, and an end node;
  - Relationships can have properties of their own;
  - Adding new relationships is easy.
  - Changing existing relationships is similar to a data migration process (i.e. change each node and relationship).

- All this information can be used to query the graph using **rich query languages** (e.g. Gremlin, Cypher).

Types of Graph Structures

- Different types of graph structures exists (not mutually exclusive).
- **Undirected/directed graphs**, in undirected graphs all relationships are symmetric; while in directed graphs relationships have directionality.
- **Labeled graphs**, vertices and edges are tagged with scalar values (labels types).
- **Attributed graphs**, key-value pairs are attached to vertices and edges, representing their properties (e.g. social networking sites).
- **Multigraphs**, multiple edges are possible between the same two vertices.
- **Hypergraphs**, where hyperedges can connect any number of vertices.
- **Nested graphs**, each vertex can be a graph.

Graph Example

(d). Facebook's social network uses the graph data model. As an illustration, suppose David along with his friend Sara visit the Eiffel Tower. David uses his cellphone to record this visit by 'checking in' to the Eiffel Tower and tagging Sara to let other friends know that she is also there. Jack writes a comment on this and John likes it.

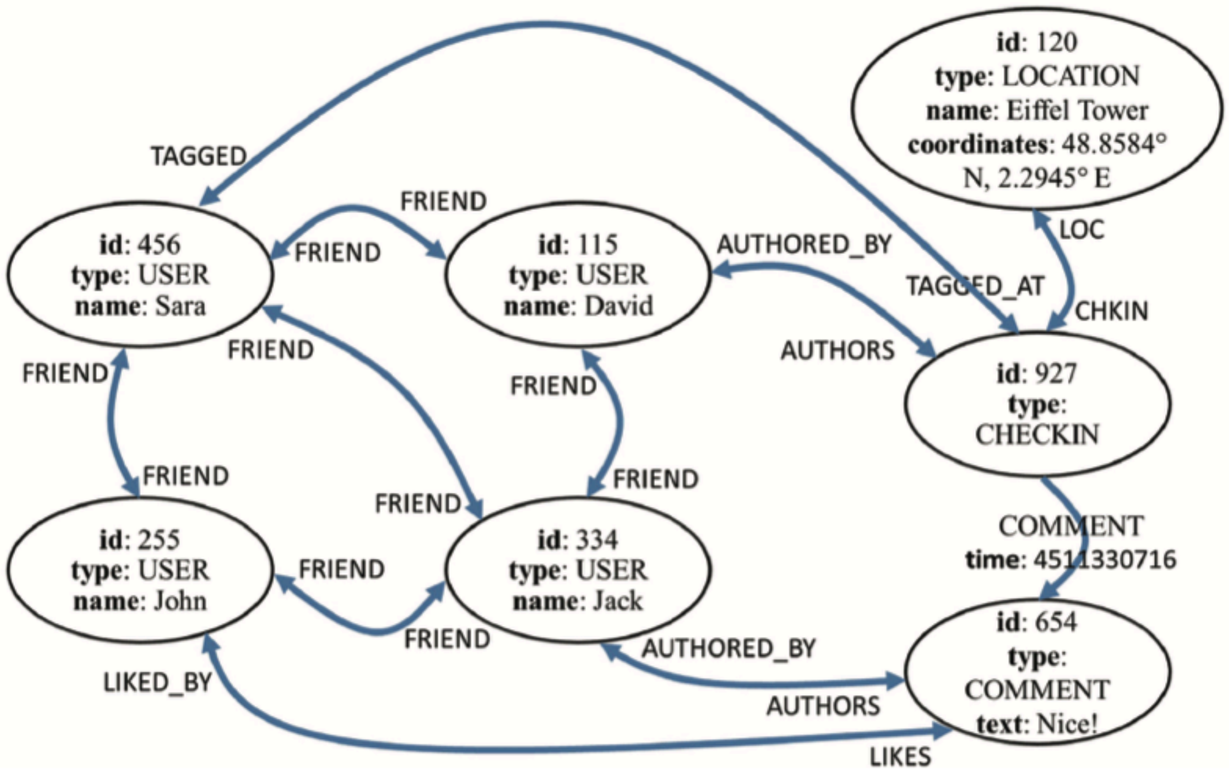


Image from "A Survey on NoSQL Stores", Davodian et al. (2018)

Graph Example (2)

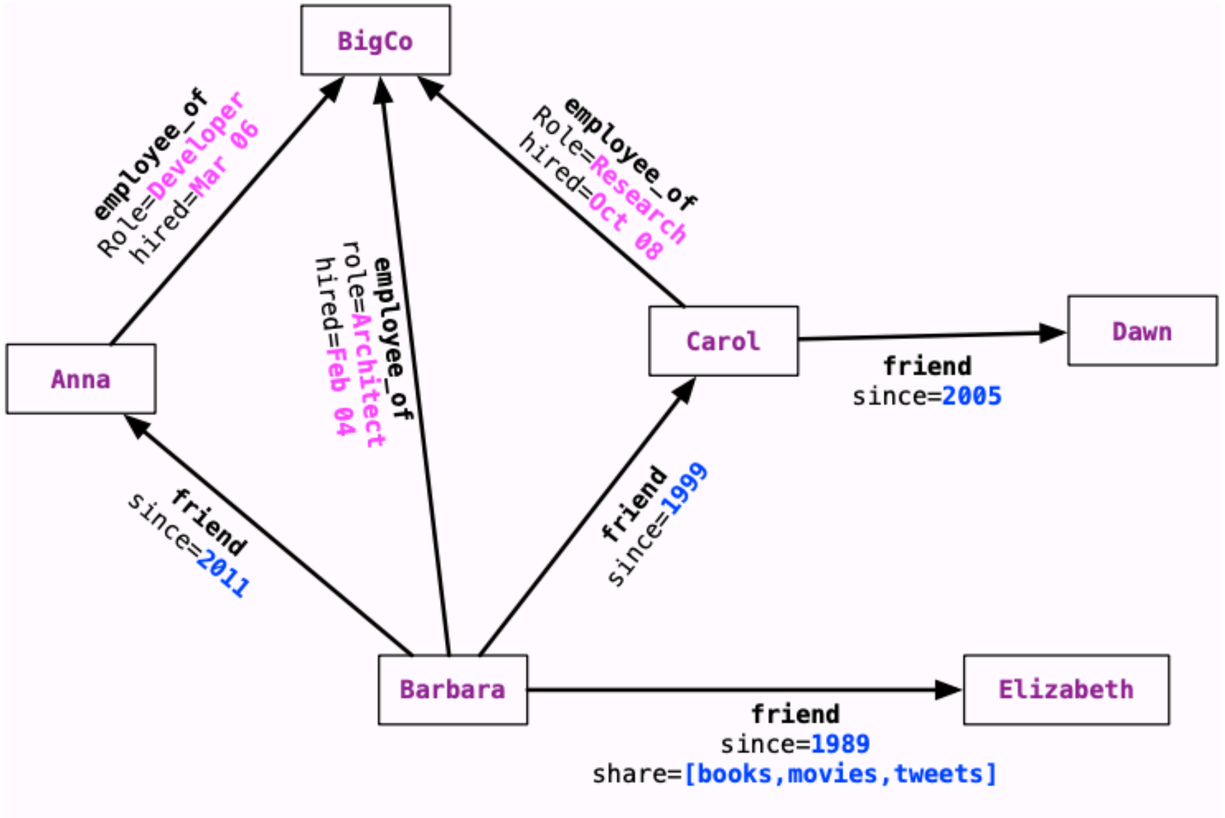


Figure 11.2 *Relationships with properties*

Image from "NoSQL Distilled", Sadalage and Fowler (2012).

Suitable Use Cases

- Complex domains and querying needs.
- Complex data models; multiple co-existing domains; strongly connected data.
- Strengths: fast and simple querying of linked datasets; powerful modeling language.
- Limitations: no standard query language; difficult to scale due to high connectivity.
- Examples: connected data, location-based services, recommendation engines.

Graph Database Solutions

- Neo4j, neo4j.com (2007) [GNU]
- JanusGraph, https://janusgraph.org (2012, as TitanDB) [Apache]
- Giraph, giraph.apache.org (2012) [Apache]
- Several multi-model solutions supporting graph model:
  - ArangoDB [www.arangodb.com], OrientDB [orientdb.org], Virtuoso [virtuoso.openlinksw.com]

- Ontotext GraphDB, www.ontotext.com/products/graphdb [Proprietary]
- Amazon Neptune, aws.amazon.com/neptune (2018) [Proprietary]
- TigerGraph, www.tigergraph.com (2012) [Proprietary]

- More: https://db-engines.com/en/ranking/graph+dbms

NoSQL Decision Guide

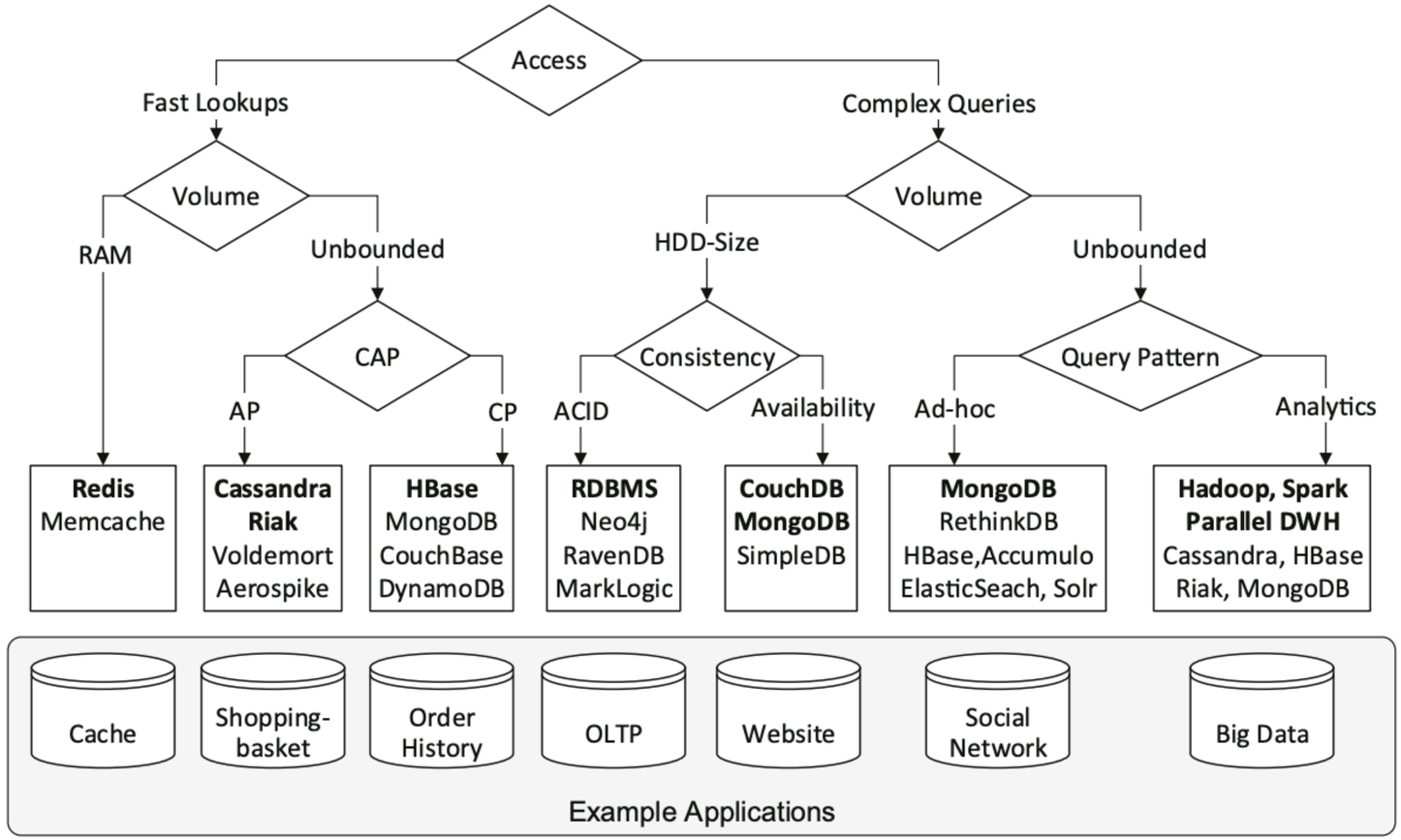


Fig. 6 A decision tree for mapping requirements to (NoSQL) database systems

Image from "NoSQL database systems: a river and decision guide", Gessert et al. (2016)

References

- **NoSQL Distilled**
  Pramod J. Sadalage and Martin Fowler
  Addison-Wesley, 2012

- **Next Generation Databases**
  Guy Harrison
  Apress, 2016

- **A Survey on NoSQL Stores**
  Ali Davoudian, Liu Chen, and Mengchi Liu
  ACM Computing Surveys, 2018