BIRMINGHAM CITY
University

CMP6207 - Assignment 1

IOThings Application Report

May, 2025

Lewis Higgins - Student ID 22133848

CMP6207 - Modern Data Stores

Module Coordinator: Konstantinos Vlachos

Contents

Introduction

I'll come back and write the introduction at a later point.

Types of NoSQL databases

Structured Query Language, or SQL, was developed by IBM following Codd (1970)'s groundbreaking publication in the ACM journal, with the first commercial SQL implementation being published by Oracle in 1979 (Oracle, 2025). SQL powers many relational database systems even today, though the problems associated with its age, most notably in the speed of its operations, are beginning to show in modern systems. Therefore, NoSQL ("Not Only SQL") was developed as an extension of SQL, allowing data to be stored in a non-tabular, non-relational format for efficient storage of semi-structured and unstructured data in a flexible, functional and scalable model for faster operations than standard relational databases in most scenarios (Google Cloud, 2025; AWS, 2025e). It is important to understand that NoSQL is **an extension** to SQL, designed to assist in the storage of unstructured data, which is a growing need across many industries (Corbellini et al., 2017), and not a total replacement, as there are still some benefits to standard relational databases that are not always offered by NoSQL alternatives, which are discussed in Chapter 2.

There are a wide variety of NoSQL database types - all of which vary in complexity, functionality and purpose, meaning that the identification of the most suitable type is paramount for maximum efficiency in terms of speed, storage and scalability.

1.1 Document database

Document databases are intuitive, flexible and horizontally scalable databases that work well in a wide variety of use cases for both transactional and analytical purposes, including IoT data and real-time analytics (MongoDB, 2025b). They store records as "documents", which store an object's data and metadata, in a format such as JSON, BSON, or XML[1]. Details and examples of these file types can be found in Appendix A.

Figure 1.1: An example of a JSON Document.

[1]JavaScript Object Notation, Binary JSON and Extensible Markup Language, respectively.

Figure 1.1 depicts an example JSON document in MongoDB, a popular DBMS for document databases. It contains the data of a singular example school teacher, storing details such as their name, age and subject expertise, as well as a unique internal object ID used by MongoDB to identify that document. The data itself is of varying types including strings, integers and arrays, which makes document databases easily integrable into a development workflow due to the direct storage of object types used in programming languages like Python and JavaScript.

Relationships in document databases can be modelled with great ease, with the ability for documents to contain other documents. For example, in MongoDB, a document for a social media user may contain the Object ID of each of their posts. Groups of documents can then be stored in **collections**, similar to tables in relational databases. These collections can be queried to perform all CRUD operations.

Popular services for document databases include Databricks (Databricks, 2023) Couchbase (Couchbase, 2025), and the previously mentioned MongoDB (MongoDB, 2025b).

1.2 Key-value database

Key-value databases are primarily reputed for their speed and simplicity, functioning by storing each record as a key-value pair. Rather than having to search through massive amounts of irrelevant data for a query, key-value databases can instead search through their stored keys to retrieve results within milliseconds or even microseconds if used in-memory (Redis, 2025a). While this is excellent for simple queries to retrieve specific known records, this same property also causes major limitations in that retrieving data based on values, such as finding all users over a given age, would require the entire database to be searched, making key-value databases best suited for real-time data access and caching where simpler queries are used (MongoDB, 2025d).

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

Figure 1.2: An example key-value pair (Redis, 2025b).

Figure 1.2 depicts an example key-value pair, mapping names to phone numbers. If a query for "Greg" was given, the associated value would be returned. There is a significant similarity between key-value stores and document stores, though key-value stores can only store simple key-value pairs, whereas document stores can have flexible schemas of complex, nested structures. Furthermore, as previously mentioned, querying key-value databases is very limited to only simple queries if speed is a decisive factor.

Notable software options used across industry for key-value databases include Amazon's DynamoDB (AWS, 2025b), Redis (Redis, 2025b), and Memcached (memcached, 2025). MongoDB can also be used as a key-value store, though it is not primarily intended to do so.

1.3 Wide-column database

Wide-column databases, also known as wide-column stores, store data across columns rather than rows, which allows for data schemas built for these databases to be very flexible. Records stored in wide-column stores do not require the same columns for every row, which can greatly help to reduce duplication and redundancy, and storage requirements by extension. Instead, Figure 1.3 depicts how wide-column databases can store data in "column groups", which would be defined when the database is created.

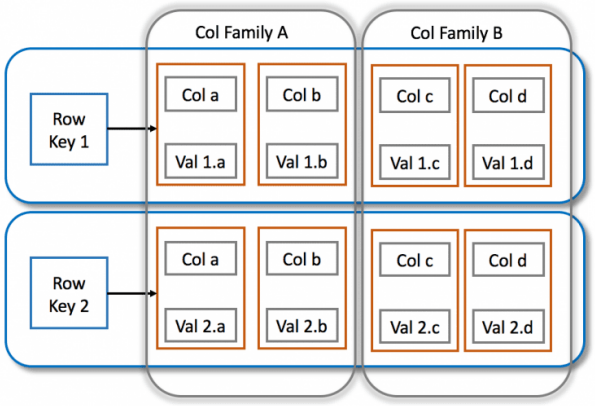

Figure 1.3: An example of wide-column store groups (Pore, 2018).

Column groups are greatly beneficial for **distributed databases**, which leverage sharding for horizontal scalability, a concept explored further in Chapter 2. For example, one node could store the columns of group A, whereas another stores group B. If done correctly, this can greatly help the speed of queries on the database, but it could also cause major slowdowns if the column groups are incorrectly defined, such as with data that would typically be queried together being split over multiple column groups. Therefore, column groups should consist of data that is frequently queried together, such as a group for personal information, then another for financial information (Cattell, 2011).

Because of this, wide-column stores are typically used for larger databases reaching petabytes in size, using software implementations such as Google's Bigtable (Chang et al., 2008) and Apache Cassandra (Apache, 2025).

1.4 Graph database

Graph databases differ heavily from the previously mentioned types, and are instead based on mathematical graph theory (AWS, 2025d). Rather than storing data as rows and columns (or keys and values), these databases store data as nodes and edges. Nodes are connected through their relationships to other nodes, which form the edges of the graph structure as depicted in Figure 1.4.

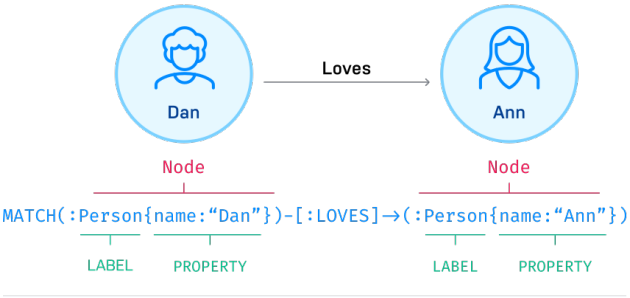

Figure 1.4: Two example nodes connected by a relationship. (Neo4j, 2025c)

Graph databases are best suited for data that is highly interconnected, and can achieve extremely fast query speeds in these scenarios due to graph traversal, where related nodes can be immediately read through the relationships between them rather than the extensive joins that would be required in a relational database equivalent (Corbellini et al., 2017). Typical use-cases of these databases include fraud detection, user recommendation systems and identity management (Neo4j, 2025b; AWS, 2025c; Memgraph, 2025a).

Software typically used for graph databases includes Neo4J (Neo4j, 2025a), Amazon Neptune (AWS, 2025c), and Memgraph (Memgraph, 2025b).

An additional key feature of graph databases is that they do not store collections of data together in groups or collections unlike other databases, with this functionality instead being dictated by labels, relationships or properties.

Label grouping

Nodes can be labelled with a tag (such as "User"), and queries can instead retrieve all nodes sharing a given label.

Relationship grouping

A node is created that represents the group, with all members of the group having a relationship with this parent node such as "Member of".

Property grouping

Nodes individually store a property such as a group ID which identifies their particular group. Queries can then fetch all nodes with the given property to identify members of the group.

1.5 The CAP theorem

The CAP theorem (Brewer, 2000) is a foundational governing principle of all distributed systems. He stated that any system that shares data across devices can only ever have two of the following three properties:

Property	Description
Consistency	All clients see the same data at the same time, irrespective of the node they're connected to.
Availability	A response is always given to a client request, even if a node (or multiple) is down.
Partition Tolerance	The system must continue to operate even if nodes temporarily lose connection to each other.

Table 1.1: The properties of the CAP theorem. (Brewer, 2000; IBM, 2022)

The CAP theorem is vitally important in the selection of which NoSQL database type to use, as different DBMS adhere to different properties of the theorem, with none being able to offer all three as stated by Brewer.

For example, the document database MongoDB adheres to Consistency and Partition Tolerance (CP), but is unable to offer total availability because of this. This is because ensuring CP means that data between nodes cannot be inconsistent, so if nodes fail to synchronise data, the availability would have to suffer to stop potentially outdated data being returned.

Another example is the wide-column store Apache Cassandra, which offers Availability and Partition Tolerance (AP), but therefore cannot offer total consistency. Cassandra instead offers "eventual consistency" through the BASE transactional model, which is described in Section 2.2.2, where clients can potentially see outdated data during simultaneous queries, though the DBMS aims to resolve inconsistencies as quickly as possible.

IBM (2022) write that consistency and availability (CA) cannot exist simultaneously in NoSQL distributed systems, as this would come at the expense of Partition Tolerance which is not possible in a distributed system. Instead, a relational database would have to be used to offer both of these.

NoSQL vs. Relational databases

Industrial needs for fast and efficient data storage continue to grow exponentially year by year, and some of these needs fail to be met by typical relational database solutions. Corbellini et al. (2017) state that the types of data that require storage have changed since the first relational database systems of the 1970s, and as such, these more mature systems struggle to meet the requirements of today's businesses with web-oriented systems storing magnitudes of unstructured data. The same can also be said of Internet of Things (IoT) systems, with Gubbi et al. (2013) stating that 'For the realization of a complete IoT vision, an efficient, secure, scalable and market oriented computing and storage resourcing is essential.'

These efficient, secure and scalable resources exist in the form of NoSQL databases. Chapter 1 provided an overview of NoSQL itself, as well as the various types of NoSQL databases, with this chapter instead offering direct comparisons between modern NoSQL solutions to more traditional relational database management systems in terms of their scalability and transactional models.

2.1 Scalability

Both Gani et al. (2016) and Katal et al. (2013) state that the amount of data stored by companies is increasing at an extreme rate, forecasting that companies will likely be storing *zettabytes* of data in the near future. Because of this, Gubbi et al. (2013)'s previously mentioned efficient and scalable storage solutions are needed to meet these ever-increasing demands.

Databases can be scaled both vertically and horizontally, with relational databases typically favouring vertical scaling and NoSQL databases favouring horizontal scaling, though both are possible in either database type.

2.1.1 Vertical scaling

Scaling vertically means to upgrade the host device(s) of the database, which refers to the physical swapping or installation of parts such as CPUs, RAM and storage drives to accelerate data processing (Cattell, 2011). A significant benefit of a vertically scaled database is that data will always be consistent as it is stored on a single device rather than distributed across many over a network connection where data could be corrupted, lost or intercepted. This allows for total ACID compliance, which is described further in Section 2.2.1.

Despite this key benefit, vertical scaling has some downsides. Server components cannot be swapped while the system is online, meaning that scaling vertically **requires** partial or total system downtime, therefore decreasing availability. Furthermore, vertical scaling introduces a single point of failure to a system, where the powerful server hosting the database could experience any issue such as a power cut that would cause a total system outage. Continuous vertical scaling can also become very costly and bear diminishing returns, with the significant cost of top-grade server parts potentially outweighing the performance benefit of their installation (MongoDB, 2025a). Vertical scaling also has a fixed limit - a server can only be as good as the best available components.

2.1.2 Horizontal scaling

Scaling horizontally means to add more devices (nodes) to a network, making the database a **distributed database**. Rather than having an entire DBMS stored on a single device and single point of failure, horizontal scaling divides the processing and storage loads of the database over many nodes, often through a process called **sharding**, which involves splitting records into independent partitions (shards) based on criteria like ID numbers (Corbellini et al., 2017), and replicating these shards over multiple nodes. Horizontal scaling is seen almost exclusively with NoSQL databases, as the normalized data models and ACID compliance of relational databases mean that scaling horizontally is not feasible (Hecht and Jablonski, 2011).

Horizontal scaling introduces considerable benefits for availability, as one node can fail, but the database can continue working without it, which allows for maintenance or the addition of extra nodes with minimal to no downtime at all (MongoDB, 2025a). Furthermore, the aforementioned processing load balancing means that each node doesn't require considerable investment in top-grade parts, as nodes can balance processing among themselves. Also, horizontal scaling has no fixed limit unlike vertical scaling, as a theoretically infinite amount of nodes can be added dependent on existing network infrastructure, though this would add considerable complication to the overall system before long. It is for these reasons that thousands of large enterprises, including industry titans like Google and Amazon, leverage horizontal scaling of their database systems to maximise their processing speeds and maximise availability (Chang et al., 2008; Amazon, 2021).

Despite the wide adoption and considerable benefits, horizontal scaling is not without its own drawbacks. Initialising a database with many nodes will introduce a larger initial setup cost, though further upgrades by adding additional nodes is typically more cost-effective than scaling vertically as depicted in Figure 2.1.

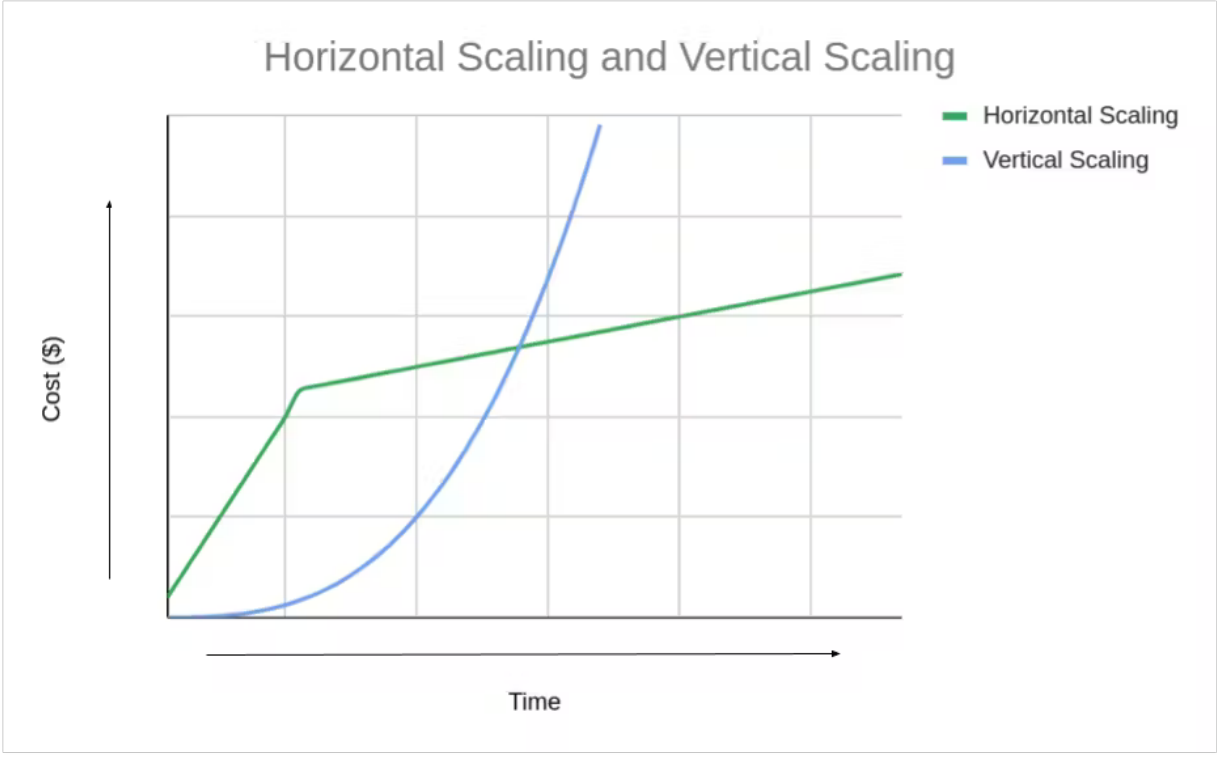

Figure 2.1: An example cost/time graph with each scaling method. (MongoDB, 2025a)

Additionally, the sharding process relies on **replication**, meaning that data will be duplicated, reducing storage efficiency despite the increase in availability. The final considerable drawback is the fact that all nodes are connected via a network, which could introduce issues with data transmission, such as the potential for data interception (man-in-the-middle) or corruption (packet loss).

2.2 Transactional models

Relational and NoSQL databases are governed by dichotomous transactional models that fundamentally change their appropriate use cases. Relational databases are designed to adhere to the ACID model that prioritises absolute integrity and consistency at the expense of both availability and scalability, whereas most NoSQL databases adhere to the opposing BASE model, prioritising availability and scalability over consistency. However, ACID is not exclusive to relational databases, and recent updates to certain NoSQL databases like MongoDB have allowed partial ACID compliance in combination with the scalability benefits of NoSQL.

2.2.1 ACID

ACID is an acronym to describe the following key attributes:

Property	Description
Atomicity	All transactions must be fully completed or not completed at all. If a transaction fails, the database must be reverted to its prior state with no changes made to the data.
Consistency	Data must meet predefined integrity constraints, and remain consistent for all users even in the event of simultaneous transactions.
Isolation	Transactions are executed sequentially, and do not interfere with each other even if they are simultaneous.
Durability	Even in the event of system failure, the database must maintain all committed records. This means that even if an error occurs, the results of any and all previous transactions must not be lost.

Table 2.1: The key properties of ACID compliance. (AWS, 2025a; Neo4j, 2023)

ACID is a very strict model that actively enforces a safe environment for data operations. Though, in maintaining such high security, a performance overhead exists with every transaction. This can massively impact the scalability of relational databases with complex join operations, which will grow slower the more users they have.

ACID compliance is not exclusive to relational databases, however; in recent years, some NoSQL databases have incorporated ACID properties to gain the best of both worlds - high speed and scalability from being non-relational, and high security and integrity from ACID compliance. However, NoSQL databases cannot be fully ACID compliant due to their architecture, as horizontal scaling while maintaining

2.2.2 BASE

BASE-compliant databases operate very differently to their ACID counterparts. Rather than the strict integrity and consistency procedures expected by ACID, the properties of BASE databases are much more relaxed:

Property	Description
Basically Available	Systems will continue to operate even if failures occur due to redundancy and replication.
Soft State	Replicas do not have to be entirely consistent with each other. This means the system can be in two different states simultaneously, which is known as a transient state.
Eventually Consistent	While updates are not occurring, transient states will converge.

Table 2.2: BASE properties. (Corbellini et al., 2017; Cattell, 2011; Neo4j, 2023; AWS, 2025a)

By forfeiting the strict regulations and associated performance overheads of ACID compliance, BASE-compliant databases can theoretically operate at much higher speeds by sacrificing their immediate consistency in favour of eventual consistency and workload division across many nodes, as was previously noted in Sections 1.5 and 2.1.2.

2.3 Overall summary

In conclusion of all factors discussed across Chapters 1 and 2, the best option for IoThings will be to leverage the heightened capabilities of a NoSQL database, specifically a **document store**, to store and query their sensor activation data.

2.3.1 Why not a relational database?

It is unlikely that IoThings will see much benefit from a relational database solution as they require **high scalability**, which is best offered by distributed NoSQL systems. As a home automation company, IoThings will likely be storing sensor readings at regular intervals, better known as **time-series data**. If this interval were 30 seconds, and they had 10,000 users, they would be storing 600,000 values a minute or 864,000,000 a day. It is not unheard of for this to be done in relational systems, though the previously mentioned lack of horizontal scalability of these systems mean that user demand will quickly outpace any vertical scaling that could be performed, possibly through the sheer storage requirement alone.

2.3.2 Why use a document store?

Based on the specific needs of the company, the best type of NoSQL database to use will be a **document store**. Within the document store, JSON/BSON objects each containing the various sensor outputs can be stored and queried with great ease and high speed. The programmatic storage of these objects using common types for languages like JavaScript and Python allows for direct integration with the IoT device's code, where sensor readings could be directly sent to the DB as part of the device's standard operations, as document stores excel in storing data of varying types, with time-series data from a thermostat containing the current date and time (ISODate type in JavaScript) and temperature (Integer or Double if decimals included) for example. One of the software options for this approach, MongoDB, already includes guides for this process in their documentation (MongoDB, 2025c).

Database design and implementation

3.1 Target problem

3.1.1 Clientele

IoThings are a UK-based small-to-medium sized enterprise specialising in Internet of Things home automation systems. Specifically, they install sensors and use MQ Telemetry Transport (MQTT) to collect their activation data and store it in a MongoDB database for analytical purposes. These sensors can be used for a variety of reasons, such as:

- Temperature control
- Lighting control
- Entertainment management
- Home security

3.1.2 Existing systems

IoThings already have an extensive data storage solution for their Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), finances, order processing / sales, and logistics. However, they do not presently have an optimal solution for their MQTT sensor activation data.

3.1.3 Requested solution

Therefore, IoThings wish to extend their current data storage implementation with an additional NoSQL database to contain their sensor activation data for the eventual purpose of giving feedback to their users about device usage. In addition to the creation of this database, the client also requests the development of an API and frontend for accessing and processing the stored data.

Due to data security and privacy concerns, it is not possible for IoThings to provide data of their own. Therefore, the solution developed in this report will use synthetic data which aims to replicate the data which would theoretically be stored.

API Implementation and Documentation

Conclusion

Overall, something was done...

Bibliography

Amazon (8th Mar. 2021). *Modernizing the Amazon Database Infrastructure*. URL: https://d1.awsstatic.com/whitepapers/modernizing-amazon-database-infrastructure.pdf (visited on 24/02/2025).

Apache (2025). *Apache Cassandra | Apache Cassandra Documentation*. Apache Cassandra. URL: https://cassandra.apache.org/ (visited on 20/02/2025).

AWS (2025a). *ACID vs BASE Databases - Difference Between Databases - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/ (visited on 21/02/2025).

AWS (2025b). *Fast NoSQL Key-Value Database – Amazon DynamoDB – AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/dynamodb/ (visited on 17/02/2025).

AWS (2025c). *Managed Graph Database - Amazon Neptune - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/neptune/ (visited on 21/02/2025).

AWS (2025d). *What Is a Graph Database? - Graph DB Explained - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/nosql/graph/ (visited on 20/02/2025).

AWS (2025e). *What Is a NoSQL Database? - Nonrelational Databases Explained - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/nosql/ (visited on 04/02/2025).

Brewer, Eric (16th July 2000). 'Towards Robust Distributed Systems'. In: PODC, p. 7. DOI: 10.1145/343477.343502.

Cattell, Rick (6th May 2011). 'Scalable SQL and NoSQL Data Stores'. In: *SIGMOD Rec.* 39 (4), pp. 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919.

Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E. Gruber (1st June 2008). 'Bigtable: A Distributed Storage System for Structured Data'. In: *ACM Trans. Comput. Syst.* 26 (2), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816.

Codd, E. F. (1st June 1970). 'A Relational Model of Data for Large Shared Data Banks'. In: *Commun. ACM* 13 (6), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685.

Corbellini, Alejandro, Cristian Mateos, Alejandro Zunino, Daniela Godoy and Silvia Schiaffino (1st Jan. 2017). 'Persisting Big-Data: The NoSQL Landscape'. In: *Information Systems* 63, pp. 1–23. ISSN: 0306-4379. DOI: 10.1016/j.is.2016.07.009.

Couchbase (2025). *Couchbase: Best Free NoSQL Cloud Database Platform*. Couchbase. URL: https://www.couchbase.com/ (visited on 17/02/2025).

Databricks (13th Oct. 2023). *The Data and AI Company*. Databricks. URL: https://www.databricks.com/ (visited on 17/02/2025).

Gani, Abdullah, Aisha Siddiqa, Shahaboddin Shamshirband and Fariza Hanum (1st Feb. 2016). 'A Survey on Indexing Techniques for Big Data: Taxonomy and Performance Evaluation'. In: *Knowledge and Information Systems* 46 (2), pp. 241–284. ISSN: 0219-3116. DOI: 10.1007/s10115-015-0830-y.

Google Cloud (2025). *What Is NoSQL? Databases Explained*. Google Cloud. URL: https://cloud.google.com/discover/what-is-nosql (visited on 04/02/2025).

Gubbi, J., R. Buyya, S. Marusic and M. Palaniswami (2013). 'Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions'. In: *Future Generation Computer Systems* 29 (7), pp. 1645–1660. DOI: 10.1016/j.future.2013.01.010.

Hecht, Robin and Stefan Jablonski (Dec. 2011). 'NoSQL Evaluation: A Use Case Oriented Survey'. In: *2011 International Conference on Cloud and Service Computing*. 2011 International Conference on Cloud and Service Computing, pp. 336–341. DOI: 10.1109/CSC.2011.6138544.

IBM (20th Dec. 2022). *What Is the CAP Theorem? | IBM*. URL: https://www.ibm.com/think/topics/cap-theorem (visited on 24/02/2025).

Katal, Avita, Mohammad Wazid and R. H. Goudar (Aug. 2013). 'Big Data: Issues, Challenges, Tools and Good Practices'. In: *2013 Sixth International Conference on Contemporary Computing (IC3)*. 2013 Sixth International Conference on Contemporary Computing (IC3), pp. 404–409. DOI: 10.1109/IC3.2013.6612229.

memcached (2025). *Memcached - a Distributed Memory Object Caching System*. URL: https://www.memcached.org/ (visited on 18/02/2025).

Memgraph (2025a). *Graph Database vs Relational Database*. URL: https://memgraph.com/blog/graph-database-vs-relational-database (visited on 21/02/2025).

Memgraph (2025b). *Memgraph Database*. URL: https://memgraph.com/memgraphdb (visited on 21/02/2025).

MongoDB (2025a). *A Guide To Horizontal Vs Vertical Scaling*. MongoDB. URL: https://www.mongodb.com/resources/basics/horizontal-vs-vertical-scaling (visited on 24/02/2025).

MongoDB (2025b). *Document Database - NoSQL*. MongoDB. URL: https://www.mongodb.com/resources/basics/databases/document-databases (visited on 16/02/2025).

MongoDB (2025c). *Model IoT Data - MongoDB Manual v8.0*. URL: https://www.mongodb.com/docs/manual/tutorial/model-iot-data/ (visited on 24/02/2025).

MongoDB (2025d). *What Is A Key-Value Database?* MongoDB. URL: https://www.mongodb.com/resources/basics/databases/key-value-database (visited on 17/02/2025).

Neo4j (11th Aug. 2023). *Data Consistency Models: ACID vs. BASE Explained*. Data Consistency Models: ACID vs. BASE Explained. URL: https://neo4j.com/blog/graph-database/acid-vs-base-consistency-models-explained/ (visited on 21/02/2025).

Neo4j (22nd Feb. 2025a). *Neo4j Graph Database*. Graph Database & Analytics. URL: https://neo4j.com/product/neo4j-graph-database/ (visited on 21/02/2025).

Neo4j (2025b). *Graph Database Use Cases*. Graph Database & Analytics. URL: https://neo4j.com/use-cases/ (visited on 21/02/2025).

Neo4j (2025c). *What Is a Graph Database - Getting Started*. Neo4j Graph Data Platform. URL: https://neo4j.com/docs/getting-started/graph-database/ (visited on 20/02/2025).

Oracle (2025). *History of SQL*. URL: https://docs.oracle.com/cd/B13789_01/server.101/b10759/intro001.htm (visited on 04/02/2025).

Pore, Akshay (16th Feb. 2018). *NoSQL Data Architecture & Data Governance: Everything You Need to Know*. Dataversity. URL: https://www.dataversity.net/nosql-data-architecture-data-governance-everything-need-know/ (visited on 20/02/2025).

Redis (2025a). *Redis FAQ*. Docs. URL: https://redis.io/docs/latest/develop/get-started/faq/ (visited on 17/02/2025).

Redis (2025b). *What Is a Key-Value Database?* Redis. URL: https://redis.io/nosql/key-value-databases/ (visited on 17/02/2025).