NebulaGraph Database Manual

v3.5.0

Min Wu, Yao Zhou, Cooper Liang, Abby Huang

2023 Vesoft Inc.

Table of contents

1. Welcome to NebulaGraph 3.5.0 Documentation

Note

This manual is revised on 2024-2-19, with GitHub commit 45177ffcc3.

Compatibility

In the version of NebulaGraph 3.2, the vertex without tags is allowed. But since NebulaGraph 3.3.0, the vertex without tags is not supported by default.

NebulaGraph is a distributed, scalable, and lightning-fast graph database. It is the optimal solution in the world capable of hosting graphs with dozens of billions of vertices (nodes) and trillions of edges (relationships) with millisecond latency.

1.1 Getting started

- Learning path & Get NebulaGraph Certifications
- What is Nebula Graph
- Quick start
- Preparations before deployment
- nGQL cheatsheet
- FAQ
- Ecosystem Tools

1.2 Release notes

- NebulaGraph Community Edition 3.5.0
- NebulaGraph Studio
- NebulaGraph Explorer
- NebulaGraph Dashboard Community Edition
- NebulaGraph Dashboard Enterprise Edition

1.3 Other Sources

- To cite NebulaGraph
- NebulaGraph Homepage
- Forum
- Blogs
- Videos
- Chinese Docs

1.4 Symbols used in this manual

Note

Additional information or operation-related notes.

Caution

Cautions that need strict observation. If not, systematic breakdown, data loss, and security issues may happen.

Danger

Operations that may cause danger. If not observed, systematic breakdown, data loss, and security issues will happen.

Performance

Operations that merit attention as for performance enhancement.

Faq

Frequently asked questions.

Compatibility

The compatibility notes between nGQL and openCypher, or between the current version of nGQL and its prior ones.

Enterpriseonly

Differences between the NebulaGraph Community and Enterprise editions.

1.5 Modify errors

This NebulaGraph manual is written in the Markdown language. Users can click the pencil sign on the upper right side of each document title and modify errors.

Last update: January 6, 2023

2. Introduction

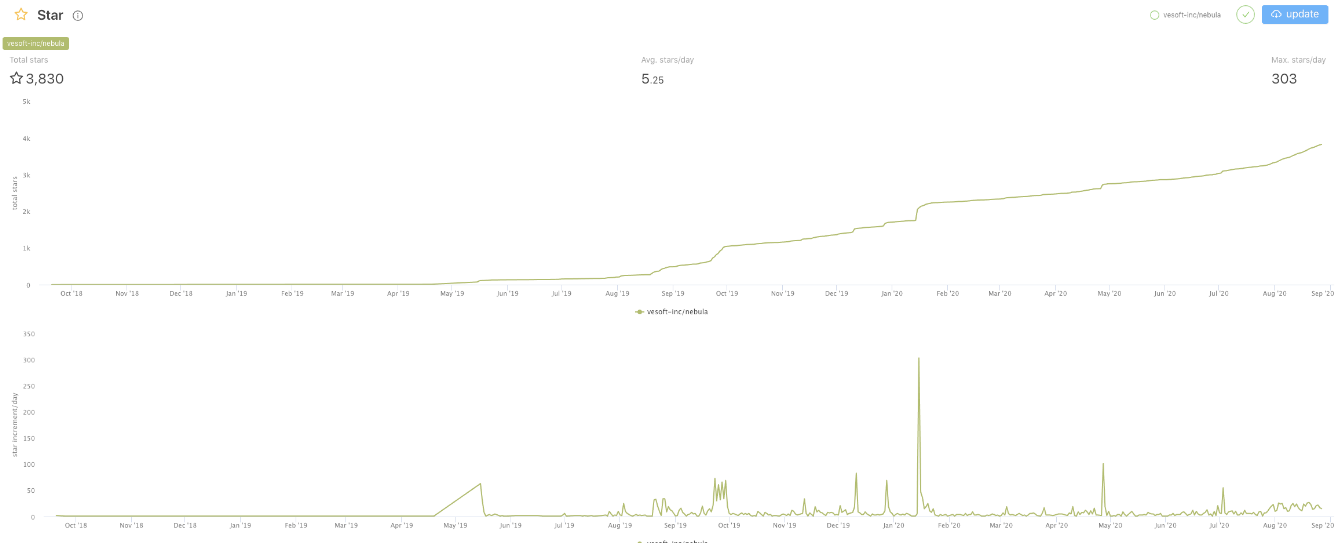
2.1 An introduction to graphs

People from tech giants (such as Amazon and Facebook) to small research teams are devoting significant resources to exploring the potential of graph databases to solve data relationships problems. What exactly is a graph database? What can it do? Where does it fit in the database landscape? To answer these questions, we first need to understand graphs.

Graphs are one of the main areas of research in computer science. Graphs can efficiently solve many of the problems that exist today. This topic will start with graphs and explain the advantages of graph databases and their great potential in modern application development, and then describe the differences between distributed graph databases and several other types of databases.

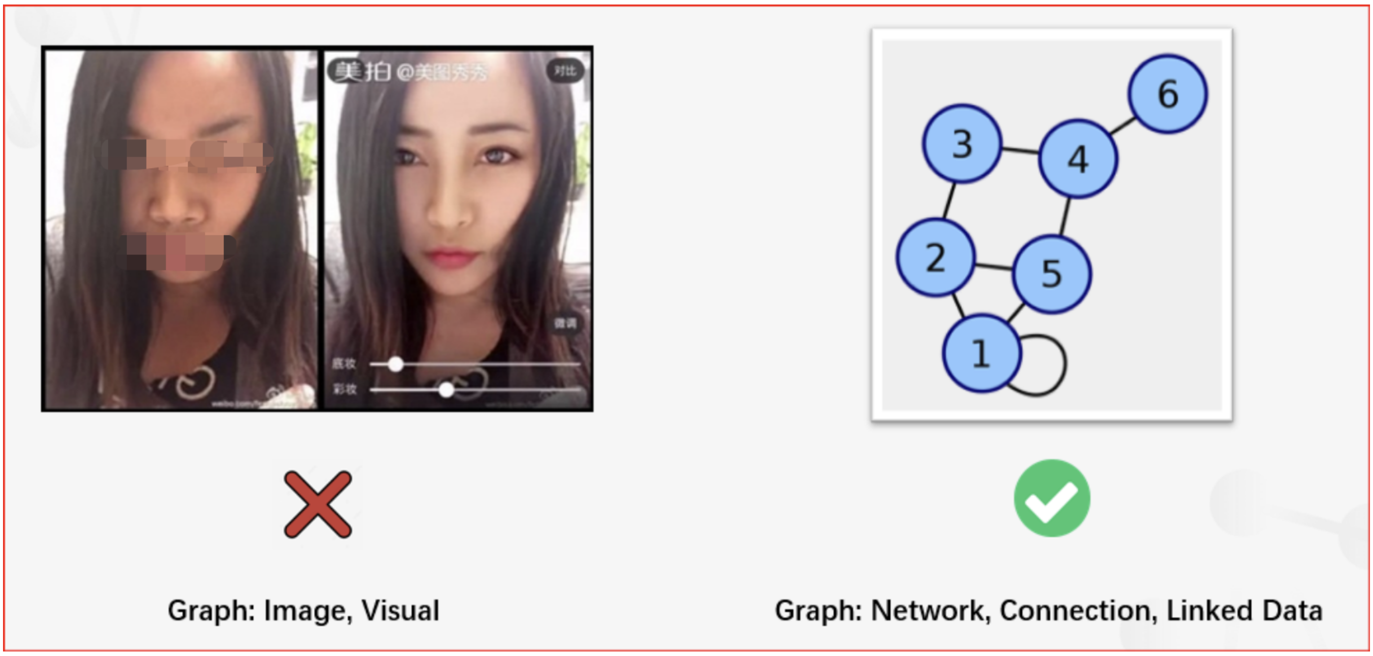
2.1.1 What are graphs?

Graphs are everywhere. When hearing the word graph, many people think of bar charts or line charts, because sometimes we call them graphs, which show the connections between two or more data systems. The simplest example is the following picture, which shows the number of NebulaGraph GitHub repository stars over time.



This type of diagram is often called a line chart. As you can see, the number of starts rises over time. A line chart can show data changes over time (depending on the scale settings). Here we have given only examples of line charts. There are various graphs, such as pie charts, bar charts, etc.

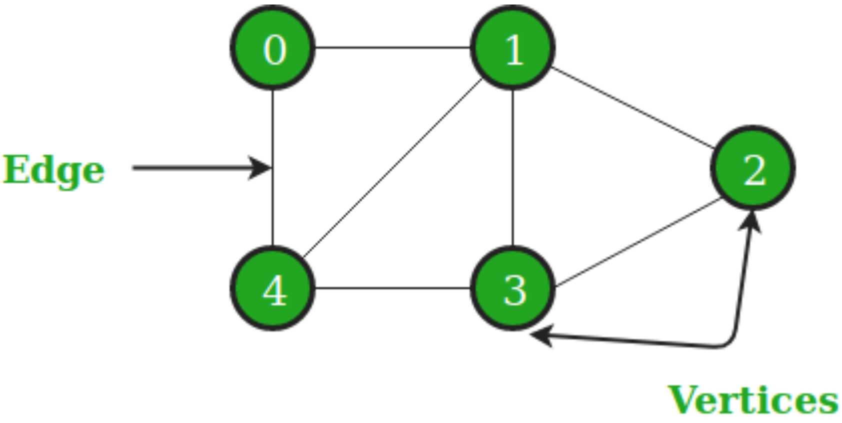
Another kind of diagram is often used in daily conversation, such as image recognition, retouched photos. This type of diagram is called a picture/photo/image.



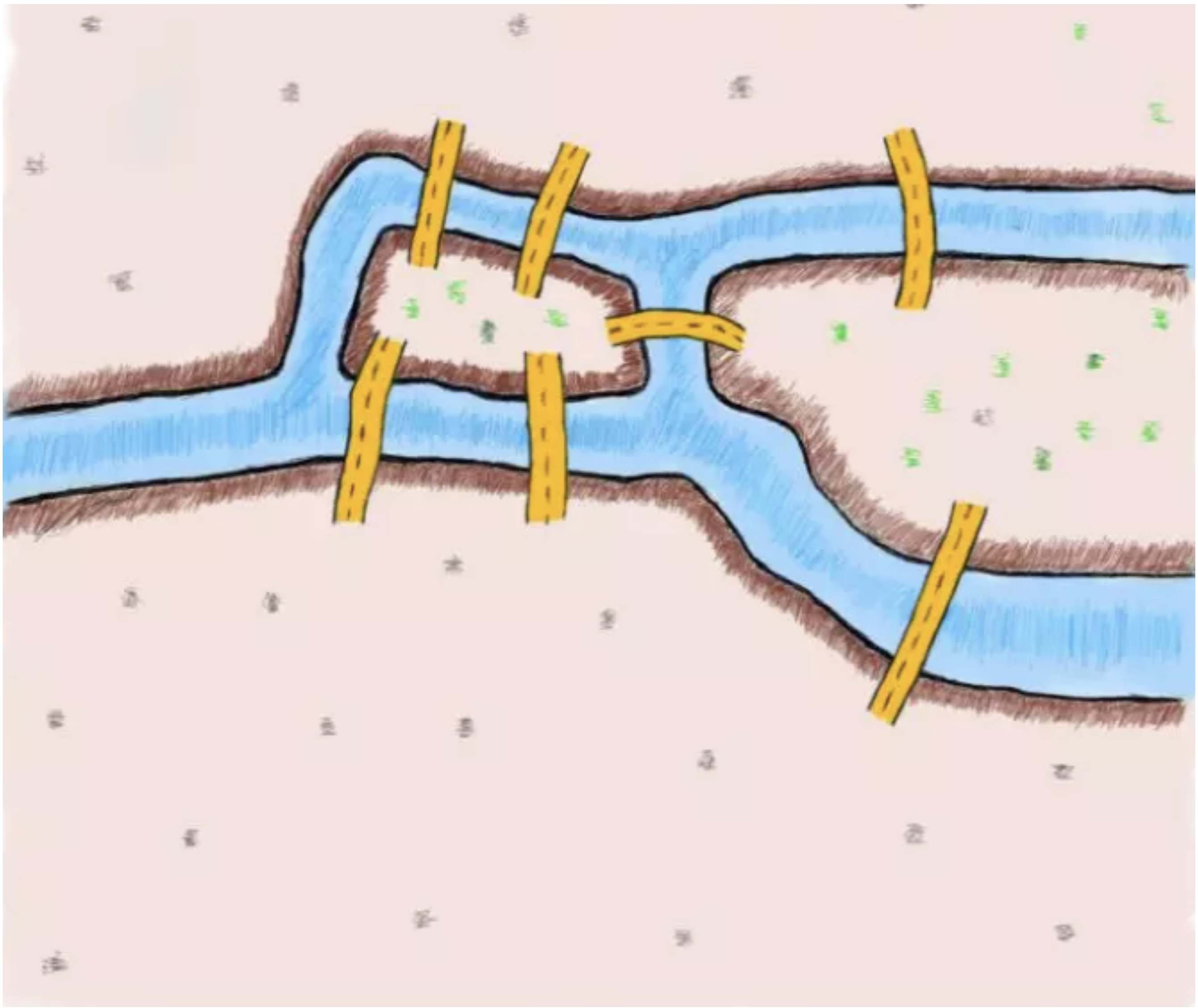
The diagram we discuss in this topic is a different concept, the graph in graph theory.

In graph theory, a branch of mathematics, graphs are used to represent the relationships between entities. A graph consists of several small dots (called vertices or nodes) and lines or curves (called edges) that connect these dots. The term graph was proposed by Sylvester in 1878.

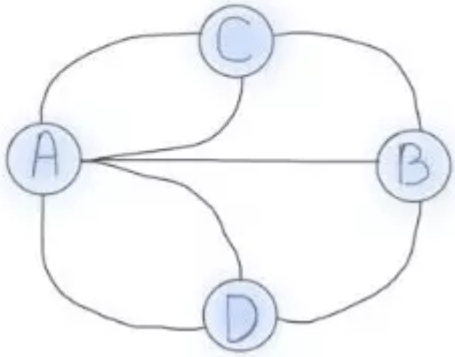
The following picture is what this topic calls a graph.



Simply put, graph theory is the study of graphs. Graph theory began in the early 18th century with the problem of the Seven Bridges of Königsberg. Königsberg was then a Prussian city (now part of Russia, renamed Kaliningrad). The river Preger crossed Königsberg and not only divided Königsberg into two parts, but also formed two small islands in the middle of the river. This divided the city into four areas, each connected by seven bridges. There was a game associated with Königsberg at the time, namely how to cross each bridge only once and navigate the entire four areas of the city. A simplified view of the seven bridges is shown below. Try to find the answer to this game if you are interested [1].



To solve this problem, the great mathematician Euler proved that the problem was unsolvable by abstracting the four regions of the city into points and the seven bridges connecting the city into edges connecting the points. The simplified abstract diagram is as follows [2].



The four dots in the picture represent the four regions of Königsberg, and the lines between the dots represent the seven bridges connecting the four regions. It is easy to see that the area connected by the even-numbered bridges can be easily passed because different routes can be chosen to come and go. The areas connected by the odd-numbered bridges can only be used as starting or endings points because the same route can only be taken once. The number of edges associated with a node is called the node degree. Now it can be shown that the Königsberg problem can only be solved if two nodes have odd degrees and the other nodes have even degrees, i.e., two regions must have an even number of bridges and the remaining regions have an odd number of bridges. However, as we know from the above picture, there is no even number of bridges in any region of Königsberg, so this puzzle is unsolvable.

2.1.2 Property graphs

From a mathematical point of view, graph theory studies the relationships between modeled objects. However, it is common to extend the underlying graph model. The extended graphs are called the **attribute graph model**. A property graph usually consists of the following components.

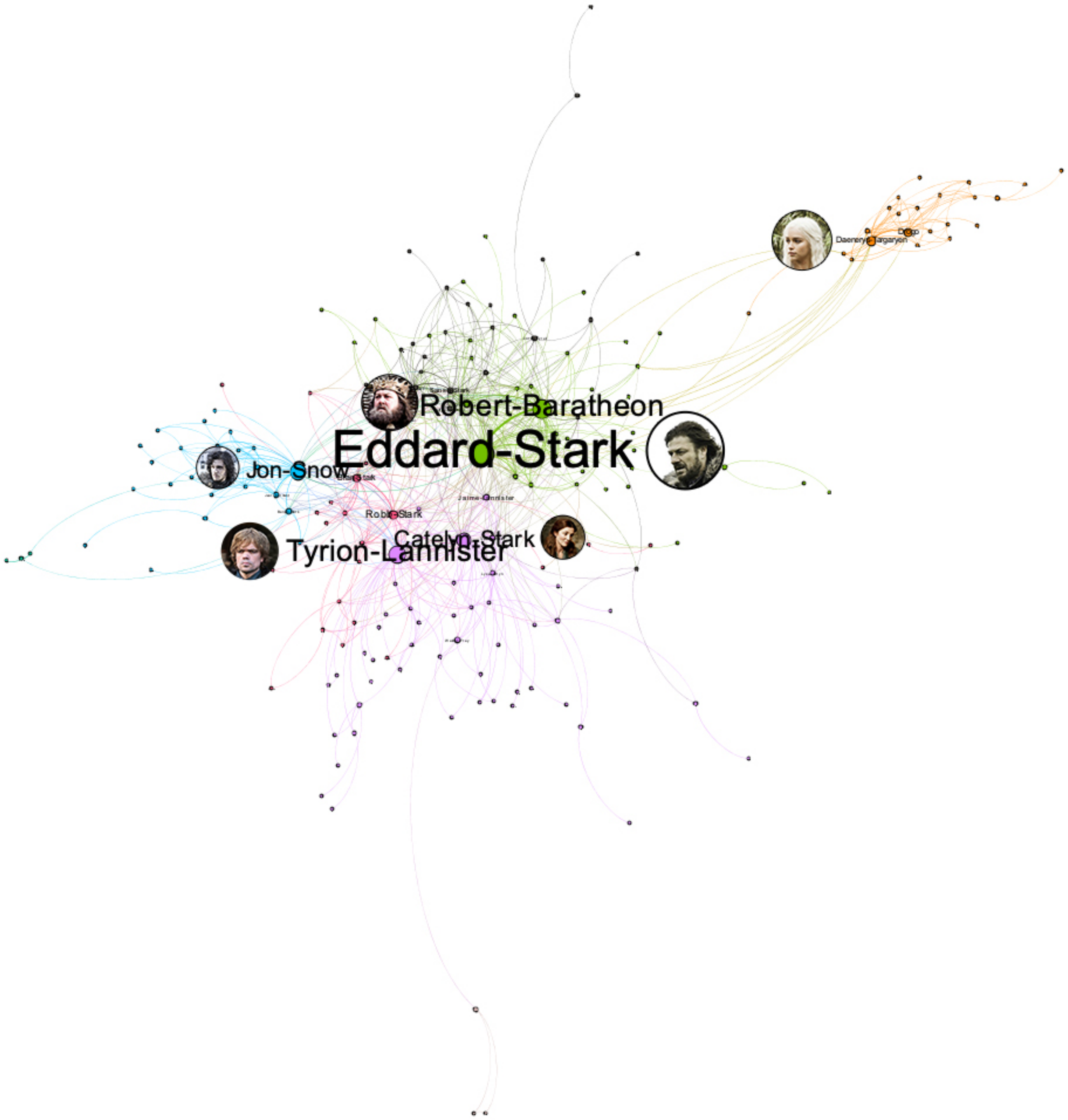
- Node, an object or entity. In this topic, nodes are called vertices.
- Relationship between nodes. In this topic, relationships are called edges. Usually, the edges can be directed or undirected to indicate a relationship between two entities.
- There can be properties on nodes and edges.

In real life, there are many examples of property graphs.

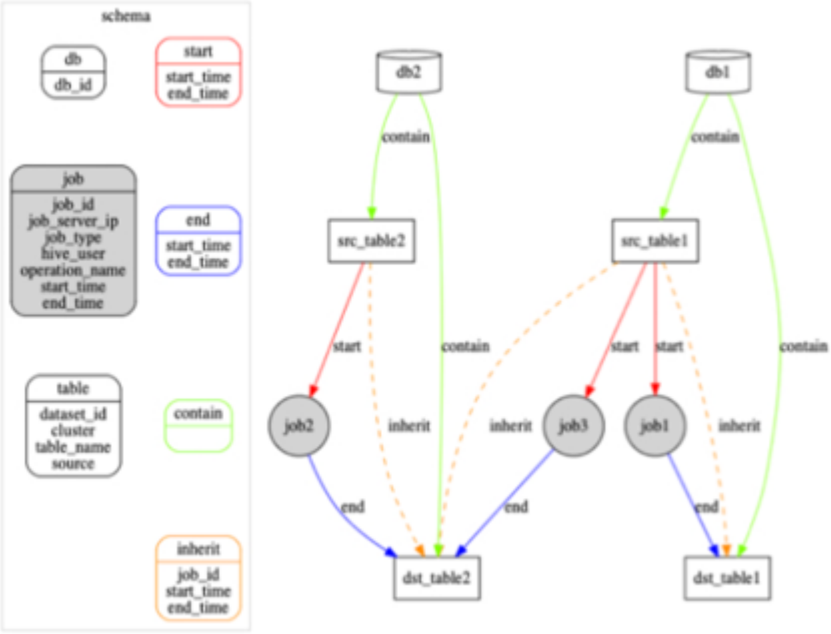
For example, Qichacha or BOSS Zhipin use graphs to model business equity relationships. A vertex usually represents a natural person or a business, and the edge represents the equity relationship between a person and a business. The properties on vertices can be the name, age, ID number, etc. of the natural person. The properties on edges can be the investment amount, investment time, position such as director and supervisor.

A vertex can be a listed company and an edge can be a correlation between listed companies. The vertex property can be a stock code, abbreviation, market capitalization, sector, etc. The edge property can be the time-series correlation coefficient of the stock price [3].

The graph relationship can also be similar to the character relationship in a TV series like Game of Thrones [4]. Vertices stand for the characters. Edges represent the interactions between the characters. Vertex properties are the character's names, ages, camps, etc., and edge properties are the number of interactions between two characters.



Graphs are also used for governance within IT systems. For example, a company like WeBank has a very large data warehouse and corresponding data warehouse management tools. These management tools record the ETL relationships between the Hive tables in the data warehouse through Job implementation [5]. Such ETL relationships can be very easily presented and managed in the form of graphs, and the root cause can be easily traced when problems arise.



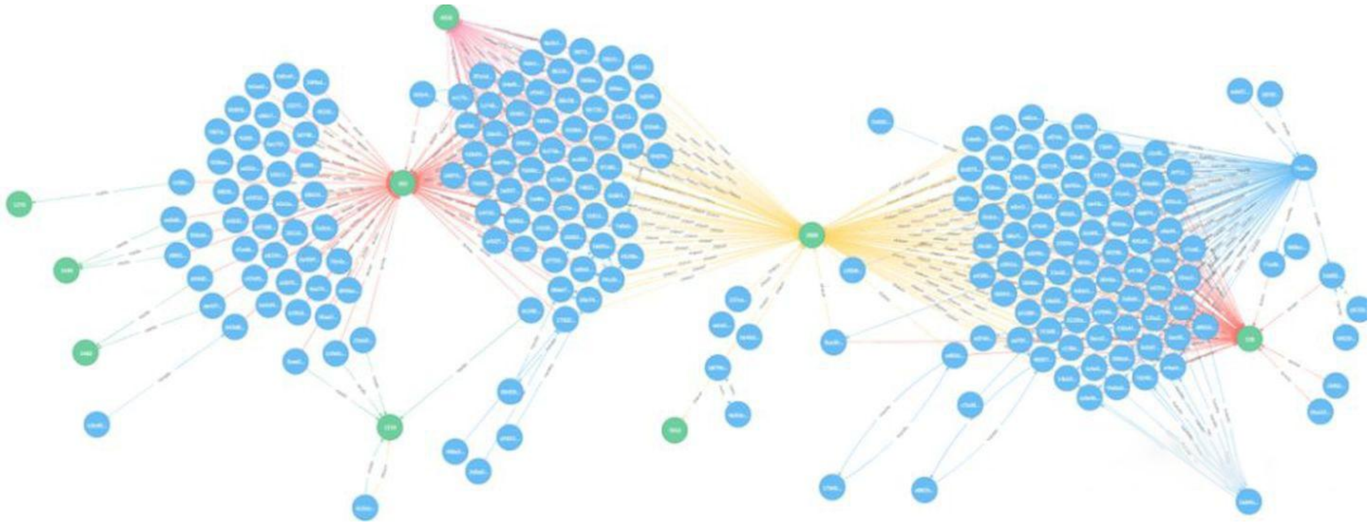
Graphs can also be used to document the invocation relationships between the intricate microservices within a large IT system [6], which is used by operations teams for service governance. Here each point represents a microservice and the edge represents the invocation relationship between two microservices; thus, Ops can easily find invocation links with availability below a threshold (99.99%) or discover microservice nodes that would be particularly affected by a failure.

Graphs are also used to record the invocation relationships between the intricate microservices [6]. Each vertex represents a microservice and an edge represents the invocation relationship between two microservices. This allows Ops to easily find call links with availability below a threshold (99.99%), or to discover microservice nodes where a failure would have a particularly large impact.

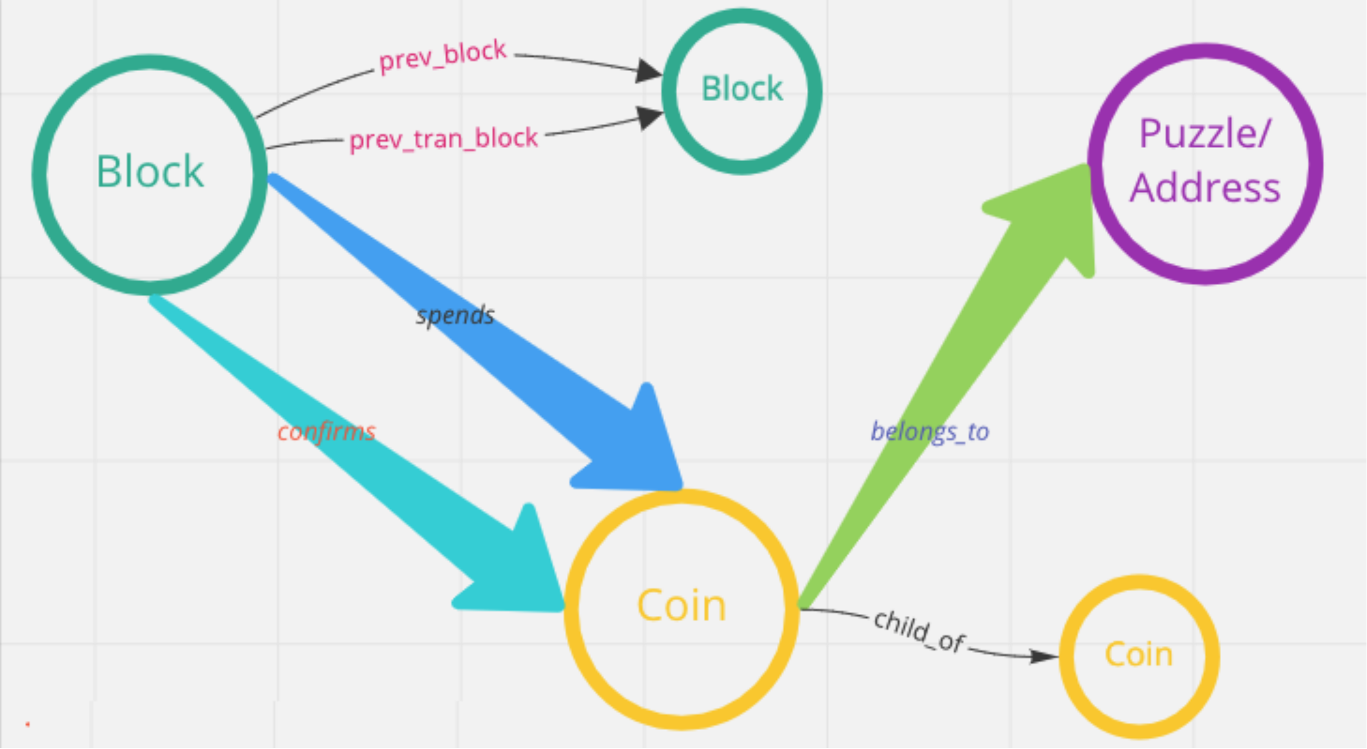
Graphs can also be used to improve the efficiency of code development. Graphs store function call relationships between codes [6] to improve the efficiency of reviewing and testing the code. In such a graph, each vertex is a function or variable, each edge is a call relationship between functions or variables. When there is a new code commit, one can more easily see other interfaces that may be affected, which helps testers better assess potential go-live risks.

In addition, we can discover more scenarios by adding some temporal information as opposed to a static property graph that does not change.

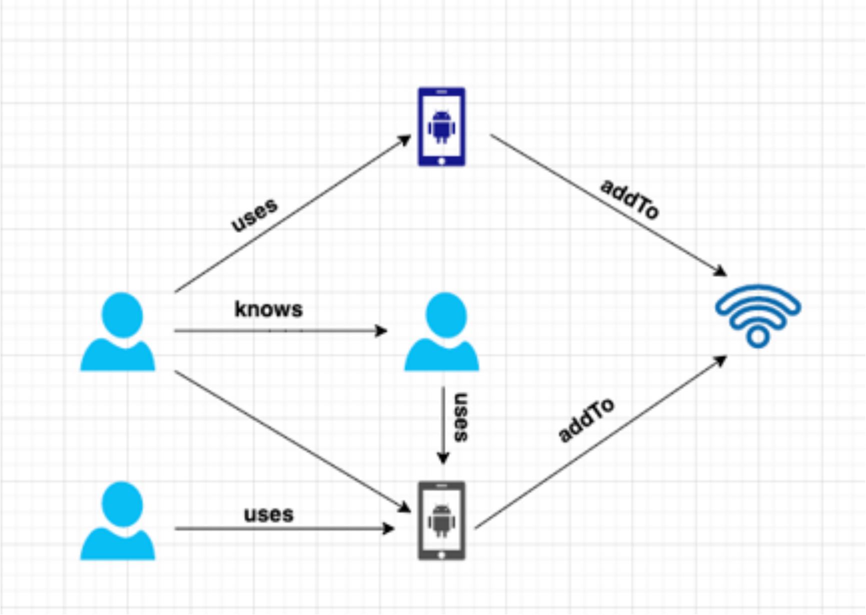
For example, inside a network of interbank account fund flows [7], a vertex is an account, an edge is the transfer record between accounts. Edge properties record the time, amount, etc. of the transfer. Companies can use graph technology to easily explore the graph to discover obvious misappropriation of funds, paying back a load to with the loan, loan gang scams, and other phenomena.



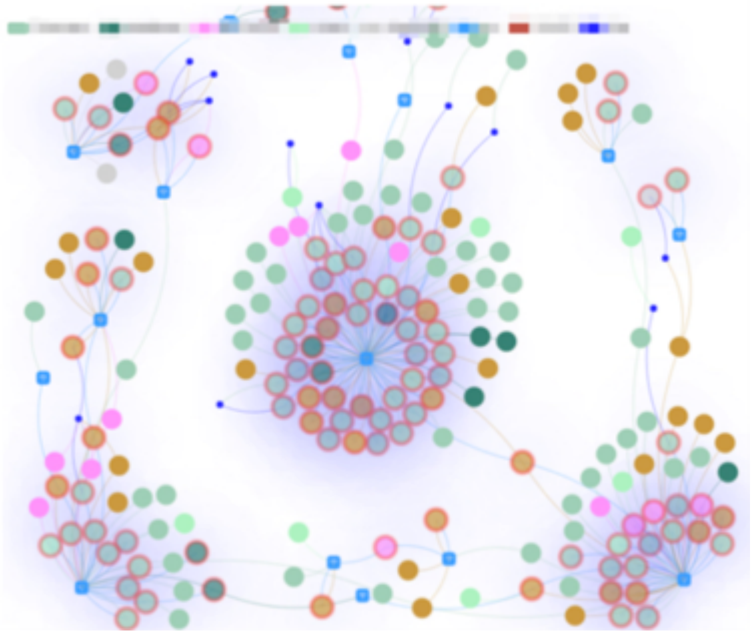
The same approach can be used to explore the discovery of the flow of cryptocurrencies.



In a network of accounts and devices [8], vertices can be accounts, mobile devices, and WIFI networks, edges are the login relationships between these accounts and mobile devices, and the access relationships between mobile devices and WIFI networks.

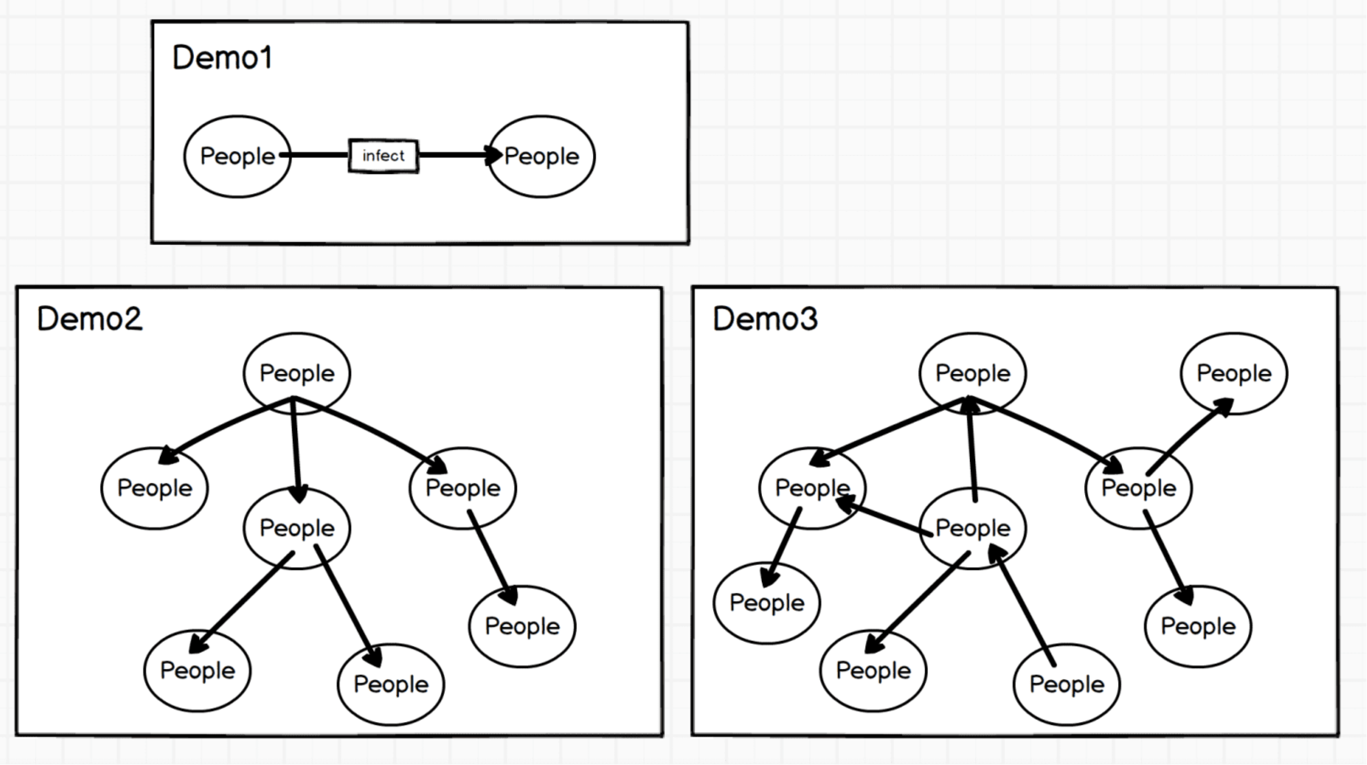


These graph data records the characteristic of the network black production operations. Some big companies such as 360 DigiTech[8], Kuaishou[9], WeChat[10], Zhihu[11], and Ctrip Finance all identified over a million crime groups through technology.



In addition to the dimension of time, you can find more scenarios for property graphs by adding some geographic location information.

For an example of tracing the source of the Coronavirus Disease (COVID-19) [12], vertices are the person and edges are the contact between people. Vertex properties are the information of the person's ID card and onset time, and edge properties are the time and geographical location of the close contact between people, etc. It provides help for health prevention departments to quickly identify high-risk people and their behavioral trajectories.



The combination of geographic location and graph is also used in some O2O scenarios, such as real-time food recommendation based on POI (Point-of-Interest) [13], which enables local life service platform companies like Meituan to recommend more suitable businesses in real-time when consumers open the APP.

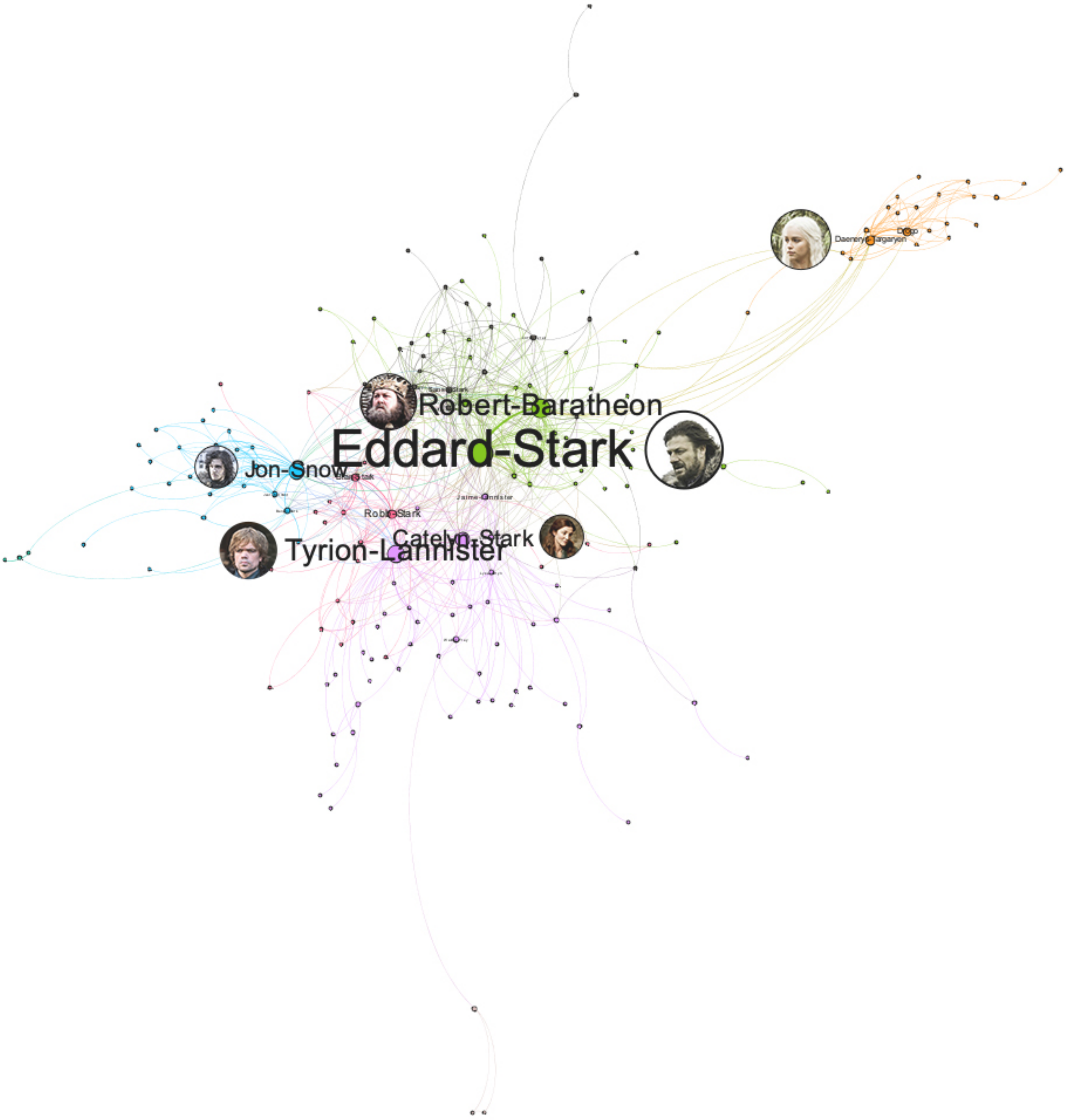
A graph is also used for knowledge inference. Huawei, Vivo, OPPO, WeChat, Meituan, and other companies use graphs for the representation of the underlying knowledge relationships.

2.1.3 Why do we use graph databases?

Although relational databases and semi-structured databases such as XML/JSON can be used to describe a graph-structured data model, a graph (database) not only describes the graph structure and stores data itself but also focuses on handling the associative relationships between the data. Specifically, graph databases have several advantages:

- Graphs are a more visual and intuitive way of representing knowledge to human brains. This allows us to focus on the business problem itself rather than how to describe the problem as a particular structure of the database (e.g., a table structure).

- It is easier to show the characteristic of the data in graphs. Such as transfer paths and nearby communities. To analyze the relationships of characters and character importance in Game of Thrones, data displayed with tables is not as intuitive as with graphs.

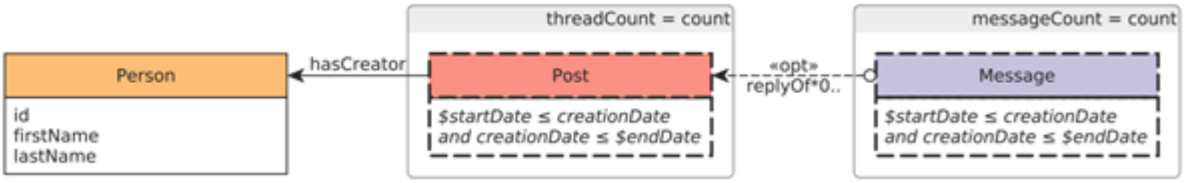


Especially when some central vertices are deleted:

Adding an edge can completely change the entire topology.

We can intuitively sense the importance of minor changes in graphs rather than in tables.

- Graph query language is designed based on graph structures. The following is a query example in LDBC. Requirements: Query the posts posted by a person, and query the corresponding replies (the replies themselves will also be replied multiple times). Since the posting time and reply time both meet certain conditions, you can sort the results according to the number of replies.


Person
id
firstName
lastName
hasCreator
threadCount = count
Post
$startDate ≤ creationDate
and creationDate ≤ $endDate
«opt»
replyOf*0..
messageCount = count
Message
$startDate ≤ creationDate
and creationDate ≤ $endDate

Write querying statements using PostgreSQL:

```
--PostgreSQL
WITH RECURSIVE post_all(psa_threadid
                      , psa_thread_creatorid, psa_messageid
                      , psa_creationdate, psa_messagetype
                       ) AS (
    SELECT m_messageid AS psa_threadid
         , m_creatorid AS psa_thread_creatorid
         , m_messageid AS psa_messageid
         , m_creationdate, 'Post'
      FROM message
     WHERE 1=1 AND m_c_replyof IS NULL -- post, not comment
       AND m_creationdate BETWEEN :startDate AND :endDate
  UNION ALL
    SELECT psa.psa_threadid AS psa_threadid
         , psa.psa_thread_creatorid AS psa_thread_creatorid
         , m_messageid, m_creationdate, 'Comment'
      FROM message p, post_all psa
     WHERE 1=1 AND p.m_c_replyof = psa.psa_messageid
     AND m_creationdate BETWEEN :startDate AND :endDate
)
SELECT p.p_personid AS "person.id"
     , p.p_firstname AS "person.firstName"
     , p.p_lastname AS "person.lastName"
     , count(DISTINCT psa.psa_threadid) AS threadCount
END) AS messageCount
     , count(DISTINCT psa.psa_messageid) AS messageCount
  FROM person p left join post_all psa on (
       1=1   AND p.p_personid = psa.psa_thread_creatorid
   AND psa_creationdate BETWEEN :startDate AND :endDate
   )
 GROUP BY p.p_personid, p.p_firstname, p.p_lastname
 ORDER BY messageCount DESC, p.p_personid
 LIMIT 100;
```

Write querying statements using Cypher designed especially for graphs:

```
--Cypher
MATCH (person:Person)<-[:HAS_CREATOR]-(post:Post)<-[:REPLY_OF*0..]-(reply:Message)
WHERE post.creationDate >= $startDate AND post.creationDate <= $endDate
  AND reply.creationDate >= $startDate AND reply.creationDate <= $endDate
RETURN
  person.id, person.firstName, person.lastName, count(DISTINCT post) AS threadCount,
  count(DISTINCT reply) AS messageCount
ORDER BY
  messageCount DESC, person.id ASC
LIMIT 100
```

- Graph traversal (corresponding to Join in SQL) is much more efficient because the storage and query engines are designed specifically for the structure of the graph.

- Graph databases have a wide range of application scenarios. Examples include data integration (knowledge graph), personalized recommendations, fraud, and threat detection, risk analysis, and compliance, identity (and control) verification, IT infrastructure management, supply chain, and logistics, social network research, etc.

- According to the literature [14], the fields that use graph technology are (from the greatest to least): information technology (IT), research in academia, finance, laboratories in industry, government, healthcare, defense, pharmaceuticals, retail, and e-commerce, transportation, telecommunications, and insurance.

- In 2019, according to Gartner's questionnaire research, 27% of customers (500 groups) are using graph databases and 20% have plans to use them.

2.1.4 RDF

This topic does not discuss the RDF data model due to space limitations.

1. Souce of the picture: https://medium.freecodecamp.org/i-dont-understand-graph-theory-1c96572a1401. ↩
2. Source of the picture: https://medium.freecodecamp.org/i-dont-understand-graph-theory-1c96572a1401 ↩
3. https://nebula-graph.com.cn/posts/stock-interrelation-analysis-jgrapht-nebula-graph/ ↩
4. https://nebula-graph.com.cn/posts/game-of-thrones-relationship-networkx-gephi-nebula-graph/ ↩
5. https://nebula-graph.com.cn/posts/practicing-nebula-graph-webank/ ↩
6. https://nebula-graph.com.cn/posts/meituan-graph-database-platform-practice/ ↩↩↩
7. https://zhuanlan.zhihu.com/p/90635957 ↩
8. https://nebula-graph.com.cn/posts/graph-database-data-connections-insight/ ↩↩
9. https://nebula-graph.com.cn/posts/kuaishou-security-intelligence-platform-with-nebula-graph/ ↩
10. https://nebula-graph.com.cn/posts/nebula-graph-for-social-networking/ ↩
11. https://mp.weixin.qq.com/s/K2QinpR5Rplw1teHpHtf4w ↩
12. https://nebula-graph.com.cn/posts/detect-corona-virus-spreading-with-graph-database/ ↩
13. https://nebula-graph.com.cn/posts/meituan-graph-database-platform-practice/ ↩
14. https://arxiv.org/abs/1709.03188 ↩

Last update: August 11, 2022

2.2 Market overview of graph databases

Now that we have discussed what a graph is, let's move on to further understanding graph databases developed based on graph theory and the property graph model.

Different graph databases may differ slightly in terms of terminology, but in the end, they all talk about vertices, edges, and properties. As for more advanced features such as labels, indexes, constraints, TTL, long tasks, stored procedures, and UDFs, these advanced features will vary significantly from one graph database to another.

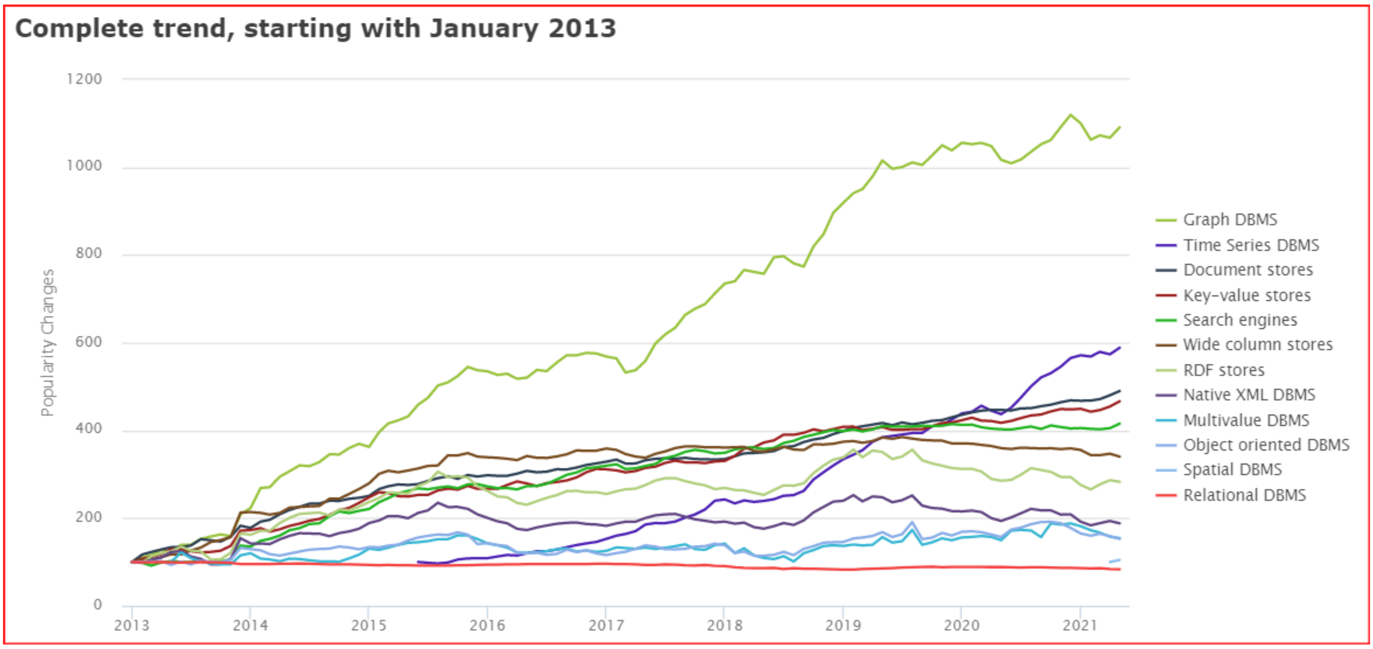
Graph databases use graphs to store data, and the graph structure is one of the structures that are closest to high flexibility and high performance. A graph database is a storage engine specifically designed to store and retrieve large information, which efficiently stores data as vertices and edges and allows high-performance retrieval and querying of these vertex-edge structures. We can also add properties to these vertices and edges.

2.2.1 Third-party services market predictions

DB-Engines ranking

According to DB-Engines.com, the world's leading database ranking site, graph databases have been the fastest growing database category since 2013 [1].

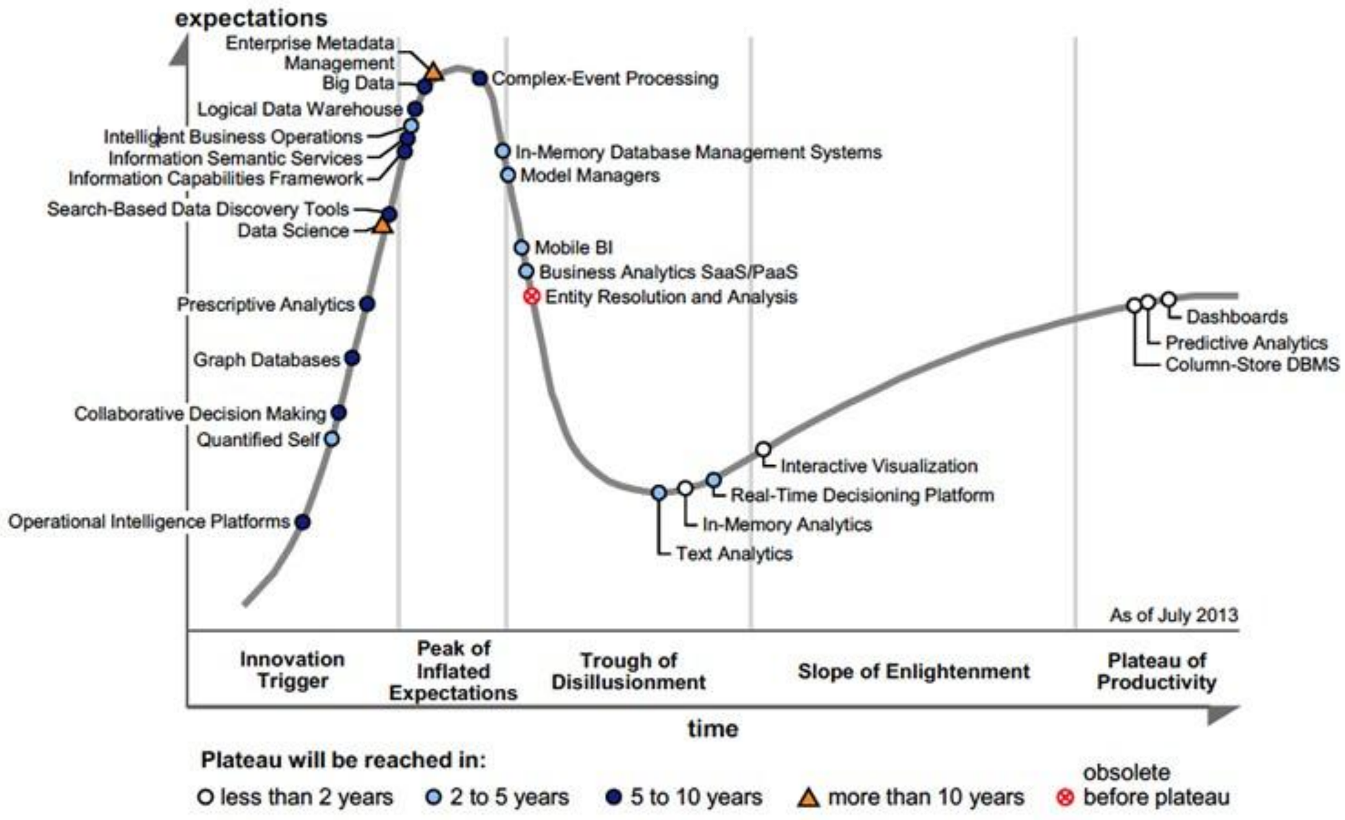
The site counts trends in the popularity of each category based on several metrics, including records and trends based on search engines such as Google, technical topics discussed on major IT technology forums and social networking sites, job posting changes on job boards. 371 database products are included in the site and are divided into 12 categories. Of these 12 categories, a category like graph databases is growing much faster than any of the others.



Gartner's predictions

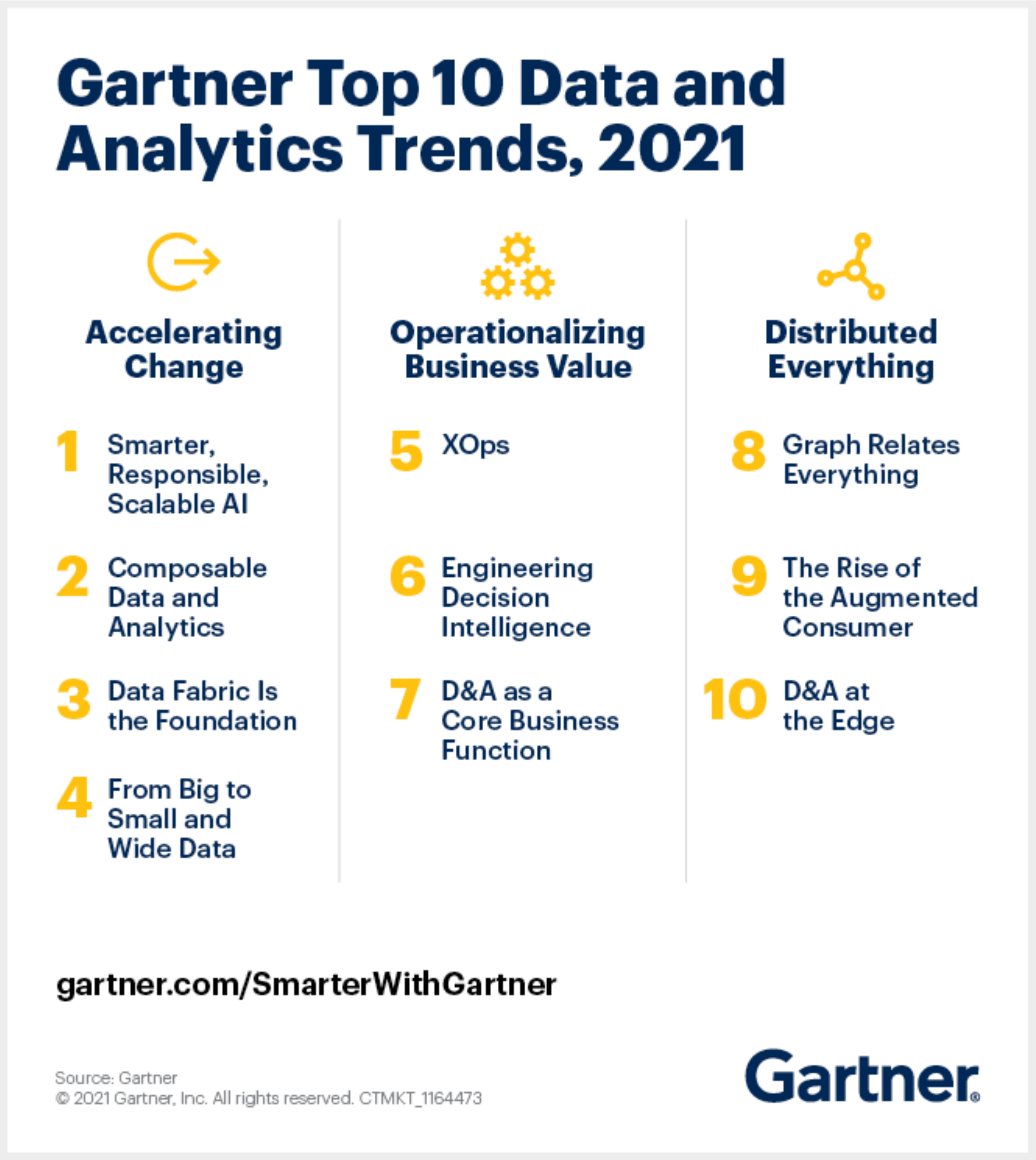
Gartner, one of the world's top think tanks, identified graph databases as a major business intelligence and analytics technology trend long before 2013 [2]. At that time, big data was hot as ever, and data scientists were in a hot position.

Figure 1. Hype Cycle for Business Intelligence and Analytics, 2013



BI = business intelligence; DBMS = database management system; SaaS = software as a service; PaaS = platform as a service

Until recently, graph databases and related graph technologies were ranked in the Top 10 Data and Analytics Trends for 2021 [3].



Gartner Top 10 Data and Analytics Trends, 2021

Accelerating Change

1 Smarter, Responsible, Scalable AI

2 Composable Data and Analytics

3 Data Fabric Is the Foundation

4 From Big to Small and Wide Data

Operationalizing Business Value

5 XOps

6 Engineering Decision Intelligence

7 D&A as a Core Business Function

Distributed Everything

8 Graph Relates Everything

9 The Rise of the Augmented Consumer

10 D&A at the Edge

gartner.com/SmarterWithGartner

Source: Gartner
© 2021 Gartner, Inc. All rights reserved. CTMKT_1164473

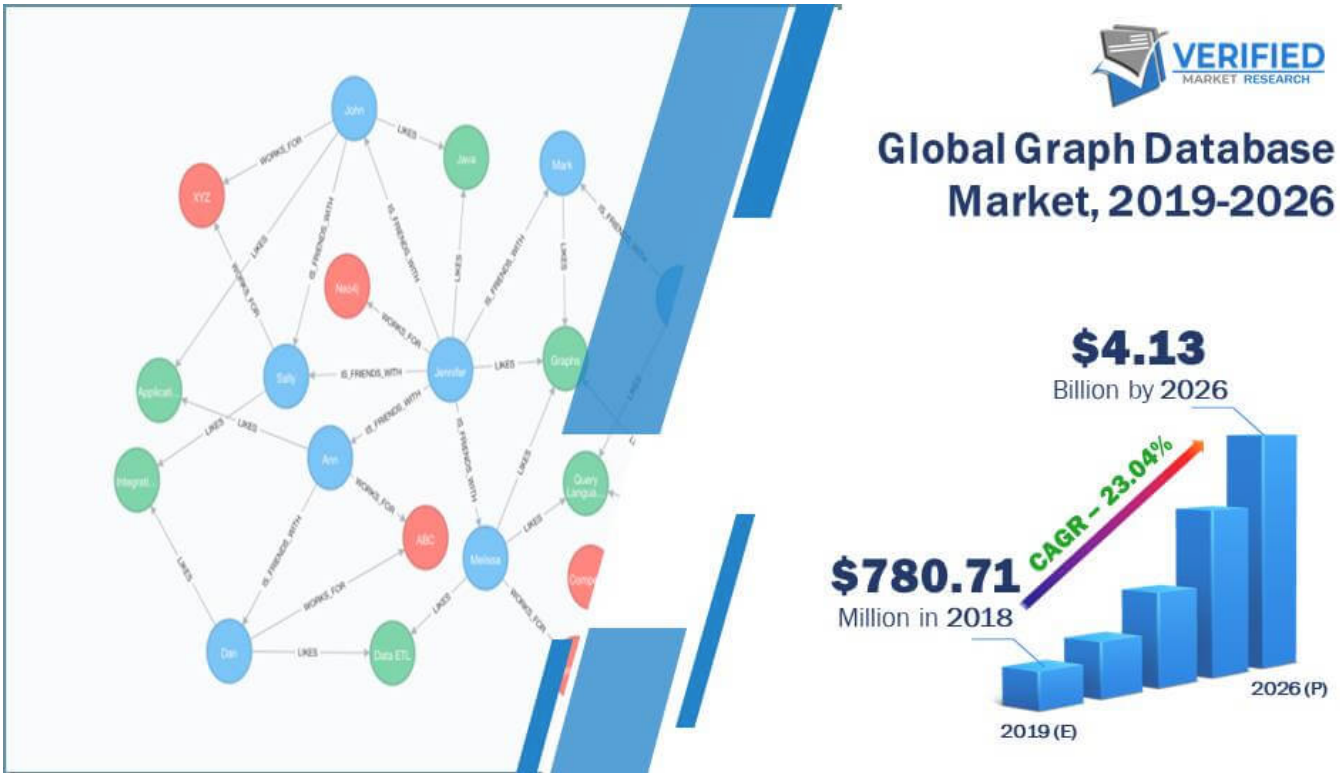
Gartner®

> **Trend 8: Graph Relates Everything**
>
> Graphs form the foundation of many modern data and analytics capabilities to find relationships between people, places, things, events, and locations across diverse data assets. D&A leaders rely on graphs to quickly answer complex business questions which require contextual awareness and an understanding of the nature of connections and strengths across multiple entities.
>
> Gartner predicts that by 2025, graph technologies will be used in 80% of data and analytics innovations, up from 10% in 2021, facilitating rapid decision-making across the organization.

It can be noted that Gartner's predictions match the DB-Engines ranking well. There is usually a period of rapid bubble development, then a plateau period, followed by a new bubble period due to the emergence of new technologies, and then a plateau period again.

Market size of graph databases

According to statistics and forecasts from Verifiedmarketresearc[4], fnfresearch[5], MarketsandMarkets[6], and Gartner[7], the global graph database market size is about to grow from about USD 0.8 billion in 2019 to USD 3-4 billion by 2026, at a Compound Annual Growth Rate (CAGR) of about 25%, which corresponds to about 5%-10% market share of the global database market.



2.2.2 Market participants

Neo4j, the pioneer of (first generation) graph databases

Although some graph-like data models and products, and the corresponding graph language G/G+ had been proposed in the 1970s (e.g. CODASYL [8]). But it is Neo4j, the main pioneer in this market, that has really made the concept of graph databases popular, and even the two main terms (labeled) property graphs and graph databases were first introduced and practiced by Neo4j.

!!! Info "This section on the history of Neo4j and the graph query language it created, Cypher, is largely excerpted from the ISO WG3 paper *An overview of the recent history of Graph Query Languages* [10] and [9]. To take into account the latest two years of development, the content mentioned in this topic has been abridged and updated by the authors of this book.

About GQL (Graph Query Language) and the development of an International Standard

Readers familiar with databases are probably aware of the Structured Query Language SQL. by using SQL, people access databases in a way that is close to natural language. Before SQL was widely adopted and standardized, the market for relational databases was very fragmented. Each vendor's product had a completely different way of accessing. Developers of the database product itself, developers of the tools surrounding the database product, and end-users of the database, all had to learn each product. When the SQL-89 standard was developed in 1989, the entire relational database market quickly focus on SQL-89. This greatly reduced the learning costs for the people mentioned above.

GQL (Graph Query Language) assumes a role similar to SQL in the field of graph databases. Uses interacts with graphs with GQL. Unlike international standards such as SQL-89, there are no international standards for GQL. Two mainstream graph languages are Neo4j’s Cypher and Apache TinkerPop's Gremlin. The former is often referred to as the DQL, Declarative Query Language. DQL tells the system "what to do", regardless of "how to do". The latter is referred to as the IQL, Imperative Query Language. IQL explicitly specifies the system's actions.

The GQL International Standard is in the process of being developed.

OVERVIEW OF THE RECENT HISTORY OF GRAPH DATABASES

- In 2000, the idea of modeling data as a network came to the founders of Neo4j.
- In 2001, Neo4j developed the earliest core part of the code.
- In 2007, Neo4j started operating as a company.
- In 2009, Neo4j borrowed XPath as a graph query language. Gremlin [11] is also similar to XPath.
- In 2010, Marko Rodriguez, a Neo4j employee, used the term Property Graph to describe the data model of Neo4j and TinkerPop (Gremlin).
- In 2011, the first public version Neo4j 1.4 was released, and the first version of Cypher was released.
- In 2012, Neo4j 1.8 enabled you to write a Cypher. Neo4j 2.0 added labels and indexes. Cypher became a declarative graph query language.
- In 2015, Cypher was opened up by Neo4j through the openCypher project.
- In 2017, the ISO WG3 organization discussed how to use SQL to query property graph data.
- In 2018, Starting from the Neo4j 3.5 GA, the core of Neo4j only for the Enterprise Edition will no longer be open source.
- In 2019, ISO officially established two projects ISO/IEC JTC 1 N 14279 and ISO/IEC JTC 1/SC 32 N 3228 to develop an international standard for graph database language.
- In 2021, the $325 million Series F funding round for Neo4j marks the largest investment round in database history.

THE EARLY HISTORY OF NEO4J

The data model property graph was first conceived in 2000. The founders of Neo4j were developing a media management system, and the schema of the system was often changed. To adapt to such changes, Peter Neubauer, one of the founders, wanted to enable the system to be modeled to a conceptually interconnected network. A group of graduate students at the Indian Institute of Technology Bombay implemented the earliest prototypes. Emil Eifrém, the Neo4j co-founder, and these students spent a week extending Peter's idea into a more abstract model: vertices were connected by relationships, and key-values were used as properties of vertices and relationships. They developed a Java API to interact with this data model and implemented an abstraction layer on top of the relational database.

Although this network model greatly improved productivity, its performance has been poor. So Johan Svensson, Neo4j co-founder, put a lot of effort into implementing a native data management system, that is Neo4j. For the first few years, Neo4j was successful as an in-house product. In 2007, the intellectual property of Neo4j was transferred to an independent database company.

In the first public release of Neo4j (Neo4j 1.4, 2011), the data model was consisted of vertices and typed edges. Vertices and edges have properties. The early versions of Neo4j did not have indexes. Applications had to construct their search structure from the root vertex. Because this was very unwieldy for the applications, Neo4j 2.0 (2013.12) introduced a new concept label on vertices. Based on labels, Neo4j can index some predefined vertex properties.

"Vertex", "Relationship", "Property", "Relationships can only have one label.", "Vertices can have zero or multiple labels.". All these concepts form the data model definitions for Neo4j property graphs. With the later addition of indexing, Cypher became the main way of interacting with Neo4j. This is because the application developer only needs to focus on the data itself, not on the search structure that the developer built himself as mentioned above.

THE CREATION OF GREMLIN

Gremlin is a graph query language based on Apache TinkerPop, which is close in style to a sequence of function (procedure) calls. Initially, Neo4j was queried through the Java API. applications could embed the query engine as a library into the application and then use the API to query the graph.

The early Neo4j employees Tobias Lindaaker, Ivarsson, Peter Neubauer, and Marko Rodriguez used XPath as a graph query. Groovy provides loop structures, branching, and computation. This was the original prototype of Gremlin, the first version of which was released in November 2009.

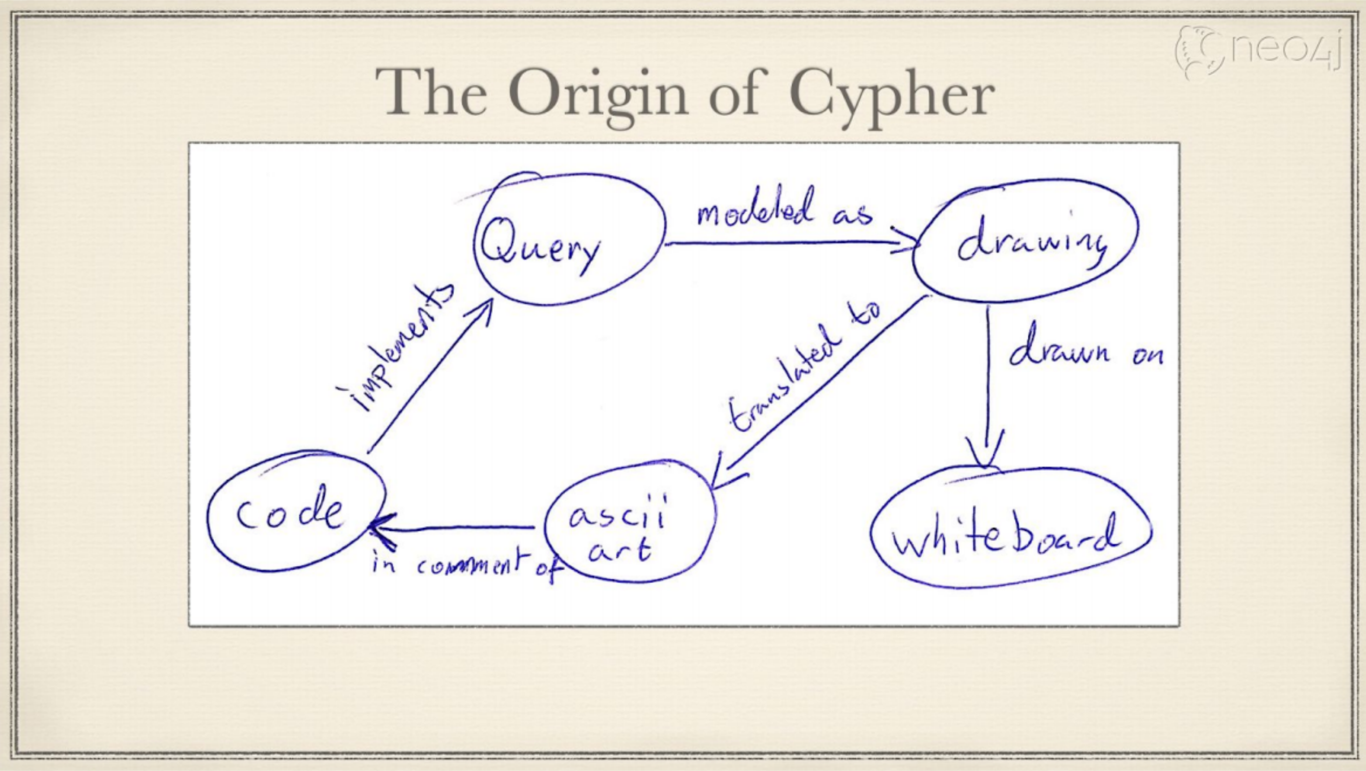
Later, Marko found a lot of problems with using two different parsers (XPath and Groovy) at the same time and changed Gremlin to a Domain Specific Language (DSL) based on Groovy.

THE CREATION OF CYPHER

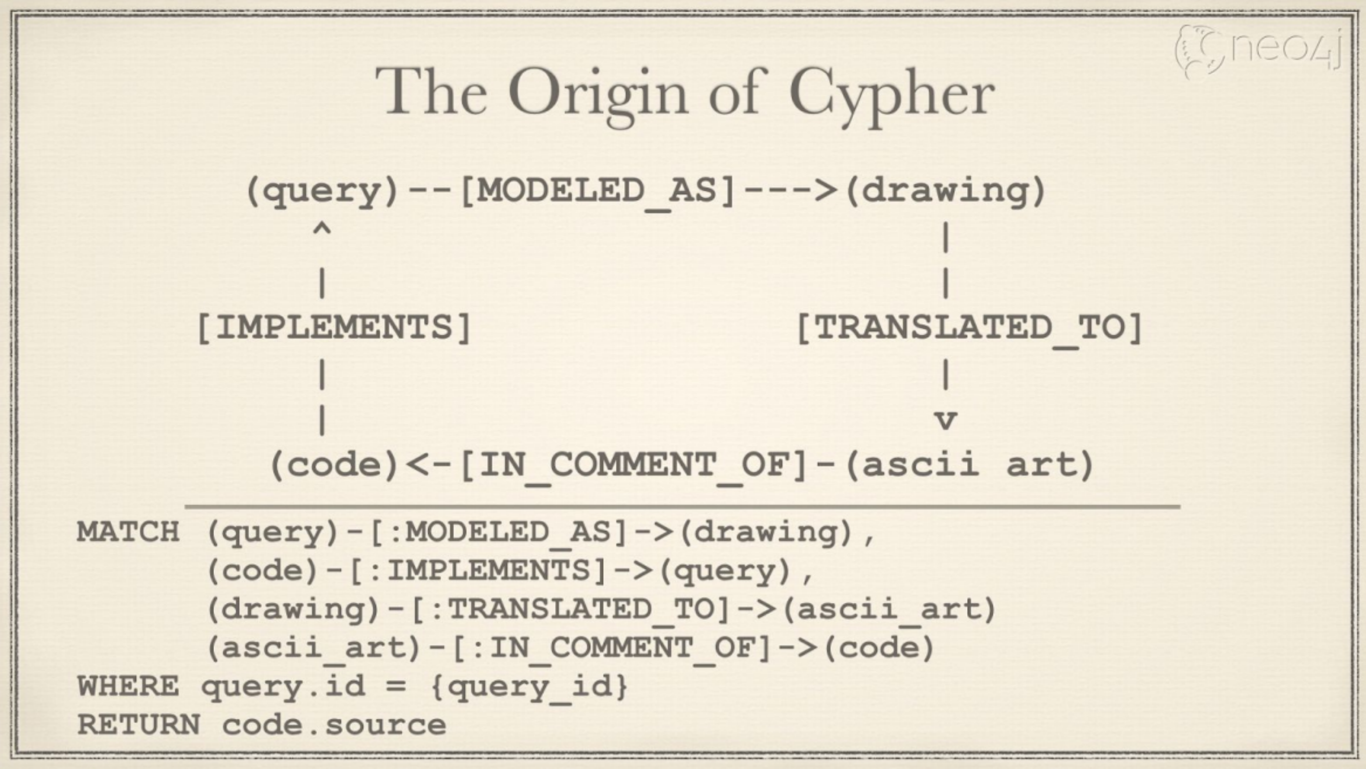
Gremlin, like Neo4j's Java API, was originally intended to be a procedural way of expressing how to query databases. It uses shorter syntaxes to query and remotely access databases through the network. The procedural nature of Gremlin requires users to know the best way to query results, which is still burdensome for application developers. Over the last 30 years, the declarative language SQL has been a great success. SQL can separate the declarative way to get data from how the engine gets data. So the Neo4j engineers wanted to develop a declarative graph query language.

In 2010, Andrés Taylor joined Neo4j as an engineer. Inspired by SQL, he started a project to develop graph query language, which was released as Neo4j 1.4 in 2011. The language is the ancestor of most graph query languages today - Cypher.

Cypher's syntax is based on the use of ASCII art to describe graph patterns. This approach originally came from the annotations on how to describe graph patterns in the source code. An example can be seen as follows.



The Origin of Cypher



The Origin of Cypher

```
(query)--[MODELED_AS]--->(drawing)
   ^                         |
   |                         |
[IMPLEMENTS]           [TRANSLATED_TO]
   |                         |
   |                         v
 (code)<-[IN_COMMENT_OF]-(ascii art)
```

```
MATCH (query)-[:MODELED_AS]->(drawing),
      (code)-[:IMPLEMENTS]->(query),
      (drawing)-[:TRANSLATED_TO]->(ascii_art)
      (ascii_art)-[:IN_COMMENT_OF]->(code)
WHERE query.id = {query_id}
RETURN code.source
```

Simply put, ASCII art uses printable text to describe vertices and edges. Cypher syntax uses `()` for vertices and `-[]->` for edges. `(query)-[modeled as]->(drawing)` is used to represent a simple graph relationship (which can also be called graph schema): `the starting vertex - query`, `the destination vertex - drawing`, and `the edge - modeled as`.

The first version of Cypher implemented graph reading, but users should specify vertices from which to start querying. Only from these vertices could graph schema matching be supported.

In a later version, Neo4j 1.8, released in October 2012, Cypher added the ability to modify graphs. However, queries still need to specify which nodes to start from.

In December 2013, Neo4j 2.0 introduced the concept of a label, which is essentially an index. This allows the query engine to use the index to select the vertices matched by the schema, without requiring the user to specify the vertex to start the query.

With the popularity of Neo4j, Cypher has a wide community of developers and is widely used in a variety of industries. It is still the most popular graph query language.

In September 2015, Neo4j established the openCypher Implementors Group (oCIG) to open source Cypher to openCypher, to govern and advance the evolution of the language itself through open source.

SUBSEQUENT EVENTS

Cypher has inspired a series of graph query languages, including:

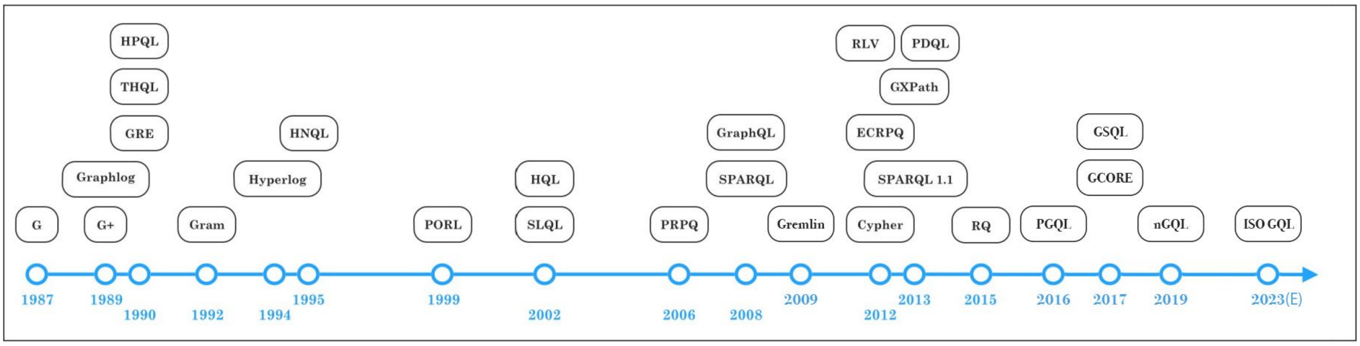
2015, Oracle released PGQL, a graph language used by the graph engine PGX.

2016, the Linked Data Benchmarking Council (short for LDBC) an industry-renowned benchmarking organization for graph performance, released G-CORE.

2018, RedisGraph, a Redis-based graph library, adopted Cypher as its graph language.

2019, the International Standards Organization ISO started two projects to initiate the process of developing an international standard for graph languages based on existing industry achievements such as openCypher, PGQL, GSQL[12], and G-CORE.

2019, NebulaGraph released NebulaGraph Query Language (nGQL) based on openCypher.



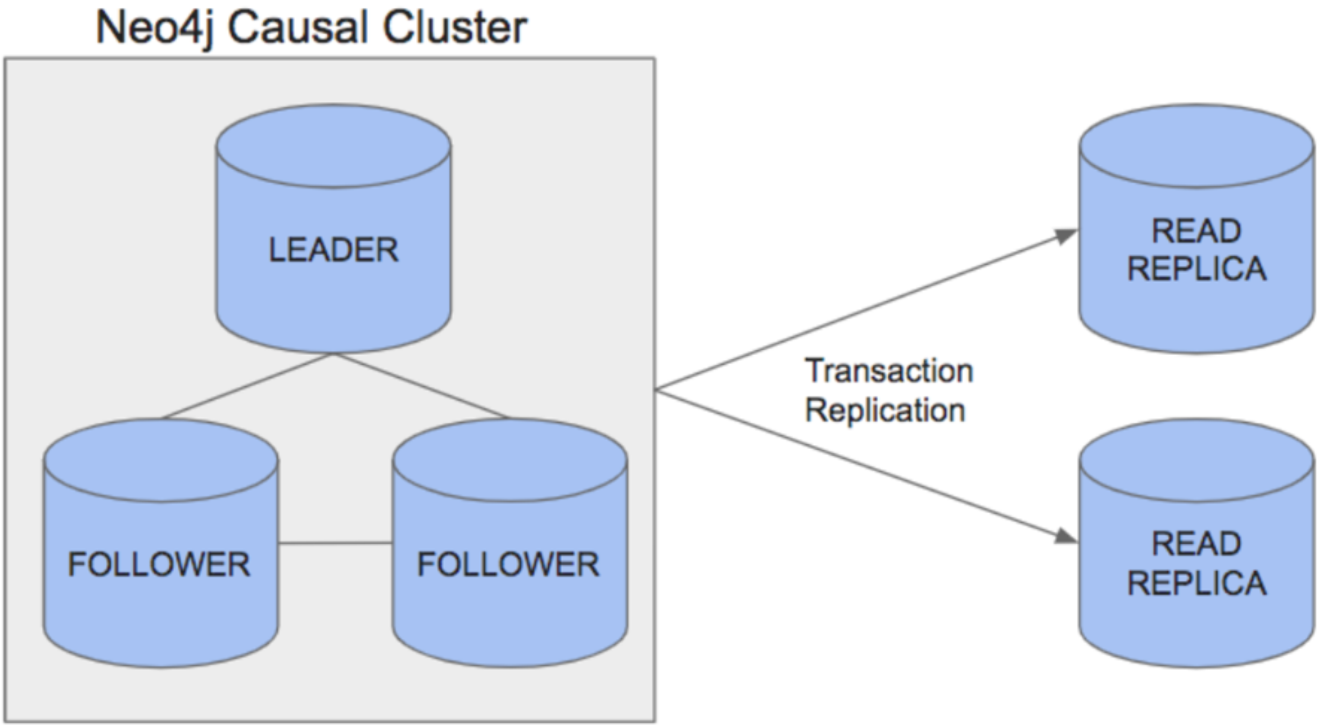
Distributed graph databases

From 2005 to 2010, with the release of Google's cloud computing "Troika", various distributed architectures became increasingly popular, including Hadoop and Cassandra, which have been open-sourced. Several implications are as follows:

1. The technical and cost advantages of distributed systems over single machines (e.g. Neo4j) or small machines are more obvious due to the increasing volume of data and computation. Distributed systems allow applications to access these thousands of machines as if they were local systems, without the need for much modification at the code level.
2. The open-source approach allows more people to know emerging technologies and feedback to the community in a more cost-effective way, including code developers, data scientists, and product managers.

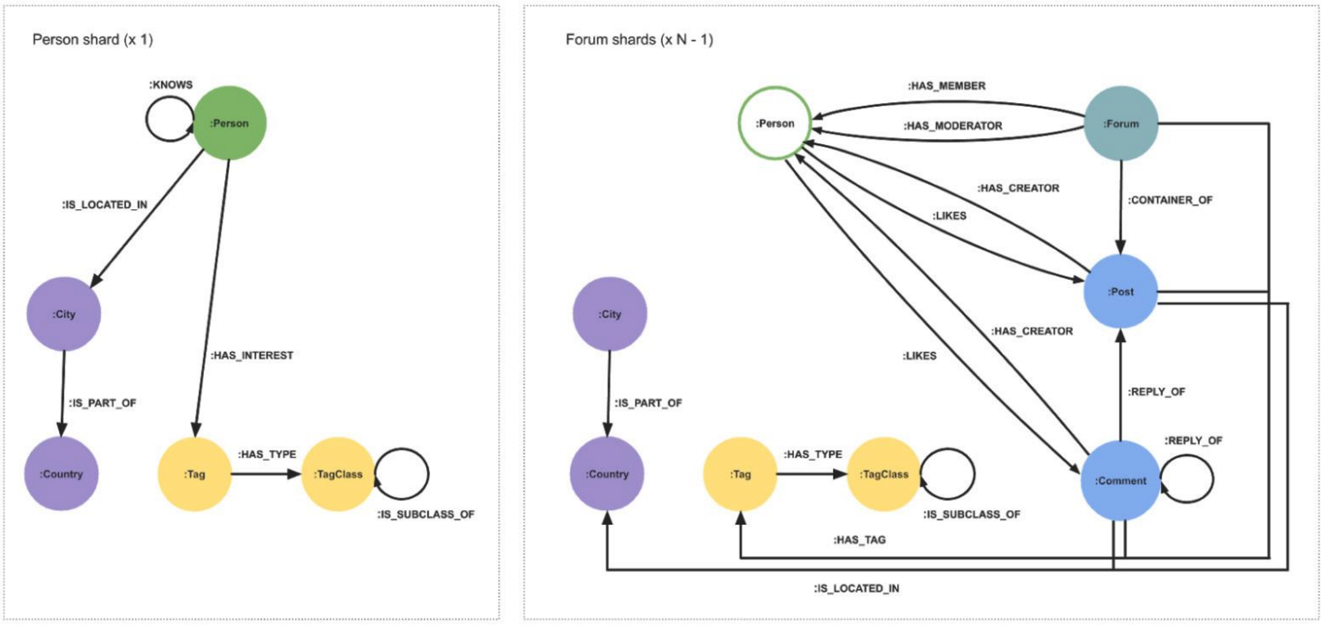
Strictly speaking, Neo4j also offers several distributed capabilities, which are quite different from the industry's sense of the distributed system.

- Neo4j 3. x requires that the full amount of data must be stored on a single machine. Although it supports full replication and high availability between multiple machines, the data cannot be sliced into different subgraphs.



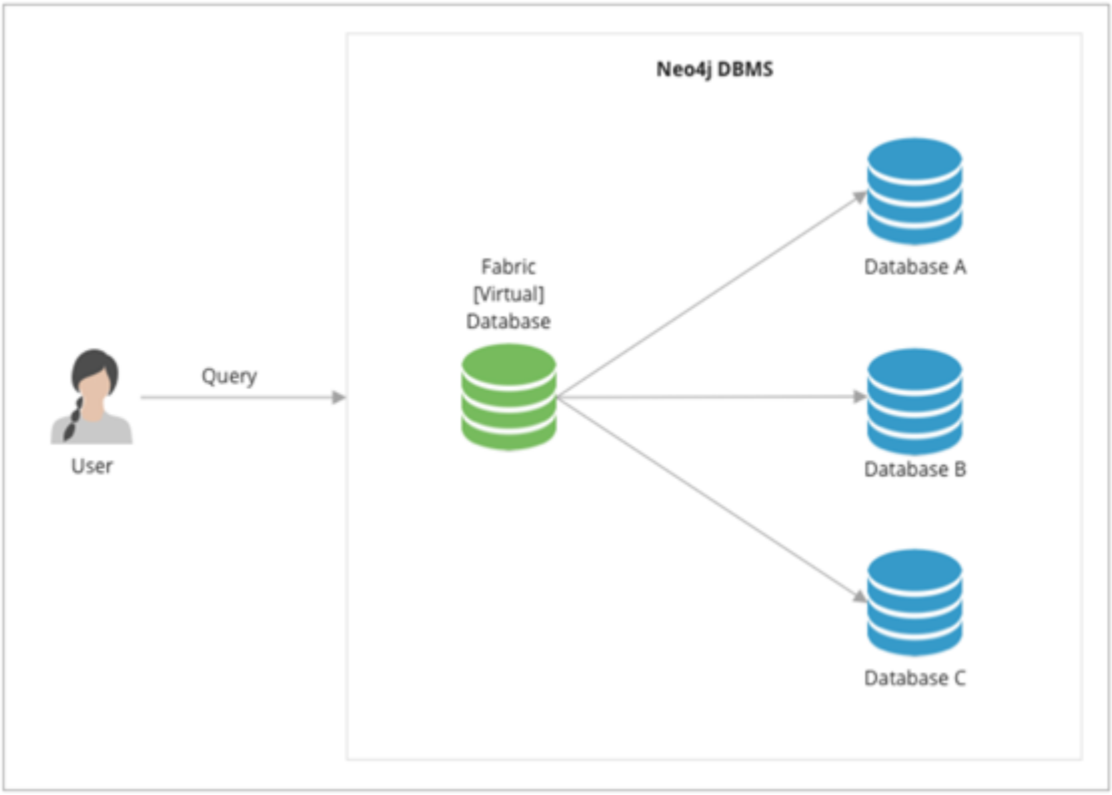
Cluster architecture

- Neo4j 4. x stores a part of data on different machines (subgraphs), and then the application layer assembles data in a certain way (called Fabric)[13] and distributes the reads and writes to each machine. This approach requires a log of involvement and work from the application layer code. For example, designing how to place different subgraphs on which machines they should be placed and how to assemble some of the results obtained from each machine into the final result.



The style of its syntax is as follows:

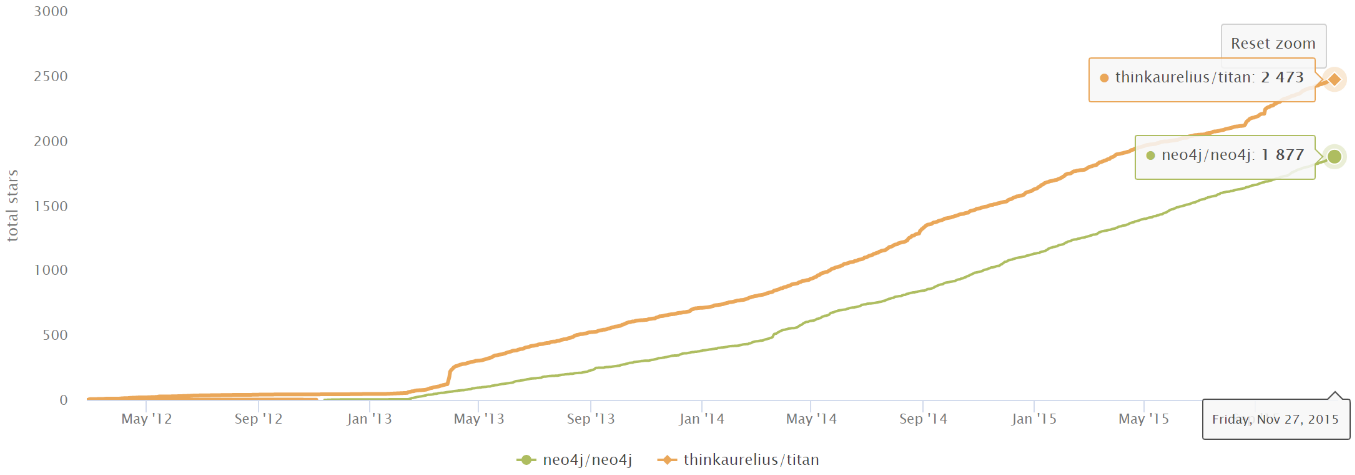
```
USE graphA
MATCH (movie:Movie)
Return movie.title AS title
    UNION
USE graphB
MATCH (move:Movie)
RETURN movie.title AS title
```



THE SECOND GENERATION (DISTRIBUTED) GRAPH DATABASE: TITAN AND ITS SUCCESSOR JANUSGRAPH

In 2011, Aurelius was founded to develop an open-source distributed graph database called Titan [14]. By the first official release of Titan in 2015, the backend of Titan can support many major distributed storage architectures (e.g. Cassandra, HBase, Elasticsearch, BerkeleyDB) and can reuse many of the conveniences of the Hadoop ecosystem, with Gremlin as a unified query language on the frontend. It is easy for programmers to use, develop and participate in the community. Large-scale graphs could be sharded and stored on HBase or Cassandra (which were relatively mature distributed storage solutions at the time), and the Gremlin language was relatively full-featured though slightly lengthy. The whole solution was competitive at that time (2011-2015).

The following picture shows the growth of Titan and Neo4j stars on Github.com from 2012 to 2015.

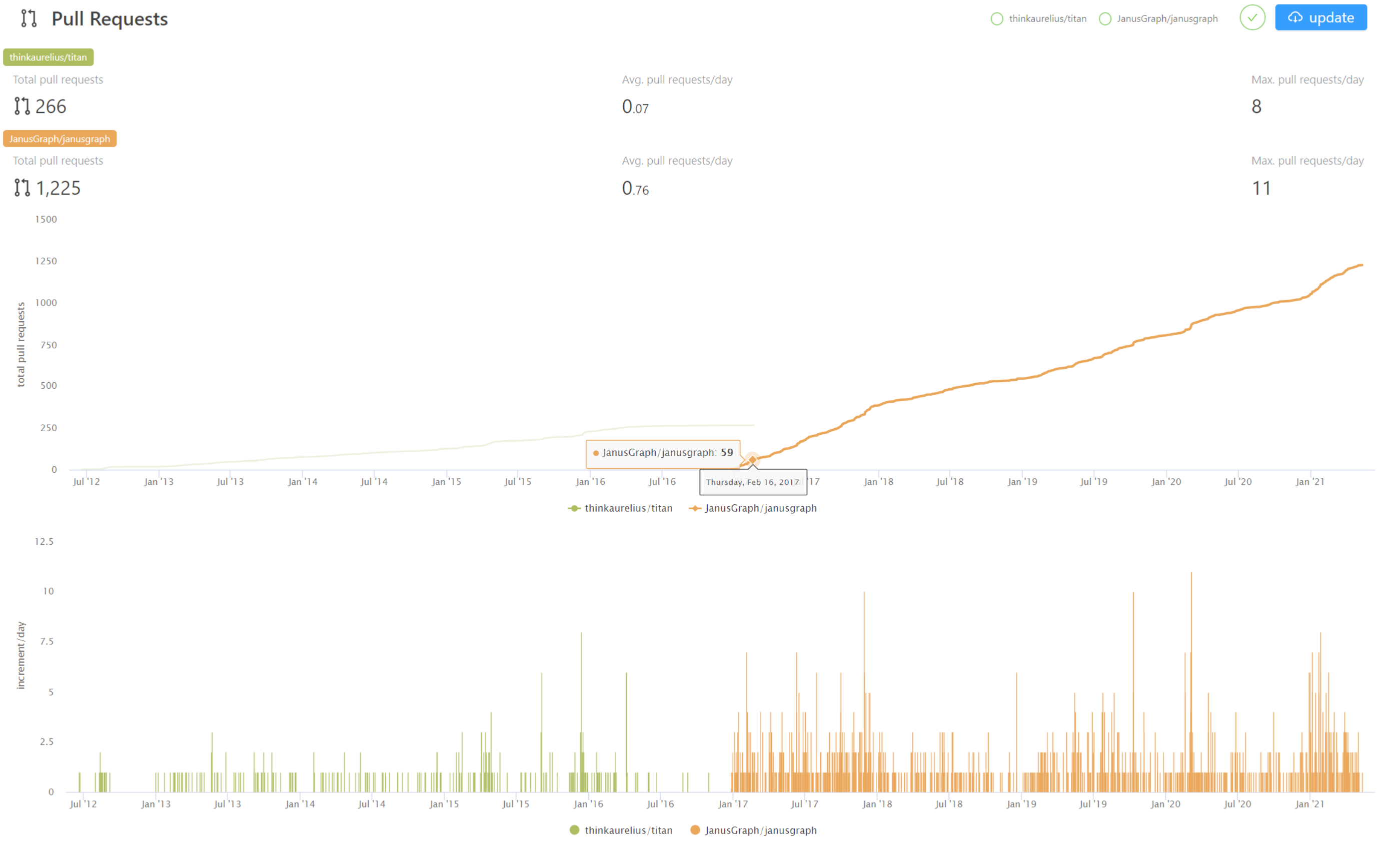


After Aurelius (Titan) was acquired by DataStax in 2015, Titan was gradually transformed into a closed-source commercial product(DataStax Enterprise Graph).

After the acquisition of Aurelius(Titan), there has been a strong demand for an open-source distributed graph database, and there were not many mature and active products in the market. In the era of big data, data is still being generated in a steady stream, far faster than Moore's Law. The Linux Foundation, along with some technology giants (Expero, Google, GRAKN.AI, Hortonworks, IBM, and Amazon) replicated and forked the original Titan project and started it as a new project JanusGraph[15]. Most of the community work including development, testing, release, and promotion, has been gradually shifted to the new JanusGraph.

The following graph shows the evolution of daily code commits (pull requests) for the two projects, and we can see:

1. Although Aurelius(Titan) still has some activity in its open-source code after its acquisition in 2015, the growth rate has slowed down significantly. This reflects the strength of the community.
2. After the new project was started in January 2017, its community became active quickly, surpassing the number of pull requests accumulated by Titan in the past 5 years in just one year. At the same time, the open-source Titan came to a halt.



Pull Requests

○ thinkaurelius/titan ○ JanusGraph/janusgraph ✓ update

thinkaurelius/titan

Total pull requests: 266 | Avg. pull requests/day: 0.07 | Max. pull requests/day: 8

JanusGraph/janusgraph

Total pull requests: 1,225 | Avg. pull requests/day: 0.76 | Max. pull requests/day: 11

FAMOUS PRODUCTS OF THE SAME PERIOD ORIENTDB, TIGERGRAPH, ARANGODB, AND DGRAPH

In addition to JanusGraph managed by the Linux Foundation, more vendors have been joined the overall market. Some distributed graph databases that were developed by commercial companies use different data models and access methods.

The following table only lists the main differences.

Vendors	Creation time	Core product	Open source protocol	Data model	Query language
OrientDB LTD (Acquired by SAP in 2017)	2011	OrientDB	Open source	Document + KV + Graph	OrientDB SQL (SQL-based extended graph abilities)
GraphSQL (was renamed TigerGraph)	2012	TigerGraph	Commercial version	Graph (Analysis)	GraphSQL (similar to SQL)
ArangoDB GmbH	2014	ArangoDB	Apache License 2.0	Document + KV + Graph	AQL (Simultaneous operation of documents, KVs and graphs)
DGraph Labs	2016	DGraph	Apache Public License 2.0 + Dgraph Community License	Originally RDF, later changed to GraphQL	GraphQL+-

TRADITIONAL GIANTS MICROSOFT, AMAZON, AND ORACLE

In addition to vendors focused on graph products, traditional giants have also entered the graph database field.

Microsoft Azure Cosmos DB[16] is a multimodal database cloud service on the Microsoft cloud that provides SQL, document, graph, key-value, and other capabilities. Amazon AWS Neptune[17] is a graph database cloud service provided by AWS support property graphs and RDF two data models. Oracle Graph[18] is a product of the relational database giant Oracle in the direction of graph technology and graph databases.

NEBULAGRAPH, A NEW GENERATION OF OPEN-SOURCE DISTRIBUTED GRAPH DATABASES

In the following topics, we will formally introduce NebulaGraph, a new generation of open-source distributed graph databases.

[1] https://db-engines.com/en/ranking_categories ↩

[2] https://www.yellowfinbi.com/blog/2014/06/yfcommunitynews-big-data-analytics-the-need-for-pragmatism-tangible-benefits-and-real-world-case-165305 ↩

[3] https://www.gartner.com/smarterwithgartner/gartner-top-10-data-and-analytics-trends-for-2021/ ↩

[4] https://www.verifiedmarketresearch.com/product/graph-database-market/ ↩

[5] https://www.globenewswire.com/news-release/2021/01/28/2165742/0/en/Global-Graph-Database-Market-Size-Share-to-Exceed-USD-4-500-Million-By-2026-Facts-Factors.html ↩

[6] https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html ↩

[7] https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the ↩

[8] https://www.amazon.com/Designing-Data-Intensive-Applications-Reliable-Maintainable/dp/1449373321 ↩

[9] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. In Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data, pages 323–330. ACM Press, May 1987. ↩

[10] "An overview of the recent history of Graph Query Languages". Authors: Tobias Lindaaker, U.S. National Expert.Date: 2018-05-14 ↩

[11] Gremlin is a graph language developed based on Apache TinkerPop. ↩

[12] https://docs.tigergraph.com/dev/gsql-ref ↩

[13] https://neo4j.com/fosdem20/ ↩

[14] https://github.com/thinkaurelius/titan ↩

[15] https://github.com/JanusGraph/janusgraph ↩

[16] https://azure.microsoft.com/en-us/free/cosmos-db/ ↩

[17] https://aws.amazon.com/cn/neptune/ ↩

[18] https://www.oracle.com/database/graph/ ↩

Last update: August 11, 2022

2.3 Related technologies

This topic introduces databases and graph-related technologies that are closely related to distributed graph databases.

2.3.1 Databases

Relational databases

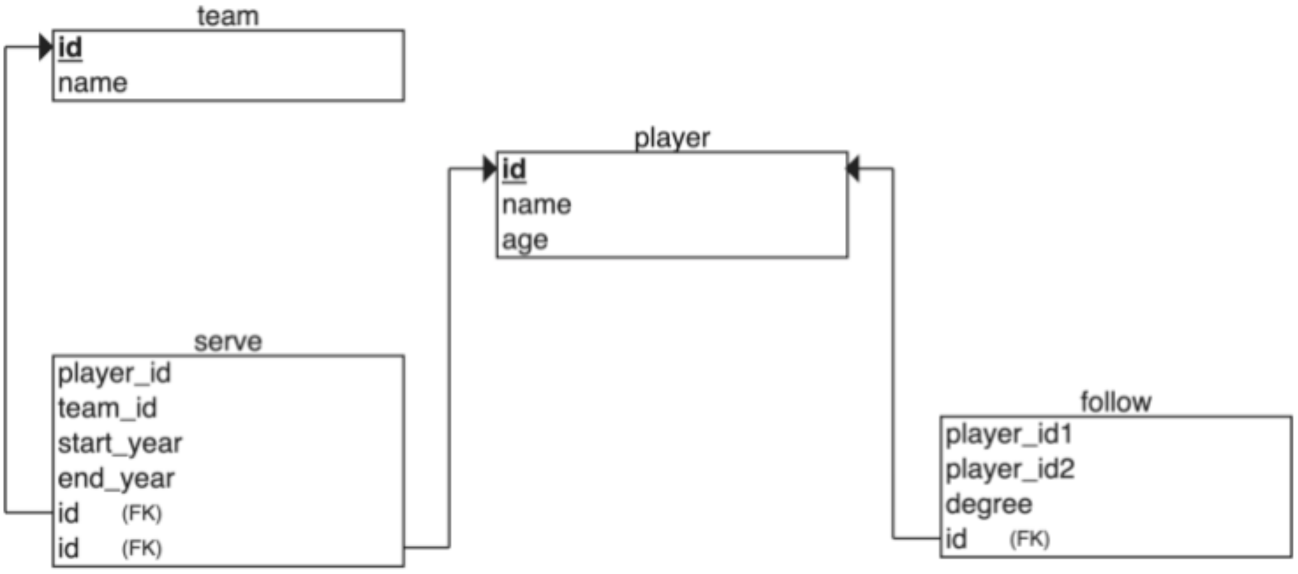
A relational database is a database that uses a relational model to organize data. The relational model is a two-dimensional table model, and a relational database consists of two-dimensional tables and the relationships between them. When it comes to relational databases, most people think of MySQL, one of the most popular database management systems that support database operations using the most common structured query language (SQL) and stores data in the form of tables, rows, and columns. This approach to storing data is derived from the relational data model proposed by Edgar Frank Codd in 1970.

In a relational database, a table can be created for each type of data to be stored. For example, the player table is used to store all player information, the team table is used to store team information. Each row of data in a SQL table must contain a primary key. The primary key is a unique identifier for the row of data. Generally, the primary key is self-incrementing with the number of rows as the field ID. Relational databases have served the computer industry very well since their inception and will continue to do so for a long time to come.

If you have used Excel, WPS, or other similar applications, you have a rough idea of how relational databases work. First, you set up the columns, then you add rows of data under the corresponding columns. You can average or otherwise aggregate the data in a column, similar to averaging in a relational database MySQL. Pivot tables in Excel are the equivalent of querying data in a relational database MySQL using aggregation functions and CASE statements. An Excel file can have multiple tables, and a single table is equivalent to a single table in MySQL. An Excel file is similar to a MySQL database.

RELATIONSHIPS IN RELATIONAL DATABASES

Unlike graph databases, edges in relational databases (or SQL-type databases) are also stored as entities in specialized edge tables. Two tables are created, player and team, and then player_team is created as an edge table. Edge tables are usually formed by joining related tables. For example, here the edge table player_team is made by joining the player table and the team table.



The way of storing edges is not a big problem when associating small data sets, but problems arise when there are too many relationships in a relational database. Specifically, when you want to query just one player's teammates, you have to join all the data in the table and then filter out all the data you don't need, which puts a huge strain on the relational database when your dataset reaches a certain size. If you want to associate multiple different tables, the system may not be able to respond before the join bombs.

ORIGINS OF RELATIONAL DATABASES

As mentioned above, the relational data model was first proposed by Edgar Frank Codd, an IBM engineer, in 1970. Codd wrote several papers on database management systems that addressed the potential of the relational data model. The relational data model does not rely on linked lists of data (mesh or hierarchical data), but more on data sets. Using the mathematical method of tuple calculus, he argued that these datasets can perform the same tasks as a navigational database. The only requirement was that the relational data model needed a suitable query language to guarantee the consistency requirements of the database. This became the inspiration for declarative query languages such as Structured Query Language (SQL). IBM's System R was one of the first implementations of such a system. But Software Development Laboratories, a small company founded by ex-IBM people and one illustrious Mr.Larry Ellison, beat IBM to the market with the product that would become known as Oracle.

Since the relational database was a trendy term at the time, many database vendors preferred to use it in their product names, even though their products were not actually relational. To prevent this and reduce the misuse of the relational data model, Codd introduced the famous Codd's 12 Rules. All relational data systems must follow Codd's 12 Rules.

NoSQL databases

Graph databases are not the only alternative that can overcome the shortcomings of relational databases. There are many non-relational database products on the market that can be called NoSQL. The term NoSQL was first introduced in the late 1990s and can be interpreted as "not SQL" or "not only SQL". For the sake of understanding, NoSQL can be interpreted as a "non-relational database" here. Unlike relational databases, the data storage and retrieval mechanisms provided by NoSQL databases are not modeled based on table relationships. NoSQL databases can be divided into four categories.

- Key-value Data Store
- Columnar Store
- Document Store
- Graph Store

The following describes the four types of NoSQL databases.

KEY-VALUE DATA STORE

Key-value databases store data in unique key-value pairs. Unlike relational databases, key-value stores do not have tables and columns. A key-value database itself is like a large table with many columns (i.e., keys). In a key-value store database, data are stored and queried by means of keys, usually implemented as hash lists. This is much simpler than traditional SQL databases, and for some web applications, it is sufficient.

The advantage of the key-value model for IT systems is that it is simple and easy to deploy. In most cases, this type of storage works well for unrelated data. If you are just storing data without querying it, there is no problem using this storage method. However, if the DBA only queries or updates some of the values, the key-value model becomes inefficient. Common key-value storage databases include Redis, Voldemort, and Oracle BDB.

COLUMNAR STORE

A NoSQL database's columnar store has many similarities to a NoSQL database's key-value store because the columnar store is still using keys for storage and retrieval. The difference is that in a columnar store database, the column is the smallest storage unit, and each column consists of a key, a value, and a timestamp for version control and conflict resolution. This is particularly useful when scaling in a distributed manner, as timestamps can be used to locate expired data when the database is updated. Because of the good scalability of columnar storage, the columnar store is suitable for very large data sets. Common columnar storage databases include HBase, Cassandra, HadoopDB, etc.

DOCUMENT STORE

A NoSQL database document store is a key-value-based database, but with enhanced functionality. Data is still stored as keys, but the values in a document store are structured documents, not just a string or a single value. That is, because of the increased information structure, document stores are able to perform more optimized queries and make data retrieval easier. Therefore, document stores are particularly well suited for storing, indexing, and managing document-oriented data or similar semi-structured data.

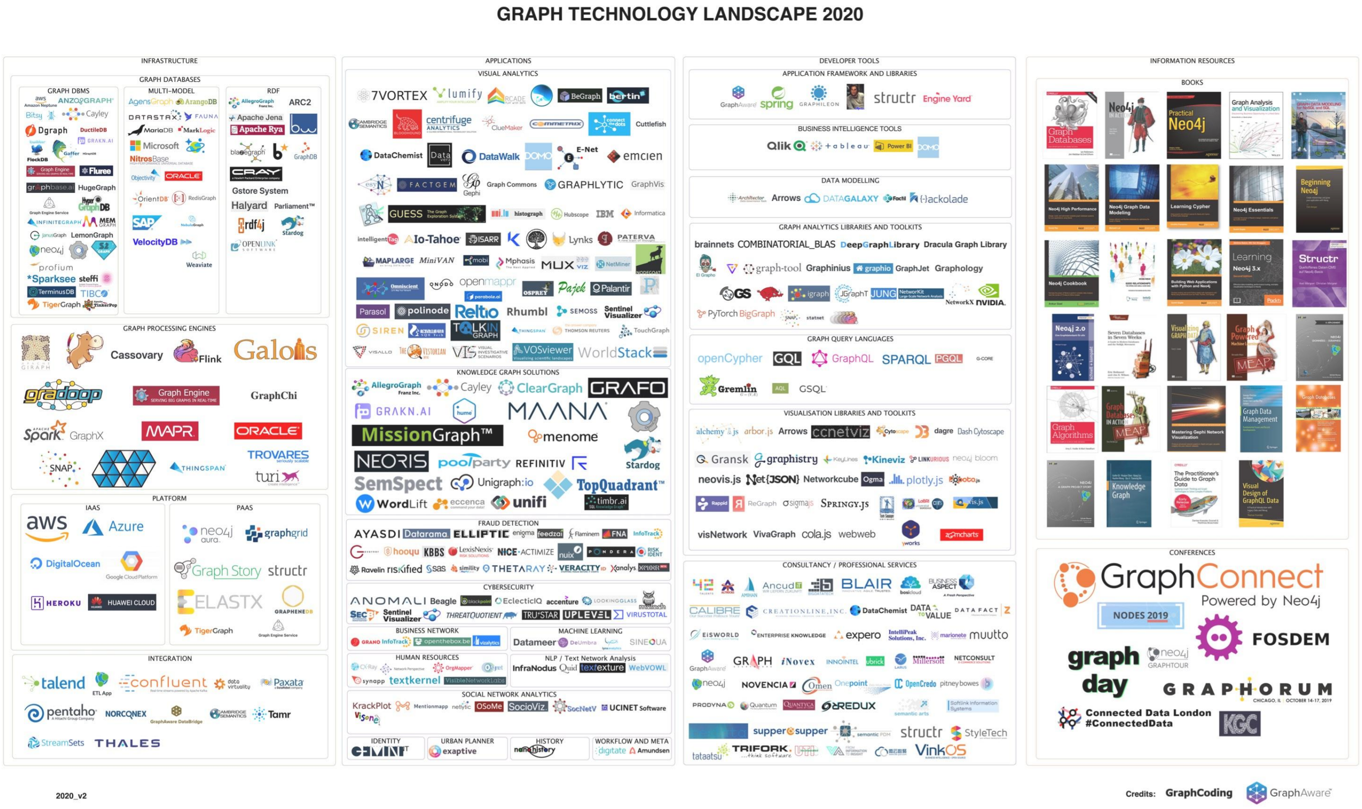
Technically speaking, as a semi-structured unit of information, a document in a document store can be any form of document available, including XML, JSON, YAML, etc., depending on the design of the database vendor. For example, JSON is a common choice. While JSON is not the best choice for structured data, JSON-type data can be used in both front-end and back-end applications. Common document storage databases include MongoDB, CouchDB, Terrastore, etc.

GRAPH STORE

The last class of NoSQL databases is graph databases. NebulaGraph, is also a graph database. Although graph databases are also NoSQL databases, graph databases are fundamentally different from the above-mentioned NoSQL databases. Graph databases store data in the form of vertices, edges, and properties. Its advantages include high flexibility, support for complex graph algorithms, and can be used to build complex relational graphs. We will discuss graph databases in detail in the subsequent topics. But in this topic, you just need to know that a graph database is a NoSQL type of database. Common graph databases include NebulaGraph, Neo4j, OrientDB, etc.

2.3.2 Graph-related technologies

Take a look at a panoramic view of graph technology in 2020 [1].



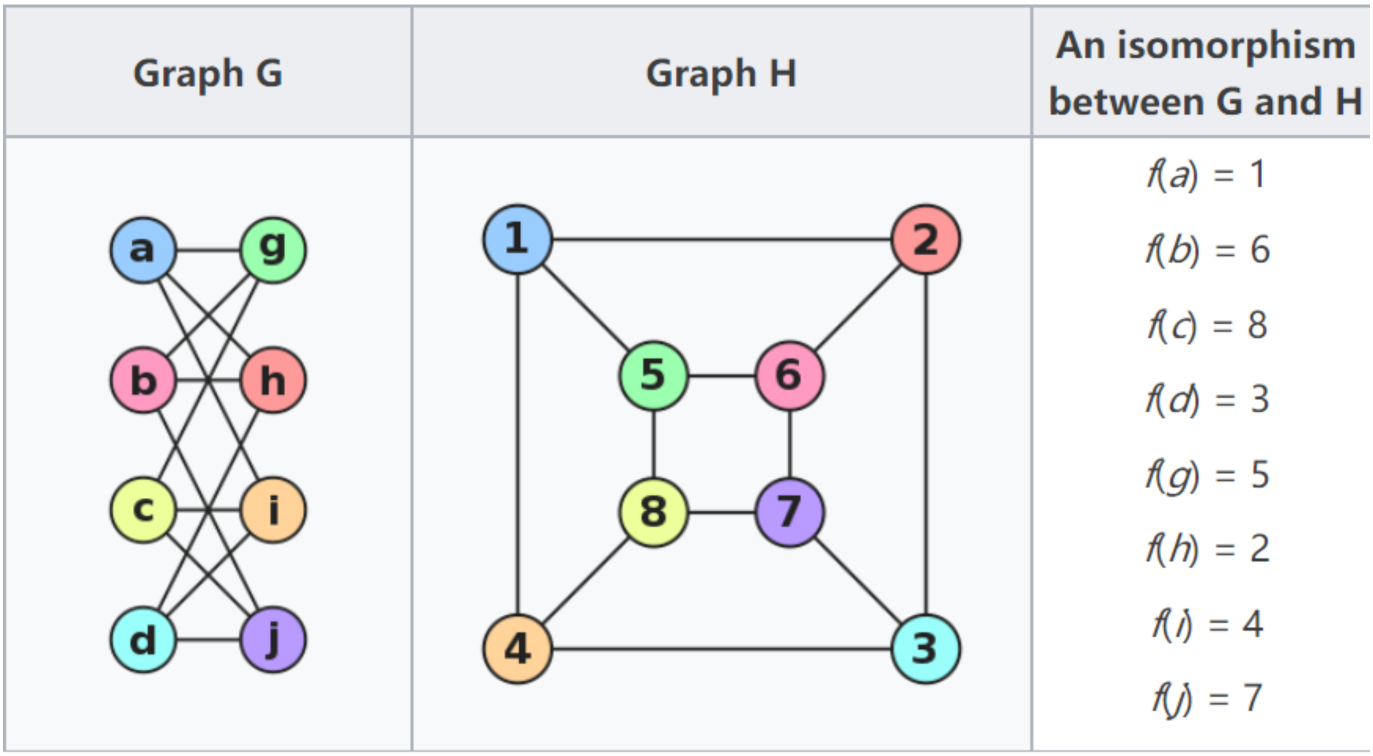
There are many technologies that are related to graphs, which can be broadly classified into these categories:

- Infrastructure: Graph databases, graph computing (processing) engines, graph deep learning, cloud services, etc.
- Applications: Visualization, knowledge graph, anti-fraud, cyber security, social network, etc.
- Development tools: Graph query languages, modeling tools, development frameworks, and libraries.
- E-books and conferences, etc.

Graph language

In the previous topic, we introduced the history of graph languages. In this section, we make a classification of the functions of graph languages.

- Nearest neighbor query (NNS): Query the neighboring edges, neighbors, or K-hops neighbors.
- Find one/all subgraphs that satisfy a given graph pattern. This problem is very close to Subgraph Isomorphism - two seemingly different graphs that are actually identical [2] as shown below.

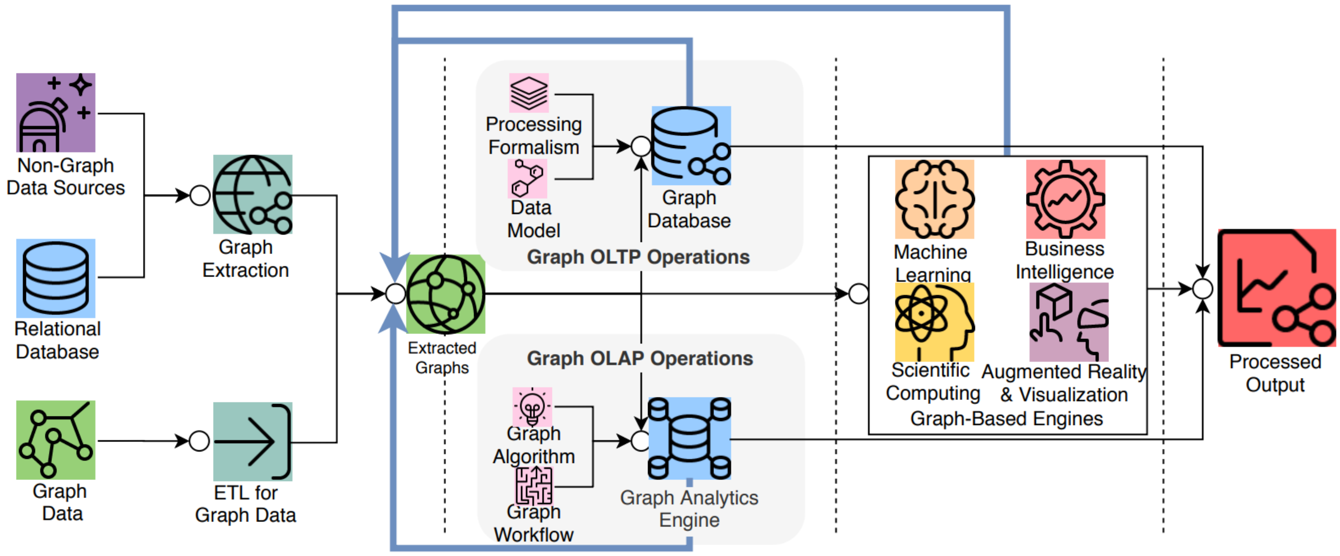


Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

- Reachability (connectivity) problems: The most common reachability problem is the shortest path problem. Such problems are usually described in terms of Regular Path Query - a series of connected groups of vertices forming a path that needs to satisfy some regular expression.
- Analytic problems: It is related to some convergent operators, such as Average, Count, Max, Vertex Degree. Measures the distance between all two vertices, the degree of interaction between a vertex and other vertices.

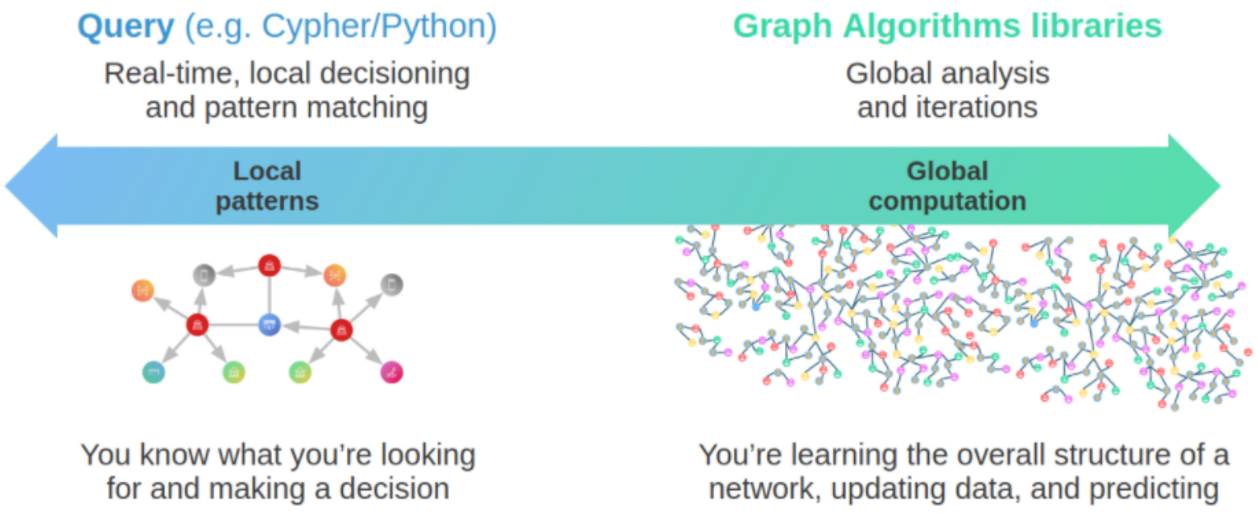
Graph database and graph processing systems

A graph system usually includes a complex data pipeline [3]. From the data source (the left side of the picture below) to the processing output (the right side), multiple data processing steps and systems are used, such as the ETL module, Graph OLTP module, OLAP module, BI, and knowledge graph.



Graph databases and graph processing systems have different origins and specialties (and weaknesses).

- (Online) The graph database is designed for persistent storage management of graphs and efficient subgraph operations. Hard disks and network are the target operating devices, physical/logical data mapping, data integrity, and (fault) consistency are the main goals. Each request typically involves only a small part of the full graph and can usually be done on a single server. Request latency is usually in milliseconds or seconds, and request concurrency is typically in the thousands or hundreds of thousands. The early Neo4j was one of the origins of the graph database space.
- (Offline) The graph processing system is for high-volume, concurrency, iteration, processing, and analysis of the full graph. Memory and network are the target operating devices. Each request involves all graph vertices and requires all servers to be involved in its completion. The latency of a single request is in the range of minutes to hours (days). The request concurrency is in single digits. Google's Pregel [4] represents the typical origin of graph processing systems. Its point-centric programming abstraction and BSP's operational model constitute a programming paradigm that is a more graph-friendly API abstraction than the previous Hadoop Map-Reduce.


Query (e.g. Cypher/Python)
Real-time, local decisioning
and pattern matching
Graph Algorithms libraries
Global analysis
and iterations
Local
patterns
Global
computation
You know what you're looking
for and making a decision
You're learning the overall structure of a
network, updating data, and predicting

[5]

Graph sharding methods

For large-scale graph data, it is difficult to store it in the memory of a single server, and even just storing the graph structure itself is not enough. By increasing the capacity of a single server, its cost price usually rises exponentially.

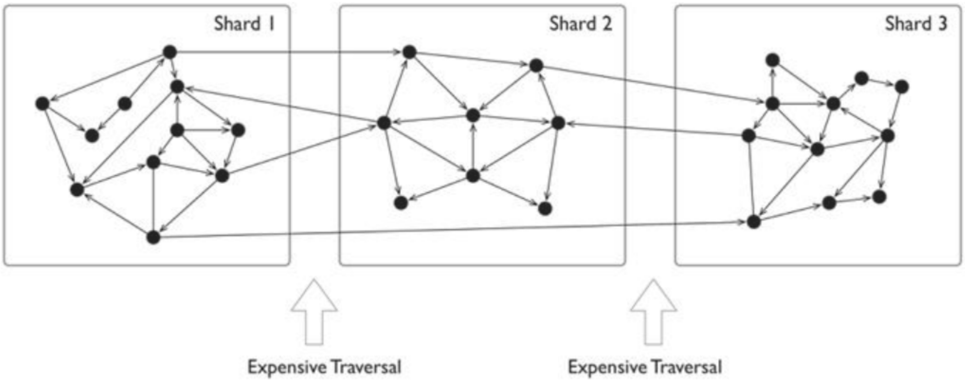
As the volume of data increases, for example, 100 billion data already exceeds the capacity of all commercially available servers on the market.

Another option is to shard data and place each shard on a different server to increase reliability and performance. For NoSQL systems, such as key-value or document systems, the sharding method is intuitive and natural. Each record and data unit can usually be placed on a different server based on the key or docID.

However, the sharding of data structures like graphs is usually less intuitive, because usually, graphs are "fully connected" and each vertex can be connected to any other vertex in usually 6 hops.

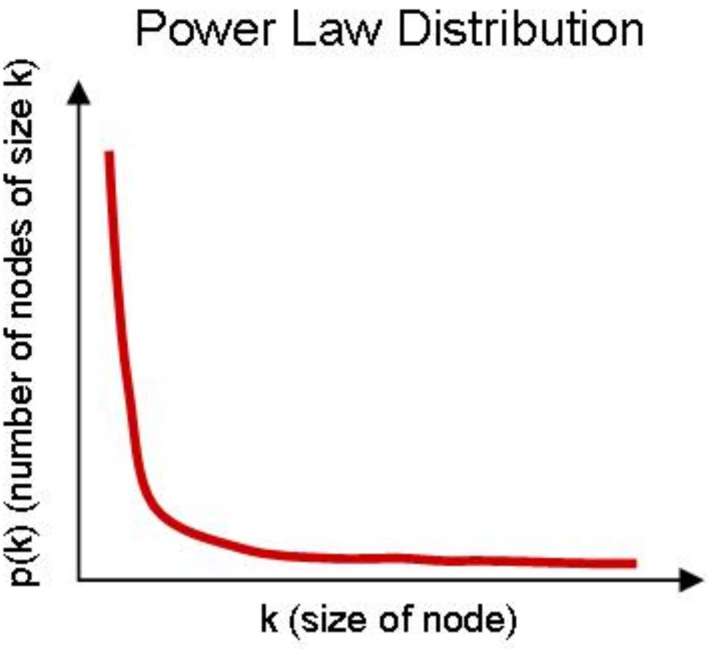
And it has been theoretically proven that the graph sharding problem is NP.

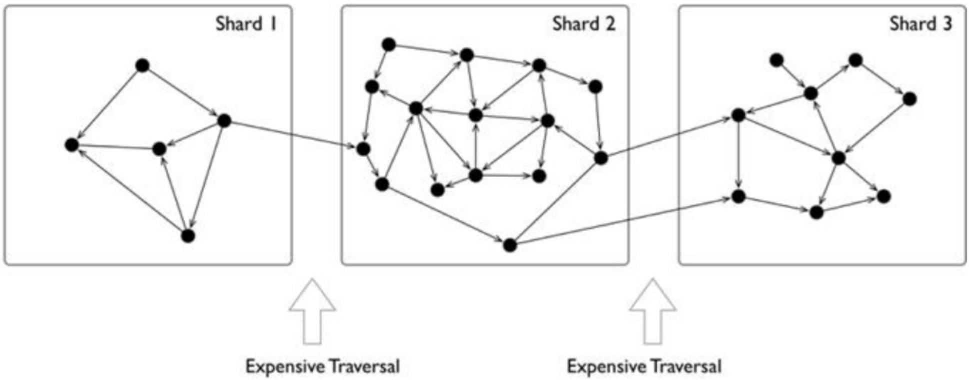
When distributing the entire graph data across multiple servers, the cross-server network access latency is 10 times higher than the hardware (memory) access time inside the same server. Therefore, for some depth-first traversal scenarios, a large number of cross-network accesses occur, resulting in extremely high overall latency.



[6]

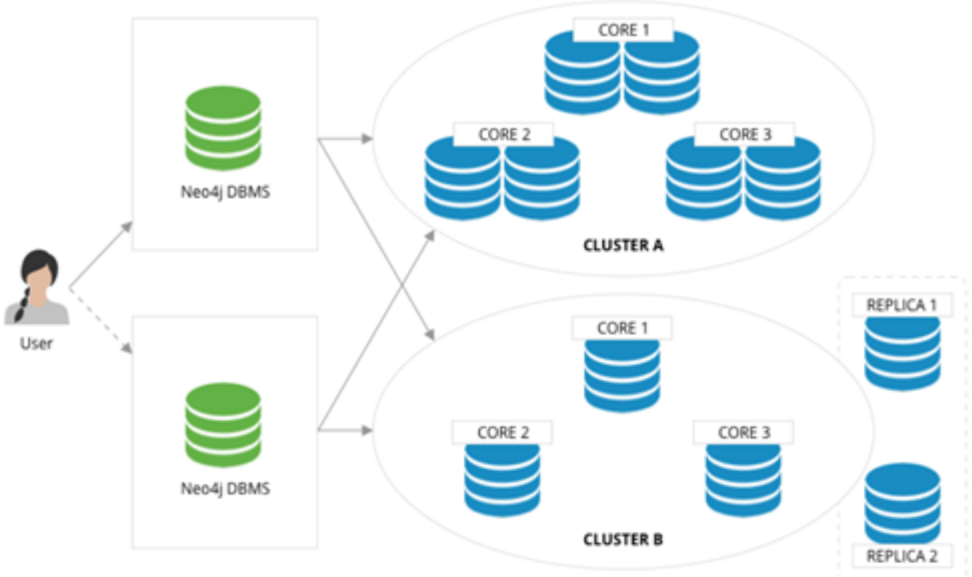
Usually, graphs have a clear power-law distribution. A small number of vertices have much denser neighboring edges than the average vertices. Though processing these vertices can usually be within the same server which reduces cross-network access, load will be far more heavier than the average.





The common graph sharding methods are as follows:

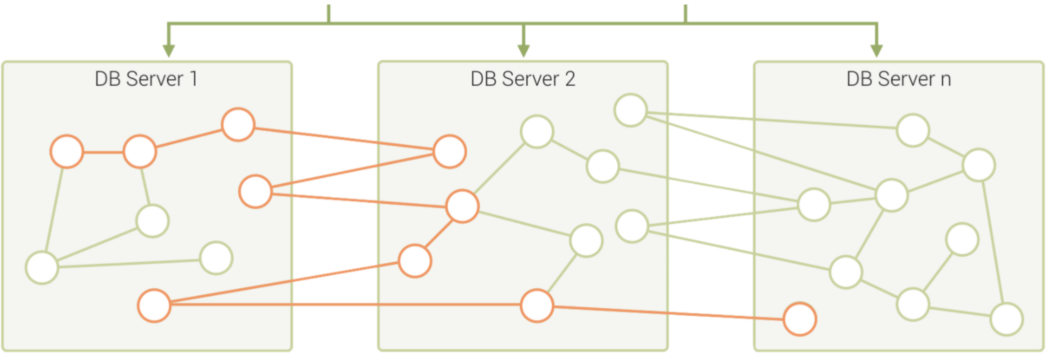
- Application-level sharding: The application layer senses and controls which shard each vertex and edge should locate on based on the type of vertices and edges. A set of vertices of the same type is placed on one sharding and another set of vertices of the same type is placed on another sharding. Of course, for high reliability, the sharding itself can also be made multiple replicas. When used by the application, the desired vertices and edges are fetched from each shard, and then on the off-application side (or some proxy server-side), the fetched data is assembled into the final result. This is typically represented by the Neo4j 4. x Fabric.

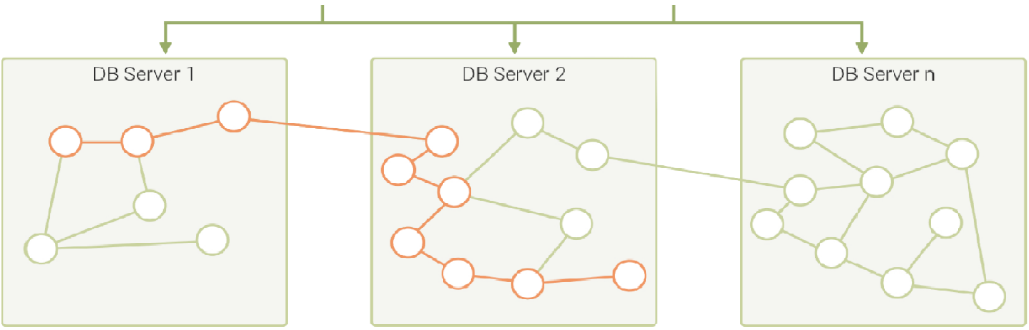


- Using a distributed cache layer: Add a memory cache layer on the top of the hard disk and cache important portions of the sharding and data and preheat that cache.

- Adding read-only replicas or views: Add read-only replicas or create a view for some of the graph sharding, and pass the heavier load of read requests through these sharding servers.

- Performing fine-grained graph sharding: Form multiple small partitions of vertices and edges instead of one large sharding, and then place the more correlated partitions on the same server as much as possible. [7].





A mixture of these approaches is also used in specific engineering practices. Usually, offline graph processing systems perform some degree of graph preprocessing to improve the locality through an ETL process, while online graph database systems usually choose a periodic data rebalancing process to improve data locality.

Technical challenges

In the literature [8], a thorough investigation of graphs and challenges is done, and the following lists the top ten graph technology challenges.

- Scalability: Loading and upgrading big graphs, performing graph computation and graph traversal, use of triggers and supernodes
- Visualization: Customizable layouts, rendering and display big images, and display dynamic and updated display
- Query language and programming API: Language expressiveness, standards compatibility, compatibility with existing systems, design of subqueries, and associative queries across multiple graphs
- Faster graph algorithms
- Easy to use (configuration and usage)
- Performance metrics and testing
- General graph technology software (e.g., to handle offline, online, streaming computations.)
- ETL
- Debug and test

Open-source graph tools on single machines

There is a common misconception about graph databases that any data access involving graph structure needs to be stored in a graph database.

When the amount of data is not large, single machine memory is enough to store the data. You can use some single-machine open-source tools to store tens of millions of vertices and edges.

- JGraphT[9]: A well-known open-source Java graph theory library, which implements a considerable number of efficient graph algorithms.
- igraph[10]: A lightweight and powerful library, supporting R, Python, and C++.
- NetworkX[11]: The first choice for data scientists doing graph theory analysis.
- Cytoscape[12]: A powerful visual open-source graph analysis tool.
- Gephi[13]: A powerful visual open-source graph analysis tool.
- arrows.app[14]: A simple brain mapping tool for visually generating Cypher statements.

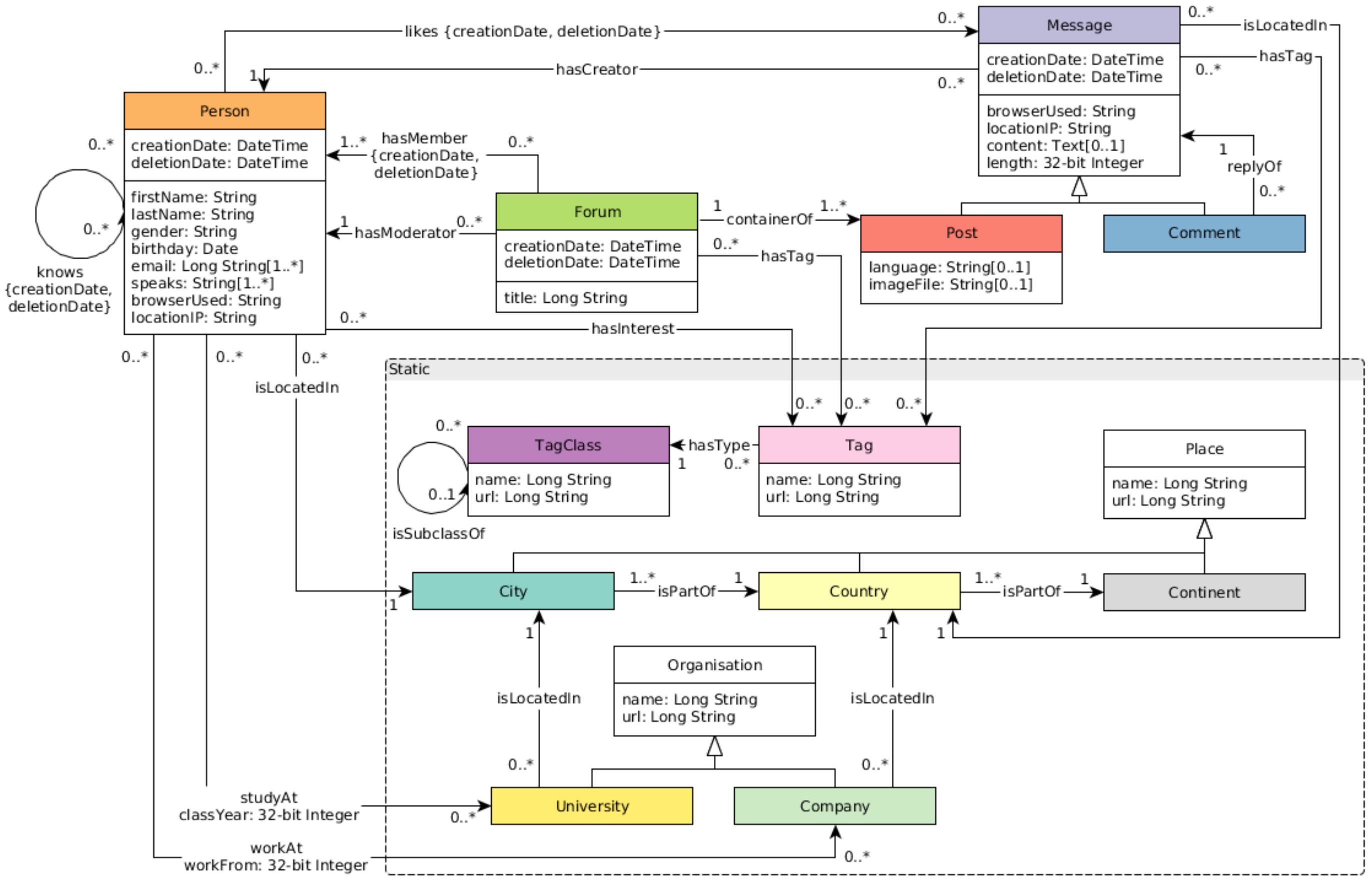
Industry databases and benchmarks

LDBC

LDBC[15] (Linked Data Benchmark Council) is a non-profit organization composed of hardware and software giants such as Oracle, Intel and mainstream graph database vendors such as Neo4j and TigerGraph, which is the benchmark guide developer and test result publisher for graphs and has a high influence in the industry.

SNB (Social Network Benchmark) is one of the benchmarks developed by the Linked Data Benchmark Committee (LDBC) for graph databases and is divided into two scenarios: interactive query (Interactive) and business intelligence (BI). Its role is similar to that of TPC-C, TPC-H, and other tests in SQL-type databases, which can help users compare the functions, performance, and capacity of various graph database products.

An SNB dataset simulates the relationship between people and posts of a social network, taking into account the distribution properties of the social network, the activity of people, and other social information.

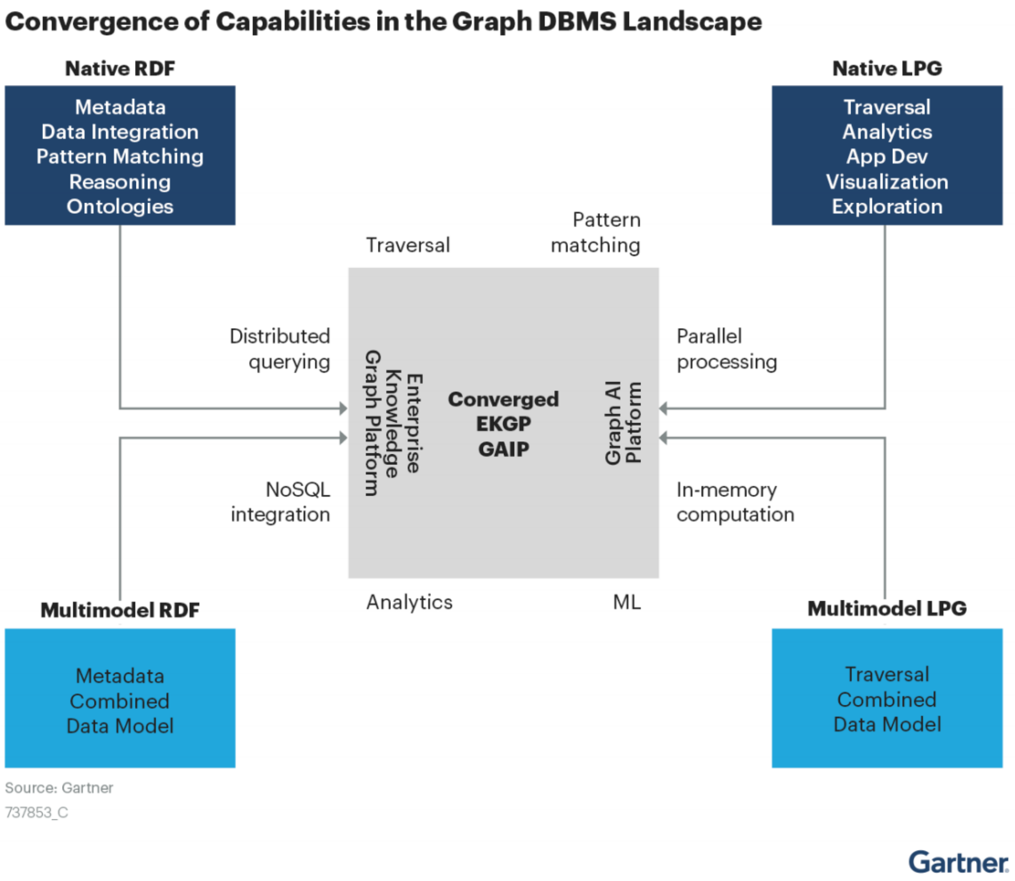


The standard data size ranges from 0.1 GB (scale factor 0.1) to 1000 GB (sf 1000). Larger data sets of 10 TB and 100 TB can also be generated. The number of vertices and edges is as shown below.

Scale Factor	0.1	0.3	1	3	10	30	100	300	1000
# of Persons	1.5K	3.5K	11K	27K	73K	182K	499K	1.25M	3.6M
# of nodes	327.6K	908K	3.2M	9.3M	30M	88.8M	282.6M	817.3M	2.7B
# of edges	1.5M	4.6M	17.3M	52.7M	176.6M	540.9M	1.8B	5.3B	17B

2.3.3 Trends

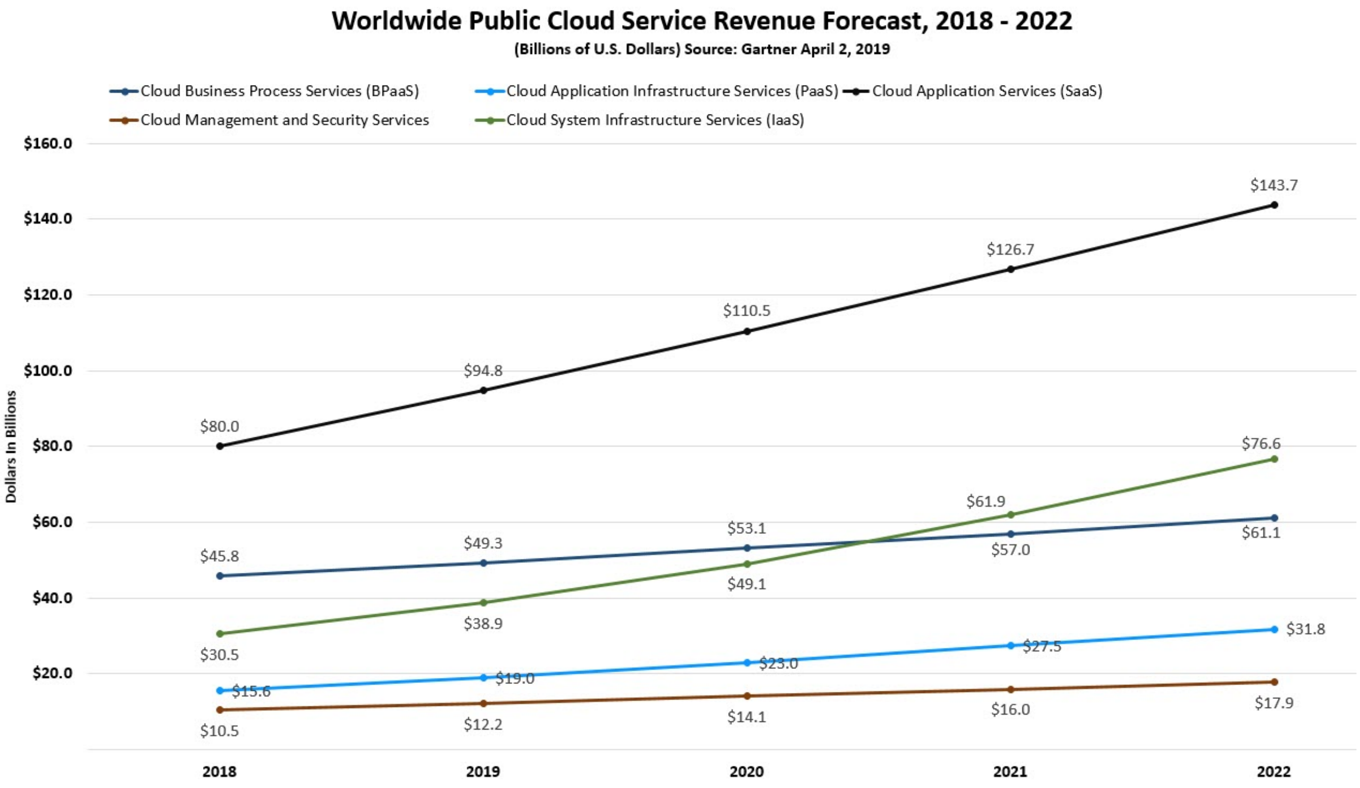
Graph technologies of different origins and goals are learning from and integrating with each other



Source: Gartner
737853_C

The trends in cloud computing place higher demands on scalability.

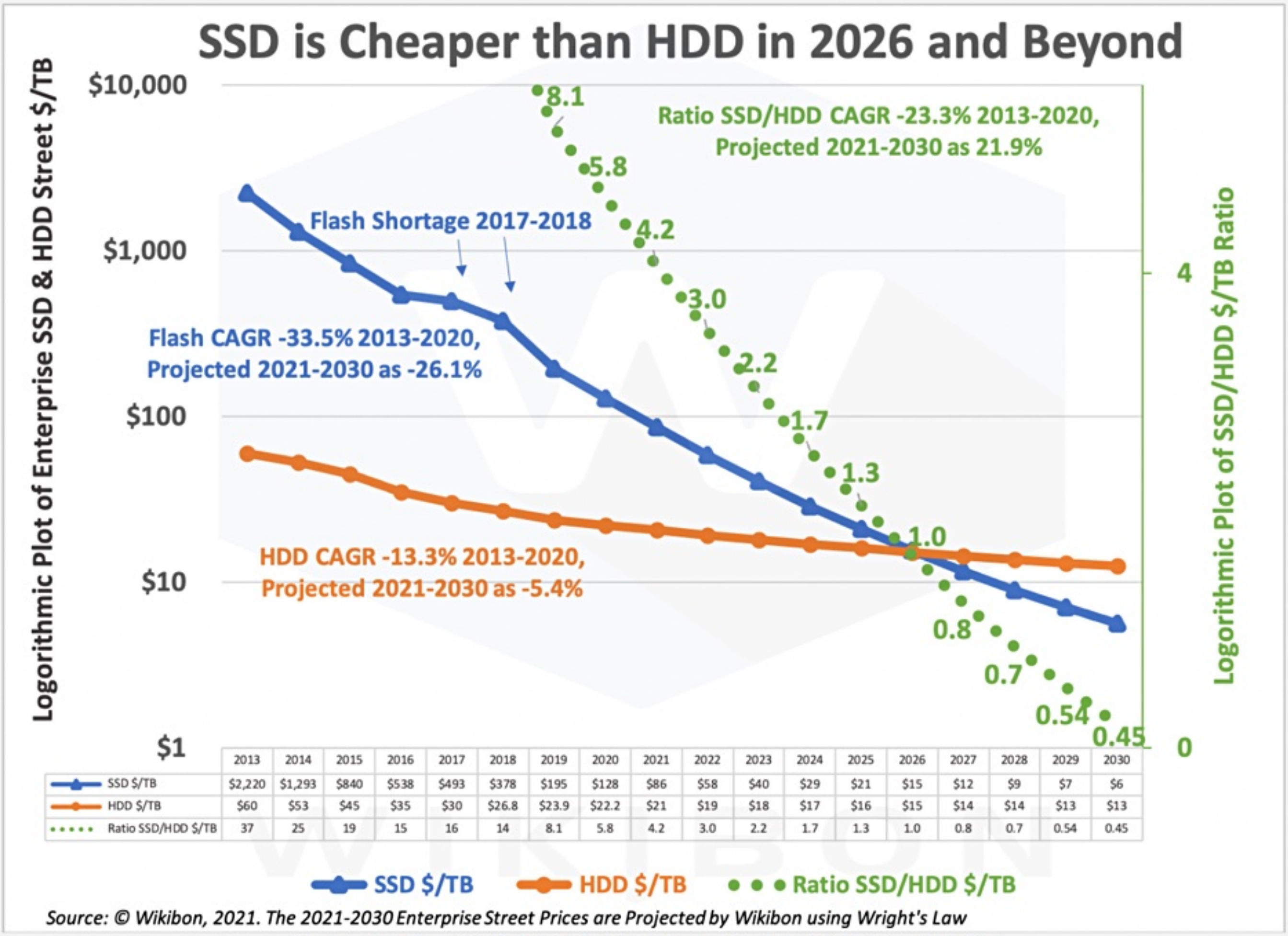
According to Gartner's projections, cloud services have been growing at a rapid rate and penetration [16]. A large number of commercial software is gradually moving from a completely local and private model 10 years ago to a cloud services-based business model. One of the major advantages of cloud services is that they offer near-infinite scalability. It requires that various cloud infrastructure-based software must have a better ability to scale quickly and elastically.



Trends in hardware that SSD will be the mainstream persistent device

Hardware determines software architecture. From the 1950s, when Moore's Law was discovered, to the 00s, when multi-core was introduced, hardware trends and speeds have profoundly determined software architecture. Database systems are mostly designed around "hard disk + memory", high-performance computing systems are mostly designed around "memory + CPU", and distributed systems are designed completely differently for 1 gigabit, 10 gigabits, and RDMA.

Graph traversals are featured as random access. Early graph database systems adopted the large memory + HDD architecture. By designing some data structure in memory, random access can be achieved in memory (B+ trees, Hash tables) for the purpose of optimizing graph topology traversal. And then the random access was converted into sequential reads and writes suitable for HDDs. The entire software architecture (including the storage and compute layers) must be based on and built around such IO processes. With the decline in SSD prices [17], SSDs are replacing HDDs as the dominant device. Friendly random access, deep IO queue, fast access are the features of SSD that differ from HDD's highly repetitive sequence, random latency, and easily damaged disk. The redesign for all software architectures becomes a heavy historical technical burden.



	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030
SSD $/TB	$2,220	$1,293	$840	$538	$493	$378	$195	$128	$86	$58	$40	$29	$21	$15	$12	$9	$7	$6
HDD $/TB	$60	$53	$45	$35	$30	$26.8	$23.9	$22.2	$21	$19	$18	$17	$16	$15	$14	$14	$13	$13
Ratio SSD/HDD $/TB	37	25	19	15	16	14	8.1	5.8	4.2	3.0	2.2	1.7	1.3	1.0	0.8	0.7	0.54	0.45

Source: © Wikibon, 2021. The 2021-2030 Enterprise Street Prices are Projected by Wikibon using Wright's Law

Figure 4 - SSD/HDD Pricing Ratio 2013 - 2030

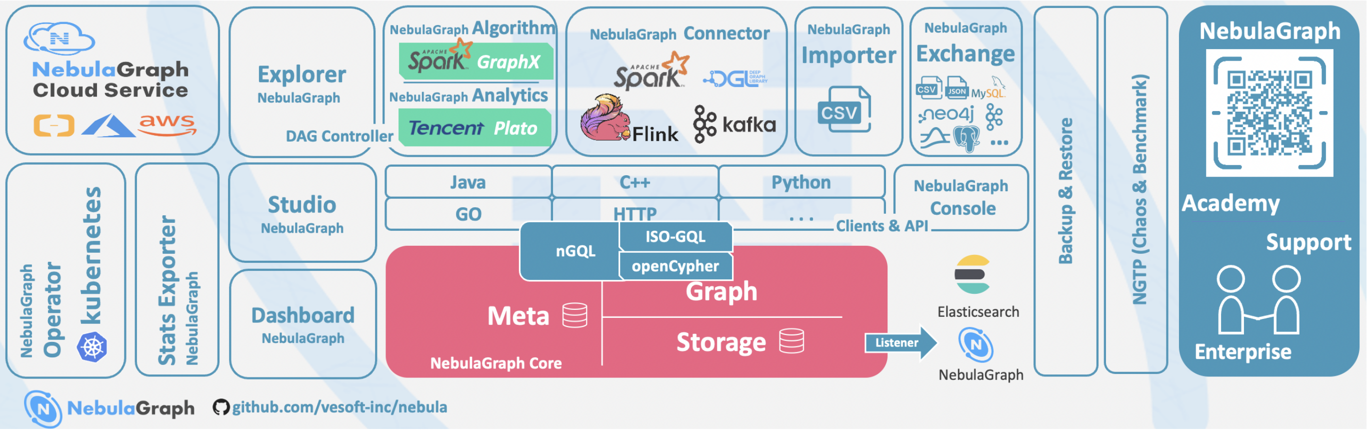
Source: © Wikibon, 2021.

1. https://graphaware.com/graphaware/2020/02/17/graph-technology-landscape-2020.html ↩
2. https://en.wikipedia.org/wiki/Graph_isomorphism ↩
3. The Future is Big Graphs! A Community View on Graph Processing Systems. https://arxiv.org/abs/2012.06171 ↩
4. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the International Conference on Management of data (SIGMOD), pages 135–146, New York, NY, USA, 2010. ACM ↩
5. https://neo4j.com/graphacademy/training-iga-40/02-iga-40-overview-of-graph-algorithms/ ↩
6. https://livebook.manning.com/book/graph-powered-machine-learning/welcome/v-8/ ↩
7. https://www.arangodb.com/learn/graphs/using-smartgraphs-arangodb/ ↩
8. https://arxiv.org/abs/1709.03188 ↩
9. https://jgrapht.org/ ↩
10. https://igraph.org/ ↩
11. https://networkx.org/ ↩
12. https://cytoscape.org/ ↩
13. https://gephi.org/ ↩
14. https://arrows.app/ ↩
15. https://github.com/ldbc/ldbc_snb_docs ↩
16. https://cloudcomputing-news.net/news/2019/apr/15/public-cloud-soaring-to-331b-by-2022-according-to-gartner/ ↩
17. https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/ ↩

Last update: August 21, 2023

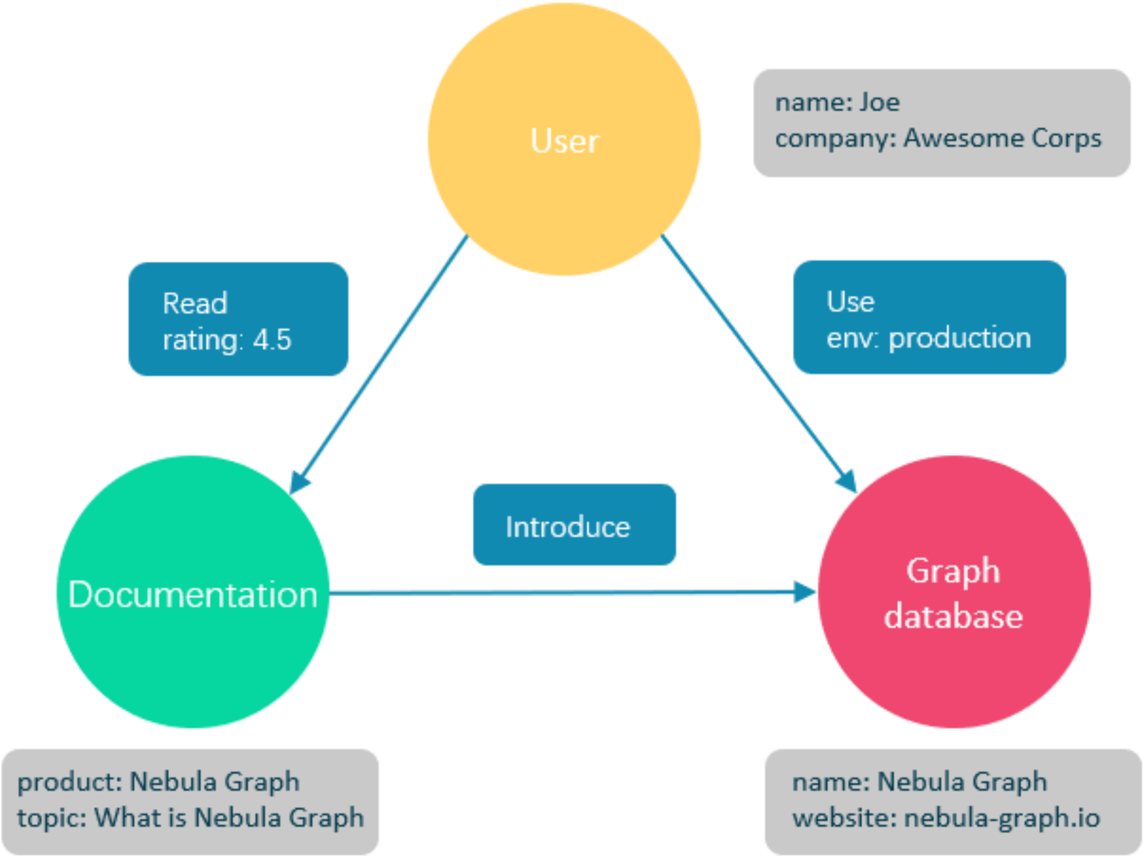
2.4 What is NebulaGraph

NebulaGraph is an open-source, distributed, easily scalable, and native graph database. It is capable of hosting graphs with hundreds of billions of vertices and trillions of edges, and serving queries with millisecond-latency.



2.4.1 What is a graph database

A graph database, such as NebulaGraph, is a database that specializes in storing vast graph networks and retrieving information from them. It efficiently stores data as vertices (nodes) and edges (relationships) in labeled property graphs. Properties can be attached to both vertices and edges. Each vertex can have one or multiple tags (labels).



Graph databases are well suited for storing most kinds of data models abstracted from reality. Things are connected in almost all fields in the world. Modeling systems like relational databases extract the relationships between entities and squeeze them into table columns alone, with their types and properties stored in other columns or even other tables. This makes data management time-consuming and cost-ineffective.

NebulaGraph, as a typical native graph database, allows you to store the rich relationships as edges with edge types and properties directly attached to them.

2.4.2 Advantages of NebulaGraph

Open source

NebulaGraph is open under the Apache 2.0 License. More and more people such as database developers, data scientists, security experts, and algorithm engineers are participating in the designing and development of NebulaGraph. To join the opening of source code and ideas, surf the NebulaGraph GitHub page.

Outstanding performance

Written in C++ and born for graphs, NebulaGraph handles graph queries in milliseconds. Among most databases, NebulaGraph shows superior performance in providing graph data services. The larger the data size, the greater the superiority of NebulaGraph.For more information, see NebulaGraph benchmarking.

High scalability

NebulaGraph is designed in a shared-nothing architecture and supports scaling in and out without interrupting the database service.

Developer friendly

NebulaGraph supports clients in popular programming languages like Java, Python, C++, and Go, and more are under development. For more information, see NebulaGraph clients.

Reliable access control

NebulaGraph supports strict role-based access control and external authentication servers such as LDAP (Lightweight Directory Access Protocol) servers to enhance data security. For more information, see Authentication and authorization.

Diversified ecosystem

More and more native tools of NebulaGraph have been released, such as NebulaGraph Studio, NebulaGraph Console, and NebulaGraph Exchange. For more ecosystem tools, see Ecosystem tools overview.

Besides, NebulaGraph has the ability to be integrated with many cutting-edge technologies, such as Spark, Flink, and HBase, for the purpose of mutual strengthening in a world of increasing challenges and chances.

OpenCypher-compatible query language

The native NebulaGraph Query Language, also known as nGQL, is a declarative, openCypher-compatible textual query language. It is easy to understand and easy to use. For more information, see nGQL guide.

Future-oriented hardware with balanced reading and writing

Solid-state drives have extremely high performance and they are getting cheaper. NebulaGraph is a product based on SSD. Compared with products based on HDD and large memory, it is more suitable for future hardware trends and easier to achieve balanced reading and writing.

Easy data modeling and high flexibility

You can easily model the connected data into NebulaGraph for your business without forcing them into a structure such as a relational table, and properties can be added, updated, and deleted freely. For more information, see Data modeling.

High popularity

NebulaGraph is being used by tech leaders such as Tencent, Vivo, Meituan, and JD Digits. For more information, visit the NebulaGraph official website.

2.4.3 Use cases

NebulaGraph can be used to support various graph-based scenarios. To spare the time spent on pushing the kinds of data mentioned in this section into relational databases and on bothering with join queries, use NebulaGraph.

Fraud detection

Financial institutions have to traverse countless transactions to piece together potential crimes and understand how combinations of transactions and devices might be related to a single fraud scheme. This kind of scenario can be modeled in graphs, and with the help of NebulaGraph, fraud rings and other sophisticated scams can be easily detected.

Real-time recommendation

NebulaGraph offers the ability to instantly process the real-time information produced by a visitor and make accurate recommendations on articles, videos, products, and services.

Intelligent question-answer system

Natural languages can be transformed into knowledge graphs and stored in NebulaGraph. A question organized in a natural language can be resolved by a semantic parser in an intelligent question-answer system and re-organized. Then, possible answers to the question can be retrieved from the knowledge graph and provided to the one who asked the question.

Social networking

Information on people and their relationships is typical graph data. NebulaGraph can easily handle the social networking information of billions of people and trillions of relationships, and provide lightning-fast queries for friend recommendations and job promotions in the case of massive concurrency.

2.4.4 Related links

- Official website
- Docs
- Blogs
- Forum
- GitHub

Last update: January 6, 2023

2.5 Data modeling

A data model is a model that organizes data and specifies how they are related to one another. This topic describes the Nebula Graph data model and provides suggestions for data modeling with NebulaGraph.

2.5.1 Data structures

NebulaGraph data model uses six data structures to store data. They are graph spaces, vertices, edges, tags, edge types and properties.

- **Graph spaces**: Graph spaces are used to isolate data from different teams or programs. Data stored in different graph spaces are securely isolated. Storage replications, privileges, and partitions can be assigned.

- **Vertices**: Vertices are used to store entities.

- In NebulaGraph, vertices are identified with vertex identifiers (i.e. `VID`). The `VID` must be unique in the same graph space. VID should be int64, or fixed_string(N).
- A vertex has zero to multiple tags.

> **Compatibility**
>
> In NebulaGraph 2.x a vertex must have at least one tag. And in NebulaGraph 3.5.0, a tag is not required for a vertex.

- **Edges**: Edges are used to connect vertices. An edge is a connection or behavior between two vertices.
- There can be multiple edges between two vertices.
- Edges are directed. `->` identifies the directions of edges. Edges can be traversed in either direction.
- An edge is identified uniquely with `<a source vertex, an edge type, a rank value, and a destination vertex>`. Edges have no EID.
- An edge must have one and only one edge type.
- The rank value is an immutable user-assigned 64-bit signed integer. It identifies the edges with the same edge type between two vertices. Edges are sorted by their rank values. The edge with the greatest rank value is listed first. The default rank value is zero.

- **Tags**: Tags are used to categorize vertices. Vertices that have the same tag share the same definition of properties.

- **Edge types**: Edge types are used to categorize edges. Edges that have the same edge type share the same definition of properties.

- **Properties**: Properties are key-value pairs. Both vertices and edges are containers for properties.

> **Note**
>
> Tags and Edge types are similar to "vertex tables" and "edge tables" in the relational databases.

2.5.2 Directed property graph

NebulaGraph stores data in directed property graphs. A directed property graph has a set of vertices connected by directed edges. Both vertices and edges can have properties. A directed property graph is represented as:

$G = < V, E, P_V, P_E >$

- **V** is a set of vertices.
- **E** is a set of directed edges.
- **P_V** is the property of vertices.
- **P_E** is the property of edges.

The following table is an example of the structure of the basketball player dataset. We have two types of vertices, that is **player** and **team**, and two types of edges, that is **serve** and **follow**.

Element	Name	Property name (Data type)	Description
Tag	**player**	name (string) age (int)	Represents players in the team.
Tag	**team**	name (string)	Represents the teams.
Edge type	**serve**	start_year (int) end_year (int)	Represents actions taken by players in the team. An action links a player with a team, and the direction is from a player to a team.
Edge type	**follow**	degree (int)	Represents actions taken by players in the team. An action links a player with another player, and the direction is from one player to the other player.

Note

NebulaGraph supports only directed edges.

Compatibility

NebulaGraph 3.5.0 allows dangling edges. Therefore, when adding or deleting, you need to ensure the corresponding source vertex and destination vertex of an edge exist. For details, see INSERT VERTEX, DELETE VERTEX, INSERT EDGE, and DELETE EDGE.

The MERGE statement in openCypher is not supported.

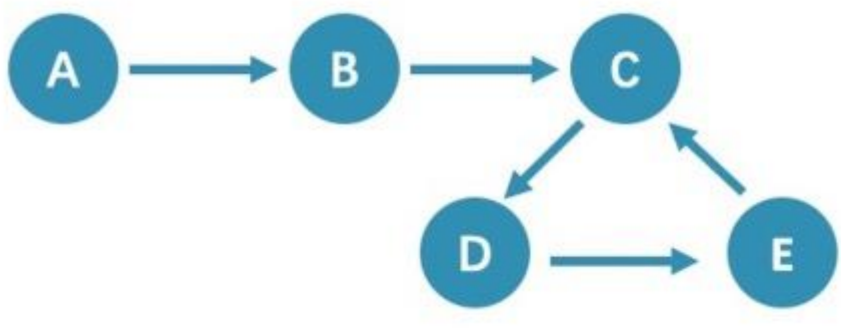
Last update: August 11, 2022

2.6 Path types

In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices. Paths are fundamental concepts of graph theory.

Paths can be categorized into 3 types: `walk`, `trail`, and `path`. For more information, see Wikipedia.

The following figure is an example for a brief introduction.



2.6.1 Walk

A `walk` is a finite or infinite sequence of edges. Both vertices and edges can be repeatedly visited in graph traversal.

In the above figure C, D, and E form a cycle. So, this figure contains infinite paths, such as `A->B->C->D->E`, `A->B->C->D->E->C`, and `A->B->C->D->E->C->D`.

Note

`GO` statements use `walk`.

2.6.2 Trail

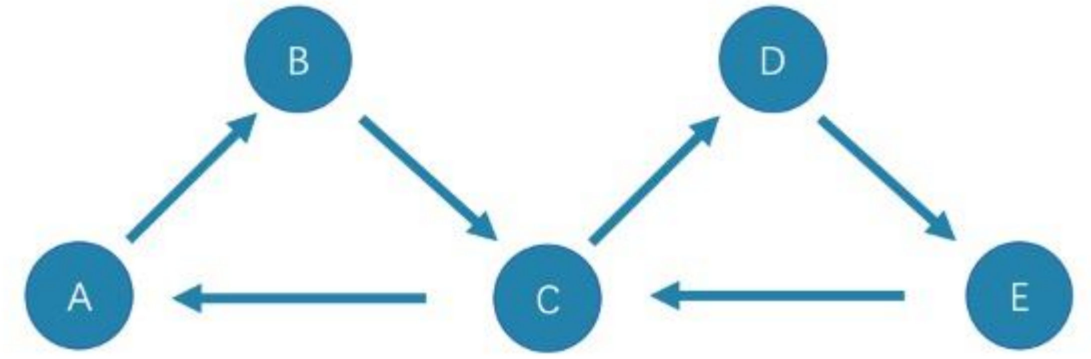
A `trail` is a finite sequence of edges. Only vertices can be repeatedly visited in graph traversal. The Seven Bridges of Königsberg is a typical `trail`.

In the above figure, edges cannot be repeatedly visited. So, this figure contains finite paths. The longest path in this figure consists of 5 edges: `A->B->C->D->E->C`.

Note

`MATCH`, `FIND PATH`, and `GET SUBGRAPH` statements use `trail`.

There are two special cases of trail, `cycle` and `circuit`. The following figure is an example for a brief introduction.



- cycle

  A `cycle` refers to a closed `trail`. Only the terminal vertices can be repeatedly visited. The longest path in this figure consists of 3 edges: `A->B->C->A` or `C->D->E->C`.

- circuit

  A `circuit` refers to a closed `trail`. Edges cannot be repeatedly visited in graph traversal. Apart from the terminal vertices, other vertices can also be repeatedly visited. The longest path in this figure: `A->B->C->D->E->C->A`.

2.6.3 Path

A `path` is a finite sequence of edges. Neither vertices nor edges can be repeatedly visited in graph traversal.

So, the above figure contains finite paths. The longest path in this figure consists of 4 edges: `A->B->C->D->E`.

Last update: March 23, 2022

2.7 VID

In a graph space, a vertex is uniquely identified by its ID, which is called a VID or a Vertex ID.

2.7.1 Features

- The data types of VIDs are restricted to `FIXED_STRING(<N>)` or `INT64`. One graph space can only select one VID type.
- A VID in a graph space is unique. It functions just as a primary key in a relational database. VIDs in different graph spaces are independent.
- The VID generation method must be set by users, because NebulaGraph does not provide auto increasing ID, or UUID.
- Vertices with the same VID will be identified as the same one. For example:
- A VID is the unique identifier of an entity, like a person's ID card number. A tag means the type of an entity, such as driver, and boss. Different tags define two groups of different properties, such as driving license number, driving age, order amount, order taking alt, and job number, payroll, debt ceiling, business phone number.
- When two `INSERT` statements (neither uses a parameter of `IF NOT EXISTS`) with the same VID and tag are operated at the same time, the latter `INSERT` will overwrite the former.
- When two `INSERT` statements with the same VID but different tags, like `TAG A` and `TAG B`, are operated at the same time, the operation of `Tag A` will not affect `Tag B`.
- VIDs will usually be indexed and stored into memory (in the way of LSM-tree). Thus, direct access to VIDs enjoys peak performance.

2.7.2 VID Operation

- NebulaGraph 1.x only supports `INT64` while NebulaGraph 2.x supports `INT64` and `FIXED_STRING(<N>)`. In `CREATE SPACE`, VID types can be set via `vid_type`.
- `id()` function can be used to specify or locate a VID.
- `LOOKUP` or `MATCH` statements can be used to find a VID via property index.
- Direct access to vertices statements via VIDs enjoys peak performance, such as `DELETE xxx WHERE id(xxx) == "player100"` or `GO FROM "player100"`. Finding VIDs via properties and then operating the graph will cause poor performance, such as `LOOKUP | GO FROM $-.ids`, which will run both `LOOKUP` and `|` one more time.

2.7.3 VID Generation

VIDs can be generated via applications. Here are some tips:

- (Optimal) Directly take a unique primary key or property as a VID. Property access depends on the VID.
- Generate a VID via a unique combination of properties. Property access depends on property index.
- Generate a VID via algorithms like snowflake. Property access depends on property index.
- If short primary keys greatly outnumber long primary keys, do not enlarge the `N` of `FIXED_STRING(<N>)` too much. Otherwise, it will occupy a lot of memory and hard disks, and slow down performance. Generate VIDs via BASE64, MD5, hash by encoding and splicing.
- If you generate int64 VID via hash, the probability of collision is about 1/10 when there are 1 billion vertices. The number of edges has no concern with the probability of collision.

2.7.4 Define and modify a VID and its data type

The data type of a VID must be defined when you create the graph space. Once defined, it cannot be modified.

A VID is set when you insert a vertex and cannot be modified.

2.7.5 Query `start vid` and global scan

In most cases, the execution plan of query statements in NebulaGraph (`MATCH`, `GO`, and `LOOKUP`) must query the `start vid` in a certain way.

There are only two ways to locate `start vid`:

1. For example, `GO FROM "player100" OVER` explicitly indicates in the statement that `start vid` is "player100".
2. For example, `LOOKUP ON player WHERE player.name == "Tony Parker"` or `MATCH (v:player {name:"Tony Parker"})` locates `start vid` by the index of the property `player.name`.

Last update: April 25, 2023

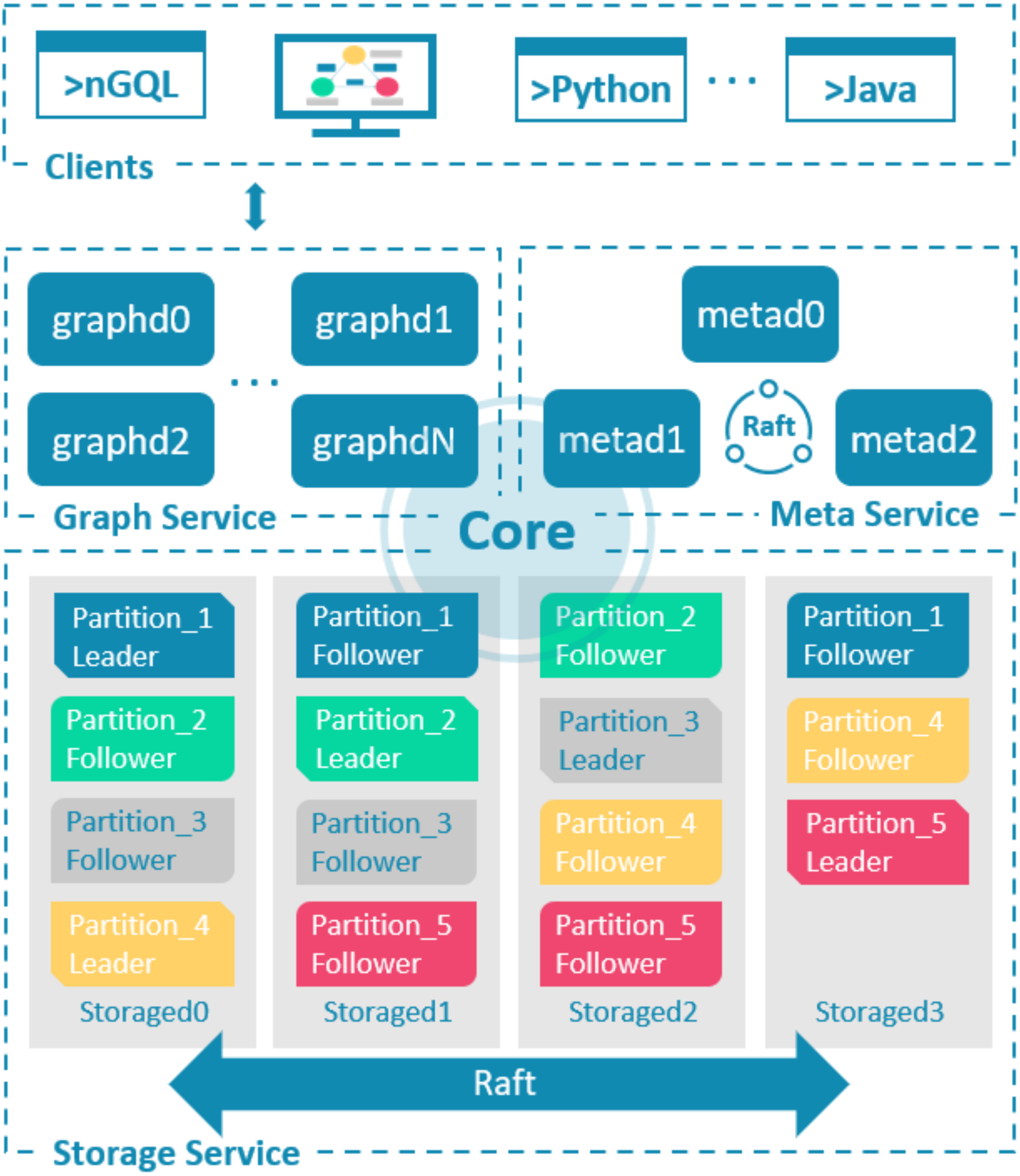
2.8 NebulaGraph architecture

2.8.1 Architecture overview

NebulaGraph consists of three services: the Graph Service, the Storage Service, and the Meta Service. It applies the separation of storage and computing architecture.

Each service has its executable binaries and processes launched from the binaries. Users can deploy a NebulaGraph cluster on a single machine or multiple machines using these binaries.

The following figure shows the architecture of a typical NebulaGraph cluster.



The Meta Service

The Meta Service in the NebulaGraph architecture is run by the nebula-metad processes. It is responsible for metadata management, such as schema operations, cluster administration, and user privilege management.

For details on the Meta Service, see Meta Service.

The Graph Service and the Storage Service

NebulaGraph applies the separation of storage and computing architecture. The Graph Service is responsible for querying. The Storage Service is responsible for storage. They are run by different processes, i.e., nebula-graphd and nebula-storaged. The benefits of the separation of storage and computing architecture are as follows:

- Great scalability

  The separated structure makes both the Graph Service and the Storage Service flexible and easy to scale in or out.

- High availability

  If part of the Graph Service fails, the data stored by the Storage Service suffers no loss. And if the rest part of the Graph Service is still able to serve the clients, service recovery can be performed quickly, even unfelt by the users.

- Cost-effective

  The separation of storage and computing architecture provides a higher resource utilization rate, and it enables clients to manage the cost flexibly according to business demands.

- Open to more possibilities

  With the ability to run separately, the Graph Service may work with multiple types of storage engines, and the Storage Service may also serve more types of computing engines.

For details on the Graph Service and the Storage Service, see Graph Service and Storage Service.

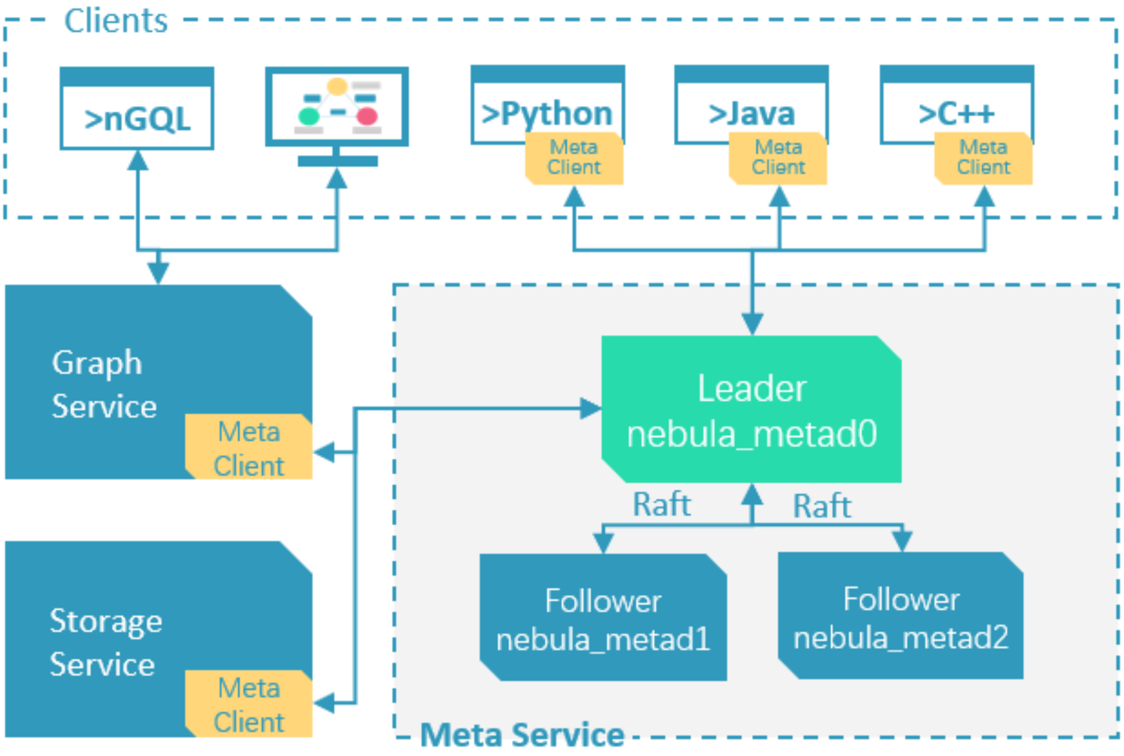
Last update: August 11, 2022

2.8.2 Meta Service

This topic introduces the architecture and functions of the Meta Service.

The architecture of the Meta Service

The architecture of the Meta Service is as follows:



The Meta Service is run by nebula-metad processes. Users can deploy nebula-metad processes according to the scenario:

- In a test environment, users can deploy one or three nebula-metad processes on different machines or a single machine.
- In a production environment, we recommend that users deploy three nebula-metad processes on different machines for high availability.

All the nebula-metad processes form a Raft-based cluster, with one process as the leader and the others as the followers.

The leader is elected by the majorities and only the leader can provide service to the clients or other components of NebulaGraph. The followers will be run in a standby way and each has a data replication of the leader. Once the leader fails, one of the followers will be elected as the new leader.

> **Note**
>
> The data of the leader and the followers will keep consistent through Raft. Thus the breakdown and election of the leader will not cause data inconsistency. For more information on Raft, see Storage service architecture.

Functions of the Meta Service

MANAGES USER ACCOUNTS

The Meta Service stores the information of user accounts and the privileges granted to the accounts. When the clients send queries to the Meta Service through an account, the Meta Service checks the account information and whether the account has the right privileges to execute the queries or not.

For more information on NebulaGraph access control, see Authentication.

MANAGES PARTITIONS

The Meta Service stores and manages the locations of the storage partitions and helps balance the partitions.

MANAGES GRAPH SPACES

NebulaGraph supports multiple graph spaces. Data stored in different graph spaces are securely isolated. The Meta Service stores the metadata of all graph spaces and tracks the changes of them, such as adding or dropping a graph space.

MANAGES SCHEMA INFORMATION

NebulaGraph is a strong-typed graph database. Its schema contains tags (i.e., the vertex types), edge types, tag properties, and edge type properties.

The Meta Service stores the schema information. Besides, it performs the addition, modification, and deletion of the schema, and logs the versions of them.

For more information on NebulaGraph schema, see Data model.

MANAGES TTL INFORMATION

The Meta Service stores the definition of TTL (Time to Live) options which are used to control data expiration. The Storage Service takes care of the expiring and evicting processes. For more information, see TTL.

MANAGES JOBS

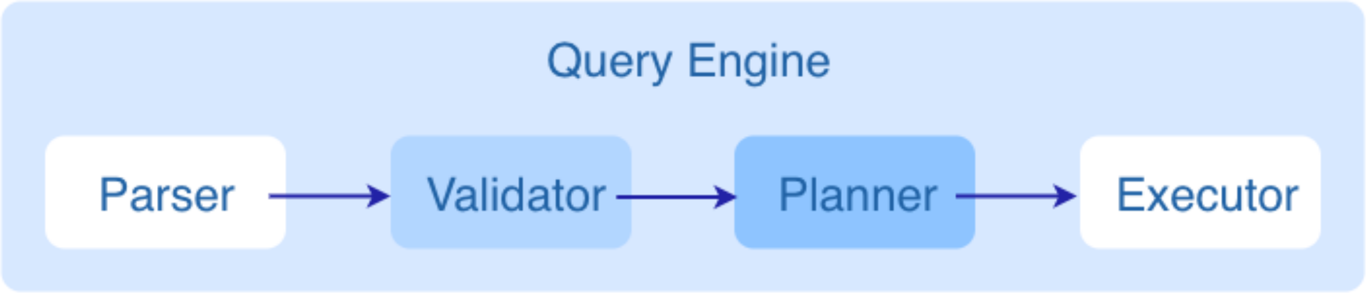
The Job Management module in the Meta Service is responsible for the creation, queuing, querying, and deletion of jobs.

Last update: August 11, 2022

2.8.3 Graph Service

The Graph Service is used to process the query. It has four submodules: Parser, Validator, Planner, and Executor. This topic will describe the Graph Service accordingly.

The architecture of the Graph Service



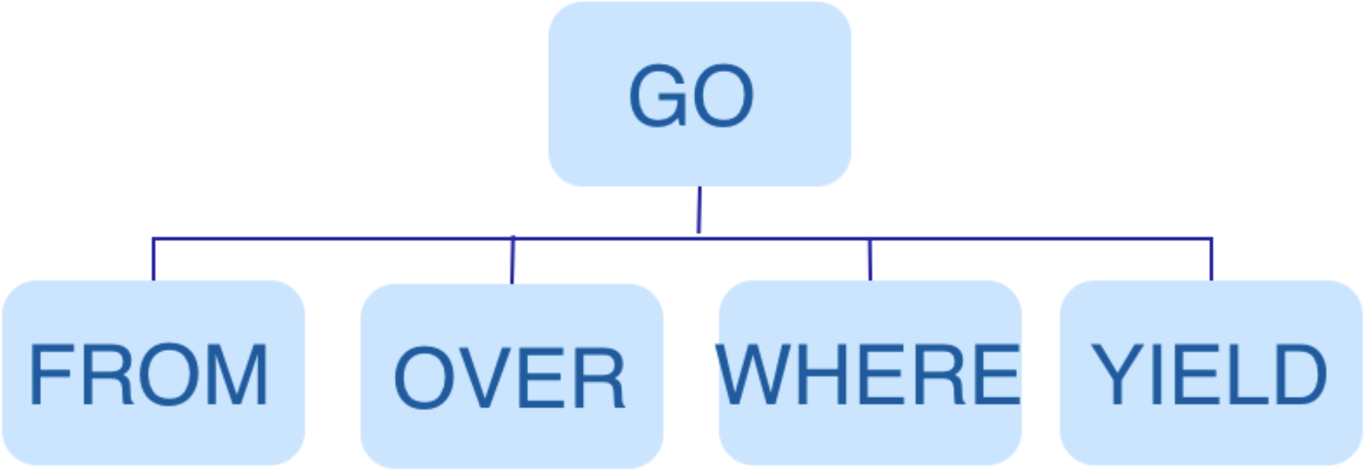
After a query is sent to the Graph Service, it will be processed by the following four submodules:

1. **Parser**: Performs lexical analysis and syntax analysis.
2. **Validator**: Validates the statements.
3. **Planner**: Generates and optimizes the execution plans.
4. **Executor**: Executes the plans with operators.

Parser

After receiving a request, the statements will be parsed by Parser composed of Flex (lexical analysis tool) and Bison (syntax analysis tool), and its corresponding AST will be generated. Statements will be directly intercepted in this stage because of their invalid syntax.

For example, the structure of the AST of `GO FROM "Tim" OVER like WHERE properties(edge).likeness > 8.0 YIELD dst(edge)` is shown in the following figure.



Validator

Validator performs a series of validations on the AST. It mainly works on these tasks:

- Validating metadata

  Validator will validate whether the metadata is correct or not.

  When parsing the `OVER`, `WHERE`, and `YIELD` clauses, Validator looks up the Schema and verifies whether the edge type and tag data exist or not. For an `INSERT` statement, Validator verifies whether the types of the inserted data are the same as the ones defined in the Schema.

- Validating contextual reference

  Validator will verify whether the cited variable exists or not, or whether the cited property is variable or not.

  For composite statements, like `$var = GO FROM "Tim" OVER like YIELD dst(edge) AS ID; GO FROM $var.ID OVER serve YIELD dst(edge)`, Validator verifies first to see if `var` is defined, and then to check if the `ID` property is attached to the `var` variable.

- Validating type inference

  Validator infers what type the result of an expression is and verifies the type against the specified clause.

  For example, the `WHERE` clause requires the result to be a `bool` value, a `NULL` value, or `empty`.

- Validating the information of `*`

  Validator needs to verify all the Schema that involves `*` when verifying the clause if there is a `*` in the statement.

  Take a statement like `GO FROM "Tim" OVER * YIELD dst(edge), properties(edge).likeness, dst(edge)` as an example. When verifying the `OVER` clause, Validator needs to verify all the edge types. If the edge type includes `like` and `serve`, the statement would be `GO FROM "Tim" OVER like,serve YIELD dst(edge), properties(edge).likeness, dst(edge)`.

- Validating input and output

  Validator will check the consistency of the clauses before and after the `|`.

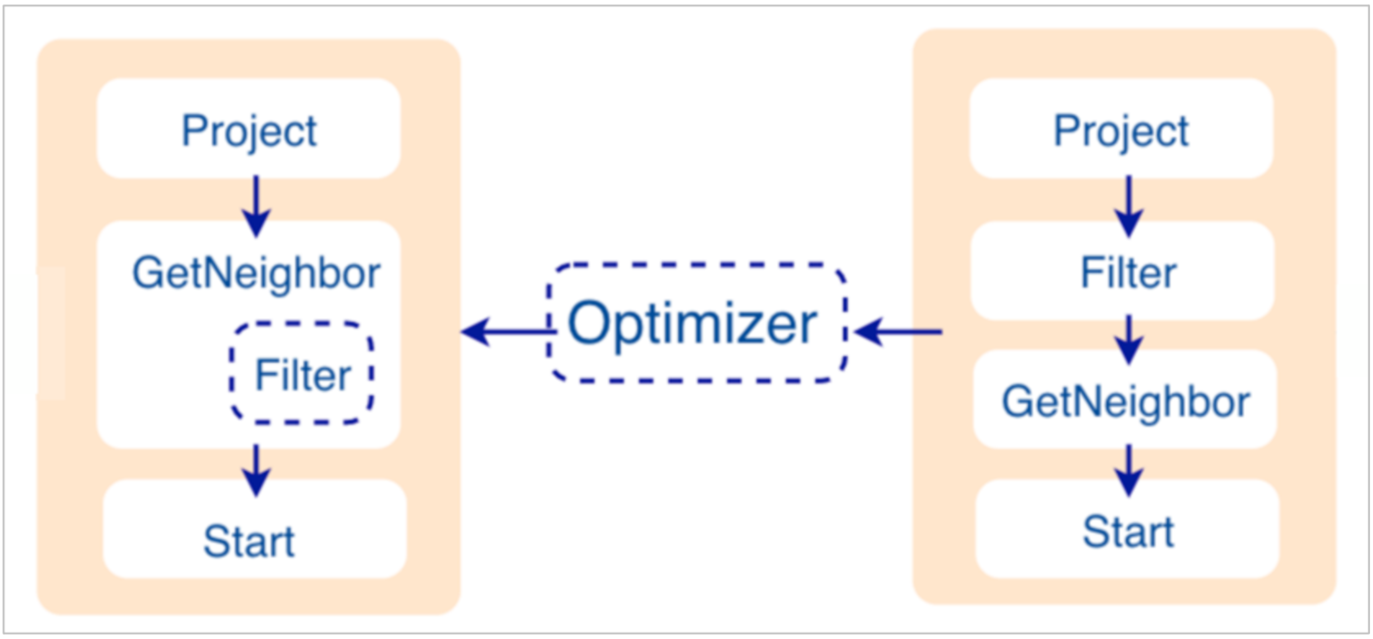
  In the statement `GO FROM "Tim" OVER like YIELD dst(edge) AS ID | GO FROM $-.ID OVER serve YIELD dst(edge)`, Validator will verify whether `$-.ID` is defined in the clause before the `|`.

When the validation succeeds, an execution plan will be generated. Its data structure will be stored in the `src/planner` directory.

Planner

In the `nebula-graphd.conf` file, when `enable_optimizer` is set to be `false`, Planner will not optimize the execution plans generated by Validator. It will be executed by Executor directly.

In the `nebula-graphd.conf` file, when `enable_optimizer` is set to be `true`, Planner will optimize the execution plans generated by Validator. The structure is as follows.



- Before optimization

  In the execution plan on the right side of the preceding figure, each node directly depends on other nodes. For example, the root node `Project` depends on the `Filter` node, the `Filter` node depends on the `GetNeighbor` node, and so on, up to the leaf node `Start`. Then the execution plan is (not truly) executed.

  During this stage, every node has its input and output variables, which are stored in a hash table. The execution plan is not truly executed, so the value of each key in the associated hash table is empty (except for the `Start` node, where the input variables hold the starting data), and the hash table is defined in `src/context/ExecutionContext.cpp` under the `nebula-graph` repository.

  For example, if the hash table is named as `ResultMap` when creating the `Filter` node, users can determine that the node takes data from `ResultMap["GN1"]`, then puts the result into `ResultMap["Filter2"]`, and so on. All these work as the input and output of each node.

- Process of optimization

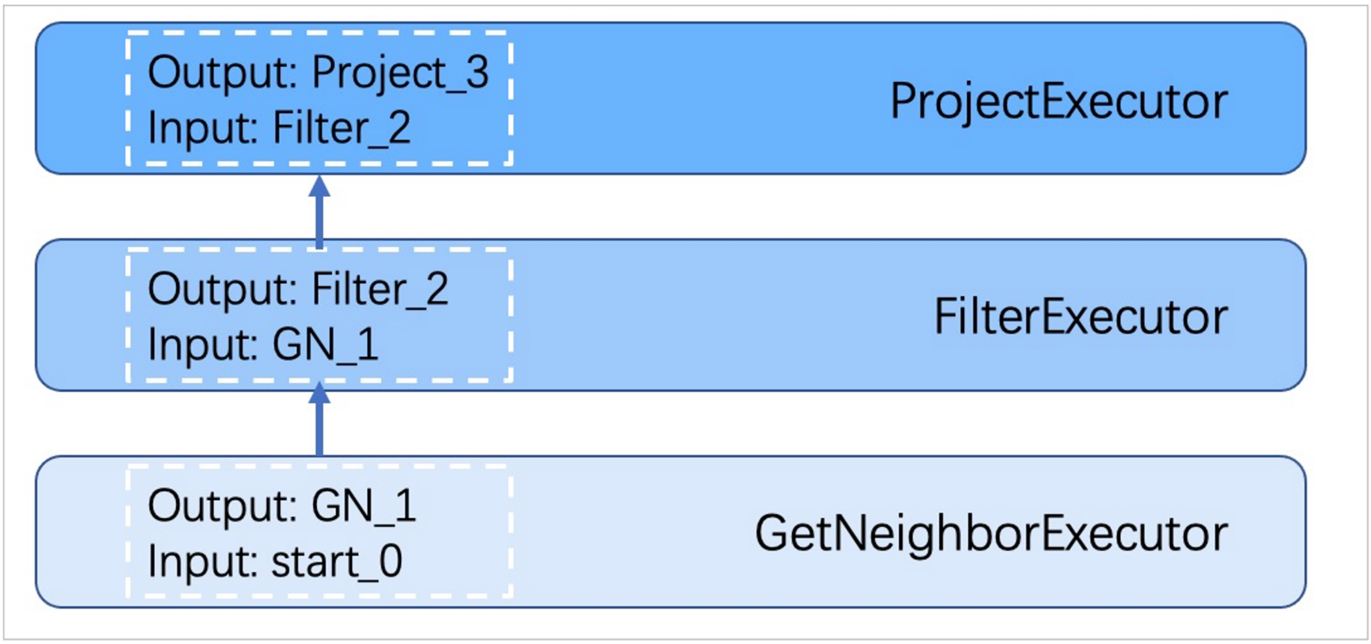
  The optimization rules that Planner has implemented so far are considered RBO (Rule-Based Optimization), namely the pre-defined optimization rules. The CBO (Cost-Based Optimization) feature is under development. The optimized code is in the `src/optimizer/` directory under the `nebula-graph` repository.

  RBO is a “bottom-up” exploration process. For each rule, the root node of the execution plan (in this case, the `Project` node) is the entry point, and step by step along with the node dependencies, it reaches the node at the bottom to see if it matches the rule.

  As shown in the preceding figure, when the `Filter` node is explored, it is found that its children node is `GetNeighbors`, which matches successfully with the pre-defined rules, so a transformation is initiated to integrate the `Filter` node into the `GetNeighbors` node, the `Filter` node is removed, and then the process continues to the next rule. Therefore, when the `GetNeighbor` operator calls interfaces of the Storage layer to get the neighboring edges of a vertex during the execution stage, the Storage layer will directly filter out the unqualified edges internally. Such optimization greatly reduces the amount of data transfer, which is commonly known as filter pushdown.

Executor

The Executor module consists of Scheduler and Executor. The Scheduler generates the corresponding execution operators against the execution plan, starting from the leaf nodes and ending at the root node. The structure is as follows.



Each node of the execution plan has one execution operator node, whose input and output have been determined in the execution plan. Each operator only needs to get the values for the input variables, compute them, and finally put the results into the corresponding output variables. Therefore, it is only necessary to execute step by step from `Start`, and the result of the last operator is returned to the user as the final result.

Source code hierarchy

The source code hierarchy under the nebula-graph repository is as follows.

```
|--src
   |--graph
      |--context    //contexts for validation and execution
      |--executor   //execution operators
      |--gc         //garbage collector
      |--optimizer  //optimization rules
      |--planner    //structure of the execution plans
      |--scheduler  //scheduler
      |--service    //external service management
      |--session    //session management
      |--stats      //monitoring metrics
      |--util       //basic components
      |--validator  //validation of the statements
      |--visitor    //visitor expression
```

Last update: August 9, 2022

2.8.4 Storage Service

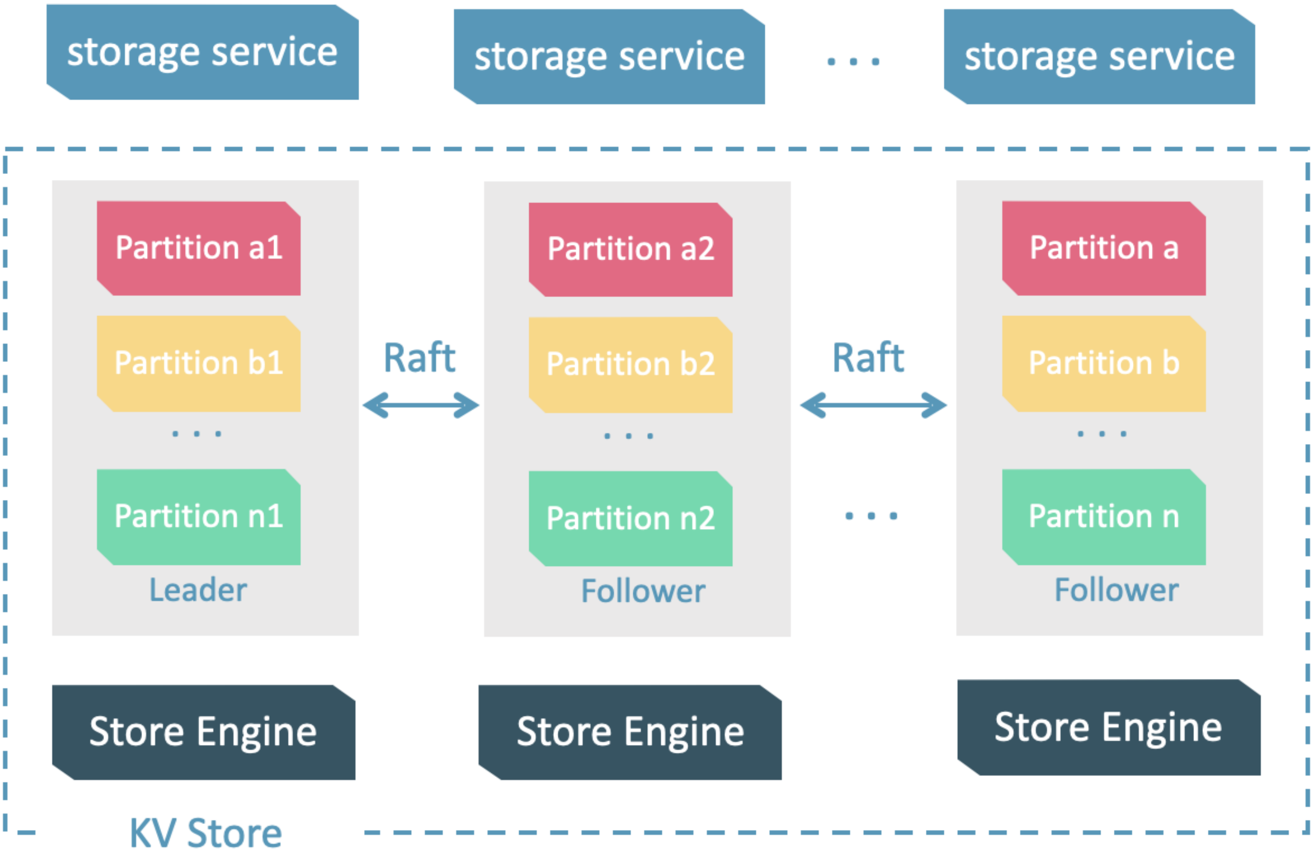
The persistent data of NebulaGraph have two parts. One is the Meta Service that stores the meta-related data.

The other is the Storage Service that stores the data, which is run by the nebula-storaged process. This topic will describe the architecture of the Storage Service.

Advantages

- High performance (Customized built-in KVStore)
- Great scalability (Shared-nothing architecture, not rely on NAS/SAN-like devices)
- Strong consistency (Raft)
- High availability (Raft)
- Supports synchronizing with the third party systems, such as Elasticsearch.

The architecture of the Storage Service



The Storage Service is run by the nebula-storaged process. Users can deploy nebula-storaged processes on different occasions. For example, users can deploy 1 nebula-storaged process in a test environment and deploy 3 nebula-storaged processes in a production environment.

All the nebula-storaged processes consist of a Raft-based cluster. There are three layers in the Storage Service:

- Storage interface

  The top layer is the storage interface. It defines a set of APIs that are related to the graph concepts. These API requests will be translated into a set of KV operations targeting the corresponding Partition. For example:

- `getNeighbors`: queries the in-edge or out-edge of a set of vertices, returns the edges and the corresponding properties, and supports conditional filtering.
- `insert vertex/edge`: inserts a vertex or edge and its properties.
- `getProps`: gets the properties of a vertex or an edge.

  It is this layer that makes the Storage Service a real graph storage. Otherwise, it is just a KV storage.

- Consensus

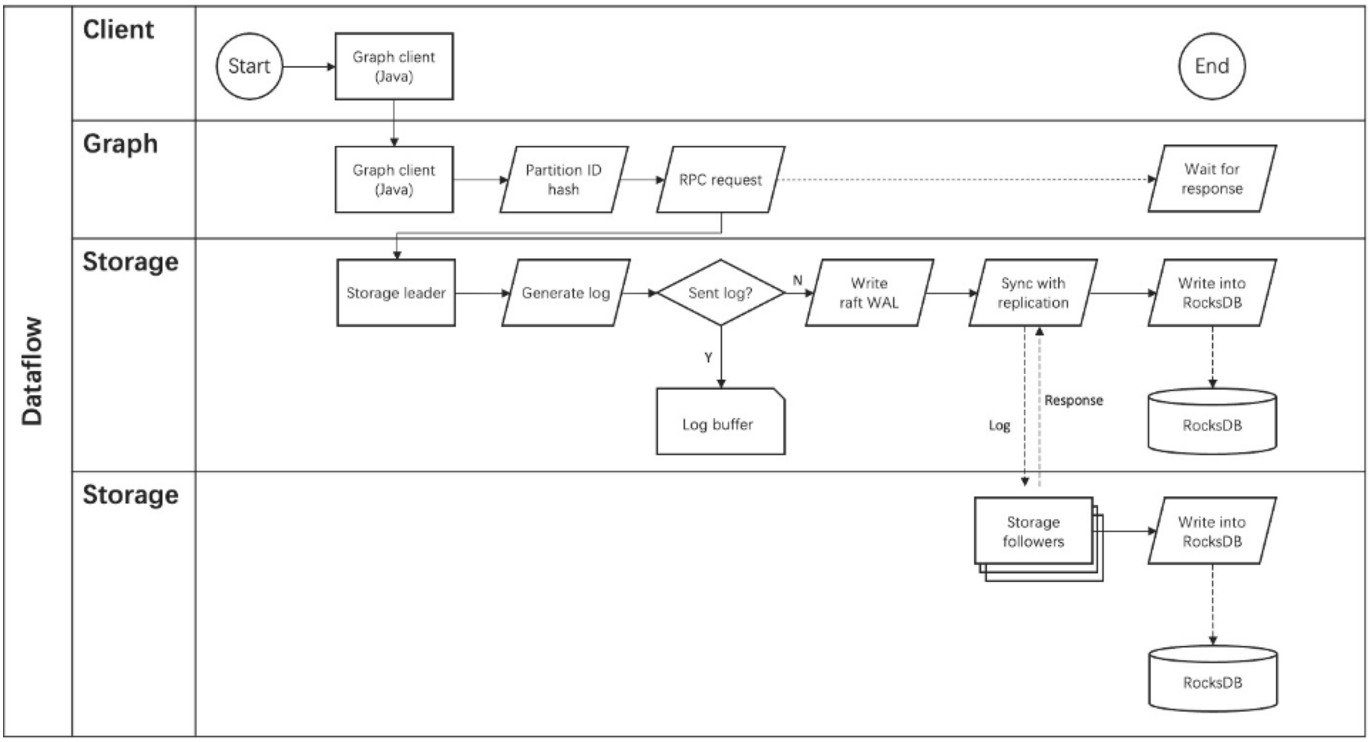
  Below the storage interface is the consensus layer that implements Multi Group Raft, which ensures the strong consistency and high availability of the Storage Service.

- Store engine

  The bottom layer is the local storage engine library, providing operations like `get`, `put`, and `scan` on local disks. The related interfaces are stored in `KVStore.h` and `KVEngine.h` files. You can develop your own local store plugins based on your needs.

The following will describe some features of the Storage Service based on the above architecture.

Storage writing process



KVStore

NebulaGraph develops and customizes its built-in KVStore for the following reasons.

- It is a high-performance KVStore.
- It is provided as a (kv) library and can be easily developed for the filter pushdown purpose. As a strong-typed database, how to provide Schema during pushdown is the key to efficiency for NebulaGraph.
- It has strong data consistency.

Therefore, NebulaGraph develops its own KVStore with RocksDB as the local storage engine. The advantages are as follows.

- For multiple local hard disks, NebulaGraph can make full use of its concurrent capacities through deploying multiple data directories.
- The Meta Service manages all the Storage servers. All the partition distribution data and current machine status can be found in the meta service. Accordingly, users can execute a manual load balancing plan in meta service.

> **Note**
>
> NebulaGraph does not support auto load balancing because auto data transfer will affect online business.

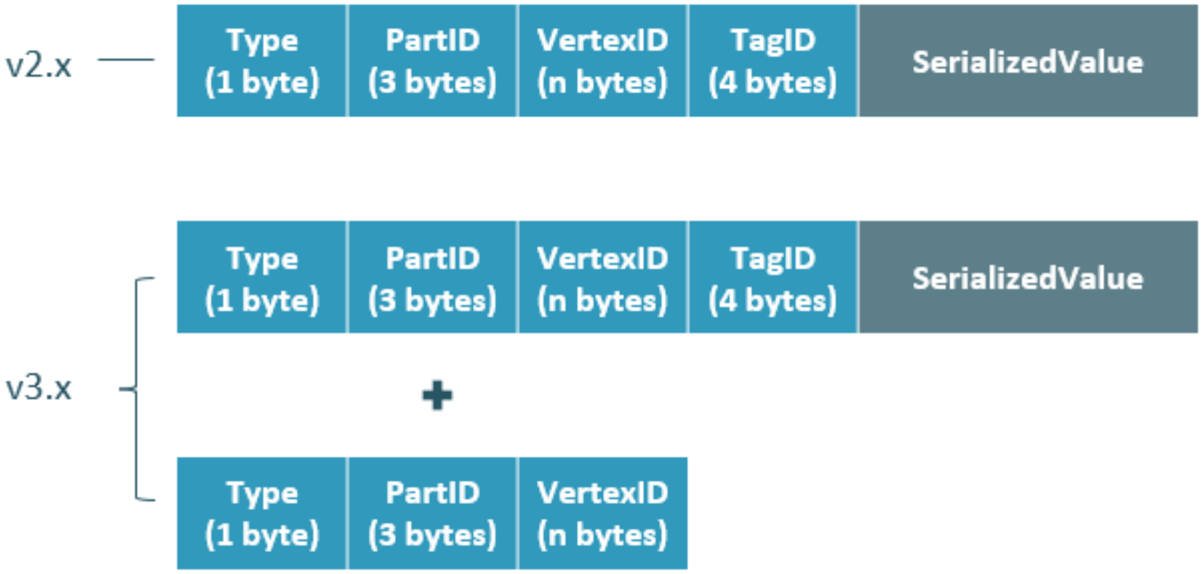
- NebulaGraph provides its own WAL mode so one can customize the WAL. Each partition owns its WAL.
- One NebulaGraph KVStore cluster supports multiple graph spaces, and each graph space has its own partition number and replica copies. Different graph spaces are isolated physically from each other in the same cluster.

Data storage structure

Graphs consist of vertices and edges. NebulaGraph uses key-value pairs to store vertices, edges, and their properties. Vertices and edges are stored in keys and their properties are stored in values. Such structure enables efficient property filtering.

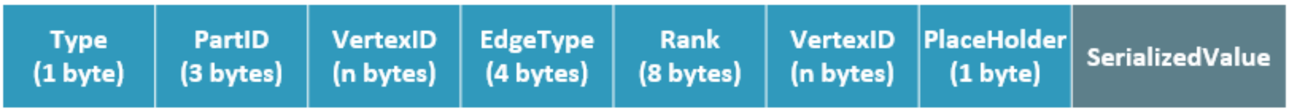
- The storage structure of vertices

  Different from NebulaGraph version 2.x, version 3.x added a new key for each vertex. Compared to the old key that still exists, the new key has no `TagID` field and no value. Vertices in NebulaGraph can now live without tags owing to the new key.



Field	Description
`Type`	One byte, used to indicate the key type.
`PartID`	Three bytes, used to indicate the sharding partition and to scan the partition data based on the prefix when re-balancing the partition.
`VertexID`	The vertex ID. For an integer VertexID, it occupies eight bytes. However, for a string VertexID, it is changed to `fixed_string` of a fixed length which needs to be specified by users when they create the space.
`TagID`	Four bytes, used to indicate the tags that vertex relate with.
`SerializedValue`	The serialized value of the key. It stores the property information of the vertex.

- The storage structure of edges



Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	EdgeType (4 bytes)	Rank (8 bytes)	VertexID (n bytes)	PlaceHolder (1 byte)	SerializedValue

Field	Description
`Type`	One byte, used to indicate the key type.
`PartID`	Three bytes, used to indicate the partition ID. This field can be used to scan the partition data based on the prefix when re-balancing the partition.
`VertexID`	Used to indicate vertex ID. The former VID refers to the source VID in the outgoing edge and the dest VID in the incoming edge, while the latter VID refers to the dest VID in the outgoing edge and the source VID in the incoming edge.
`Edge Type`	Four bytes, used to indicate the edge type. Greater than zero indicates out-edge, less than zero means in-edge.
`Rank`	Eight bytes, used to indicate multiple edges in one edge type. Users can set the field based on needs and store weight, such as transaction time and transaction number.
`PlaceHolder`	One byte. Reserved.
`SerializedValue`	The serialized value of the key. It stores the property information of the edge.

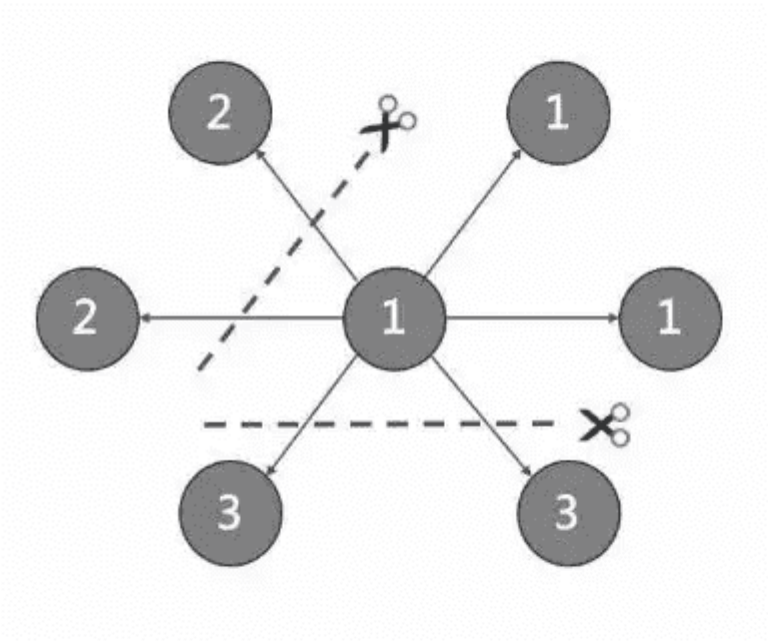
PROPERTY DESCRIPTIONS

NebulaGraph uses strong-typed Schema.

NebulaGraph will store the properties of vertex and edges in order after encoding them. Since the length of fixed-length properties is fixed, queries can be made in no time according to offset. Before decoding, NebulaGraph needs to get (and cache) the schema information in the Meta Service. In addition, when encoding properties, NebulaGraph will add the corresponding schema version to support online schema change.

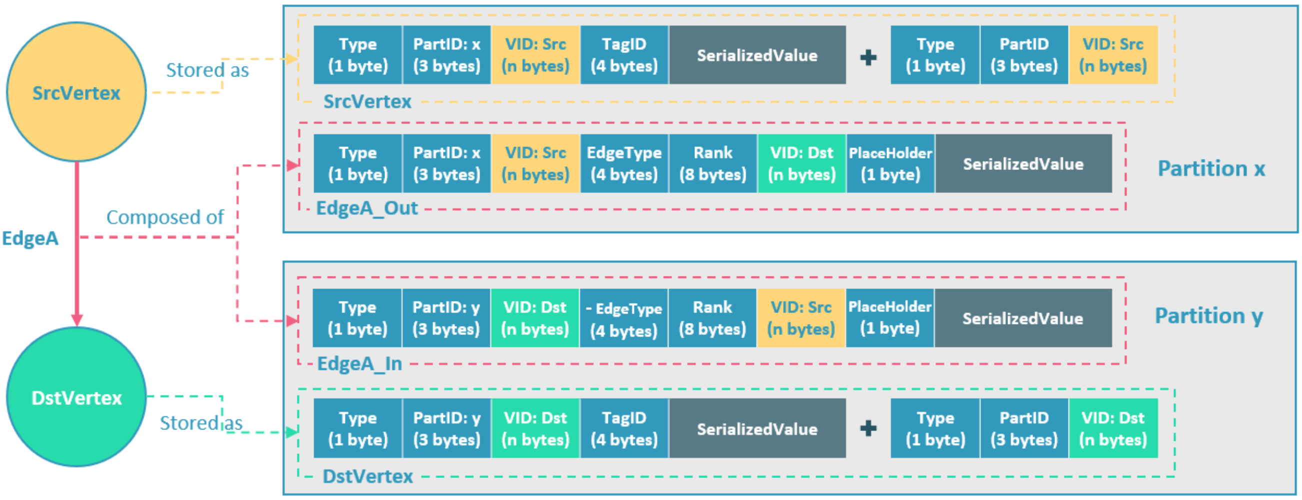
Data partitioning

Since in an ultra-large-scale relational network, vertices can be as many as tens to hundreds of billions, and edges are even more than trillions. Even if only vertices and edges are stored, the storage capacity of both exceeds that of ordinary servers. Therefore, NebulaGraph uses hash to shard the graph elements and store them in different partitions.



EDGE PARTITIONING AND STORAGE AMPLIFICATION

In NebulaGraph, an edge corresponds to two key-value pairs on the hard disk. When there are lots of edges and each has many properties, storage amplification will be obvious. The storage format of edges is shown in the figure below.



In this example, SrcVertex connects DstVertex via EdgeA, forming the path of `(SrcVertex)-[EdgeA]->(DstVertex)`. SrcVertex, DstVertex, and EdgeA will all be stored in Partition x and Partition y as four key-value pairs in the storage layer. Details are as follows:

- The key value of SrcVertex is stored in Partition x. Key fields include Type, PartID(x), VID(Src), and TagID. SerializedValue, namely Value, refers to serialized vertex properties.
- The first key value of EdgeA, namely EdgeA_Out, is stored in the same partition as the SrcVertex. Key fields include Type, PartID(x), VID(Src), EdgeType(+ means out-edge), Rank(0), VID(Dst), and PlaceHolder. SerializedValue, namely Value, refers to serialized edge properties.
- The key value of DstVertex is stored in Partition y. Key fields include Type, PartID(y), VID(Dst), and TagID. SerializedValue, namely Value, refers to serialized vertex properties.
- The second key value of EdgeA, namely EdgeA_In, is stored in the same partition as the DstVertex. Key fields include Type, PartID(y), VID(Dst), EdgeType(- means in-edge), Rank(0), VID(Src), and PlaceHolder. SerializedValue, namely Value, refers to serialized edge properties, which is exactly the same as that in EdgeA_Out.

EdgeA_Out and EdgeA_In are stored in storage layer with opposite directions, constituting EdgeA logically. EdgeA_Out is used for traversal requests starting from SrcVertex, such as `(a)-[]->()`; EdgeA_In is used for traversal requests starting from DstVertex, such as `()-[]->(a)`.

Like EdgeA_Out and EdgeA_In, NebulaGraph redundantly stores the information of each edge, which doubles the actual capacities needed for edge storage. The key corresponding to the edge occupies a small hard disk space, but the space occupied by Value is proportional to the length and amount of the property value. Therefore, it will occupy a relatively large hard disk space if the property value of the edge is large or there are many edge property values.

PARTITION ALGORITHM

NebulaGraph uses a **static Hash** strategy to shard data through a modulo operation on vertex ID. All the out-keys, in-keys, and tag data will be placed in the same partition. In this way, query efficiency is increased dramatically.

Note

The number of partitions needs to be determined when users are creating a graph space since it cannot be changed afterward. Users are supposed to take into consideration the demands of future business when setting it.

When inserting into NebulaGraph, vertices and edges are distributed across different partitions. And the partitions are located on different machines. The number of partitions is set in the CREATE SPACE statement and cannot be changed afterward.

If certain vertices need to be placed on the same partition (i.e., on the same machine), see Formula/code.

The following code will briefly describe the relationship between VID and partition.

```
// If VertexID occupies 8 bytes, it will be stored in int64 to be compatible with the version 1.0.
uint64_t vid = 0;
if (id.size() == 8) {
    memcpy(static_cast<void*>(&vid), id.data(), 8);
} else {
    MurmurHash2 hash;
    vid = hash(id.data());
}
PartitionID pId = vid % numParts + 1;
```

Roughly speaking, after hashing a fixed string to int64, (the hashing of int64 is the number itself), do modulo, and then plus one, namely:

```
pId = vid % numParts + 1;
```

Parameters and descriptions of the preceding formula are as follows:

Parameter	Description
`%`	The modulo operation.
`numParts`	The number of partitions for the graph space where the `VID` is located, namely the value of `partition_num` in the CREATE SPACE statement.
`pId`	The ID for the partition where the `VID` is located.

Suppose there are 100 partitions, the vertices with `VID` 1, 101, and 1001 will be stored on the same partition. But, the mapping between the partition ID and the machine address is random. Therefore, we cannot assume that any two partitions are located on the same machine.

Raft

RAFT IMPLEMENTATION

In a distributed system, one data usually has multiple replicas so that the system can still run normally even if a few copies fail. It requires certain technical means to ensure consistency between replicas.

Basic principle: Raft is designed to ensure consistency between replicas. Raft uses election between replicas, and the (candidate) replica that wins more than half of the votes will become the Leader, providing external services on behalf of all replicas. The rest Followers will play backups. When the Leader fails (due to communication failure, operation and maintenance commands, etc.), the rest Followers will conduct a new round of elections and vote for a new Leader. The Leader and Followers will detect each other's survival through heartbeats and write them to the hard disk in Raft-wal mode. Replicas that do not respond to more than multiple heartbeats will be considered faulty.

> **Note**
>
> Raft-wal needs to be written into the hard disk periodically. If hard disk bottlenecks to write, Raft will fail to send a heartbeat and conduct a new round of elections. If the hard disk IO is severely blocked, there will be no Leader for a long time.

Read and write: For every writing request of the clients, the Leader will initiate a Raft-wal and synchronize it with the Followers. Only after over half replicas have received the Raft-wal will it return to the clients successfully. For every reading request of the clients, it will get to the Leader directly, while Followers will not be involved.

Failure: Scenario 1: Take a (space) cluster of a single replica as an example. If the system has only one replica, the Leader will be itself. If failure happens, the system will be completely unavailable. Scenario 2: Take a (space) cluster of three replicas as an example. If the system has three replicas, one of them will be the Leader and the rest will be the Followers. If the Leader fails, the rest two can still vote for a new Leader (and a Follower), and the system is still available. But if any of the two Followers fails again, the system will be completely unavailable due to inadequate voters.

> **Note**
>
> Raft and HDFS have different modes of duplication. Raft is based on a quorum vote, so the number of replicas cannot be even.

MULTI GROUP RAFT

The Storage Service supports a distributed cluster architecture, so NebulaGraph implements Multi Group Raft according to Raft protocol. Each Raft group stores all the replicas of each partition. One replica is the leader, while others are followers. In this way, NebulaGraph achieves strong consistency and high availability. The functions of Raft are as follows.

NebulaGraph uses Multi Group Raft to improve performance when there are many partitions because Raft-wal cannot be NULL. When there are too many partitions, costs will increase, such as storing information in Raft group, WAL files, or batch operation in low load.

There are two key points to implement the Multi Raft Group:

- To share transport layer

  Each Raft Group sends messages to its corresponding peers. So if the transport layer cannot be shared, the connection costs will be very high.

- To share thread pool

  Raft Groups share the same thread pool to prevent starting too many threads and a high context switch cost.

BATCH

For each partition, it is necessary to do a batch to improve throughput when writing the WAL serially. As NebulaGraph uses WAL to implement some special functions, batches need to be grouped, which is a feature of NebulaGraph.

For example, lock-free CAS operations will execute after all the previous WALs are committed. So for a batch, if there are several WALs in CAS type, we need to divide this batch into several smaller groups and make sure they are committed serially.

TRANSFER LEADERSHIP

Transfer leadership is extremely important for balance. When moving a partition from one machine to another, NebulaGraph first checks if the source is a leader. If so, it should be moved to another peer. After data migration is completed, it is important to balance leader distribution again.

When a transfer leadership command is committed, the leader will abandon its leadership and the followers will start a leader election.

PEER CHANGES

To avoid split-brain, when members in a Raft Group change, an intermediate state is required. In such a state, the quorum of the old group and new group always have an overlap. Thus it prevents the old or new group from making decisions unilaterally. To make it even simpler, in his doctoral thesis Diego Ongaro suggests adding or removing a peer once to ensure the overlap between the quorum of the new group and the old group. NebulaGraph also uses this approach, except that the way to add or remove a member is different. For details, please refer to addPeer/removePeer in the Raft Part class.

Differences with HDFS

The Storage Service is a Raft-based distributed architecture, which has certain differences with that of HDFS. For example:

- The Storage Service ensures consistency through Raft. Usually, the number of its replicas is odd to elect a leader. However, DataNode used by HDFS ensures consistency through NameNode, which has no limit on the number of replicas.
- In the Storage Service, only the replicas of the leader can read and write, while in HDFS all the replicas can do so.
- In the Storage Service, the number of replicas needs to be determined when creating a space, since it cannot be changed afterward. But in HDFS, the number of replicas can be changed freely.
- The Storage Service can access the file system directly. While the applications of HDFS (such as HBase) have to access HDFS before the file system, which requires more RPC times.

In a word, the Storage Service is more lightweight with some functions simplified and its architecture is simpler than HDFS, which can effectively improve the read and write performance of a smaller block of data.

Last update: August 10, 2023

3. Licensing

3.1 About NebulaGraph licenses

Enterpriseonly

NebulaGraph licenses applies only to the NebulaGraph Enterprise Edition.

3.1.1 What NebulaGraph licenses do

NebulaGraph licenses are the legal permissions granted by Vesoft Co., Ltd., allowing you to utilize the capabilities of a NebulaGraph Enterprise Edition database and its associated software. You can buy a NebulaGraph license on a cloud marketplace or by contacting Vesoft's sales team. Currently, the only cloud marketplace available is the AWS Marketplace. You can purchase a NebulaGraph license from NebulaGraph Enterprise (by Node) on the AWS Marketplace.

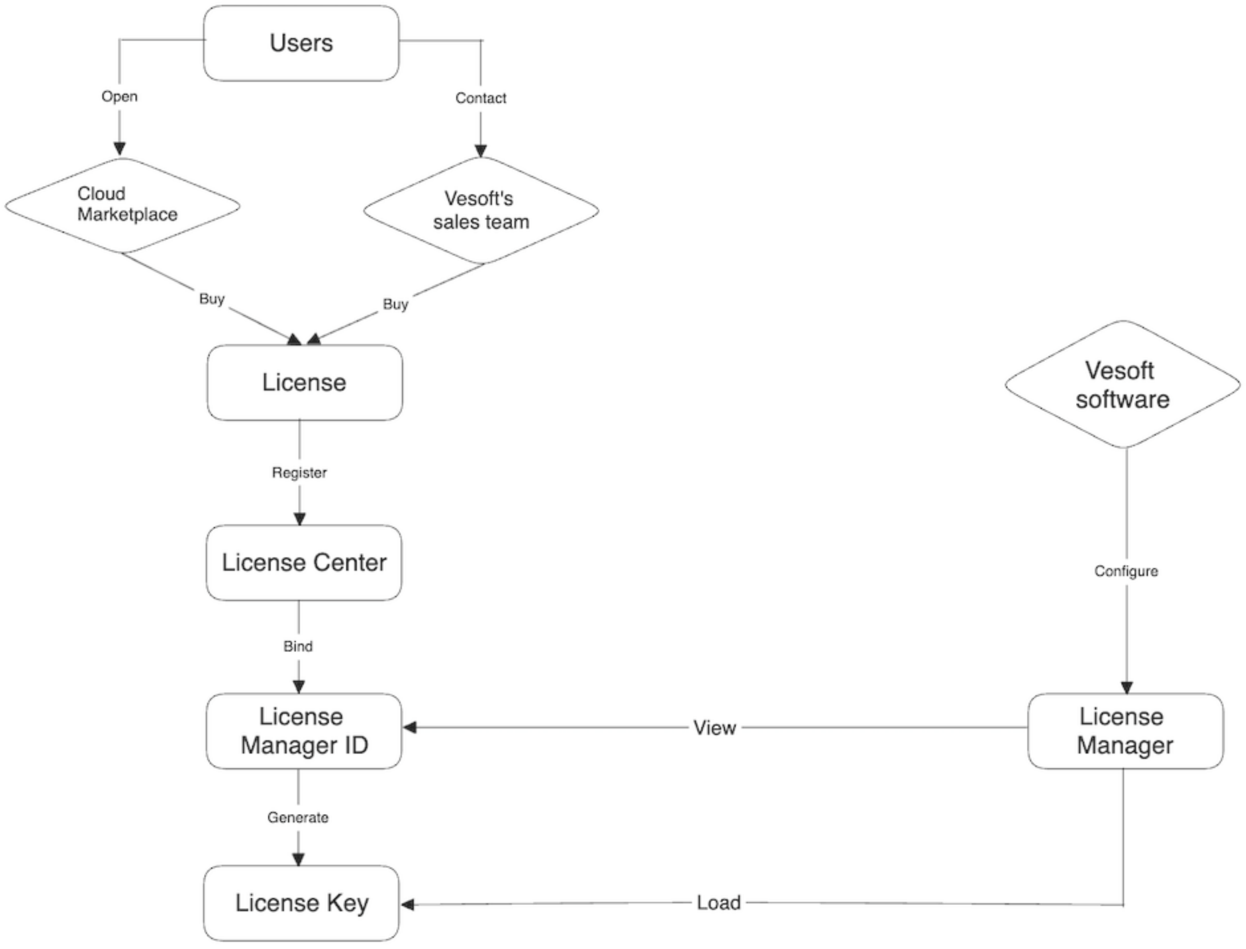
After purchasing a NebulaGraph license, you must obtain a license key by binding an LMID through the LC. Once the license key is obtained, you need to use the LM service to load the license key. When starting the NebulaGraph Enterprise and associated software, the LM service will check the validity of the license. If the license is valid, then the graph database and associated software will function normally. Otherwise, the graph database and associated software will not be functional.

You can view the license information, including the expiration date, nodes purchased, and license key on the LC or by using the LM client to query the license information via the command line.

3.1.2 License key

A license key is an encrypted string containing authorization information and serves as the unique credential for you to obtain access to the NebulaGraph Enterprise and its associated software features. There are two forms of license keys: online license keys and offline license keys. For more information, see License key.

3.1.3 Licensing process flowchart



3.1.4 Licensing process

Purchasing licenses on cloud marketplaces

1. Create a contract for the purchase of a NebulaGraph license through a cloud marketplace service.
2. Follow the setup account link to set up your LC account.
3. Receive email attachments that contain NebulaGraph Enterprise and LM installation packages.
4. View and copy the LMID on your LM.
5. Bind the LMID to generate a license key on LC.
6. Load the license key on LM.
7. Configure the LM address in the NebulaGraph and associated software.
8. Start NebulaGraph and associated software.

Purchasing licenses through Vesoft sales personnel

1. Contact Vesoft's sales personnel to purchase a NebulaGraph license and obtain NebulaGraph and LM installation packages.
2. Receive an email to set up your LC account.
3. View and copy the LMID on your LM.
4. Bind the LMID to generate a license key on LC.
5. Load the license key on LM.

6. Configure the LM address in the NebulaGraph and associated software.
7. Start NebulaGraph and associated software.

Last update: July 11, 2023

3.2 License management suites

3.2.1 License management suites overview

The license management suites are a combination of a platform and services designed to enable you to obtain authorized access to the NebulaGraph Enterprise Edition database and its associated software. These suites include license purchase services, the publicly accessible license management platform known as the License Center (LC), and the client-side license management service called the License Manager (LM).

NebulaGraph Enterprise (by Node)

NebulaGraph Enterprise (by Node) is a service offered by Vesoft on AWS Marketplace, which allows you to easily sign contracts, purchase, or update Vesoft licenses. For more information on this service, see Purchase Licenses.

Note

Currently, the license purchase service is available exclusively on AWS Marketplace. However, Vesoft plans to expand to more cloud marketplaces for license purchases in the future.

License Center

License Center (LC) by Vesoft is a publicly accessible platform that is used to record and manage all purchased licenses. Its main purpose is to enable you to view your license information through public access, including the license key, valid duration, number of querying nodes, number of storage nodes, and other relevant details. For more information, see License Center (LC).

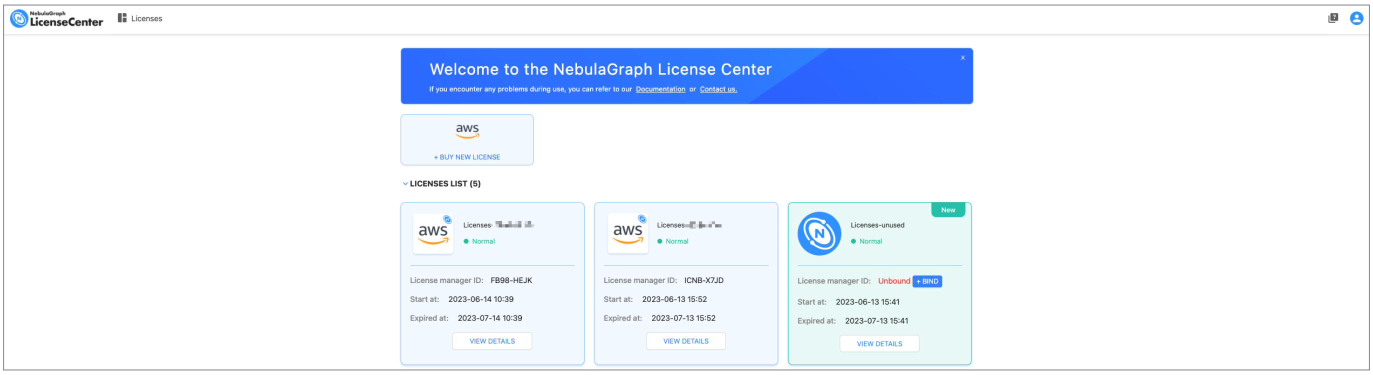
License Manager

Vesoft's License Manager (LM) is an essential service that operates in the background to manage your NebulaGraph licenses and license the NebulaGraph Enterprise Edition database and associated software. The LM client tool enables you to query and view your license information conveniently from the client side. This information includes the license key, validation period, and the number of resources purchased. For more information, see License Manager (LM).

Last update: May 12, 2023

3.2.2 License Center

License Center (LC) provided by Vesoft is an online platform for managing licenses that is accessible through public networks. On the LC platform, you can track all your purchased license information, including details such as license type, number of purchased resources, the status of the license, and expiration date.



Welcome to the NebulaGraph License Center

If you encounter any problems during use, you can refer to our Documentation or Contact us.

LICENSES LIST (5)

Licenses [redacted]
Normal
License manager ID: FB98-HEJK
Start at: 2023-06-14 10:39
Expired at: 2023-07-14 10:39

Licenses [redacted]
Normal
License manager ID: ICNB-X7JD
Start at: 2023-06-13 15:52
Expired at: 2023-07-13 15:52

Licenses-unused
Normal
License manager ID: Unbound
Start at: 2023-06-13 15:41
Expired at: 2023-07-13 15:41

To generate a license key, you need to bind the ID of your License Manager (LM) on LC. The license key must then be loaded into the installed LM service. And after specifying the LM access address in the software, you can authorize the license which enables you to use NebulaGraph Enterprise.

This article introduces how to set up an LC account, bind the LMID, and generate the license key.

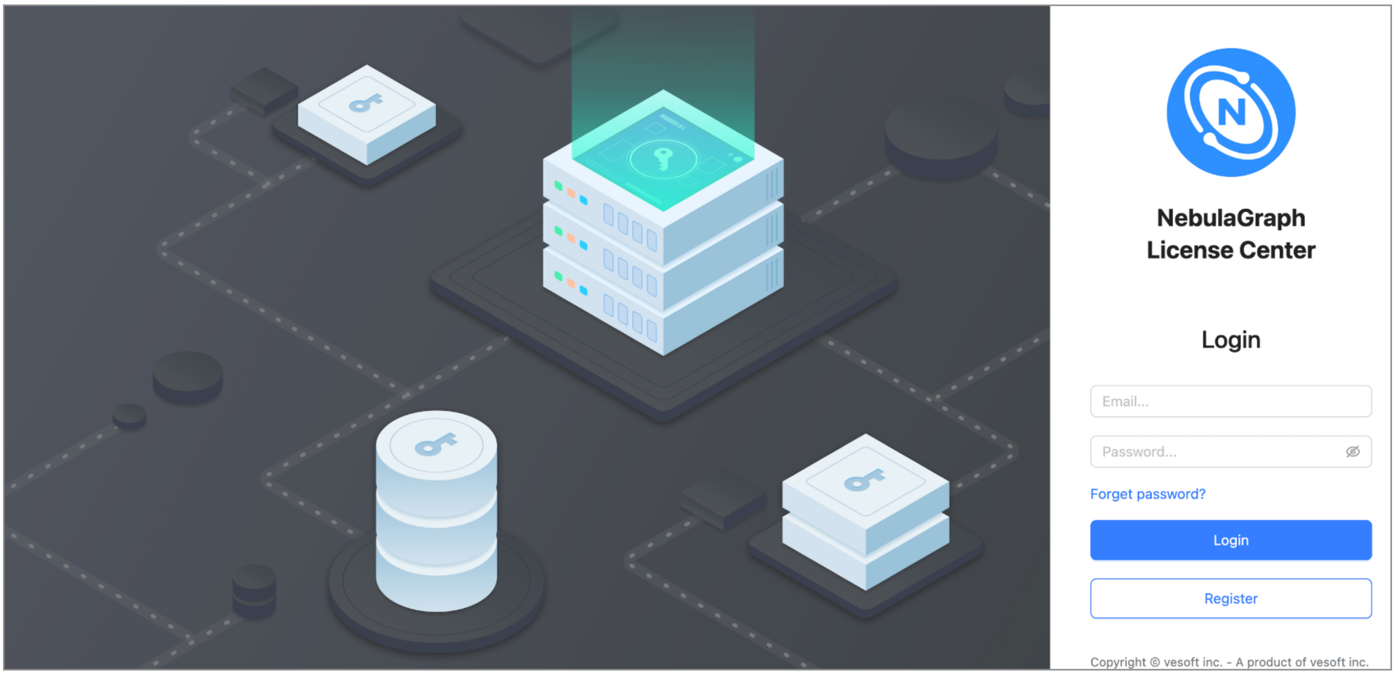
Preparations

To use LC, you must first purchase a NebulaGraph license. For more information, see Purchase a license.

Set up an LC account

To use LC, you must first set up an LC account.

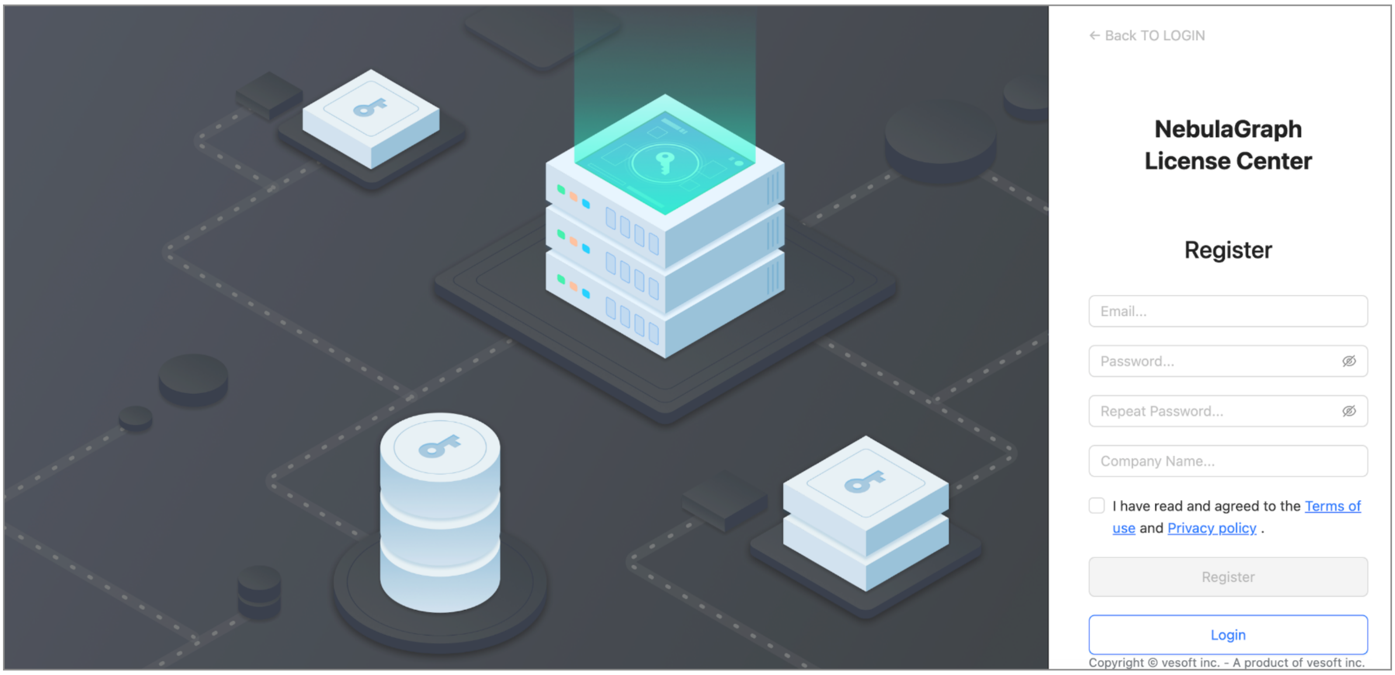
1. Go to the LC account setup page.



The entry to the LC account setup page varies depending on how you purchase your license:

- For purchasing a license on a cloud marketplace, go to the cloud marketplace service page and then click **Click here to set up your account**.
- For purchasing a license through Vesoft sales personnel, go to the email sent by Vesoft and then click **Setup License**.

2. Click **Register**.


← Back TO LOGIN
NebulaGraph
License Center
Register
Email...
Password...
Repeat Password...
Company Name...
□ I have read and agreed to the Terms of use and Privacy policy .
Register
Login
Copyright © vesoft inc. - A product of vesoft inc.

3. Fill in your email address, password, and company name, and tick the **I have read and agreed to the Terms of Use and Privacy Policy** box.

Caution

- Make sure the email address is valid, as you will receive a verification email after registration.
- The password must be between 12 and 30 characters long and contain numbers, letters, and special characters.

4. Click **Register** to complete the registration.
5. Open the verification email you received, and click on **Activate** to go to the LC login page.
6. Enter your email address and password, and click **Login** to log in to LC.

Bind LMID to generate a license key

After you log in to LC, you need bind the ID of your LM to generate a license key.

Caution

Each license can only be bound to one LMID, and the unbinding of LMIDs is not supported.

QUICKLY BIND LMID

You are guided to bind the LMID every time you log in to LC after you buy a new license. Binding the LMID is a prerequisite to generate a license key for using the license. You can also skip the quick binding and bind the LMID on the license information page.

The following describes how to quickly bind the LMID:

1. On the quick binding page, check the information of the purchased license, and click **Next**.
2. Bind the LMID by the following steps and then click **Next**.

a. Install the LM service. For how to install the LM service, see LM.

b. View the LMID. For how to view the LMID, see LM.

c. Fill in the LMID and select **Online** or **Offline**.

- Online

  Select the **Online** mode to generate an online license key.

- Offline

  Select the **Offline** mode to generate an offline license key. After you enter the offline license key into your LM, the LM service stores fixed license information.

  For more information, see License key.

d. Click **BIND LMID** to complete the binding.

3. View the license key generated after binding the LMID and click **Close** to complete the binding.
4. (Optional) Copy the license key and load it into the LM service. For how to load the license key, see LM.

Note

You can choose a license key type based on your LM accessibility.

- If your LM is accessible from the internet, you can select either **Online** or **Offline** mode. The **Online** mode is recommended, as it generates an online license key.
- If your LM is not accessible from the internet, then **Offline** mode is the only option available for generating an offline license key, as it can't reach out to the license server to validate the key itself.

BIND LMID ON THE LICENSE INFORMATION PAGE

If you skip the quick binding, you can still bind the LMID on the license information page.

1. On the targeted license details page, click **Bind License Manager ID**.
2. In the pop-up panel, enter the ID of your LM. For how to view LMID, see LM.
3. Select **Online** or **Offline**, and then click **CONFIRM** to bind the LMID.

- Select **Online** to generate an online license key, so that LM can get the latest license information from LC every 1 ~ 2 hours.
- Select **Offline** to generate an offline license key, which means LM obtains fixed license information. If you need to update the license information, you must obtain a new offline license key.

4. In the **License Key** section, view the license key generated after binding the LMID.
5. (Optional) Copy the license key and load it into the LM service. For how to load the license key, see LM.

License information

In the **LICENSES LIST** section of the LC homepage, click **VIEW DETAILS** to access the **License Info** page.

BASIC INFORMATION

- **LMID**: Indicates the ID of the LM service that you installed (If not bound, this field will be empty).
- **License Type**: Currently limited to the purchase of node-based resources.
- **Start At** and **Expire Time**: Indicates the active and expiry dates of the license.

RESOURCES

In the **Purchased Resources** section, you can view the purchased query and storage node quantities and statuses, as well as the complimentary software names and statuses.

LICENSE KEY

After you bind the LMID, a license key is automatically generated and the **License Key** section displays the license key information.

- Online license keys

  An online license allows you to obtain the latest license information from LC.

  When binding your LMID, select the **Online** mode to generate an online license key. After you load the key into the LM service, the LM can retrieve the latest license information regularly.

- Offline license keys

  Compared to an online license key, an offline license key contains fixed license information. If the license information is updated, a new offline license key must be obtained.

  When binding your LMID, select the **Offline** mode to generate an offline license key. Compared to an online license key, after you load an offline license key into your LM, the LM service stores fixed license information. If the license information is updated, a new offline license key must be obtained.

SUBSCRIPTION

This section is only displayed when you purchase a license on a cloud marketplace. In this section, you can view the subscription ID of the cloud marketplace where your license is purchased, your subscription platform account, product ID, and subscription details.

Last update: July 27, 2023

3.2.3 License Manager

A License Manager (LM) is an essential service that runs on a server for you to manage your license and license the NebulaGraph enterprise edition database and its associated software. You can use an LM client that communicates with the LM service to load license keys and view license information, including the license validity period and purchased nodes. By configuring the LM service address in the NebulaGraph database and its associated software, the validity of the license can be verified to ensure the normal use of the NebulaGraph database and its associated software.

This article introduces how to deploy and use an LM service in a Linux environment and how to configure it within the Nebula Graph database and its associated software. For information on how to deploy the LM in a K8s cluster, see Deploy LM.

Preparations

To use an LM, you need to make sure the following:

- You have purchased a NebulaGraph license.

- You have obtained the desired LM installation package.

  > **Note**
  >
  > LM installation packages are sent to you by email after you purchase a license.

- LM uses `9119` as the default port, make sure that port is not occupied.

Notes

- An LM is a single-process service. To ensure the reliability and continuity of the LM, it is recommended that you use systemd to manage the LM and set a restart policy for the LM.
- The time on the LM server must be synchronized with the services (including the NebulaGraph database and associated software) connected to the LM. If the times are not in sync, license verification will fail, which can prevent the service from being used.

Install and start LM

An LM can be installed on Linux amd64 or arm64 systems, or installed through Dashboard.

USING THE TAR PACKAGE

1. Unpack the LM TAR package.

```
tar -zxvf <name.tar.gz> -C <path>
```

- `<name.tar.gz>`: The name of the LM TAR package.
- `<path>`: The installation path for the unpacked LM. If the `-C` parameter is not specified, it defaults to the current directory.

2. Start the LM service using systemd.

a. Create the LM service file `/etc/systemd/system/nebula-license-manager.service` with the following contents:

```
[Unit]
Description=License Manager
[Service]
Type=simple
ExecStart=<path>/nebula-license-manager/nebula-license-manager
WorkingDirectory=<path>/nebula-license-manager
Restart=always
[Install]
WantedBy=multi-user.target
```

- `<path>`: Refers to the directory where the LM package is extracted.

b. Start the LM service:

```
sudo systemctl start nebula-license-manager
```

3. Set up LM to start automatically on boot.

```
sudo systemctl enable nebula-license-manager
```

USING THE RPM PACKAGE

1. Unpack the LM RPM package.

```
sudo rpm -ivh <name.rpm>
```

- `<name.rpm>`: The name of the LM RPM package.
- The default installation path is `/usr/local/nebula-license-manager`, which cannot be changed.

2. Start LM.

```
sudo systemctl start nebula-license-manager
```

3. Set up LM to start automatically on boot.

```
sudo systemctl enable nebula-license-manager
```

USING THE DEB PACKAGE

1. Unpack the LM DEB package.

```
sudo dpkg -i <name.deb>
```

- `<name.deb>`: The name of the LM DEB package.
- The default installation path is `/usr/local/nebula-license-manager`, which cannot be changed.

2. Start LM.

```
sudo systemctl start nebula-license-manager
```

3. Set up LM to start automatically on boot.

```
sudo systemctl enable nebula-license-manager
```

USING DASHBOARD

LM can be installed and started through Dashboard. For more information, see Connect to Dashboard.

View LM configuration file

The configuration file name for LM is `nebula-license-manager.yaml`.

- For RPM and DEB packages, the default path is `/usr/local/nebula-license-manager/etc/nebula-license-manager.yaml`.
- For the TAR package, the path is `nebula-license-manager/etc/nebula-license-manager.yaml` under the LM installation directory.

The contents of the LM configuration file are as follows:

```
Name: nebula-license-manager
Host: 0.0.0.0                 # The host address LM binds to.
Port: 9119                    # The port number LM listens on. The default is 9119.
Timeout: 3000                 # The timeout period for LM waiting for client requests, in milliseconds. The default is 3000 milliseconds.
DataPath: data                # The path for storing LM status data.
Notify:                       # LM notification related configuration.
  Mail:                       # Mail notification related configuration.
    Host: ""                  # SMTP server address.
    Port: 465                 # SMTP server port number.
    User: ""                  # SMTP email.
    Password: ""              # SMTP email password.
    To: []                    # The list of emails to receive notifications.
Log:                          # Logging related configuration.
  Mode: file                  # The logging mode. It can be file (output to file) or console (output to console). The default is file.
  Path: logs                  # The path for storing log files.
  Level: info                 # The log level. It can be debug, info, error, or severe. The default is info.
  KeepDays: 30                # The longest number of days to keep logs. The default is 30 days.
```

Use LM

After your LM starts, in the LM installation path you can use the LM CLI to view license information.

VIEW LM CLI VERSION

```
./nebula-license-manager-cli version
```

Note

When LM starts, its version information is printed in the logs.

LOAD A LICENSE KEY

After generating a license key, you need to use the LM client tool to load the license key.

```
./nebula-license-manager-cli load --key <license-key> --force
```

- `<license-key>`: The license key string, such as `MSY2-LGQ60-69521-XXXXX-XXXXX`.
- `--force`: Loads the license key without checking the current license status. This flag is optional.

VIEW LICENSE INFORMATION

```
./nebula-license-manager-cli info
```

- When the license key is not loaded, the output is as follows:

```
            LMID:  RUZB-XXXX
   LicenseStatus:  NotExist
```

- When the license key is loaded, the output is as follows:

```
            LMID:  RUZB-XXXX
   LicenseStatus:  Normal
      LicenseKey:  MM90U-9H4QO-W093M-XXXXX-XXXXX
            Type:  NODE
      Query Node:  3
    Storage Node:  3
        ExpireAt:  2023-06-25 12:00:00 +0800 CST
```

The information items of the license in the output are described as follows:

Items	Description
`LMID`	The ID of your LM. When you obtain a license key, this LMID needs to be bound. For more information, see Generate a license key.
`LicenseStatus`	The status of the license. It includes: `Normal` : The license can be used normally. `NotExist` : The license key does not exist. `Invalid` : The license key is invalid. `Syncing` : Synchronizing the license information from LC. `Expiring` : The license is about to expire. `Expired` : The license has expired.
`LicenseKey`	An encrypted string containing authorization information, which is the only credential for you to obtain the authorization of the NebulaGraph database and its associated software. For details, see License key.
`Type`	The type of resources purchased. Currently, only node-based resources can be purchased.
`Query Node`	The number of query nodes purchased
`Storage Node`	The number of storage nodes purchased
`ExpireAt`	The expiration time of the license.

SYNCHRONIZE LICENSE INFO

When the license key loaded into LM is in online mode, the LM periodically synchronizes the license information from LC every one to two hours. You can also manually synchronize the license key using the following command.

```
./nebula-license-manager-cli sync
```

CHECK THE LICENSE QUOTA USAGE

You can run the following command to check the current usage of the license quota (the number of nodes purchased) and the usage status of the associated software.

```
./nebula-license-manager-cli usage
```

VIEW THE LICENSE INFORMATION ON A SPECIFIED LM

For a database administrator (DBA), there may be a need to view the license information on a specified LM. To achieve this, run the following command:

```
./nebula-license-manager-cli <command> --addr <host>:9119
```

- `<command>` : The command to be executed. Options include `info` , `usage` , `sync` , and `load --key <license-key> --force` .
- `<host>` : The IP address of the host where the specified LM is located.

Monitor LM

MONITOR LM STATUS

You can use monitoring tools to monitor the status of the LM service.

- Use Dashboard Enterprise

  When LM is running normally, the **License Manager** page displays the status of LM as **Running**, otherwise, it displays **Exited**. For more information, see License Manager.

- Use Prometheus

  You need to configure the Prometheus server before monitoring LM.

  Add the following configuration to the Prometheus configuration file. For more information about the Prometheus configuration file, see Prometheus Configuration.

  ```
  ...
  - job_name: license-manager
    scrape_interval: 15s              # The interval for pulling data.
    metrics_path: /metrics            # The monitoring metrics path of LM, which is `/metrics`.
    scheme: http                      # The protocol type of LM, which is HTTP.
    static_configs:                   # The address and port of LM (default 9119).
    - targets:
      - [<ip:lm_port>]
  ...
  ```

  After the configuration is complete, you can monitor the status of LM through its built-in metric `up`. If the value is `1`, it means that LM is running normally; if the value is `0`, it means that LM is not running. To view the metric, enter `up` in the Prometheus query box as shown below:

  ```
  up{instance="<ip:lm_port>", job="job_name"}
  ```

- `<ip:lm_port>`: The IP address and port of LM.
- `job_name`: The name of the job.

  For example, up{instance="192.168.8.xxx:9119", job="license-manager"}.

> **Note**
>
> By default, LM uses port `9119`. If you need to change the port number, you can modify the value of the `Port` field in the LM configuration file above, or modify the value of `port` in the YAML file of Deploying LM in K8s.

VIEW LM METRICS

You can view the built-in metrics of LM through the following URL:

```
http://<ip:lm_port>/metrics
```

- `<ip:lm_port>`: The IP address and port of LM.

You can also collect LM metrics data through monitoring tools. To use Prometheus, you need to configure the Prometheus server before collecting LM metrics. For more information, see the above section **Monitor LM status**.

Configure connection to LM

CONFIGURE LM IN NEBULAGRAPH

In the NebulaGraph database Meta service configuration file (`nebula-metad.conf`), set the `license_manager_url` value to reflect the IP address of the LM host and port number `9119` in the format like `192.168.8.xxx:9119`. For more information, see Meta service configuration.

After the configuration is complete, restart the Meta service.

CONFIGURE LM IN EXPLORER

In the Explorer installation directory, enter the `config` folder and modify the `app-config.yaml` file. Set the value of `LicenseManagerURL` to reflect the IP address of the LM host and port number `9119` in the format like `192.168.8.xxx:9119`.

After the configuration is complete, restart Explorer. For more information, see Deploy Explorer.

CONFIGURE LM IN DASHBOARD

In the Dashboard installation directory, enter the `etc` folder and modify the `config.yaml` file. Set the value of `LicenseManagerURL` to reflect the IP address of the LM host and port number `9119` in the format like `192.168.8.xxx:9119`.

After the configuration is complete, restart Dashboard. For more information, see Deploy Dashboard.

CONFIGURE LM IN ANALYTICS

In the Analytics installation directory, enter the `scripts` folder and modify the `analytics.conf` file. Set the value of `license_manager_url` to the IP address of the LM host and the port number `9119`, for example, `192.168.8.xxx:9119`.

After the configuration is complete, run `./run_pagerank.sh` in the `scripts` folder to start the Analytics service. For more information, see NebulaGraph Analytics.

CONFIGURE LM IN NEBULAGRAPH OPERATOR

- When deploying the cluster using Kubectl, configure the address and port of the LM through the `spec.metad.licenseManagerURL` field in the cluster configuration file. For more details, see Deploying with Kubectl.
- When deploying the cluster using Helm, specify the address and port of the LM with `--set nebula.metad.licenseManagerURL`. For more details, see Deploying with Helm.

FAQ

Q: Can I change the host on which my LM is located?

A: No. The LM is bound to the host where it is installed. If you need to change the host, or the host is unable to be used, you need to contact Vesoft sales to rebind the LMID.

Last update: July 14, 2023

3.3 Purchase a NebulaGraph license

To utilize the features of the NebulaGraph database and associated software, you must obtain a NebulaGraph license. The license can be procured either via the cloud marketplace or by directly contacting Vesoft sales.

Currently, AWS Marketplace is the only cloud marketplace from which a license can be obtained. This article assists you in purchasing a NebulaGraph license on the AWS Marketplace.

3.3.1 Preparations

You have registered an AWS Marketplace account and logged in.

3.3.2 Steps

> **Note**
>
> Before purchasing a license on the AWS Marketplace, it is recommended that you contact Vesoft sales for detailed information about the license.

1. Open the AWS Marketplace NebulaGraph Enterprise (by Node) service page.
2. Click **View purchase options** to enter the license purchase contract signing page.
3. Configure the contract items, which include the license validity period, auto-renewal setting, and the number of nodes to be purchased.

- **How long do you want your contract to run**: Choose the validity period for the license, either 1 month or 1 year.
- **Renewal Settings**: Whether to automatically renew the license after its validity period.
- **Yes**: The license will be automatically renewed after its validity period.
- **No**: The license will not be automatically renewed after its validity period.
- **Contract Options**: Select the number of resources to purchase, currently supporting the purchase of query nodes and storage nodes.

4. Click **Create contract**.
5. In the pop-up panel, confirm purchase information and click **Pay now**.
6. Click **Set up your account** for LC registration and to start managing the license. For details, see set up an LC account.

   You can view the license information on LC.

> **Note**
>
> Once the registration process is finished, you will receive an email from Vesoft within one business day containing the complete NebulaGraph packages, which include not only the database, but also other software such as LM, NebulaGraph Explorer, and more.

3.3.3 Next to do

After purchasing a license, you must generate a license key and then load it into the LM service. Following this, the NebulaGraph database and its associated software will verify the validity of the license key through the LM service at startup. If the license key is valid, the NebulaGraph database and associated software can operate normally. The following steps describe how to generate and load a license key:

- Install LM
- Generate the license key
- Load the license key

For more information about how to use a license, see Licensing process.

Last update: July 5, 2023

3.4 Manage licenses

This article provides instructions on managing licenses, including license renewal, license node expansion, and viewing online and offline license keys.

3.4.1 Preparations

- You have generated a license key on LC
- You have loaded the license key into the LM

3.4.2 Renew licenses

- For licenses purchased through Vesoft's sales team, you need to contact the sales team to renew them.
- For licenses purchased on the cloud marketplace platform, follow these steps for renewal:

a. On the LC homepage, navigate to the **LICENSES LIST** section, and find the target license card.

b. Click **RENEW** to enter the cloud marketplace renewal page.

c. Click **Modify renewal terms** and select the renewal period, which can be either 1 month or 1 year.

d. Click **Modify renewal**.

After a successful renewal, if the license key loaded in the LM is online, your LM will automatically synchronize the license information. If it's offline, you need to copy the new offline license key from LC, and then reload this new offline license key into your LM.

3.4.3 Expand license node count

- For licenses purchased through sales team, you need to contact the sales team to increase the number of nodes.
- For licenses purchased on the cloud marketplace platform, follow these steps to expand the license node count:

a. In the **LICENSES LIST** section of the LC homepage, find the target license card.

b. Click **RENEW** to enter the cloud marketplace renewal page.

c. Click **Upgrade current contract** and select the number of nodes to be added.

d. Click **Modify current contract**.

After successfully expanding the node count, if the license key loaded in the LM is online, the LM will automatically synchronize the license information. If it's an offline license key, you need to copy the new offline license key from LC, and then reload this new offline license key into the LM.

3.4.4 View online and offline license keys

1. In the **LICENSES LIST** section of the LC homepage, find the target license card.
2. Click **VIEW DETAILS** to enter the license information page.
3. In the **Basic Info** section, click **EDIT LM ID**.
4. In the pop-up panel, select **Online** or **Offline**, and click **CONFIRM**.
5. In the **LICENSES KEY** section, view the corresponding license key in the selected mode.

Last update: July 27, 2023

4. Quick start

4.1 Quickly deploy NebulaGraph using Docker

You can quickly get started with NebulaGraph by deploying NebulaGraph with Docker Desktop or Docker Compose.

Using Docker Desktop **Using Docker Compose**

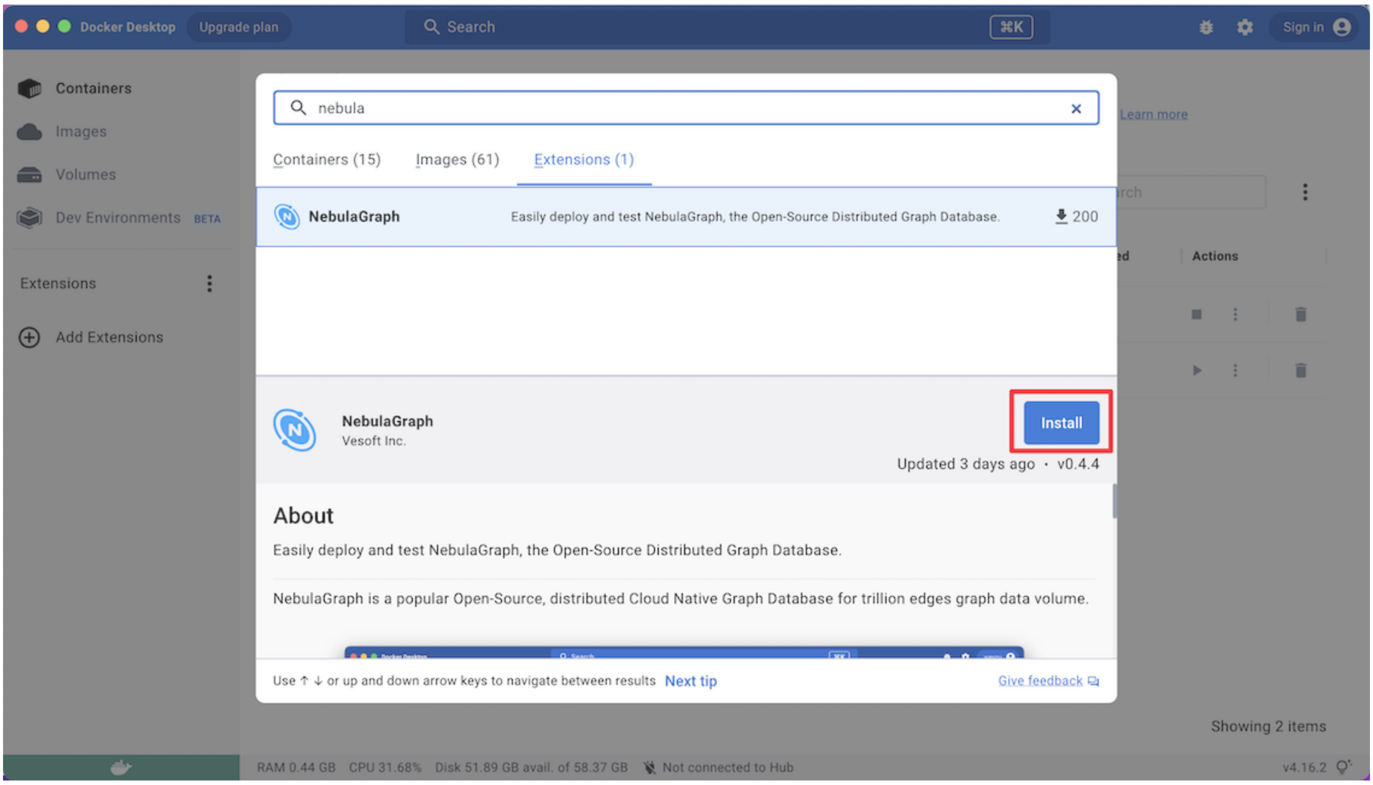
NebulaGraph is available as a Docker Extension that you can easily install and run on your Docker Desktop. You can quickly deploy NebulaGraph using Docker Desktop with just one click.

1. Install Docker Desktop

- Install Docker Desktop on Mac
- Install Docker Desktop on Windows

> **Caution**
>
> To install Docker Desktop, you need to install WSL 2 first.

2. In the left sidebar of Docker Desktop, click **Extensions** or **Add Extensions**.
3. On the Extensions Marketplace, search for NebulaGraph and click **Install**.



nebula

NebulaGraph Easily deploy and test NebulaGraph, the Open-Source Distributed Graph Database.

NebulaGraph
Vesoft Inc.

Updated 3 days ago • v0.4.4

About

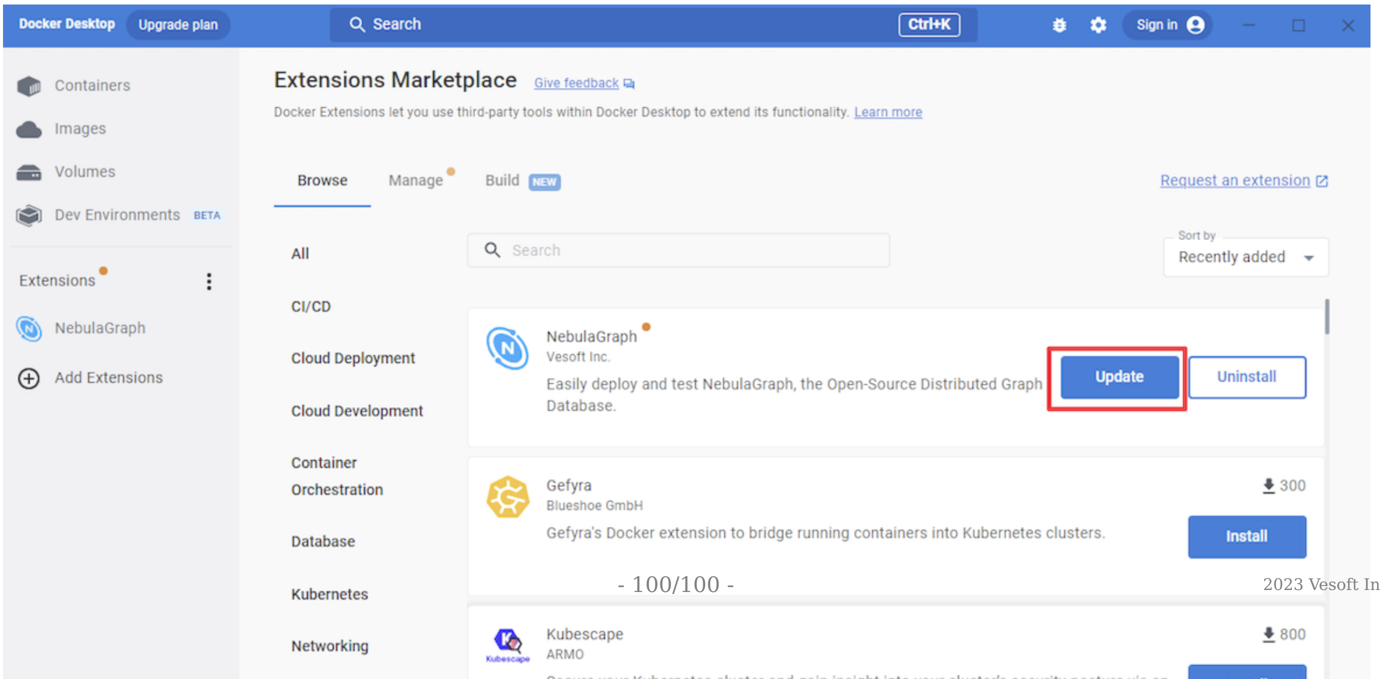
Easily deploy and test NebulaGraph, the Open-Source Distributed Graph Database.

NebulaGraph is a popular Open-Source, distributed Cloud Native Graph Database for trillion edges graph data volume.

Use ↑ ↓ or up and down arrow keys to navigate between results

RAM 0.44 GB CPU 31.68% Disk 51.89 GB avail. of 58.37 GB Not connected to Hub

Click **Update** to update NebulaGraph to the latest version when a new version is available.



Extensions Marketplace

Docker Extensions let you use third-party tools within Docker Desktop to extend its functionality. Learn more

NebulaGraph
Vesoft Inc.
Easily deploy and test NebulaGraph, the Open-Source Distributed Graph Database.

Gefyra
Blueshoe GmbH
Gefyra's Docker extension to bridge running containers into Kubernetes clusters.

Kubescape
ARMO