Database Management

| Big data, distributed DB, and NoSQL

Ling-Chieh Kung

Department of Information Management
National Taiwan University

(This set of slides was made with the help from
彭曉蓮 and 盧沛宏, two B11 NTU IM students)

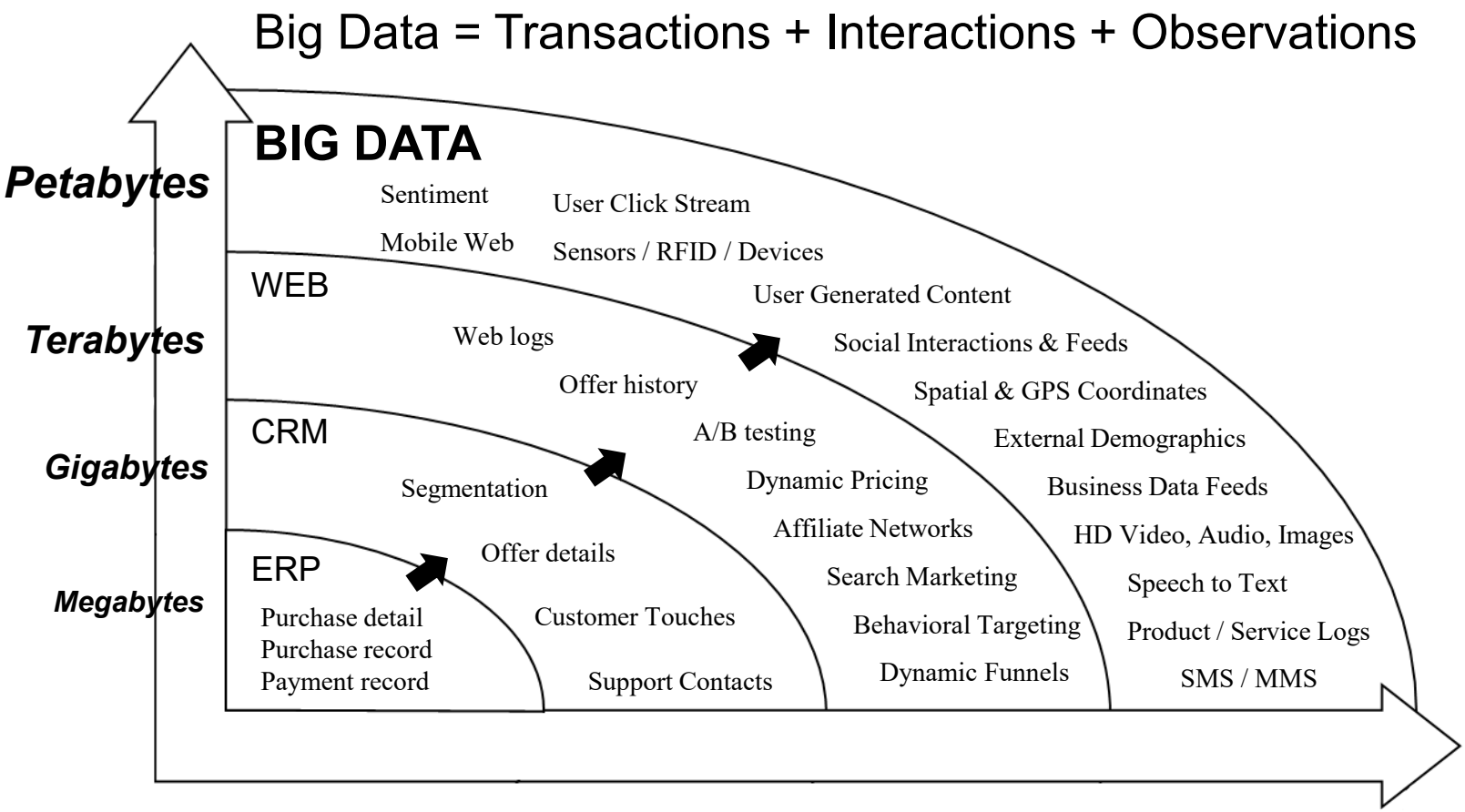
Agenda

- **Big data**
- Distributed databases
- NoSQL

What is big data?

- Wikipedia: "**Big data** (大數據) is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications."
- Early applications include Gmail, Facebook, etc.
  - Most companies' transaction data are not considered "big".
  - The data behind a **technology company's digital products/services** are considered "big".
- The challenges include capture, curation, storage, search, sharing, transfer, analysis, and visualization.

The 3V of big data



The 3V of big data: volume (scale)

- Data **volume**: From 0.8 zettabytes (2009) to 50zb (2020).
  - Machine-generated data, contents on Internets, digital multi-media, etc.
- Data volume is increasing **exponentially**.

The 3V of big data: variety (complexity)

- Traditionally, we only work on **structured data**.
  - Relational data (tables/transaction).
- Today we have huge variety of structured and **unstructured data**:
  - Text data (Web).
  - Semi-structured data (XML) .
  - Graph data (e.g., social network)
  - Streaming data.
  - Big public data (online, weather, finance, etc.)
- A single application can be generating/collecting many types of data.
- To extract knowledge, all these types of data need to linked together.

The 3V of big data: velocity (speed)

- Data is being generated fast and needs to be processed at high **velocity**.
  - Online (real-time) data analytics.
  - Late decisions mean missed opportunities.
- Examples:
  - **Location-based service:** Based on your current location, your purchase history, and what you like, the system sends promotions right away for stores close to you.
  - **Healthcare monitoring:** With sensors monitoring your motion and body conditions, any abnormal measurements require an immediate reaction.
- The value is no longer hindered by the ability to collect data.
  - It is by the ability to **manage, analyze, summarize, and visualize** the collected data in a timely manner and in a scalable fashion.

The 4V, 5V, … of big data

- Some people also add some other Vs as the characteristics of big data.
  - **Value**.
  - **Veracity** (quality, accuracy, correctness).

Three levels of data analytics/processing

- **OLTP**: Online transaction processing (relational DBMS).
- **OLAP**: Online analytical processing (data warehousing).
- **RTAP**: Real-time analytics & processing (big data architecture/technology).

Application (?): YouBike relocation

交通局向捷安特調閱熱門YouBike站完整借還車數據，分平日、假日分析每小時平均借還車數量，發現站點鄰近捷運、觀光區、辦公場所、圖書館等不同區域，借還車有不同模式，可作為車輛調度管理參考。

大數據分析發現，捷運淡水站的YouBike假日下午4點到6點還車數暴量，比平時高出千輛之多。由於淡水與鄰近的北市北投、關渡也有距離，要把還車潮後暴量的YouBike馬上外運難度高。

目前業者已在捷運淡水站周邊覓地設置臨時調度場，可就近移走車輛，再慢慢運出。交通局在淡水增設YouBike站，已有11站，若淡水站已無還車空間，遊客還能轉移陣地。

交通局表示，利用大數據分析結果，可在各站借還車高峰前預布調度人力因應，減少民眾遭遇「無車可借」、「無位可還」。隨YouBike設站增加，使用模式會不斷變化，未來會持續追蹤數據分析。

"Big" data analysis

- 建構預測與調度系統的必要條件：收集借還記錄
  - 幾點幾分、在哪、借或還
  - 這些**交易資料**（transaction data）挺好的
- 這就是**大數據**（big data）了嗎？
- 還有**外部資料**（external data）：
  - 氣溫、是否降雨、當天是否放假…
- 還有**行為資料**（behavioral data）：
  - 有多少人想借借不到？想還還不了？
  - 有多少人用 app 查詢剩餘車輛數？

Application: Pay how you drive (PHYD)

Pay How You Drive Insurance Guide

August 5, 2013 | Robert Prime | No Comments | Guides for Telematics

Like this post? Like +1 Tweet Pin it 3 Share

Pay how you drive insurance is a special type of motor vehicle insurance that take into consideration how you drive. This simply means your driving habits dictate your premiums i.e. speeding, braking, parking, positioning, stops e.t.c. If you happen to be a rough driver who speeds, breaks suddenly and positions themselves near obstacles/objects, you will obviously pay higher premiums compared to the careful driver who breaks gradually and leaves enough space between the car and objects. Pay how you drive insurance is simply aimed at

(https://www.telematics.com/pay-how-you-drive-insurance-guide/)

Application: apparel retailing

- 每件衣服上都有 RFID
  - 藏在防盜扣裡
  - 衣服的身分證：材質、樣式……
- 「只要有人試穿，衣服就會『告訴』店家」

(http://www.businesstoday.com.tw/article/category/80394/post/201509170022/物聯網趨勢超熱%20全球都在靠它賺錢！)

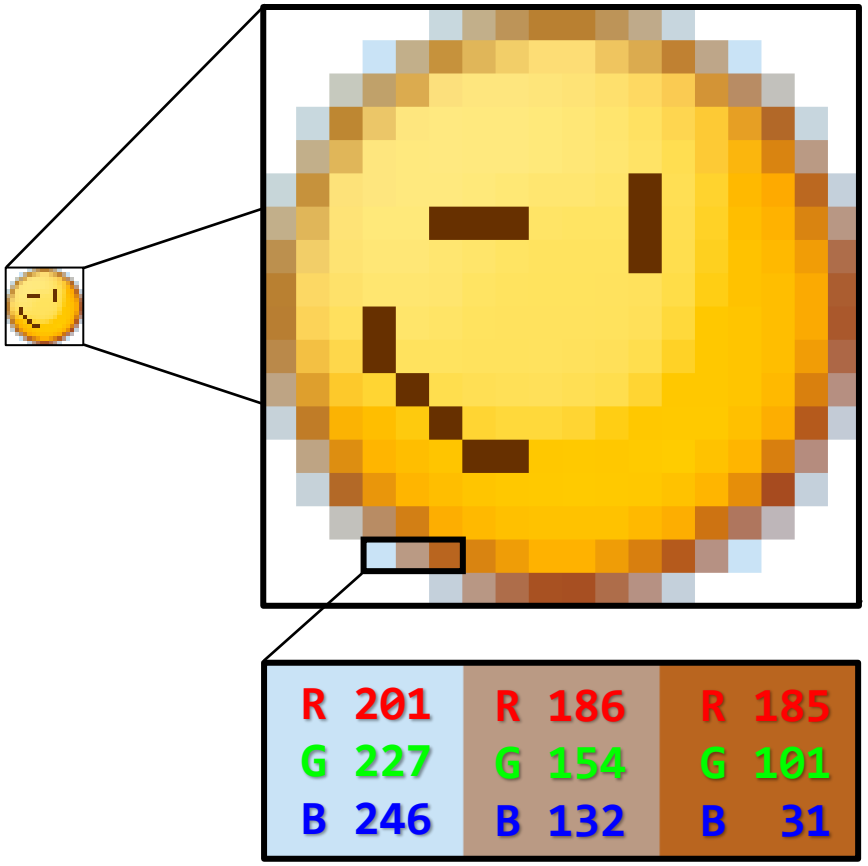
Unstructured data

- 上述資料大致上都屬於**結構化資料**（structured data）
  - 好好地放在一張表裡的數字或簡單文字
- 大數據分析也強調**非結構化資料**（unstructured data）分析
  - 文字、聲音、圖片、影片……
  - 常見應用：文意辨識、聲音轉文字、圖片辨識
- 分析上，可能可以先把非結構化資料**轉成**結構化資料
  - 用機器學習原理做訓練以建構預測模型
  - 有很多新的挑戰

(https://ipetgroup.com/article/uSPXllGGYmI.html)

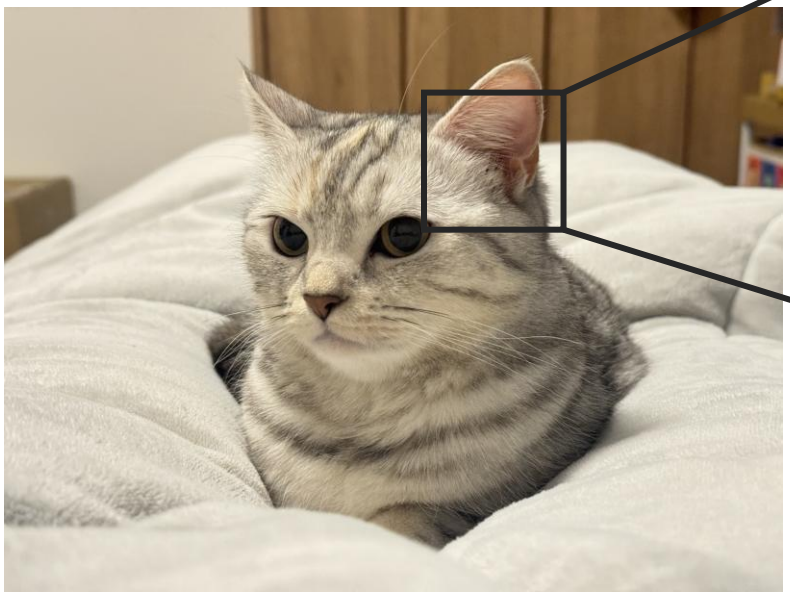
Storing a picture in numbers

- 點陣圖（Bitmap）為電腦儲存圖片的格式之一
  - 一張圖片為許多小點所構成
  - 每個小點使用三個數字代表這個點的顏色
  - 這三個數字代表紅色（R）、綠色（G）與藍色（B）



Storing a picture in numbers

The Problem: Sematic Gap



```
[ 79  79  79  80  79  73  63  55  50  52  57  95 149 145 110  67  73  82  78  88]
[ 81  80  80  80  78  72  63  56  51  51  55  96 152 142 107  76  71  79  77  89]
[ 81  80  79  78  76  71  64  60  56  51  64 113 158 129  88  68  74  78  77  89]
[ 85  83  81  78  74  69  64  61  55  41  74 133 157 113  80  72  81  80  78  89]
[ 90  88  84  79  74  69  65  62  65  43  90 147 142  94  77  78  82  80  80  90]
[ 85  83  79  76  74  70  66  64  70  59 115 155 129  82  72  73  78  78  81  90]
[ 83  80  77  75  73  71  66  62  57  73 131 149 120  84  74  73  76  77  81  87]
[ 92  87  82  78  75  70  63  57  59  93 141 134 107  80  69  70  77  77  80  83]
[ 87  88  85  79  69  58  57  64  65 110 144 124  94  68  67  68  74  77  79  80]
[ 84  80  77  74  67  61  60  63  67 135 149 104  76  69  73  64  70  73  78  84]
[ 86  80  77  75  67  62  62  63  81 149 139  98  75  64  67  71  74  75  82  89]
[ 85  80  81  79  67  60  63  67  97 150 121  97  84  62  58  74  80  81  85  88]
[ 83  77  75  76  69  60  63  71 114 151 109  84  80  68  62  68  78  82  84  83]
[ 89  78  69  71  72  62  64  81 138 146 102  76  74  70  70  70  72  80  84  83]
[ 90  80  67  69  73  62  71 105 150 129  97  77  72  68  77  77  72  79  86  88]
[ 80  76  65  66  71  60  80 132 141 113  96  76  67  69  83  75  75  79  85  89]
[ 76  78  71  72  66  58  95 155 130 100  87  64  57  76  75  77  73  77  84  92]
[ 78  78  69  71  64  63 107 153 123  85  77  64  63  76  77  80  74  75  79  85]
[ 85  79  65  65  58  72 125 152 114  68  69  64  69  73  76  79  77  76  77  81]
[ 87  79  66  67  62  89 142 145 104  60  69  67  73  69  71  72  77  76  76  82]
[ 81  76  70  73  76 111 150 130  90  59  75  68  72  68  68  67  74  73  76  84]
[ 79  74  68  66  81 125 148 121  75  61  78  68  70  70  67  69  73  74  79  86]
[ 80  74  68  60  86 138 144 117  68  65  78  67  68  71  64  73  76  77  80  85]
[ 78  75  75  67 101 152 142 112  67  68  76  66  66  71  60  74  77  77  78  80]
[ 77  76  71  74 119 150 120 101  67  70  66  68  73  65  62  76  78  81  82  81]
[ 78  73  70  84 131 144 112  88  66  69  65  64  68  61  60  73  79  76  78  86]
[ 83  74  73 100 149 138 105  73  65  70  66  62  63  59  60  70  77  74  75  82]
[ 84  77  79 114 159 131  99  62  63  70  68  63  62  62  63  70  76  79  78  74]
[ 81  79  86 122 157 120  93  57  61  68  69  66  65  66  68  71  77  82  83  78]
[ 78  83  97 131 150 110  87  61  60  64  68  68  67  68  70  71  76  77  81  88]
```

An image is just a big grid of numbers between [0, 255];

e.g. 800 x 600 x 3 (3 channels RGB)

非結構化資料的挑戰

- 由非結構化資料轉成的結構化資料，通常變數量超大
  - 一張 800×600 的點陣圖，結構化後會有 1,440,000 個自變數
- 要做應用（例如影像辨識），又需要超多資料
- 這大量的資料自然構成大數據

From transaction data to big data

- **OLTP** systems that deal with **transaction data** are still very important.
  - All organizations/companies, tech or non-tech, need one (or many).
  - Data are mainly structured and generated by transactions made by human beings.
- However, many modern applications now need to deal with **big data**.
  - Technology companies need them.
  - Unstructured data, Internet data, machine-generated (behavioral) data, etc.
- The three Vs of big data: volume, variety, and velocity.
  - We need **distributed databases** (**DDB**, 分散式資料庫).
  - We need **more flexible** database architectures, e.g., **NoSQL** (非關聯式資料庫).
- Distributed databases and NoSQL can be used **individually**.
  - They are often used **together**.

Agenda

- Big data
- **Distributed databases**
- NoSQL

Distributed databases and scalability

- A **distributed database** is a collection of multiple logically interrelated databases distributed over a computer network.
  - A **distributed database management system** (**DDBMS**) is a software system that manages a distributed database.
  - Hopefully the distribution is **transparent** to users, i.e., users are not aware of the fact that the database is actually distributed.
- In a distributed database, each machine (or a set of machines tightly connected as a unit in a physical location) is also called a **node**.
  - These nodes are connected in a "graph".
- When is distribution a good idea?

Vertical and horizontal scalability

- When we have a lot of data, there are two ways to enhance **efficiency**:
  - We may **vertically scale** the database by expanding the capacity (e.g., storage, computation power, etc.) of the machine hosting the database.
  - We may **horizontally scale** the database by expanding the number of machines hosting the (distributed) database.
- Which one to use?
  - For a traditional environment (e.g., a database recording transaction data in a middle-scale company), vertical scalability may be good enough.
  - For big data, vertical scalability is not enough (or too expensive), and horizontal scalability is required.

A very naïve example

- Consider the following SQL program:

```
SELECT rt.arrive_station_id, m.name
FROM RESERVED_TICKET AS rt
    JOIN MEMBER AS m ON rt.citizen_id = m.citizen_id
WHERE rt.travel_date = '2023-08-01'
    AND rt.depart_station_id <= 1000
    AND m.name LIKE '% A%'
```

- If all data are in the same node:
  - Filtering data in `RESERVED_TICKET`.
  - Filtering data in `MEMBER`.
  - Join the previous two results.
- Not bad, but may we do better?

A very naïve example

- We may create a DDB in the following **three nodes**:
  - Node 1: Storing all tuples in `RESERVED_TICKET`.
  - Node 2: Storing all tuples in `MEMBER`.
  - Node 0: Receiving a user's query, asking the two nodes to send required tuples and columns to itself, conducting the join operation.
- This is under the **master-slave architecture**, where node 0 is the **master node** and nodes 1 and 2 are the **work nodes** (**slave nodes**).
- With nodes 1 and 2 processing their data **parallelly**, node 0 may return the result to the user faster.
  - Parallel computing on a single node can also achieve this.
  - However, the cost of having a strong machine is more expensive than having multiple okay machines in general.

Reliability (availability)

- Making a database distributed can also enhance **reliability** (**availability**).
  - The degree of reliability can be defined as the proportion of time that the system is functional, the proportion of requests that can be completed, etc.
  - Distribution makes it easier (less costly) to achieve high reliability because now **single point of failure** does not make the whole system down.
  - Some people think reliability and availability are different.
- Consider the previous example:
  - If node 1 fails, queries that are not related with `reserved_ticket` can still be done.
  - If node 2 fails, queries that are not related with `member` can still be done.
  - Distribution does increase reliability.
- If node 0 fails, however, the whole system is down.

Autonomy

- Note that in the previous example, the failure of node 0 is fatal.
  - For a distributed system in a master-slave architecture, the failure of the master node is fatal.
- We say the degree of **autonomy** of the example DDB is low.
  - In general, if the degree of autonomy of a DDB is high, most nodes can operate independently (i.e., even if other nodes fail).
- Regarding reliability, a DDB under a **master-master architecture** (also called peer-to-peer or fully distributed) is better.
  - All nodes are autonomous.
  - No need to worry about single point of failure.
  - Can be achieved only with a strong DDBMS with high coordination cost.
  - The design of a distributed system is beyond the scope of this course.

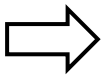
Fragmentation and replication

- When we have multiple nodes, data **fragmentation** (分割) and **replication** (複製) should be considered and applied.
- Fragmentation is to split the data in a table into **fragments**.
  - Mainly to enhance efficiency and make recovery easier.
  - In many cases different fragments are in different nodes. However, different fragments may also appear in a single node. This may still enhance efficiency.
- Replication is to make certain data stored on **multiple nodes**.
  - Mainly to increase availability and reliability.
  - Can also enhance efficiency (by, e.g., reducing the amount of data transfer among nodes.

Horizontal and vertical fragmentation

- There are two kinds of fragmentation (and they can be applied together).
- **Horizontal fragmentation** (水平分割, also known as **sharding**, 資料庫分片) is to split the set of **rows** in a table into multiple subsets of rows. Each subset forms a **horizontal fragment** (or a **shard**).

ID	A	B	C
1			
2			
3			
4			
5			



ID	A	B	C
1			
2			
3			

ID	A	B	C
4			
5			

- Splitting a database **according to tables** (i.e., there are some tables stored only in a subset of nodes) is also called horizontal fragmentation.

Horizontal and vertical fragmentation

- When we do horizontal fragmentation:
  - If a fragmented table is referred by other table(s) through foreign keys, **derived horizontal fragmentation** (衍生水平分割) applies the same partitioning of the referred table relating to those referring tables.
  - For example, if a table `STORE` is horizontally fragmented so that the records of all stores in the eastern area are in site 1 and all others are in site 2, then typically for all those sales records in the table `SALES`, those sales records occurred in eastern stores are also stored in site 1 and all other sales records are in site 2.

Horizontal and vertical fragmentation

- Vertical fragmentation is to split the set of non-primary-key **columns** in a table into multiple subsets of columns. Each subset and the primary-key column(s) together form a **vertical fragment** (垂直分割).

ID	A	B	C
1			
2			
3			
4			
5			

⇨

ID	A
1	
2	
3	
4	
5	

✚

ID	B	C
1		
2		
3		
4		
5		

- If there are n fragments, the primary-key column(s) are replicated n times to make sure that the fragment may be combined when needed.
- Non-primary-key columns may also be replicated.

Example: the database RETAIL

EMPLOYEE

id	name	gender	birthday	monthly_salary	supervisor_id	store_id
A473564811	Chih-Yuan Lee	M	1994/10/1	63000	A465113807	1
L645977505	Tzu-Hsuan Wang	W	1990/12/30	150000	(null)	0
A465113807	Shih-Han Chang	W	1998/6/3	92000	L645977505	1
S657432198	Chen-Hsi Liu	M	1995/1/10	102000	L645977505	2
S153004821	Po-Lin Chen	M	2000/5/17	48000	S657432198	2
E972346850	Hao-Tien Chu	M	2001/8/22	48000	S657432198	2
A521870632	Ching-Yi Chao	W	1998/5/24	53000	A465113807	1
A671224980	Chieh Sun	M	1996/4/9	58000	A465113807	1

STORE

id	postal_code	addr_str	mgr_id
0	407	No. 10, Section 1, Zhongke Road, Xitun District, Taichung City	L645977505
1	100	No. 113, Section 4, Chongqing South Road, Zhongzheng District, Taipei City	A465113807
2	803	No. 7, Section 3, Dayong Road, Yancheng District, Kaohsiung City	S657432198

PRODUCT

id	name
1	VCare Shampoo
2	Skin Moisturizing Lotion
3	Soft Tissue
4	Baby Comfort Diapers
5	Antibacterial Handwash
6	Clean Beauty Wipes

STORE_PHONE

store_id	phone_number
0	(04)2483-7890
0	(04)5566-8253
0	(04)7364-9901
1	(02)6667-5112
1	(02)9841-0037
2	(07)548-6634

Example: the database RETAIL

STORE_PRODUCT

store_id	product_id
1	1
1	2
1	3
1	4
1	5
1	6
2	1
2	2
2	3
2	5

SALES

receipt_no	date_time	store_id
EC23456792	2025/1/2 13:04	1
EC67890124	2025/1/2 15:14	1
ER78901246	2025/1/3 9:25	2
ER34567891	2025/1/3 11:35	2
EC10345786	2025/1/3 14:48	1
EC48786019	2025/1/4 10:24	1
ER84785499	2025/1/4 12:08	2
EC78901235	2025/1/4 13:40	1
EC67890514	2025/1/4 18:23	1
EC65414902	2025/1/5 16:50	1

SALES_DETAIL

receipt_no	product_id	unit_price	qty
EC23456792	1	200	1
EC67890124	1	200	3
EC67890124	2	190	3
EC67890124	5	110	5
ER78901246	1	200	1
ER34567891	2	190	2
ER34567891	3	35	10
EC10345786	6	50	6
EC48786019	5	110	2
ER84785499	1	200	5

Business requirements

- Suppose that the retailer has three sites, one for each store.
- Sites 1 and 2 are for store 1 and 2, respectively.
  - At each sites, we expect frequent access to `EMPLOYEE` and `SALES` for the employees who work in that stores and the data **related to that store**, as well as the products it sells.
  - These local sites mainly access only **a subset of attributes** of employees, including `id`, `name`, `gender`, `supervisor_id`, and `store_id` of `EMPLOYEE`.
- Site 0, the headquarter, is for store 0.
  - Site 0 accesses **all** employee and sales information regularly.

One way to fragment the database

- According to the requirements, we can do the following fragmentation.
  - **The whole database** can have one replication stored at site 0.
  - We **horizontally fragment** `STORE` by its primary key `id`. Then we apply derived fragmentation to the `EMPLOYEE`, `SALES`, `SALES_DETAIL`, and `STORE_PRODUCT` relations based on their foreign keys referencing `STORE.id`.
  - We **vertically fragment** the resulting `EMPLOYEE` fragments to include only the attributes `id`, `name`, `gender`, `supervisor_id`, and `store_id`.
- In this way:
  - Fragmentation and replication are applied at the same time.
  - Many queries at sites 1 and 2 can be satisfied **at local sites** with smaller tables. Efficiency is enhanced.
  - All sites have some degrees of autonomy. Reliability is improved.

The data stored in store 1

EMPLOYEE

id	name	gender	birthday	monthly_salary	supervisor_id	store_id
A473564811	Chih-Yuan Lee	M	1994/10/1	63000	A465113807	1
A465113807	Shih-Han Chang	W	1998/6/3	92000	L645977505	1
A521870632	Ching-Yi Chao	W	1998/5/24	53000	A465113807	1
A671224980	Chieh Sun	M	1996/4/9	58000	A465113807	1

SALES

receipt_no	date_time	store_id
EC23456792	2025/1/2 13:04	1
EC67890124	2025/1/2 15:14	1
EC10345786	2025/1/3 14:48	1
EC48786019	2025/1/4 10:24	1
EC78901235	2025/1/4 13:40	1
EC67890514	2025/1/4 18:23	1
EC65414902	2025/1/5 16:50	1

STORE_PRODUCT

store_id	product_id
1	1
1	2
1	3
1	4
1	5
1	6

SALES_DETAIL

receipt_no	product_id	unit_price	qty
EC23456792	1	200	1
EC67890124	1	200	3
EC67890124	2	190	3
EC67890124	5	110	5
EC10345786	6	50	6
EC48786019	5	110	2

STORE_PHONE

store_id	phone_number
1	(02)6667-5112
1	(02)9841-0037

The data stored in store 2

EMPLOYEE

id	name	gender	birthday	monthly_salary	supervisor_id	store_id
S657432198	Chen-Hsi Liu	M	1995/1/10	102000	L645977505	2
S153004821	Po-Lin Chen	M	2000/5/17	48000	S657432198	2
E972346850	Hao-Tien Chu	M	2001/8/22	48000	S657432198	2

SALES

receipt_no	date_time	store_id
ER78901246	2025/1/3 9:25	2
ER34567891	2025/1/3 11:35	2
ER84785499	2025/1/4 12:08	2

SALES_DETAIL

receipt_no	product_id	unit_price	qty
ER78901246	1	200	1
ER34567891	2	190	2
ER34567891	3	35	10
ER84785499	1	200	5

STORE_PRODUCT

store_id	product_id
2	1
2	2
2	3
2	5

STORE_PHONE

store_id	phone_number
2	(07)548-6634

Typical sharding methods

- **Business rule sharding**:
  - E.g., domestic students in shard 1, international students in shard 2.
- **Range sharding**:
  - E.g., ages in [10, 20) in shard 1, [20, 30) in shard 2, etc.
- **Entity group sharding**:
  - Storing related entities in the same shard (popular in sharding relational databases).
  - E.g., manufacturing-related tables in the factory site, customer-related tables in the headquarter site, etc.

Typical sharding methods

- **One customer, one database**:
  - Especially if (1) your solution is for businesses, and (2) different companies' database operations are completed separated.
- **One giant table, one database**:
  - Especially if this table is rarely joined with other tables.
- **Hash sharding**:
  - Putting each record into a shard according to a hash function.
- These methods again can be mixed.

Typical sharding principles

- One common idea is to have **multiple "layers"** based on **access frequencies**.
  - The most frequently accessed data are put in the first node (probably with the strongest computation resources).
  - The least frequently accessed data are put in the third node (probably with the weakest computation resources).
  - Other data are put in the second node.
- For example, a university can do this:
  - Put all records of current students and those who graduated within five years in the first node.
  - Put all records of other graduated students in the second node.

Distributed query planning

- When data are distributed, the DDBMS does **distributed query planning** (分散式查詢規劃).
  - Now the bottleneck in many cases is network transmission.
  - The optimization objective becomes **minimizing the amount of data transmitted** through the network.

An example: the data

- Suppose that in a company the data are distributed in the following way:

Site 1:

EMPLOYEE

id	name	gender	birthday	monthly_salary	supervisor_id	store_id

1,000 records
each record is 100 bytes long

`id` is 10 bytes long
`name` is 60 bytes long
`store_id` is 10 bytes long

Site 2:

STORE

id	postal_code	addr_str	mgr_id

10 records
each record is 50 bytes long

`id` is 10 bytes long
`mgr_id` is 10 bytes long

Distributed query planning: an example

- Consider a query: For each store, retrieve the store ID and the name of the store manager.
  - The result site (the site making the query request) is site 3.
- Consider the following three plans:
  - Plan 1: Transfer `EMPLOYEE` and `STORE` to site 3 and perform the join there. The transfer amount is $80{,}000 + 200 = 80{,}200$ bytes.
  - Plan 2: Transfer `EMPLOYEE` to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is $10 \times (10 + 60) = 700$ bytes (10 records, each with two attributes), so $80{,}000 + 700 = 80{,}700$ bytes must be transferred.
  - Plan 3: Transfer `STORE` to site 1, execute the join at site 1, and send the result to site 3. The transfer amount is $200 + 700 = 900$ bytes.
- Plan 3 is the best due to its smallest network transfer amount.

Distributed query planning: an example

- What if the result site is site 2?
- Consider the following three plans:
  - Plan 1: Transfer `EMPLOYEE` and `STORE` to site 2 and perform the join there. The transfer amount is 80,000 bytes.
  - Plan 2: Transfer `STORE` to site 1, execute the join at site 1, and send the result back to site 2. The transfer amount is $200 + 700 = 900$ bytes.
- Plan 2 is better.

A (usually) better strategy

- Sometimes we may do better in joining two relations R and S that are stored in different sites.
  - First, we send **only the joining column(s)** of R to the site storing S.
  - This column(s) is then joined with S.
  - This column(s) and **only the required columns** are shipped back to the original site and joined with R.

A (usually) better strategy: example

- Consider still the query: For each store, retrieve the store ID and the name of the store manager.
  - The result site (the site making the query request) is site 3.
- Recall Plan 3: Transfer `STORE` to site 1, execute the join at site 1, and send the result to site 3. The transfer amount is $200 + 700 = 900$ bytes.
- Now let's apply the strategy:
  - Send the join attributes of `STORE` to site 1. We transfer `mgr_id`, whose size is $10 \times 10 = 100$ bytes.
  - Join the transferred file with `EMPLOYEE` at site 1 and transfer only the required attributes from the resulting file to site 2. We transfer {`mgr_id, name`}, whose size is $10 \times (10 + 60) = 700$ bytes.
  - Execute join with `STORE` at site 2.

Pros and cons of distribution

- While distribution is great for processing queries, it raises challenges for processing **updates** (including insertion, deletion, and updating).
  - If we want to ensure that every updates **occurs at all nodes** that should be affected, we need good algorithms and sacrifice efficiency.
  - If we want the updates to be effective at all nodes **at almost the same time**, we need very good algorithms and sacrifice a lot of efficiency.
- Transaction management and concurrency control both become much more challenging in a distributed environment.

Distributing an OLTP system is hard

- An OLTP system cannot be easily distributed.
- One major reason is because for OLTP we require **consistency**.
  - To make a distributed database the most efficient, different nodes should serve different clients **concurrently**.
  - If we want to enforce data consistency among **all** nodes at **any** time, concurrent processing will be significantly limited. The distributed database will lose a large part of its advantage.

Inconsistency in a distributed system

- Consider the following example:
  - A data item X is replicated on nodes M and N.
  - Client 1 updates X on node N.
  - Some time elapses.
  - Client 2 reads X from node M.
  - Should client 2 see the update made by client 1?
- It depends!

Consistency vs. efficiency

- When is consistency needed and not needed?
- In a transaction processing system (e.g., banking, retailing, ticketing), we do need consistency.
  - Otherwise, e.g., a seat may be sold to two customers.
  - Correctness is more critical than efficiency.
- In many web applications, however, we do not need perfect consistency.
  - **Social networking sites**: It is pretty much fine for a post/reply to appear in front of different people in different places at different times (if the time difference is within several seconds or even minutes).
  - Efficiency is more critical than correctness.
  - We do not need consistency at any time; we need **eventual consistency**.

The CAP theorem

- **The CAP theorem** says: consistency, availability, and partition tolerance, choose **two**.
  - **Consistency**: A read operation always gets the most recently written results.
  - **Availability**: A node responds to a client correctly in a reasonable time as long as it is not down.
  - **Partition tolerance**: Partitioning nodes is allowed, and thus different nodes may contain different data or provide different level of availability.
- Proposed in 2000 and later proved in 2002.

The CAP theorem

- Three types of systems: **CA**, **AP**, and **CP**.
- CA: a traditional system.
  - A centralized (single-node) system has consistency and availability.
- AP: from time to time, there may be some data not synchronized.
- CP: from time to time, there may be some data unavailable to some clients.
  - If a distributed system wants consistency, only one node is allowed to accept write operations at a time. Before new records are propagated to other nodes, read operations are also disallowed (and thus low availability).
- To **scale** up and down, you must partition.
  - That leaves either consistency or availability to choose from.
  - In most cases, people would choose availability over consistency.

BASE

- For distributed databases and NoSQL, people ask for **BASE** rather than ACID.
- Let's recall the ACID property of transactions in a relational database:
  - Atomicity: Be performed either in its entirety or not performed at all.
  - Consistency preservation: If a transaction is completely executed, the resulting database should still satisfy the schema constraints.
  - Isolation: The execution of a transaction should not be affected by other concurrent transactions.
  - Durability or permanency: Once a transaction is committed, the changes made by the transaction must never be lost.
- Consistency is a too strong requirement for a distributed database.
  - Let's change it to eventual consistency.

Eventual consistency

- While consistency (at any time) is too difficult to achieve in a system with availability and partition tolerance, we require **eventual consistency**:
  - If no updates occur for a long period of time, eventually all updates should propagate through the system.
  - For a given accepted update at node A and another node B, eventually either the update reaches node B, or node B is removed from service.
  - Data on all nodes should then be consistent.
- Techniques for distributed systems are required.

BASE

- We want a distributed system to satisfy the **BASE** property.
- **Basically Available**:
  - When some partitions fail, other partitions are still available.
  - When a minor function fails, core functions are still available.
- **Soft state**:
  - Data in the system is allowed to be inconsistent for a while.
- **Eventual consistency**:
  - After a certain amount of time, all data on all nodes are consistent.
  - The definition of "a certain amount of time" depends on the application and the service provider.

BASE and relational databases

- Even if we accept BASE, distribution is still hard (or inefficient) if we work with relational databases.
  - For example, due to the referential constraints among tables.
- This is one of the reasons for people to choose to use NoSQL databases in some cases.
  - Note that even if we do not want a distributed database, there are still reasons to choose NoSQL (e.g., flexibility).

Agenda

- Big data
- Distributed databases
- **NoSQL**

Limits of relational databases

- Relational databases are useful in many scenarios.
  - However, there are still cases where they are not so efficient.
- As long as there are too many tuples (**big volume**), a relational database can be slow regardless of the content.
  - To scale a relational database, **vertical scaling** by adding computing resources on a single machine is straightforward but expensive.
  - **Horizontal scaling** to a **cluster** of machines is required, but for a relational database it is just too difficult because of the ACID requirement.

Limits of relational databases

- Another example: **web logs**.
  - We want to analyze users' **clickstream data** on a website (one kind of behavior data).
  - Click records are typically stored in **web logs** generated by **web servers**.
  - It is not efficient to store these click records in a relational database: too many insertion operations in too short a period (**high velocity**), and checking those schema constraints is time-consuming and not needed.
  - Moreover, in the future we may have the web server store more or fewer attributes for each click (**high variety**).

NoSQL

- **NoSQL** stands for "**Non-SQL**" or "**Not only SQL**".
  - Became popular starting around 2010.
  - Google's BigTable, Amazon's DynamoDB, Facebook's Cassandra, etc.
- While SQL is the name of a programming language, NoSQL is not.
  - NoSQL is **a type of a database**.
  - There are relational databases and NoSQL ones.
  - Popular NoSQL DBMS: MongoDB, Cassandra, HBase, Redis, etc.
- Major characteristics of most NoSQL databases:
  - **Schema-free** or **schema-less**: There is no fixed schema for tables.
  - **Allowing inconsistency**: from ACID to BASE.

NoSQL solutions

- NoSQL solutions fall into two major areas: key-value (schema-free) and schema-less.
- **Key-value** (or "the big hash table"):
  - There is pretty much no schema. One may search for the value of a data item only through a key but cannot search based on attributes/columns.
  - Redis, Amazon S3 (Dynamo), Voldemort, Scalaris, etc.
- **Schema-less**:
  - There is a weak concept of schema. One may still search based on attributes/columns.
  - **Document**: MongoDB, CouchDB, etc.
  - **Wide column**: Cassandra, HBase, etc.
  - **Graph**: Neo4J, etc.

A key-value store database: Redis

- The database **Redis** is a **key-value store**, which is schema-free.
  - Each records consists only a key attribute and the corresponding value.
- A natural scenario: **cache**.
  - Most of the cache are prepared to speed up some processing, e.g., web search.
  - Whenever a user input a **keyword** to search for something, a search engine typically first search within a cache.
  - The search is only based on the **keyword** in the **key attribute**.
- Data storage and data operations are in-memory and thus time-efficient.
- In such an application, typically we do not care about ACID.
  - If an application requires ACID, using a key-value store can be dangerous.

A document-based databases: MongoDB

- In **MongoDB**, each data record is stored as a **document**.
  - The format is called BSON (binary JSON).
  - The values of fields may be documents, arrays, arrays of documents, etc.
- For example:

```
{
    "name": "Ling-Chieh Kung",
    "age": 40,
    "address": {
        "street": "No. 1, Sec. 4, Roosevelt Road",
        "city": "Taipei"
    },
    "hobbies": ["reading", "sleeping", "teaching"]
}
```

- Another record may have different fields.

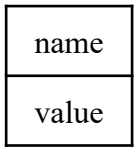
A document-based databases: MongoDB

- Note that most **relational databases** also allow document-type columns.
  - For example, in PostgreSQL, the data type of a column can be set to be **JSON**.
- Whether to use a document-based database or a relational one should be considered carefully.

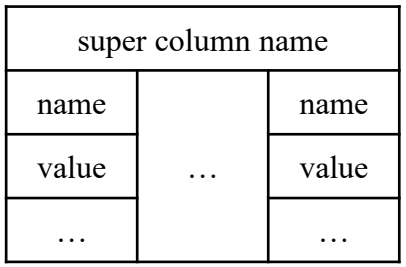
A wide-column databases: Cassandra

- In **Cassandra**, each data record is stored in a **row**.
  - A keyspace (like a database) contains column families (like tables).
  - A column family contains super columns (a collection of columns) and columns.
  - A column is a name-value pair.
  - Each row (record) has a key.
- A natural scenario: **web crawler**.
  - Adding columns are easy.

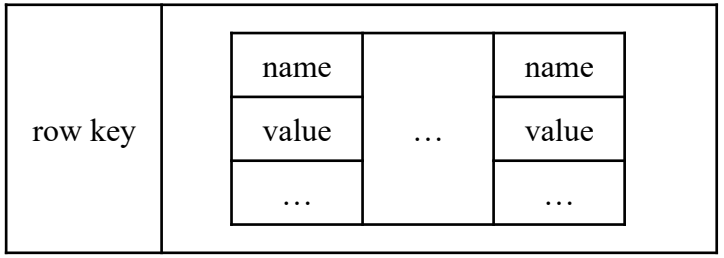
Column



Super Column



Column Family



Deal with varying columns

- Suppose that in our application there is a table that may have new columns in the future.
- In a wide-column database, typically we use just one table with many columns.
  - For example:

record_id	attribute1	attribute2	attribute3	attribute4	attribute5	attribute6
1	200	2025/6/3		Train		
2	300	2025/5/25				
3	500			Air	15	20
4		2025/5/28				
5	200		400			

- As adding columns is easy, we do not worry about varying columns.

Deal with varying columns

- Note that in a relational database, the column variation issue can still be handled.
- In a relational database, typically we use a table to store "the columns" and use another table to store the data.
  - For example:

attribute_id	attribute_name
1	attribute_name_1
2	attribute_name_2
3	attribute_name_3
4	attribute_name_4
5	attribute_name_5
6	attribute_name_6

record_id	attribute_id	attribute_value
1	1	200
1	2	2025/6/3
1	4	Train
2	1	300
2	2	2025/5/25
3	1	500
3	4	Air
3	5	15
3	6	20
4	2	2025/5/28
5	1	200
5	3	400

NoSQL databases: Cassandra

- Example programs:
  - Creating a keyspace:

```
CREATE KEYSPACE myKeyspace
WITH replication
= {'class':'SimpleStrategy',
   'replication_factor': 3};
```

  - Creating a table (column families):

```
CREATE TABLE USERS (
    id UUID PRIMARY KEY,
    name text,
    email text,
    phone text
);
```

  - Insert data:

```
INSERT INTO USERS (id, name, phone)
VALUES (uuid(), 'Jane Smith', '123-456-7890');
```

```
INSERT INTO USERS (id, name, email)
VALUES (uuid(), 'John Doe', 'johndoe@example.com');
```

NoSQL databases: Cassandra

```
CREATE TABLE USERS (
    id UUID PRIMARY KEY,
    name text,
    email text,
    phone text
);
```

```
INSERT INTO USERS (id, name, phone)
VALUES (uuid(), 'Jane Smith', '123-456-7890');
```

```
INSERT INTO USERS (id, name, email)
VALUES (uuid(), 'John Doe', 'johndoe@example.com');
```

- But wait… What's the difference between this and a relational database?
  - A relational database allows a record to include NULL values.
- The difference is at the system level:
  - In a relational database, each row has **all** those attributes with some of the values are NULL (like an adjacency array).
  - In Casandra, each row has only **part** of the attributes and **only** values of those attributes (like an adjacency list).

Accessing data in a NoSQL database

- Data in a NoSQL database are typically accessed through **APIs**.
- Let's take MongoDB as an example.
- For example, to get all documents in a collection `collectionName`:

```
db.collectionName.find()
```

- To get all documents whose value of the field `key` is equal to `value`:

```
db.collectionName.find({ key: value })
```

- To get all documents and include only `fieldName1` and `fieldName2` in the returned result:

```
db.collectionName.find({}, { fieldName1: 1, fieldName2: 1 })
```

Accessing data in a NoSQL database

- To do an OR operation:

```
db.collectionName.find({ $or: [{ key1: value1 }, { key2: value2 }] })
```

- To set the value of the field `keyToUpdate` to `newValue` for all documents whose value of the field `key` is `value`:

```
db.collectionName.update(
  { key: value },
  { $set: { keyToUpdate: newValue } }
)
```

Common advantages of NoSQL databases

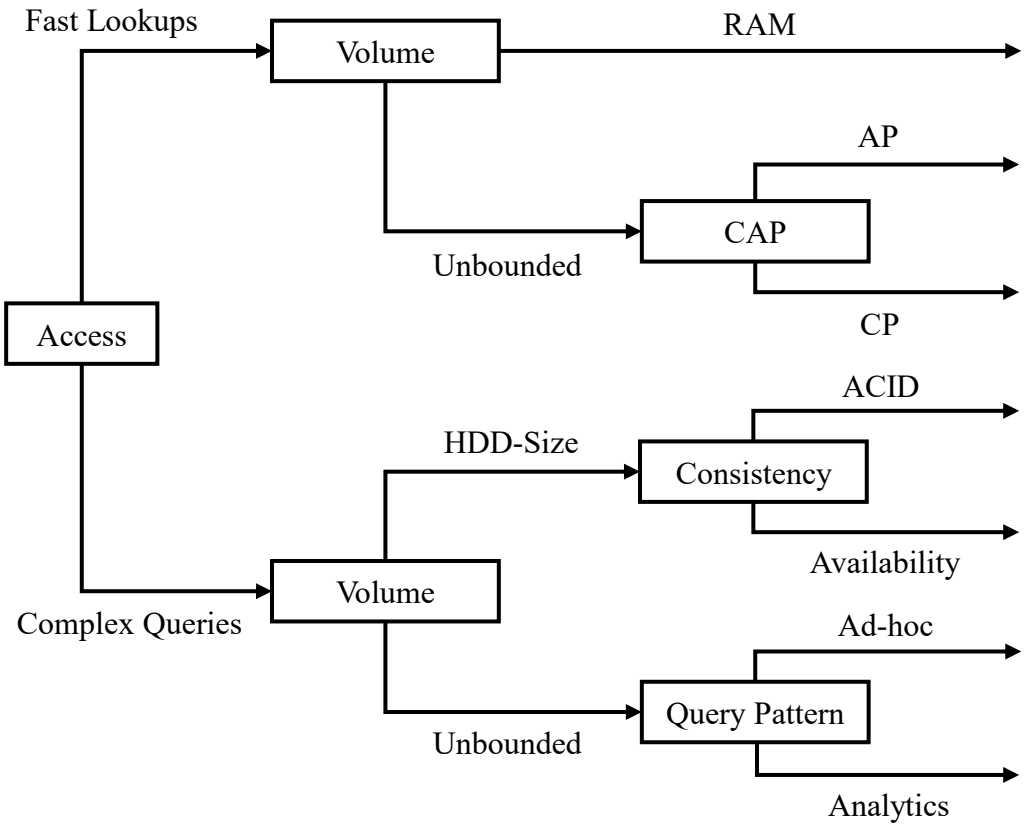
- Cheap and easy to implement (open source).
- Data can be easily **distributed** and **replicated** to **multiple nodes** (therefore **efficient** and **fault-tolerant**) and can be partitioned.
  - A node is a device on a distributed network.
  - NoSQL databases are not necessarily distributed (but most of them support distribution and partition).
- Down nodes can be easily replaced.
  - No **single point of failure**.
- Do not require a schema and thus do not require schema design.
- Scaling up and down is easy.

NoSQL solutions

Dimension	NoSQL databases				
	MongoDB	HBase	Cassandra	Riak	Redis
Data Structure	Document	Wide-column	Wide-column	Key-value	Key-value
CAP	CP	CP	AP	AP	CP
Write performance	High (I/O)	High (I/O)	High (I/O)	High (I/O)	Super high (Memory)
Sharding	Hash range-based	range-based	consistent hashing	consistent hashing	hash

(produced by Ling-Chieh Kung based on https://xiang753017.gitbook.io/zixiang-blog/database/qian-tan-nosql-zi-liao-ku-zen-me-xuan)

Database solutions



Database	Use Cases
Redis, Memcache	Cache
Cassandra, Riak, Voldemort, Aerospike	Web crawler
HBase, MongoDB, CouchBase, DynamoDB	Operations History
RDBMS (MySQL, PostgreSQL, etc.)	OLTP
CouchDB, MongoDB, SimpleDB	Website
MongoDB, RethinkDB, HBase, Accumulo, ElasticSearch, Solr	Social Network
Hadoop, Spark, Parallel DWH, Cassandra, HBase, Riak, MongoDB	Big data

Is NoSQL always good?

- The answer is obvious: it depends.
- There are still cases (e.g., OLTP) that consistency is critical.
  - Transactions are required to be ACID.
  - SQL is intuitive and powerful.
- For OLAP based on historical transactions, relational DBMS can be more efficient.
  - Operations including join, group by, order by, etc., are well optimized.
  - SQL is intuitive and powerful.
- The choice of database solution requires technical knowledge, experience, and domain knowledge.
  - Both information technology and management are important!