

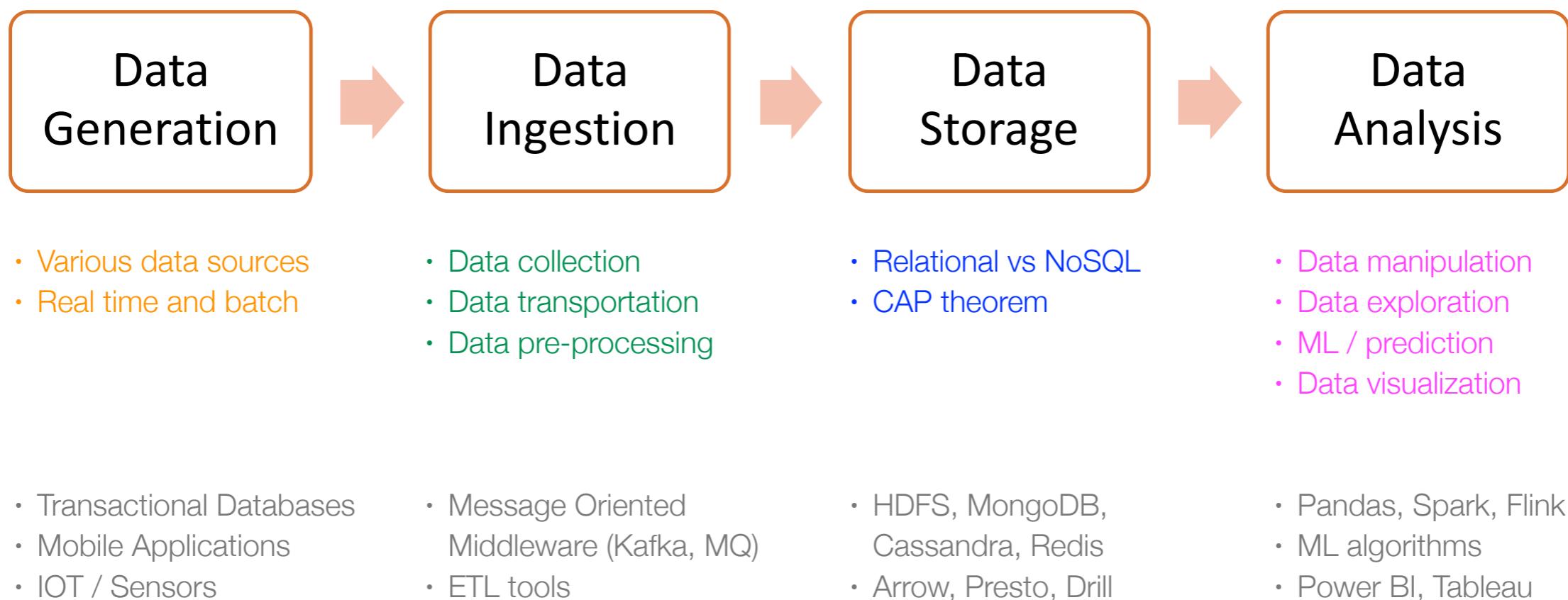


2110446 - Data Science and Data Engineering

Data Storages

Asst.Prof. Natawut Nupairoj, Ph.D.
Department of Computer Engineering
Chulalongkorn University
natawut.n@chula.ac.th

Data Lifecycle



Traditional Database

- Based on "relational model"
 - Data is split and stored into tables
 - Tables can be processed together using set-like operations
 - Data model is usually normalized to remove duplication
- Very suitable for OLTP or transaction systems
 - Provide lots of complicated SQL operations
 - Lots of inserts and updates

Advertisement

點用 未來錢

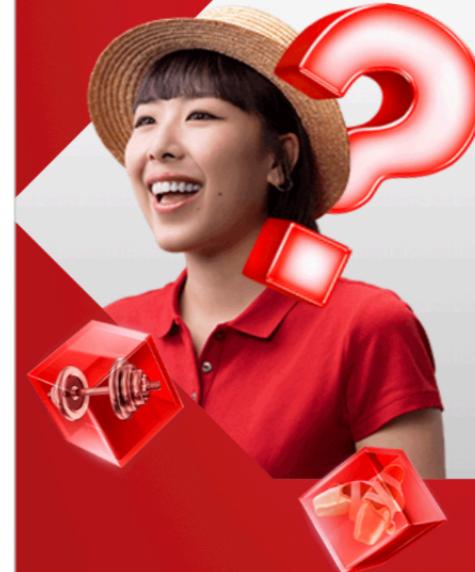


諗清楚，計劃好！

Advertisement



Advertisement

Smart
use of
**FUTURE
MONEY**

Advertisement

Prep Time:

20 mins

Total Time:

20 mins

Servings:

6

[Jump to Nutrition Facts](#)

Ingredients

- 3 cloves garlic, peeled
- 3 Thai green chiles
- 6 green beans, cut into 1-inch pieces
- 1 large unripe papaya, peeled and cut into

📍 Local Offers

00000 [Change](#)

Oops! We cannot find any ingredients on sale near you. Do we have the correct zip code?

Advertisement

**Trending Videos**

X

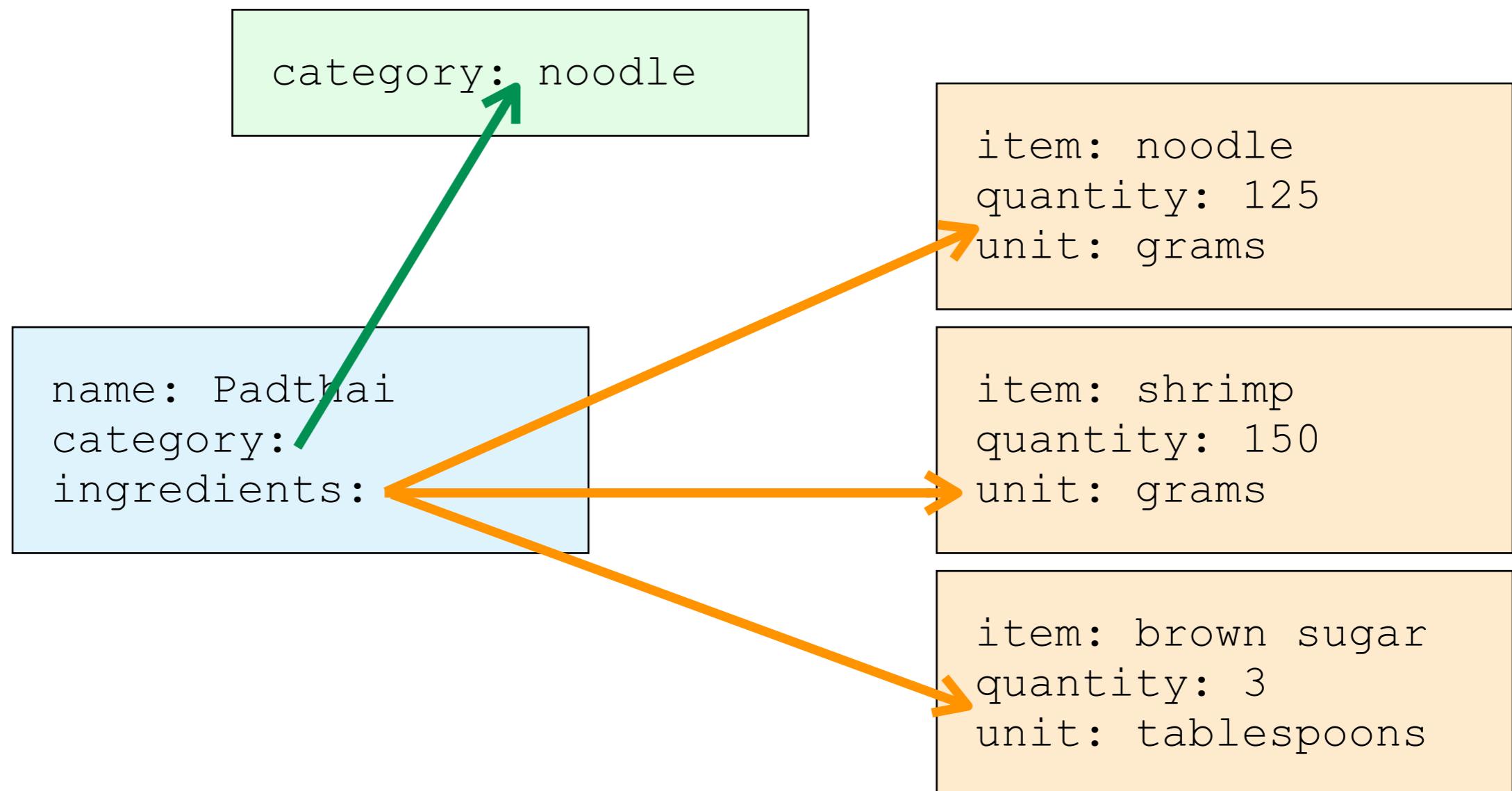


from four different chains.

allrecipes

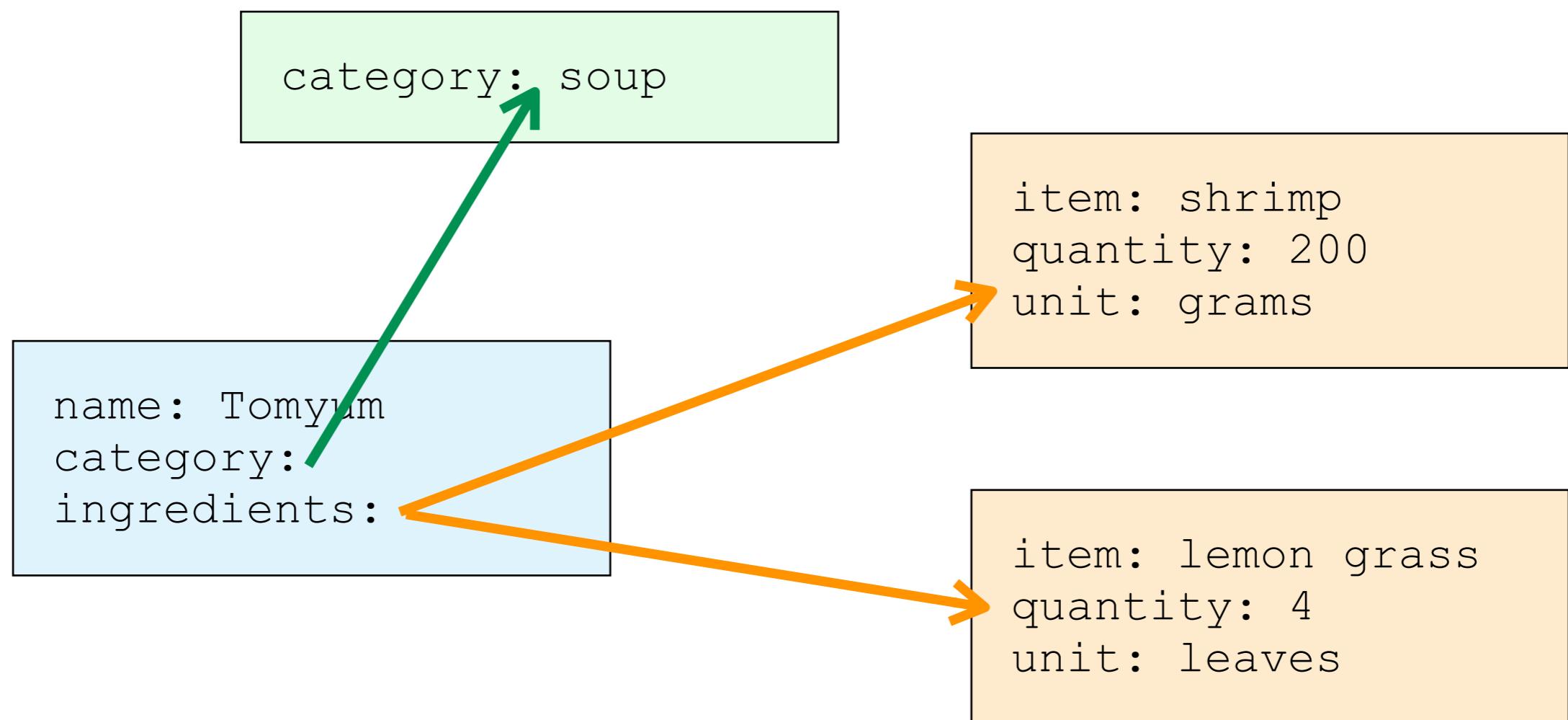
Recipe Data Example

- Padthai belongs to a noodle category with 3 main ingredients noodle, shrimp, brown sugar



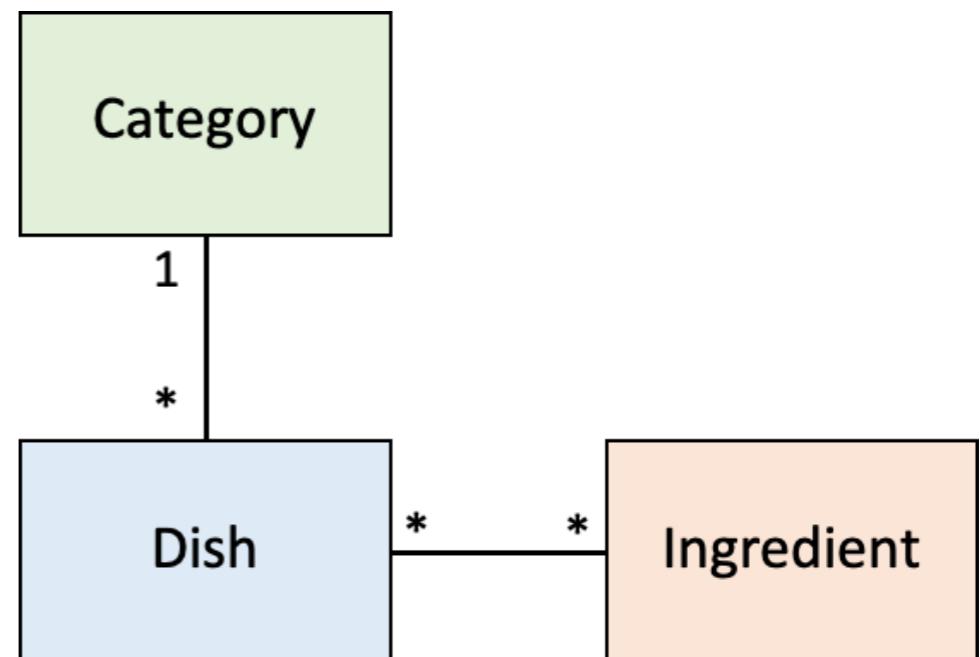
Recipe Data Example

- Tomyum belongs to a soup category; shrimp and lemon grass are main ingredients



Recipe Data is Hierarchical

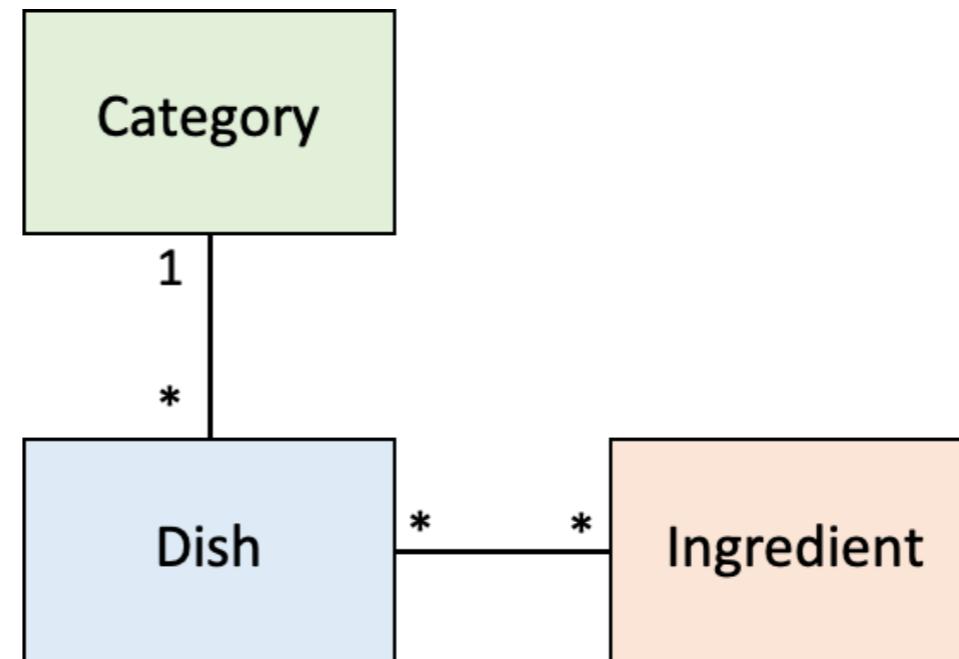
- A dish has the following data:
 - belong to a category
 - consists of one or more main ingredients
 - Ingredients are different in portions and units



Example RDBMS: Recipe

Category

id	category
1	Soup
2	Noodle
3	Rice



Dish

id	name	category	...
1203	Pad Thai	2	
1288	Chicken Rice	3	

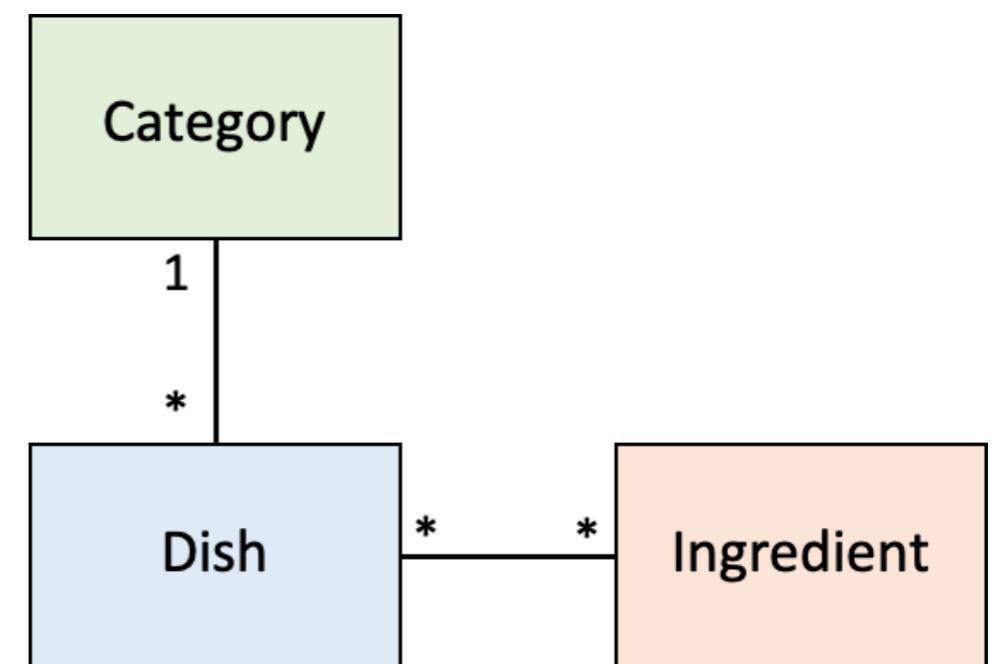
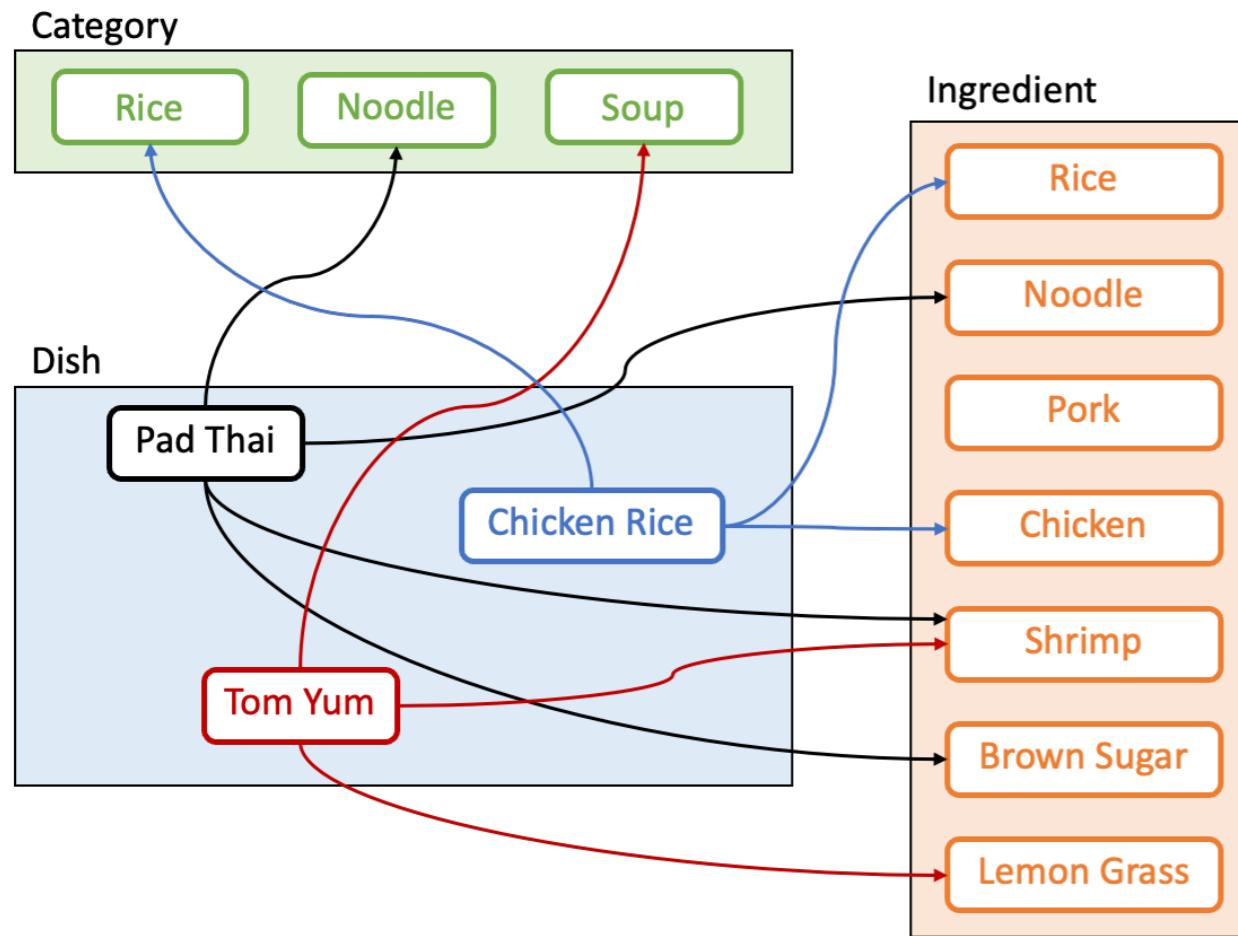
DI_link

d_id	i_id	quantity	unit
1203	45	125	g
1203	48	150	g
1203	52	3	tbsp

Ingredient

id	item
45	noodle
46	rice
48	shrimp
52	brown sugar

Relational Database: Normalized Data Model



How can we split these tables to lots of machines?

- Dishes of the same category in the same machine
- How about dishes and ingredients?

Or we can replicate data — lead to data consistency problems

Problems of Data Science Storage

- There are several needs for data analytics purposes e.g. traditional data store, caching, feature store
- Data is historical data and its volume can be huge
- Scalability is extremely important and “Relational + Consistency” can limit scalability
- SQL command can be very complex and time-consuming
 - It requires the synchronization of data accessing between multiple tables
 - It will be poor when using on more than a few servers in the same cluster

CAP Theorem (Brewer's Theorem)

- By Eric Brewer (University of California, Berkeley)
- It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees: Consistency, Availability, Partition tolerance
- When the communication failure occurs, the system has to choose either consistency or availability

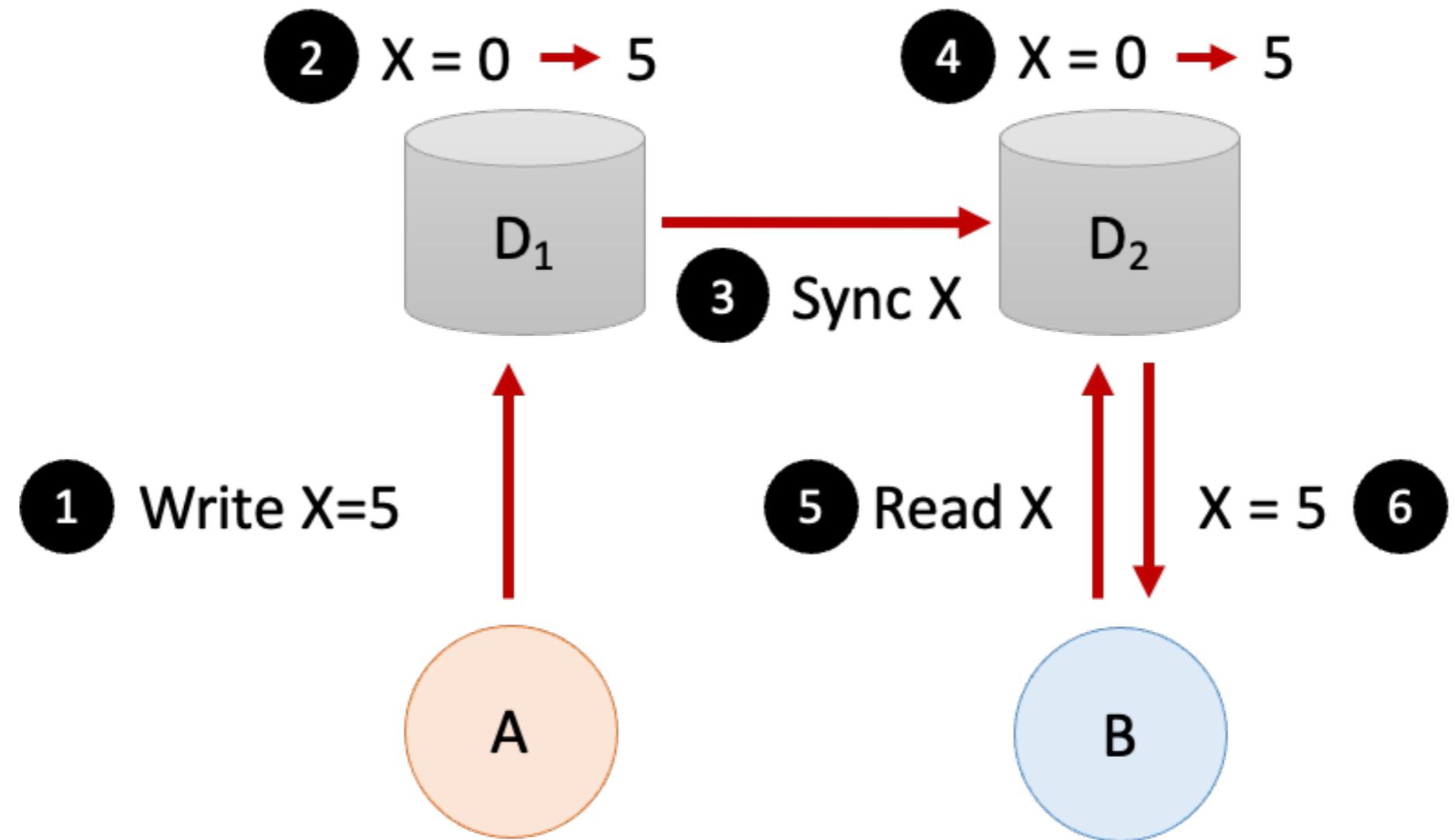
CAP Theorem Scenario

- Distributed system (clients and servers)
- Multiple servers working together
- Multiple clients may read or write on the same data at the same time

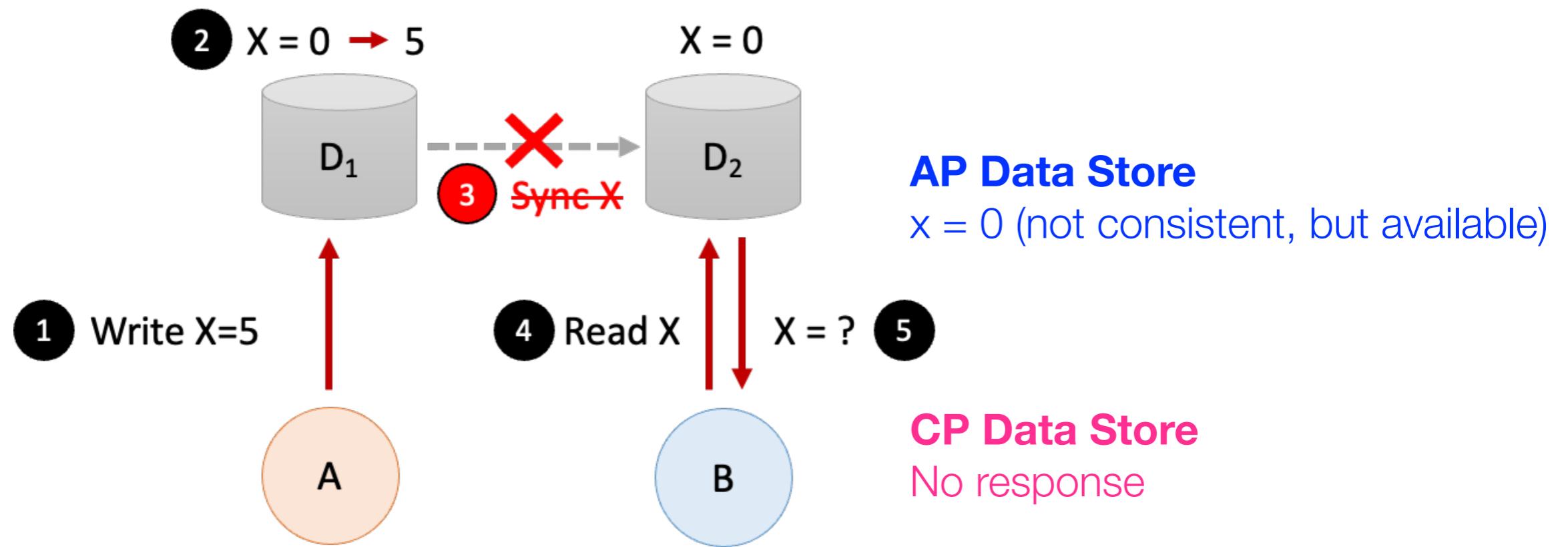
CAP Guarantees only 2 out of 3

- Consistency
 - Every read receives the most recent write or an error
- Availability
 - Every request receives a (non-error) response – without guarantee that it contains the most recent write
 - Response from any server is good
- Partition tolerance
 - The system continues to operate despite arbitrary message loss or failure of part of the system
 - Lots of servers require long synchronization time causing servers to not be able to communicate among one another within reasonable time

CA Data Store



Given P, Choosing between C and A (at step 5)



More Scenario - when partition occurs

- CA - cannot operate
- CP
 - allow read requests and reject all write requests
 - allow read and write at a group that still connects to a master node

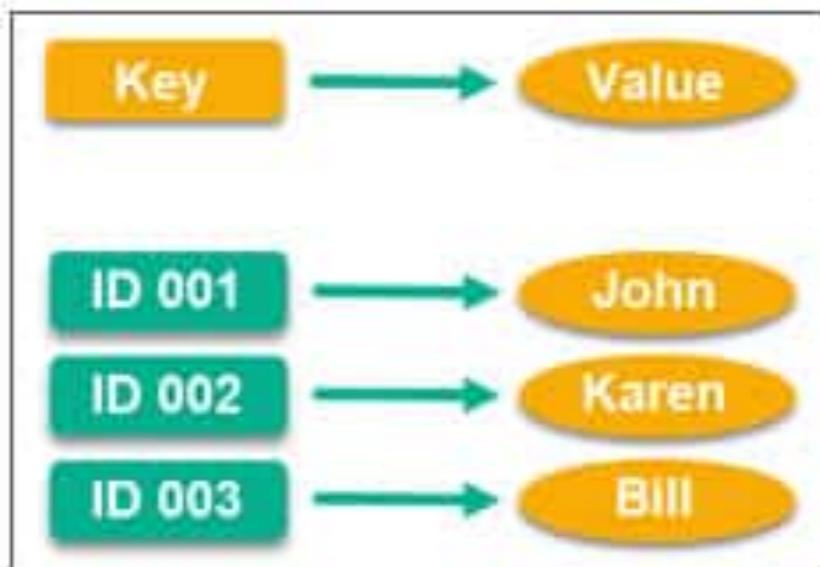
The Landscape of NoSQL

- Alternatives to SQL database
 - non-relational
 - distributed
 - horizontally scalable
- Data is shared and distributed across multiple servers
- Typically use weak consistency model (but not always)

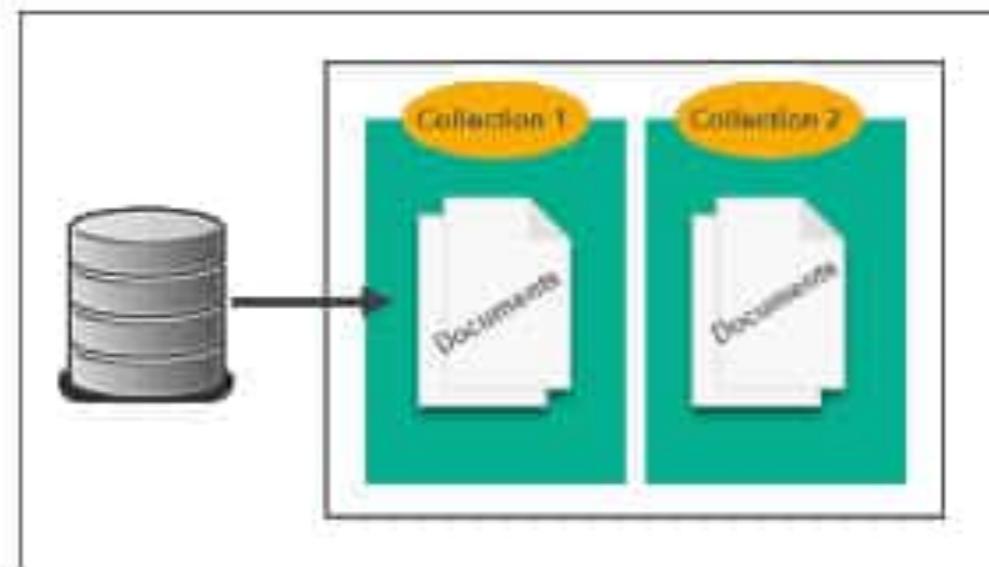
Types of NoSQL

- Document: MongoDB, DynamoDB, CosmosDB, Couchbase, Firebase
- Column: Cassandra, HBase, CosmosDB, Accumulo
- Key-value: Redis, DynamoDB, CosmosDB, MemcacheDB
- Graph: Neo4J, TigerGraph, ArrangoDB, OrientDB
- Search Engine: Elasticsearch, Splunk, Solr
- Timeseries: InfluxDB, TimescaleDB, Prometheus, Graphite
- Vector: Chroma, Pinecone,

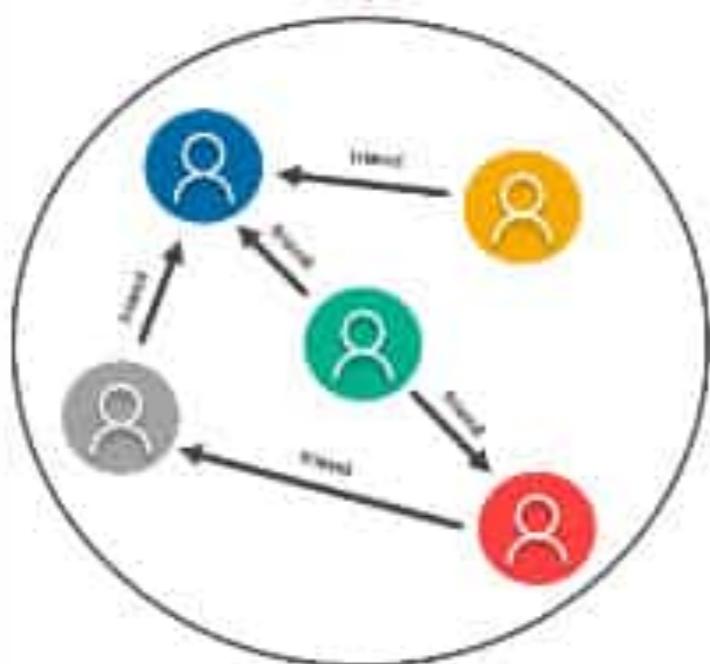
Key-value



Document



Graph



Wide-column

A diagram illustrating a wide-column database. It shows two tables: "Row-oriented" and "Column-oriented".

Row-oriented:

ID	Name	Grade	GPA
001	John	Senior	4.00
002	Karen	Freshman	3.67
003	Bill	Junior	3.33

Column-oriented:

Name	ID
John	001
Karen	002
Bill	003

Grade	ID
Senior	001
Freshman	002
Junior	003

GPA	ID
4.00	001
3.67	002
3.33	003

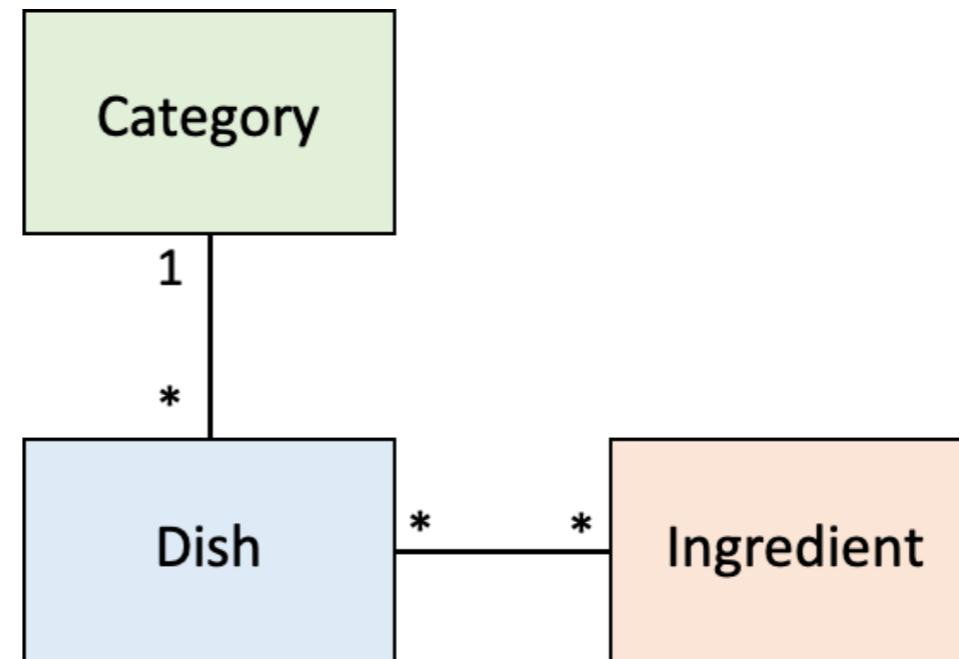
How NoSQL can “Scale”

- Principle ideas
 - Split data into chunks or shards
 - Distribute data across multiple servers
 - Must require minimum synchronization
- Have to give up some traditional features
 - No complex relational model
 - Relax consistency
 - Duplicated information (not space optimized)
 - Fast to insert new record, but not so fast to update the existing one

Example RDBMS: Recipe

Category

id	category
1	Soup
2	Noodle
3	Rice



Dish

id	name	category	...
1203	Pad Thai	2	
1288	Chicken Rice	3	

DI_link

d_id	i_id	quantity	unit
1203	45	125	g
1203	48	150	g
1203	52	3	tbsp

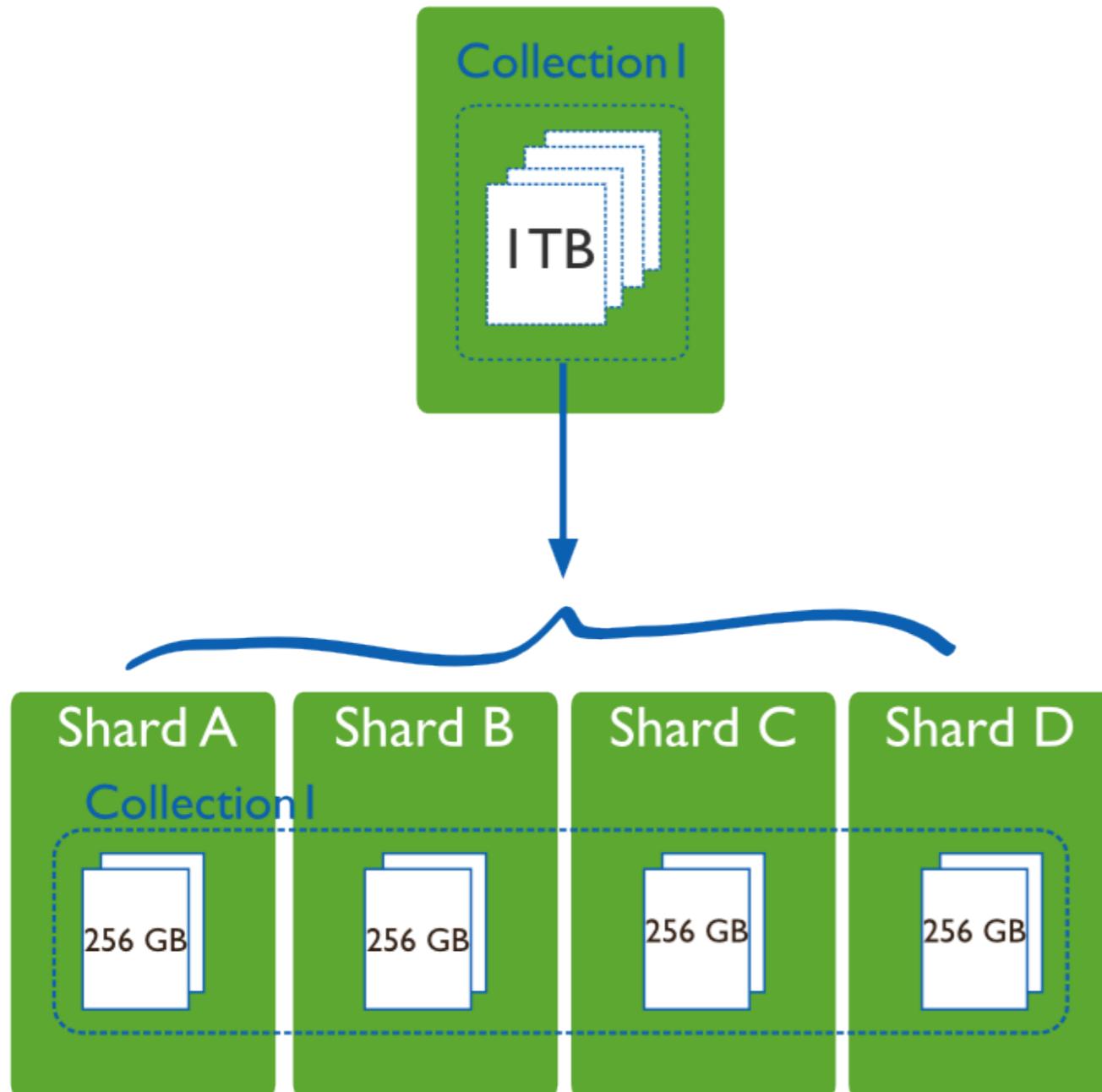
Ingredient

id	item
45	noodle
46	rice
48	shrimp
52	brown sugar

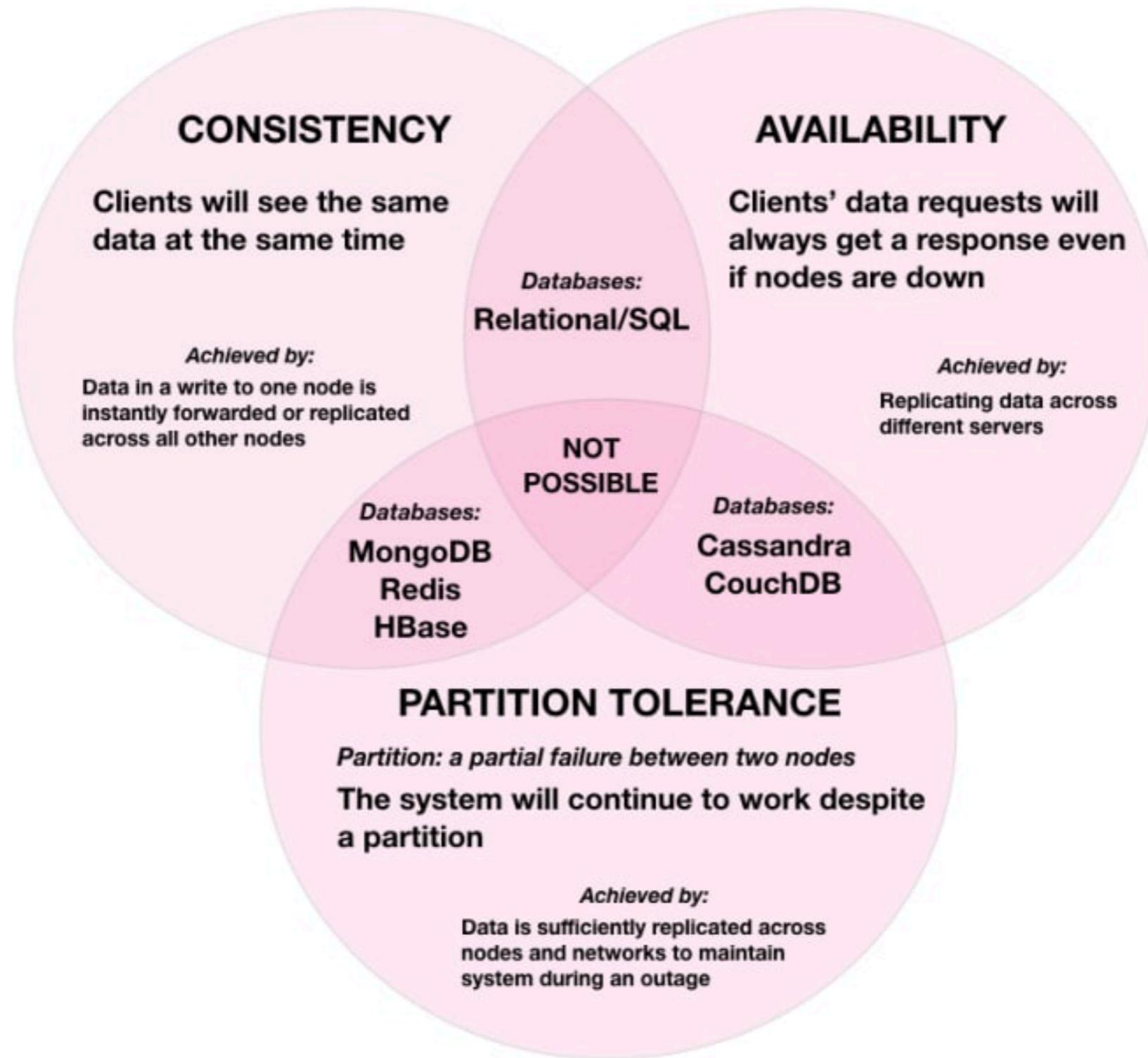
Example MongoDB: Recipe

```
{  
  _id: ObjectId ("507f191e810c19729de860ea""),  
  name: "Pad Thai",  
  cuisine: "Thai",  
  category: "Noodle",  
  rating: 4.7,  
  votes: 9301,  
  ingredients: [  
    { item: "noodle", quantity: "125", unit: "g" },  
    { item: "shrimp", quantity: "150", unit: "g" },  
    { item: "brown sugar", quantity: "3", unit: "tbsp" }  
  ]  
}
```

How MongoDB Scales - Sharing



- Large-Scale MongoDB bases on Horizontal Scaling mechanism
- Storing shards (blocks of data) across multiple machines
- Data partitioned into shards with shard keys
- Each shard handles only operations related to its block



Source: <https://dev.to/katkelly/cap-theorem-why-you-can-t-have-it-all-ga1>

Redis

Key-Value Store

Data Storage



Key-Value Stores

- A data store designed for storing, retrieving, and managing associative arrays (aka. dictionary or hash table)
- Main concept is to store data as a collection of key-value pairs in which a key serves as a unique identifier
- Simple and fast, often use in-memory architecture, ability to scale-out
- Use cases: Database/API caching, session store, shopping cart, etc.

Redis (Remote Dictionary Server)

- In-memory data structure store with clustering, transactional, time-to-live limiting, and auto-failover capabilities
- Support wide-range of data structure with lots of related operations for each structure
- Being used for database cache, message broker, streaming engine, feature store engine, etc.
- Provide CLI and support many programming languages
- Many useful modules are available to extend the functionality of Redis core e.g. RedisJSON, RedisSearch, RedisTimeSeries, Redis OM, etc.

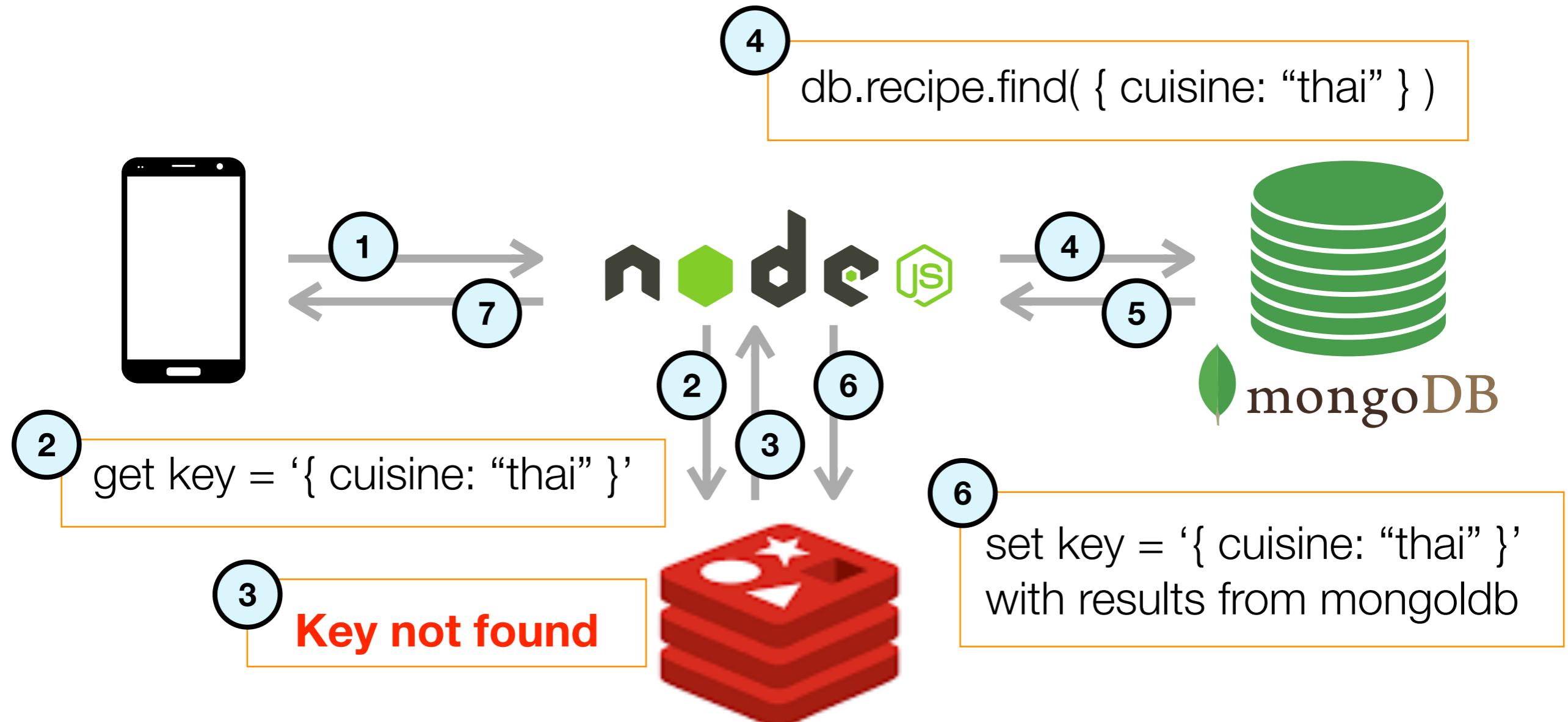
Typical Web Application Architecture



Database latency = 100ms per query
Effective latency = 100ms per query

Redis as Database Cache

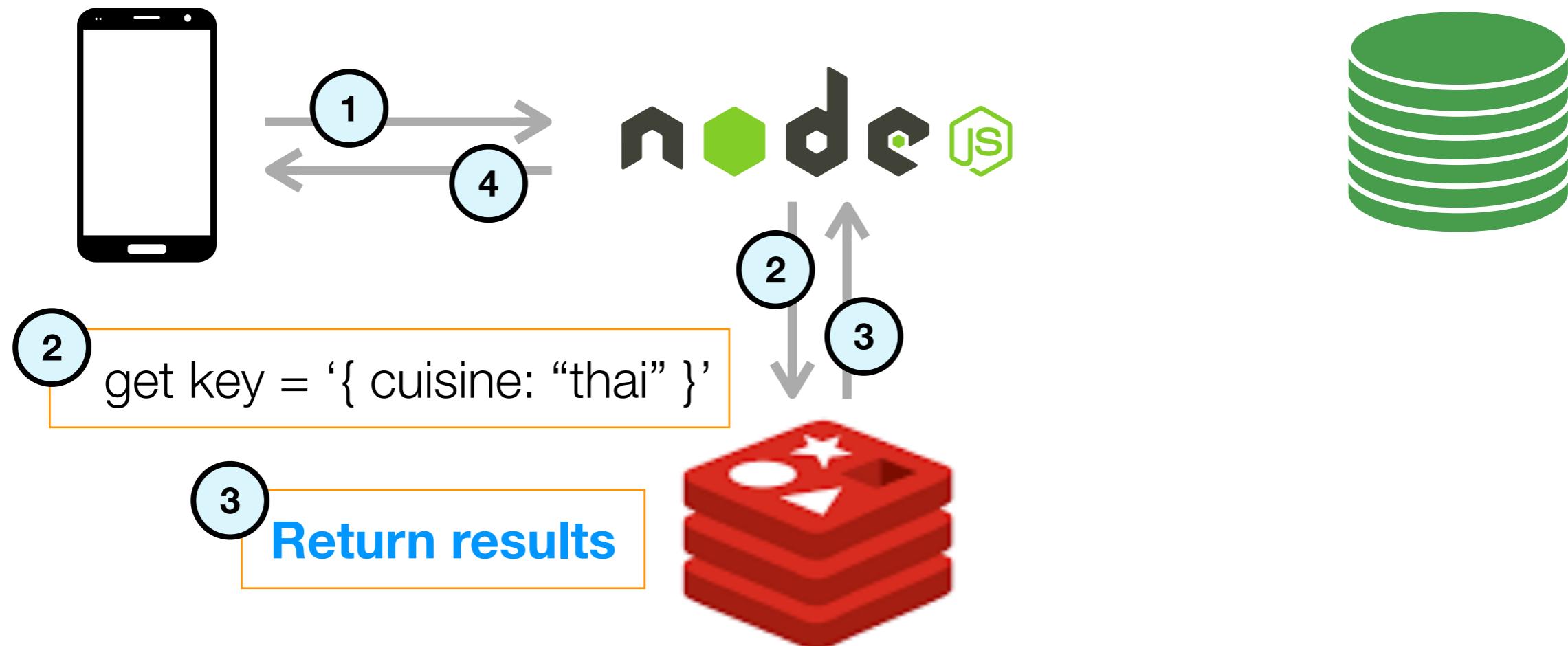
First Time = Cache Miss



Redis latency = 5ms per operation
Database latency = 100ms per query
Effective latency = 110ms per query

Redis as Database Cache

First Time = Cache Hit



Redis latency = 5ms per operation
Effective latency = 5ms per query

Benefits of Using Cache

- Each request results to either cache hit (5ms) or cache miss (100ms)
- Cache hits come from using the same request, suppose this happens **80%** of the time (**cache hit rate**)
- If cache is not hit, it is a cache miss, thus:
cache miss rate = $1 - \text{cache hit rate}$

Effective Latency =

$$\begin{aligned} & \text{Cache Hit Rate} * \text{Cache Hit Latency} + \\ & \text{Cache Miss Rate} * \text{Cache Miss Latency} \\ = & 0.8 * 5 + (1 - 0.8) * 100 \\ = & 4 + 20 \\ = & \mathbf{24 \text{ ms}} \end{aligned}$$

Running Redis

- The simplest way to run a redis instance is to use docker

```
docker pull redis
docker run -d --rm --name redis -p 6379:6379 redis
```
- This will start redis in your docker at port 6379 and map the port to your localhost
- You can also use docker-compose.yml and other example files in redis folder in datasci_architecture repo

Working with Redis

- Redis CLI
 - Standard client program to connect to any redis server
 - Come with any redis installation (see: <https://redis.io/docs/getting-started/>)

redis-cli

```
redis-cli -h 34.143.227.66
```

- You can type in redis command in the CLI input

Working with Redis

- Redis-Py
 - Standard python package for redis client

```
pip install redis
```

- Example

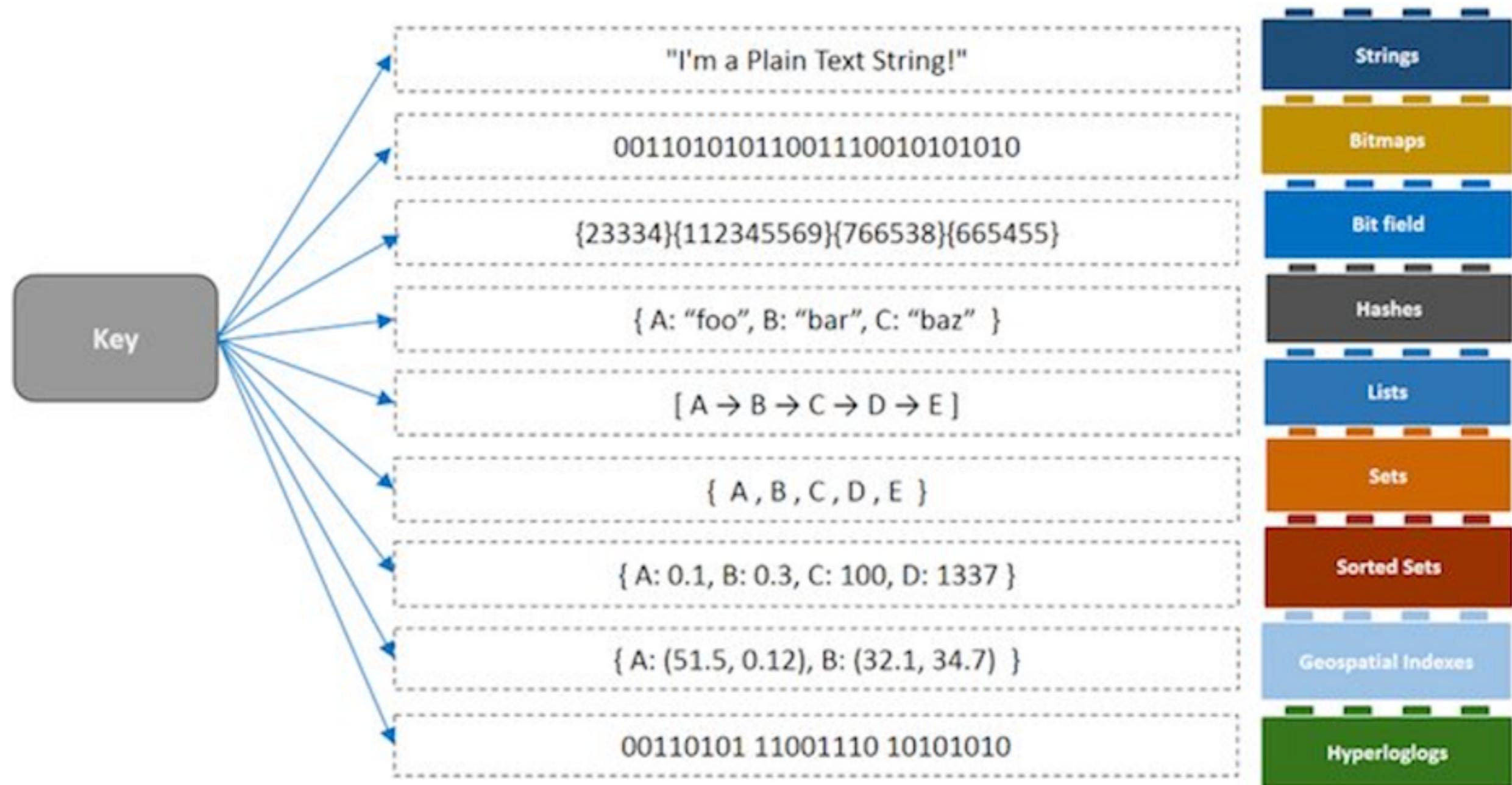
```
import redis

r = redis.Redis(host='hostname', port=port)
```

Redis Key

- Each record is “Key” and “Value”
- Redis keys are binary safe; any binary sequence as a key, from a string like "foo" to the content of a JPEG file
- Maximum key size is 512 MB
- Key should follow some consistent patterns e.g. “object-type:id” (e.g. “user:1234”)
- Key pattern design is very important, especially for data retrieval

Redis Value Data Types



String

- Similar to Python or Java Strings, maximum length of 512MB

```
SET "user" "Natawut Nupairoj"
```

```
GET "user"
```

```
DEL "user"
```

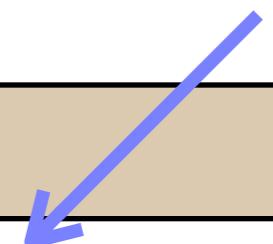
- Use cases

- Server-side object cache e.g. HTML fragments

- Database/API cache

Key	Value
cache:cuisine:thai	“[{{name:'padthai',cuisine:'thai',...},...},...]”

JSON String



Other String Commands

APPEND

INCR

SET

DECR

INCRBY

SETEX

DECRBY

INCRBYFLOAT

SETNX

GET

LCS

SETRANGE

GETDEL

MGET

STRLEN

GETEX

MSET

SUBSTR

GETRANGE

MSETNX

GETSET

PSETEX

Useful Commands

- Any item in Redis can be made to expire after or at a certain time

```
EXPIRE user 60. # in seconds
```

```
TTL user
```

- You can scan all index with scan command

```
SCAN 0
```

- You can delete item or test its existence

```
DEL mykey
```

```
EXISTS mykey
```

List

- List of strings, sorted by insertion order
- Can be used as list, queue, stack

```
LPUSH mylist abc # mylist contains "abc"
```

```
LPUSH mylist xyz # mylist contains "xyz", "abc"
```

```
RPUSH mylist 123 # mylist contains "xyz", "abc", "123"
```

- Use cases
 - Queue
 - Shopping carts
 - Timelines

Key	Value
task:queue	[“task:37492”, “task:40938”, “task:49274”, ...]

List Commands

BLMOVE

LMOVE

LSET

BLMPOP

LMPOP

LTRIM

BLPOP

LPOP

RPOP

BRPOP

LPOS

RPOPLPUSH

BRPOPLPUSH

LPUSH

RPUSH

LINDEX

LPUSHX

RPUSHNX

LINSERT

LRANGE

LLEN

LREM

Set

- Powerful data types for unordered non-duplicated keys
- Support many set operations e.g. intersection, union, etc.
`SADD user_set natawut`
`SCARD user_set`
`SMEMBERS user_set`
- Use cases
 - Set of user profiles
 - Set of inappropriate words for inappropriate content filtering

Key	Value
words:denied	{ "YLIJIL", "KJLYGKIJ", "*%&^*^%", ... }

Set Commands

SADD

SISMEMBER

SSCAN

SCARD

SMEMBERS

SUNION

SDIFF

SMISMEMBER

SUNIONSTORE

SDIFFSTORE

SMOVE

SINTER

SPOP

SINTERCARD

SRANDMEMBER

SINTERSTORE

SREM

Sorted Set

- Set of sorted items based on the score associated to each member

```
ZADD my_sortedset 5 data1
```

```
ZADD my_sortedset 1 data2 10 data3
```

```
ZRANGEBYSCORE my_sortedset 5. +inf WITHSCORES
```

- Use
 - Leader scoreboard
 - Priority queue

Sorted Set Commands

BZMPOP

ZDIFFSTORE

ZMSCORE

BZPOPMAX

ZINCRBY

ZPOPMAX

BZPOPMIN

ZINTER

ZPOPMIN

ZADD

ZINTERCARD

ZRANDOMMEMBER

ZCARD

ZINTERSTORE

ZRANGE

ZCOUNT

ZLEXCOUNT

ZRANGEBYLEX

ZDIFF

ZMPOP

ZRANGEBYSCORE

Hash

- A container of unique fields and their values

```
HMSET profile:12345 user nnp id 12345 name "Natawut Nupairoj"  
balance 10
```

```
HGETALL profile:12345
```

```
HINCRBY profile:12345 balance 5
```

- Use
 - User profile information
 - Post. Information

Key	Value
profile:12345	{ user: "nap", id: "12345", name: "Natawut..." }

Hash Commands

HDEL

HLEN

HSTRLEN

HEXISTS

HMGET

HVALS

HGET

HMSET

HGETALL

HRANDFIELD

HINCRBY

HSCAN

HINCRBYFLOAT

HSET

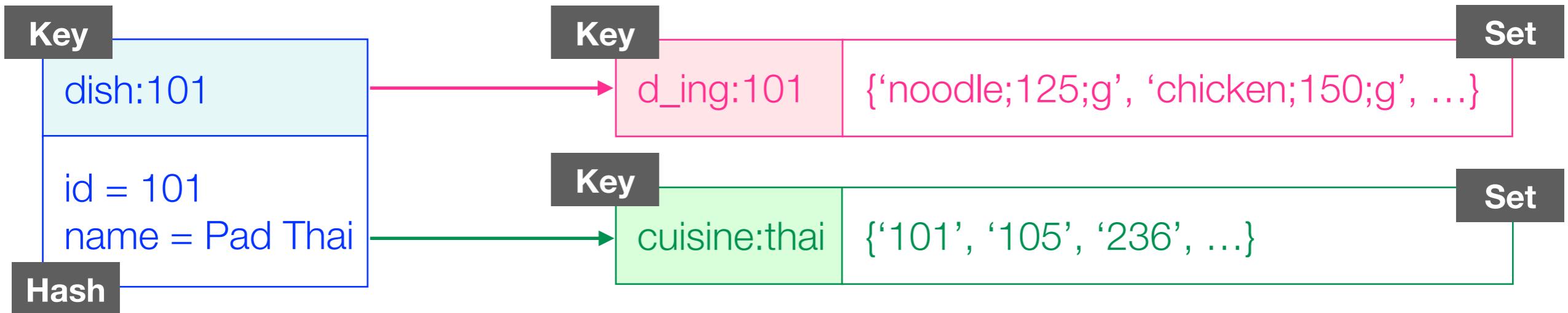
HKEYS

HSETEX

Example: Recipes in Redis

- Create data model for recipe
- Data items
 - Main data
 - Dish - hash - key = dish:dish_id, value = dish_id, name, category, cuisine
 - Dish Ingredients - set - key = d_ing:dish_id, value = string(name; size; unit)
 - For query
 - Cuisine - set - key = cuisine:cuisine_value, value = string(dish_id)
- Relationship
 - 1 dish can have multiple ingredients
 - Cuisine, as a helper data item, points back to dishes belong in the cuisine
- This is only one example of model design

Example: Recipes in Redis



```
HMSET dish:1 id 101 name "Pad Thai" cuisine "thai" category "noodle"
```

```
SADD d_ing:101 "noodle;125;g"
```

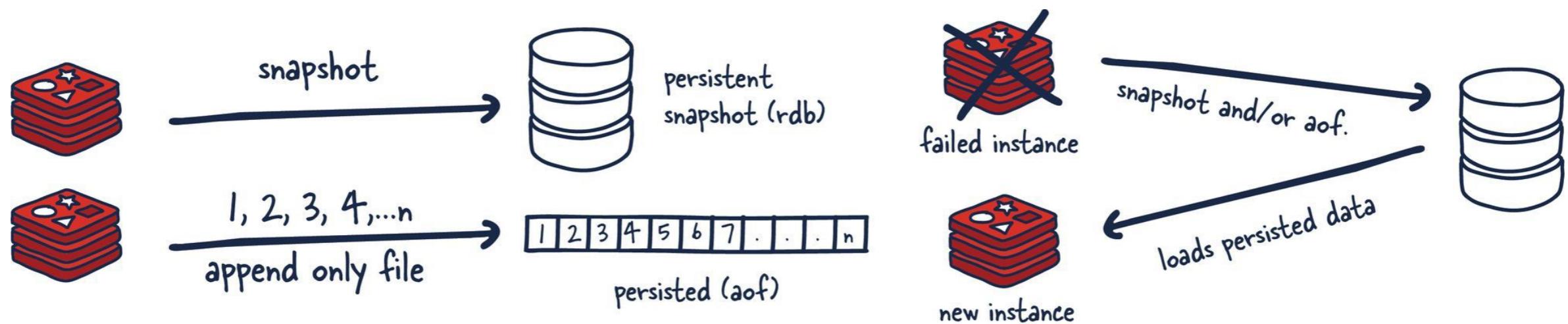
```
SADD d_ing:101 "shrimp;150;g"
```

```
SADD d_ing:101 "brown sugar;3;tbsp"
```

```
SADD cuisine:thai "101"
```

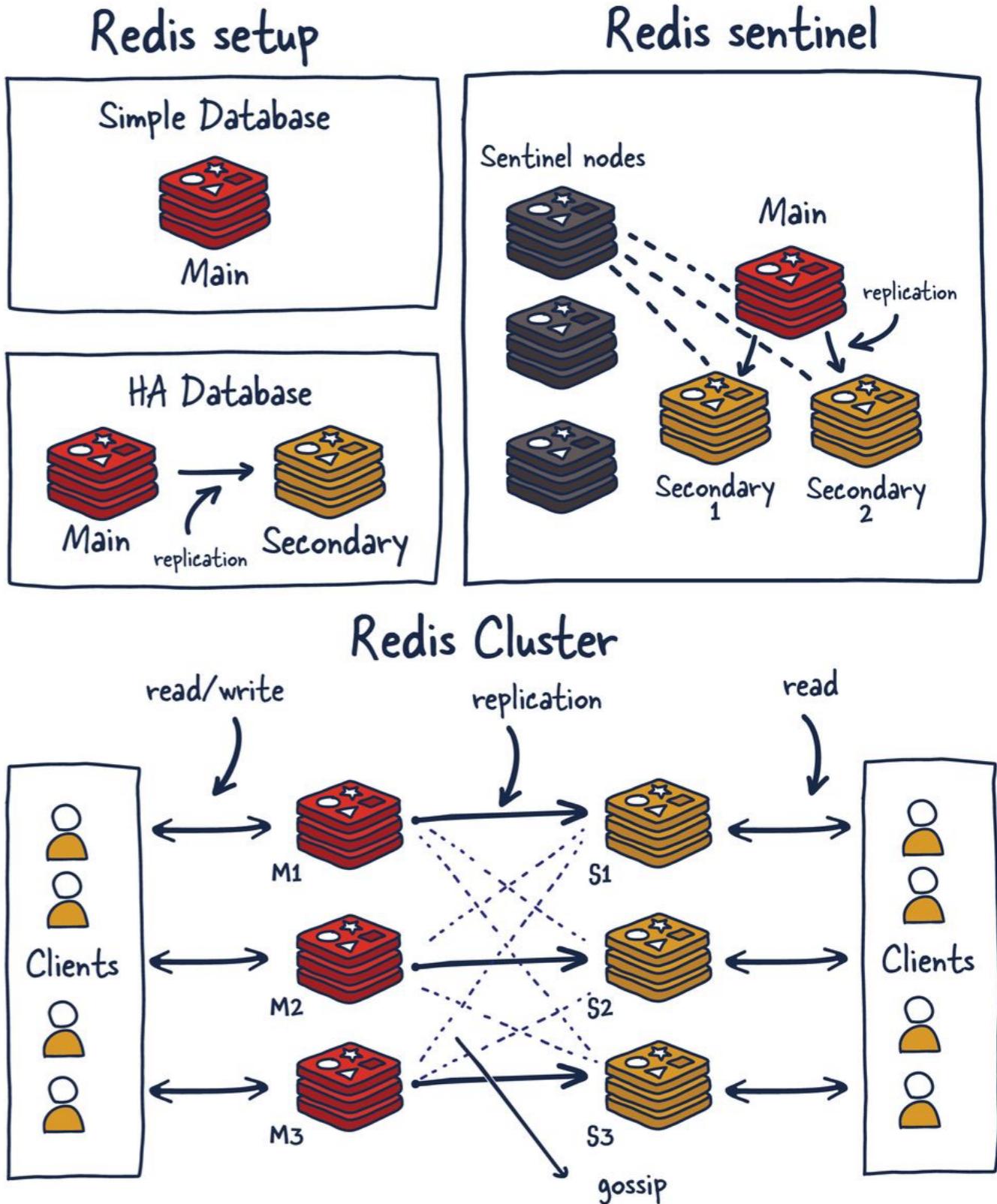
Redis Persistence

- Redis supports many level of persistence: no persistence, RDB (point-in-time snapshot), AOF (log every write), RDB+AOF



Redis Architecture

- Redis supports flexible architecture
 - Single instance
 - High Availability
 - Sentinel
 - Cluster

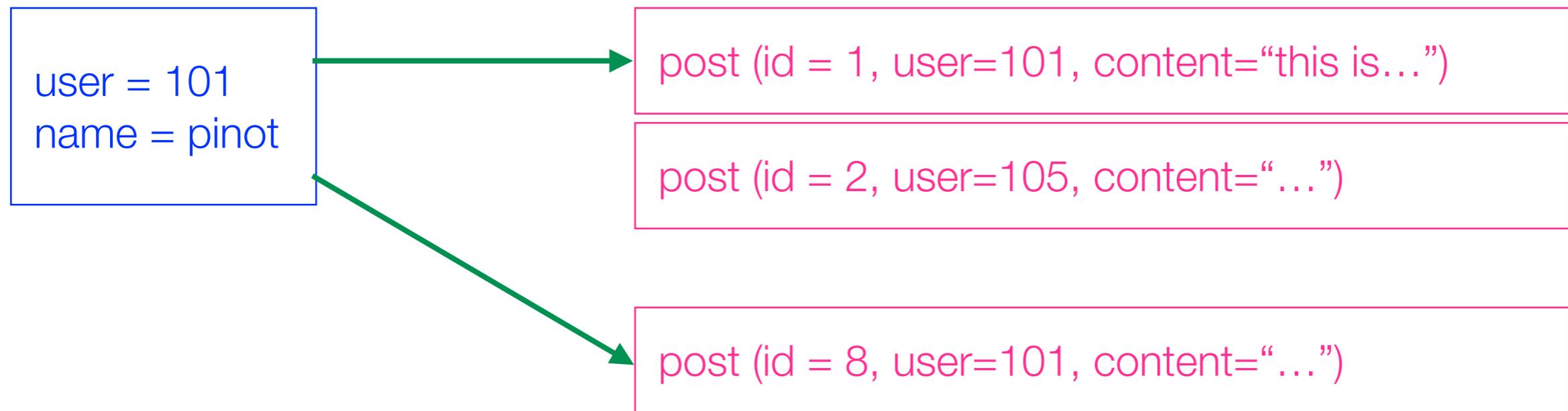


Source: <https://architecturenotes.co/redis/>

Example: Redis for “X-Like” Datastore

- Create data model for “X-Like”
- Data items
 - Users - id, name, can follow others, can be followed
 - Posts - id, content
- Relationship
 - 1 user can have many posts, each post can associate to only one user
 - User can follow one another

Users and Posts

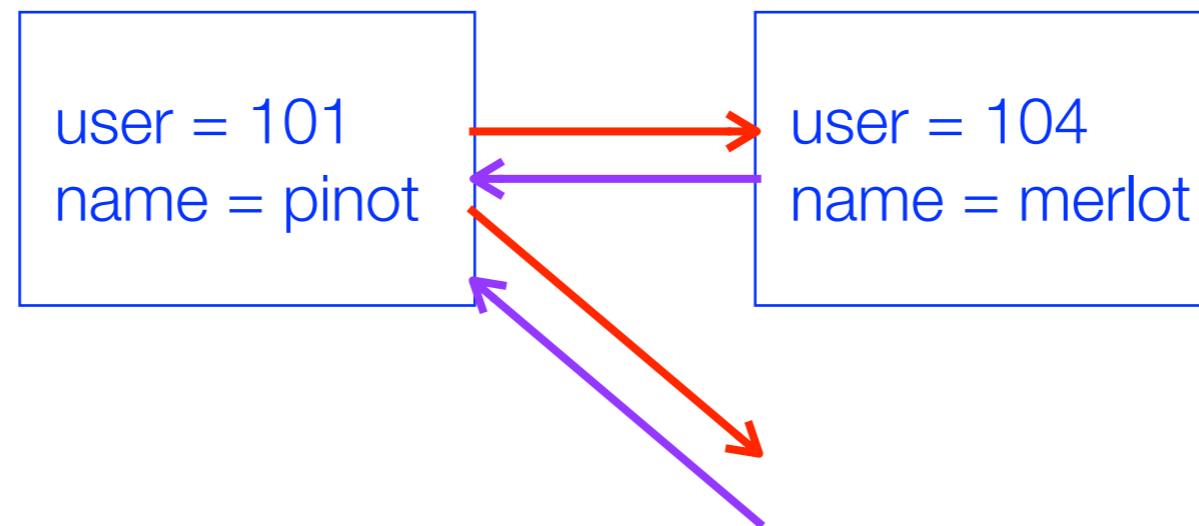


```
HMSET user:101:profile user 101 name "pinot"  
SET username:pinot:id 101           # refer back to user-id
```

```
HMSET post:1 user 101 content "this is the first post"
```

```
RPUSH user:101:post post:1  
RPUSH user:101:post post:8
```

Users and Followers



SADD **user:101:follows** 104

SADD **user:101:follows** 105

SADD **user:104:followed_by** 101

SADD **user:105:followed_by** 101

Simple Redis-Py Example

This notebook contains simple redis python commands.

For your local redis only

In [1]: `import redis`

Connect to local server -- no hostname or ip is needed

In [2]: `rd = redis.Redis(charset="utf-8", decode_responses=True)`

In [3]: `rd.set('user:101:name', 'pinot')`

Out[3]: True

In [4]: `rd.get('user:101:name')`

Out[4]: 'pinot'

In [5]: `rd.hset('post:1', 'user', 101)
rd.hset('post:1', 'content', 'this is the first post')`

Out[5]: 1

In [6]: `rd.hgetall('post:1')`

Out[6]: {'user': '101', 'content': 'this is the first post'}

In [7]: `rd.rpush('user:101:post', 1)
rd.rpush('user:101:post', 8)`

Out[7]: 2

In [8]: `rd.llen('user:101:post')`

Out[8]: 2

```
In [9]: rd.lrange('user:101:post', 0, -1)
```

```
Out[9]: ['1', '8']
```

```
In [10]: rd.sadd('user:101:follows', 104)
rd.sadd('user:101:follows', 105)
```

```
Out[10]: 1
```

```
In [11]: rd.scard('user:101:follows')
```

```
Out[11]: 2
```

```
In [12]: rd.smembers('user:101:follows')
```

```
Out[12]: {'104', '105'}
```

```
In [13]: cursor = 0
cursor, keys = rd.scan(cursor=cursor, match='user:*')
while cursor > 0:
    for key in keys:
        print('found: ', key)
    cursor, keys = rd.scan(cursor=cursor, match='user:*')

for key in keys:
    print('found: ', key)
```

```
found: user:101:post
found: user:101:follows
found: user:101:name
```

References

- ScaleGrid, “Top Redis Use Cases by. Core Data Structure Types”, <https://scalegrid.io/blog/top-redis-use-cases-by-core-data-structure-types/>
- Jerry An, “The most important Redis data structures you must understand”, <https://medium.com/analytics-vidhya/the-most-important-redis-data-structures-you-must-understand-2e95b5cf2bce>
- Brad Solomon, “How to use Redis with python”, <https://realpython.com/python-redis/>
- M. Yusuf, “Redis Explained”, <https://architecturenotes.co/redis/>