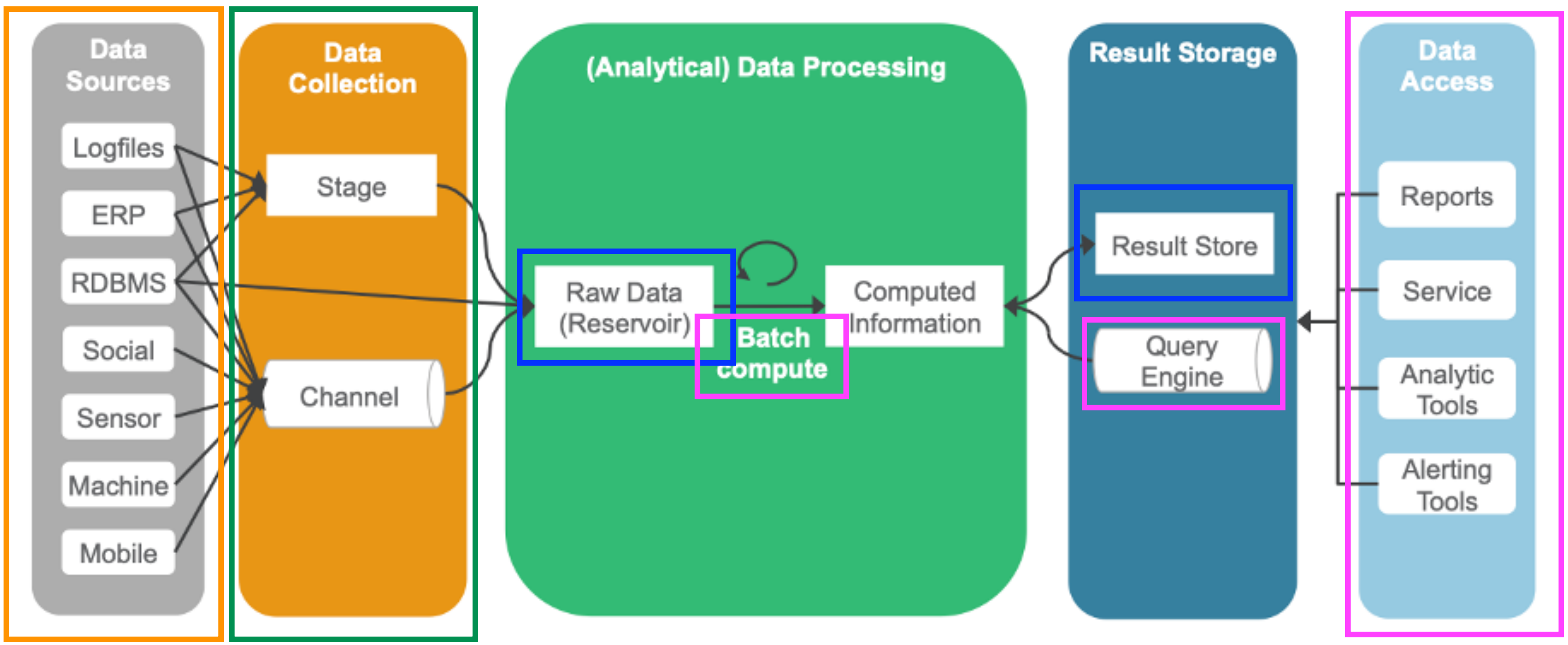
2110531 - Data Science and Data Engineering Tools

Data Storages

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

CHULA ENGINEERING
Foundation toward Innovation

Simple Big Data Analytic Architecture



Data Generation Data Ingestion Data Storage Data Analytics

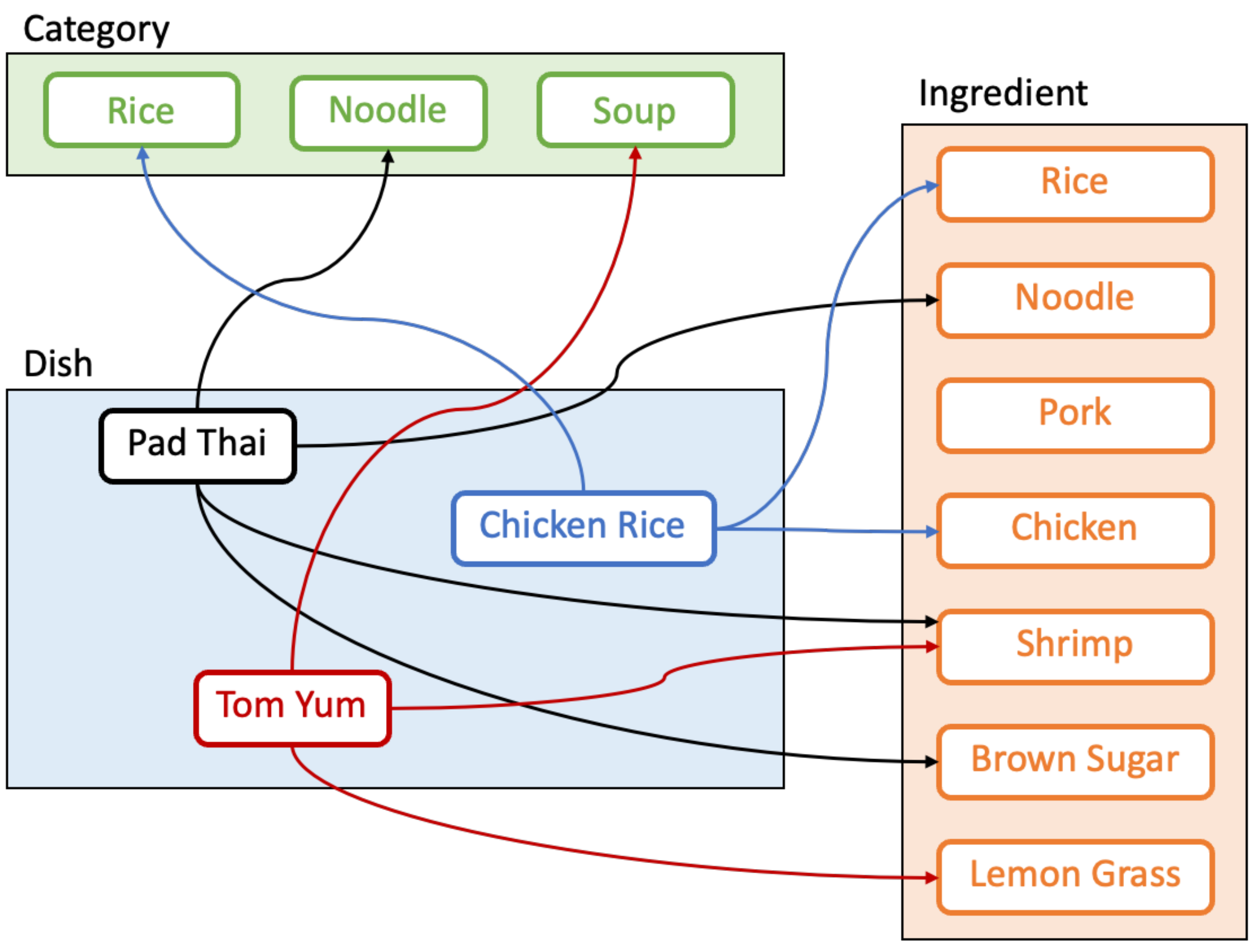
Traditional Database

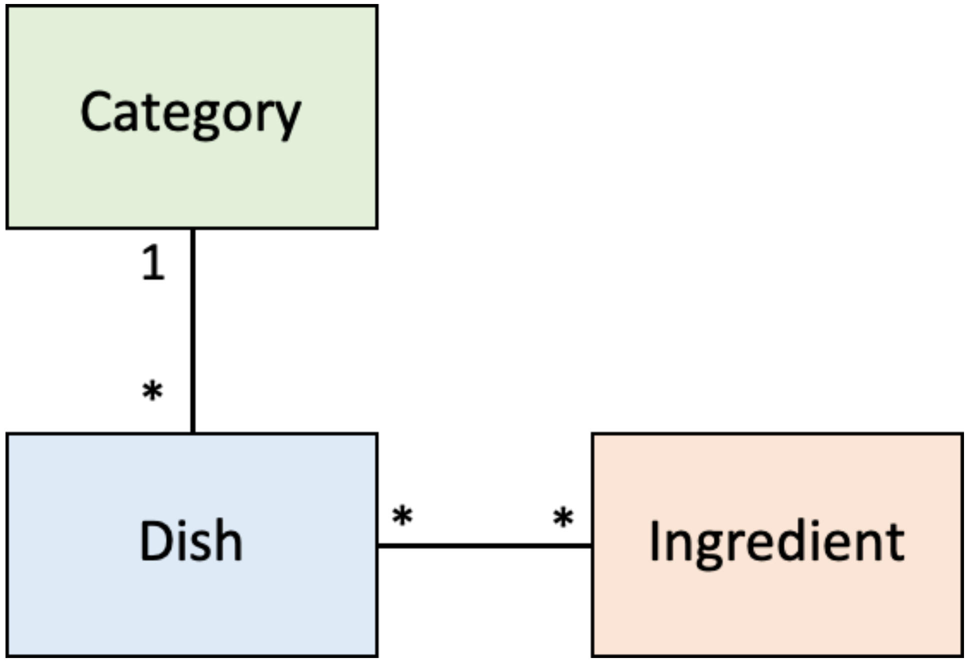
- Based on ”relational model”
  - Data is split and stored into tables
  - Tables can be processed together using set-like operations
  - Data model is usually normalized to remove duplication
- Very suitable for OLTP or transaction systems
  - Provide lots of complicated SQL operations
  - Lots of inserts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

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

Relational Database: Normalized Data Model



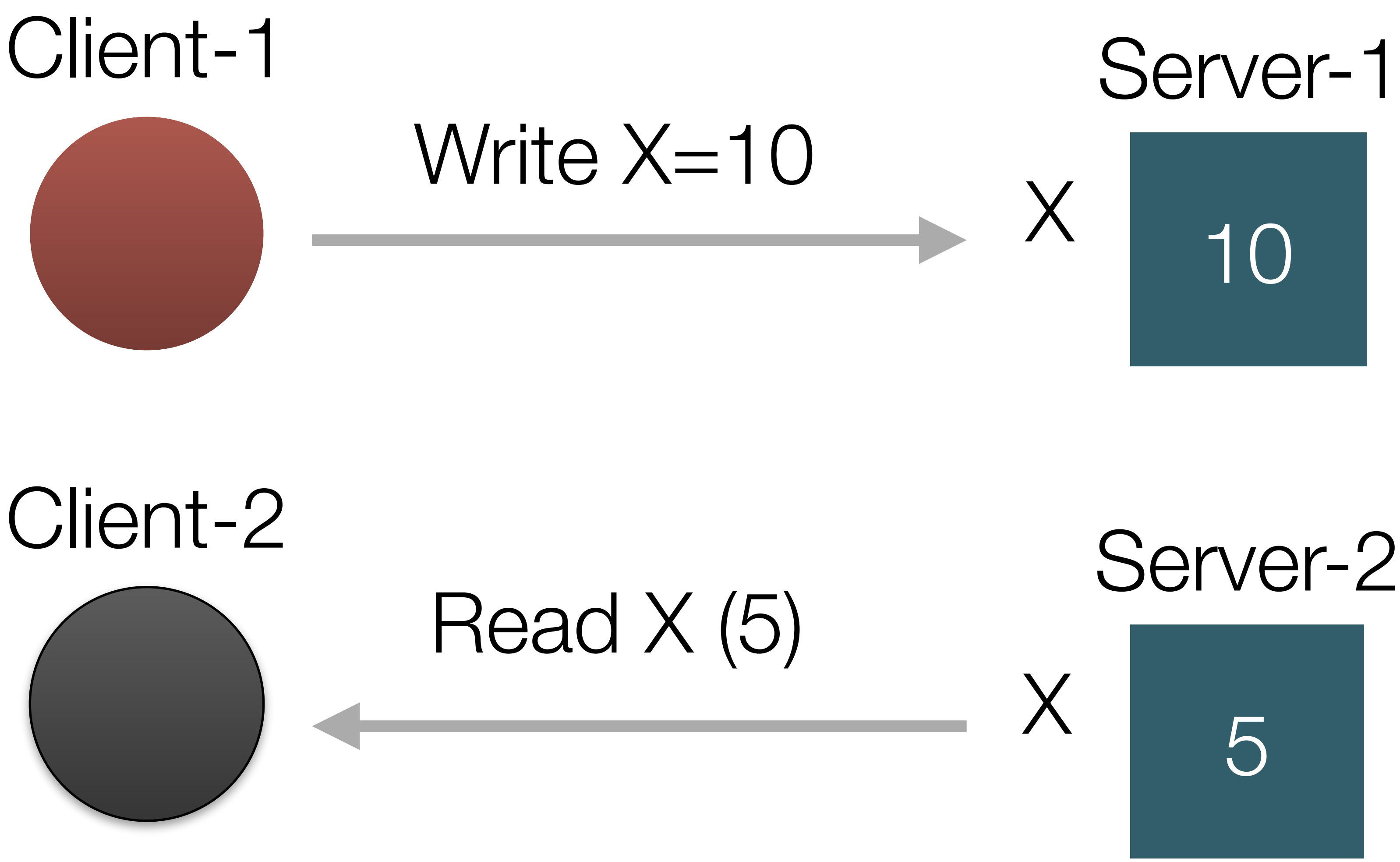


How can we split these tables to lots of machines?

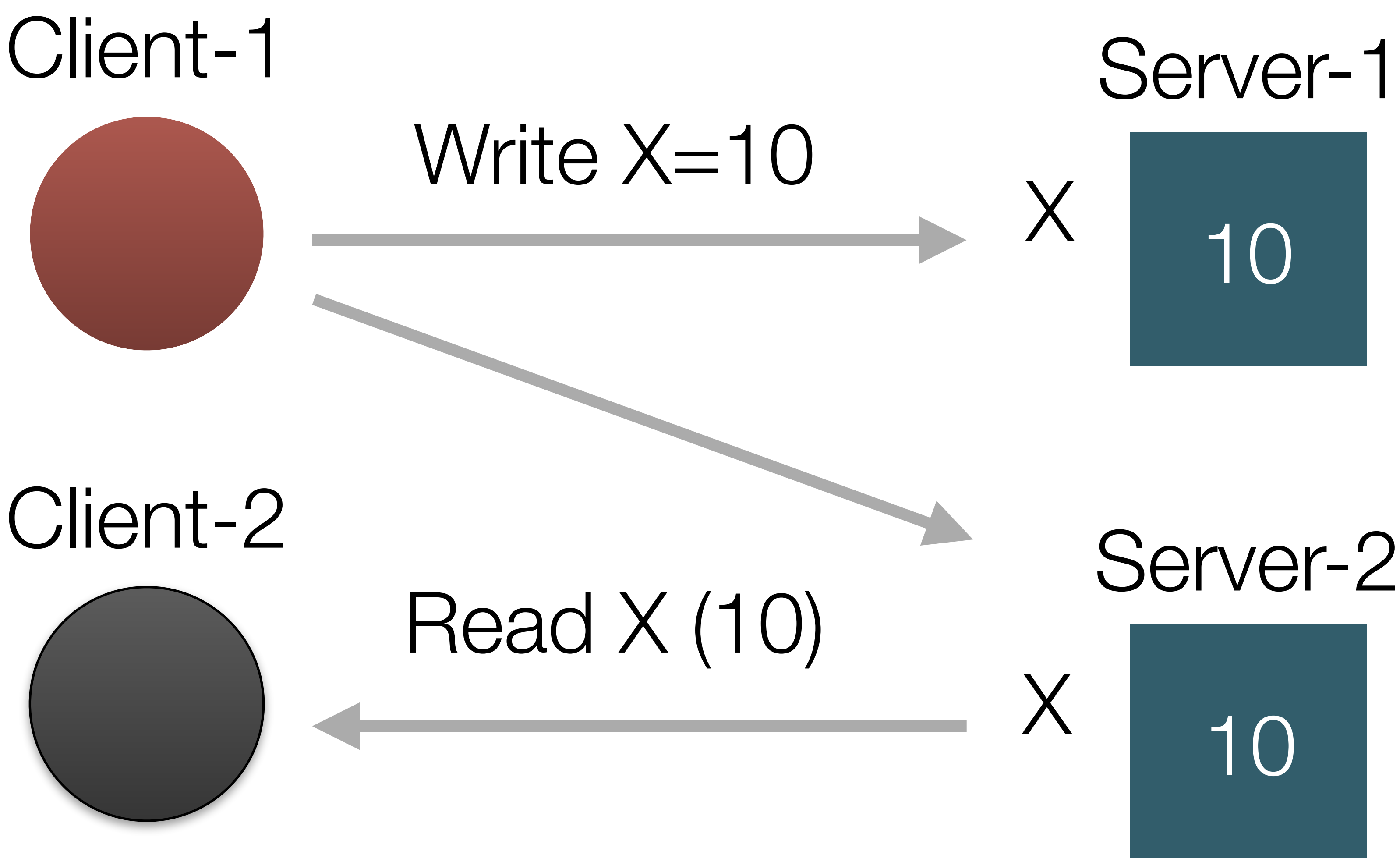
- Dishes of the same category in the same machine
- How about dishes and ingredients?

Or we can replicate data — lead to data consistency problems

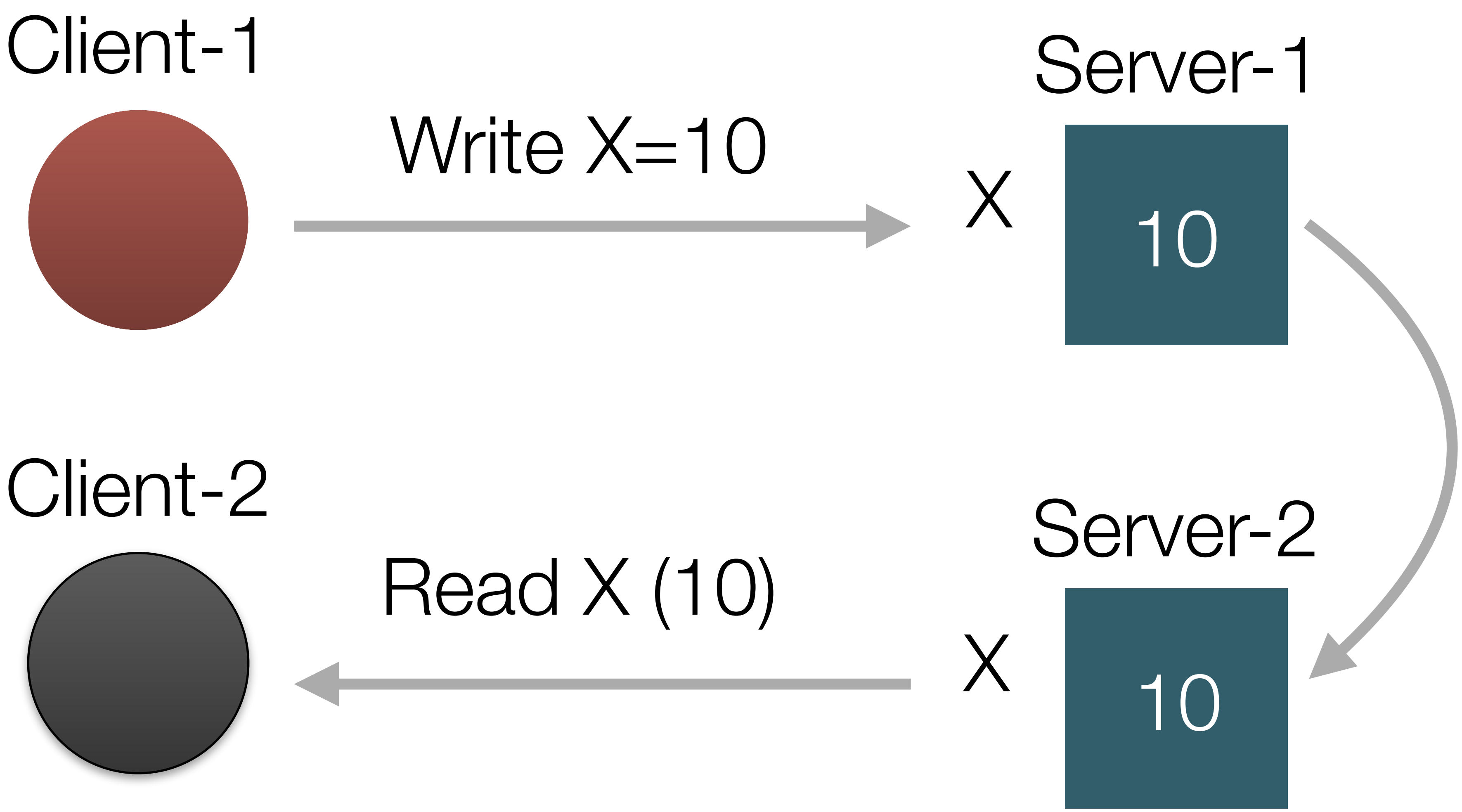
Replication and Data Consistency Problem



Solution to Data Consistency - Active Replication



Solution to Data Consistency - Passive Replication



Data Consistency Models

- Strong consistency
  - After update completes, any access will return the updated value
  - High costs for large scale system
- Weak consistency
  - Certain conditions must be met before the consistency is guaranteed
  - Time between update completion and guaranteed consistency is called inconsistency window

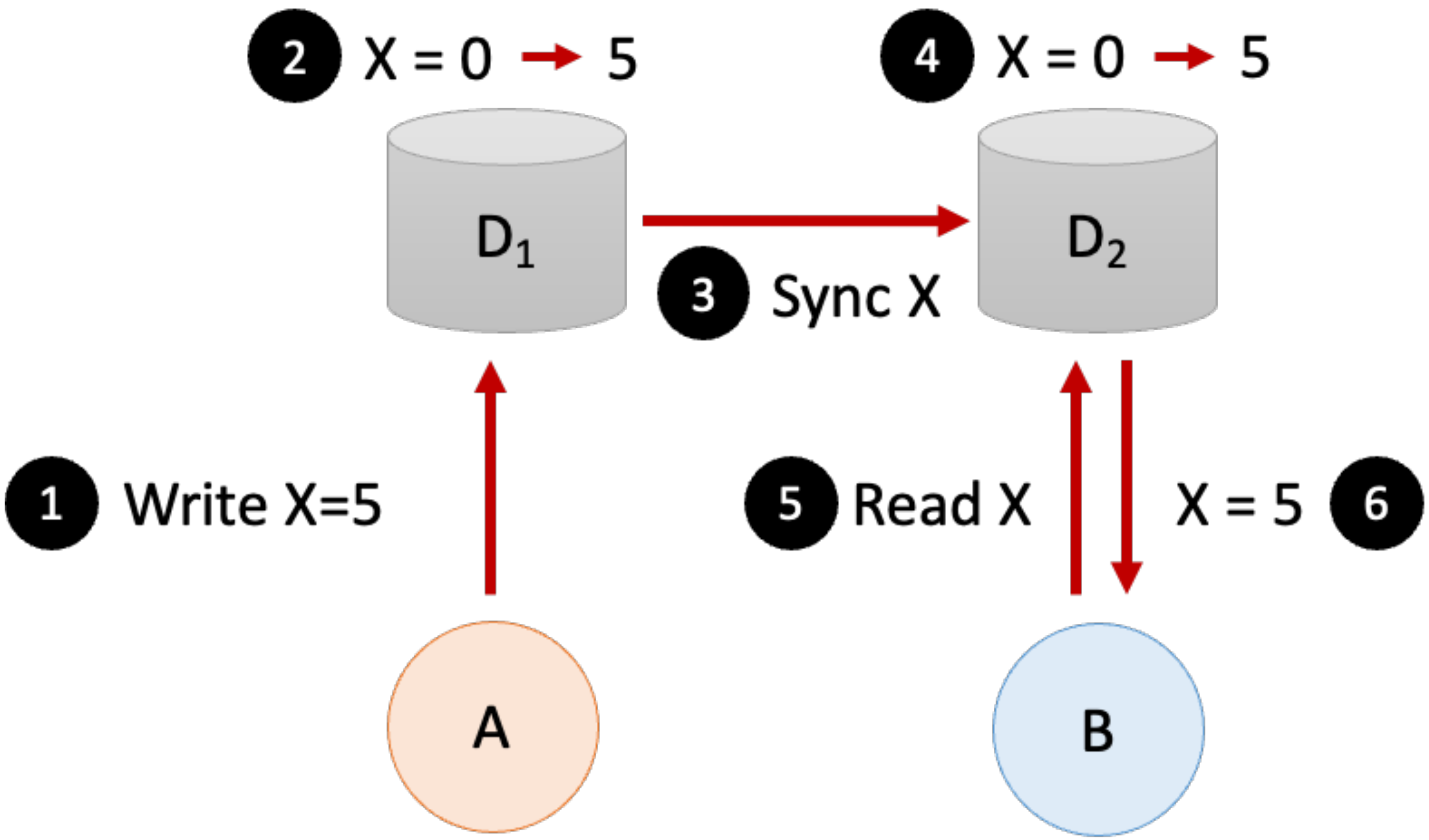
CAP Theorem (Brewer’s Theorem)

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

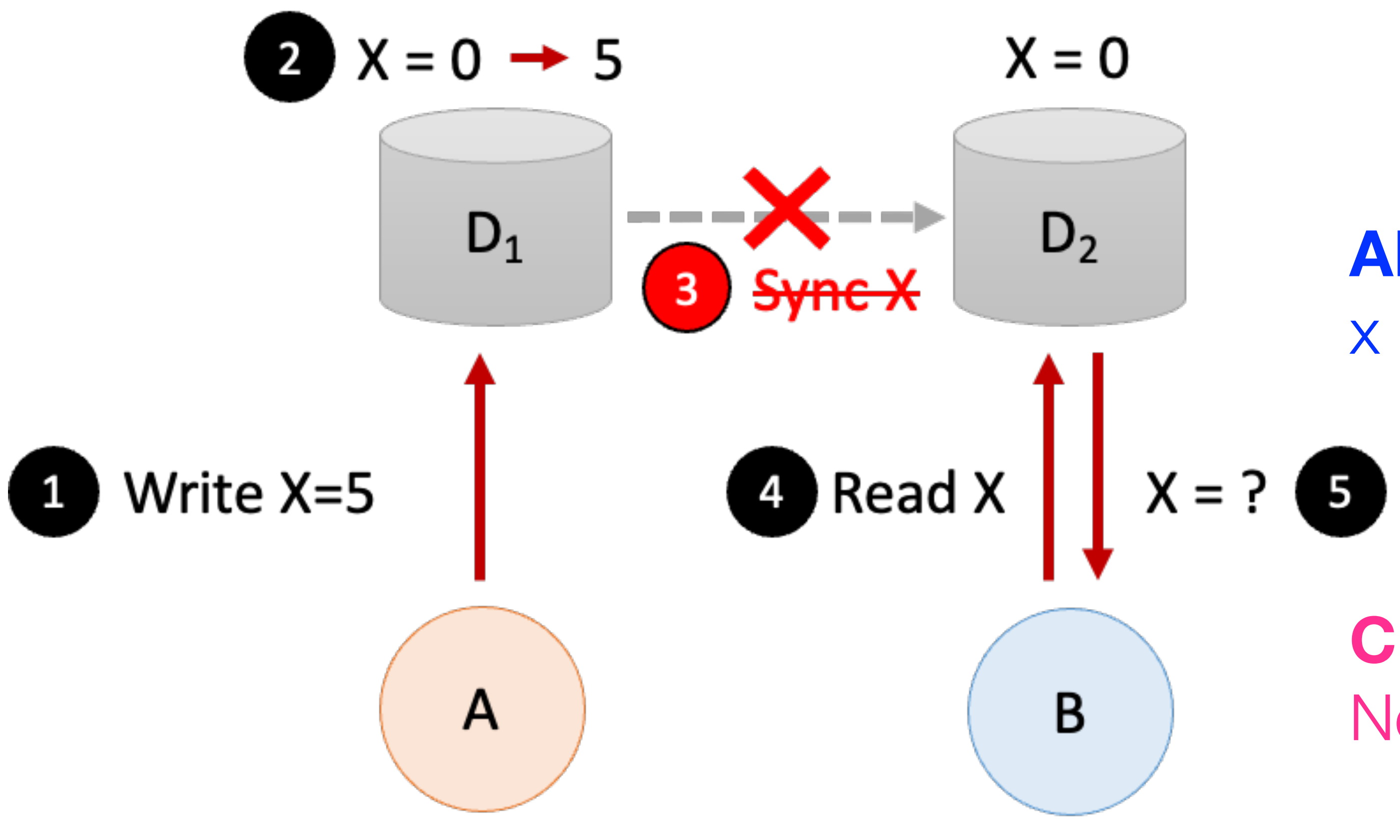
CAP Guarantees only 2 out of 3

- 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)



AP Data Store
x = 0 (not consistent, but available)

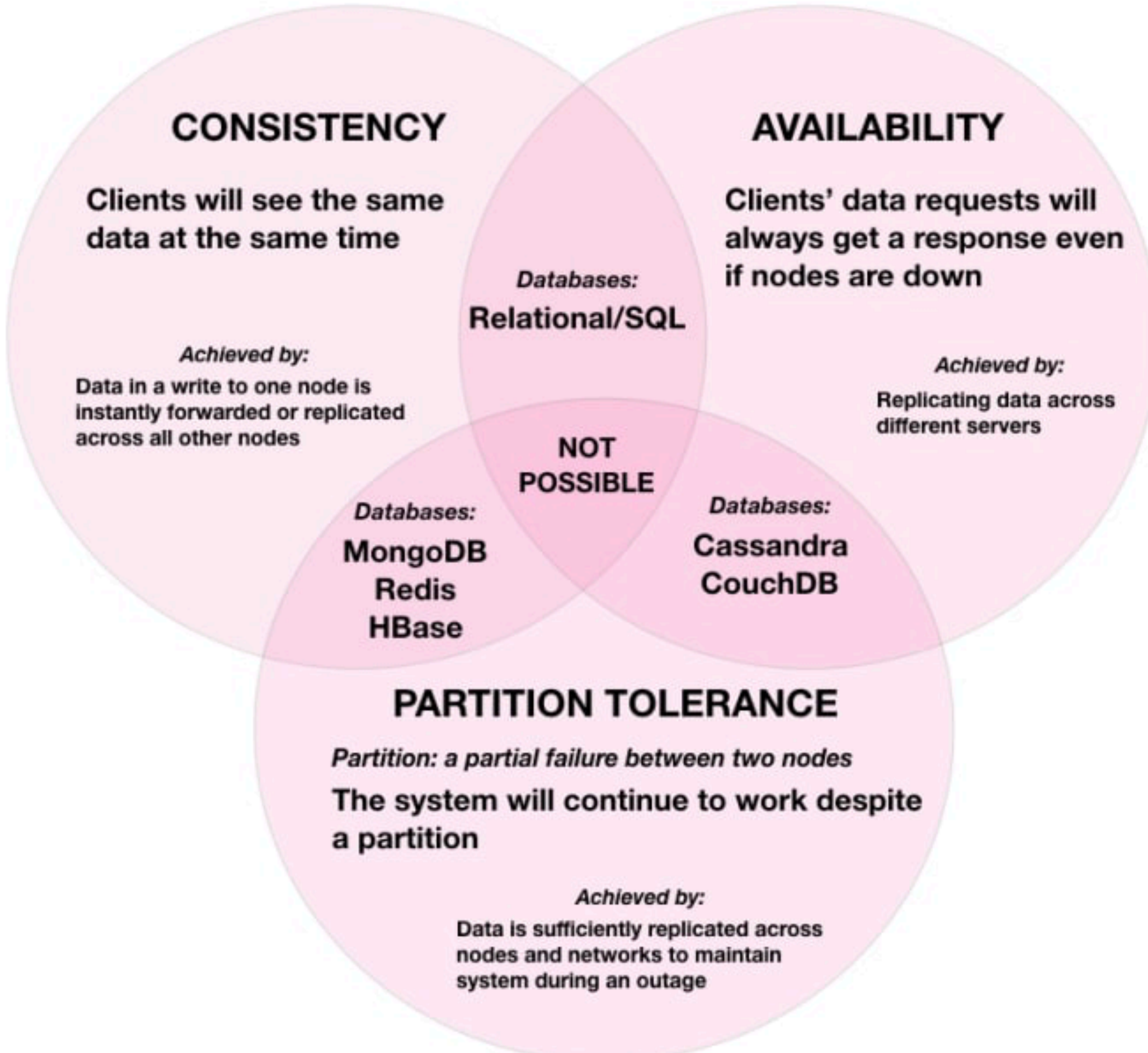
CP Data Store
No response

The Landscape of NoSQL

- Alternatives to SQL database - non-relational, distributed, and horizontally scalable
- Data is shared and distributed across multiple servers
- Typically use weak consistency model (but not always)
- Examples
  - Document: MongoDB, DynamoDB, CosmosDB, Couchbase, Firebase
  - Column: Cassandra, HBase, CosmosDB, Accumulo
  - Key-value: Redis, DynamoDB, CosmosDB, MemcacheDB
  - Graph: Neo4J, CosmosDB, ArrangoDB, OrientDB
  - Search Engine: Elasticsearch, Splunk, Solr

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



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: session store, shopping cart, SQL/API caching, task queue, etc.

Redis (Remote Dictionary Server)

- In-memory data structure store with clustering, transactional, time-to-live limiting, and auto-failover capabilities

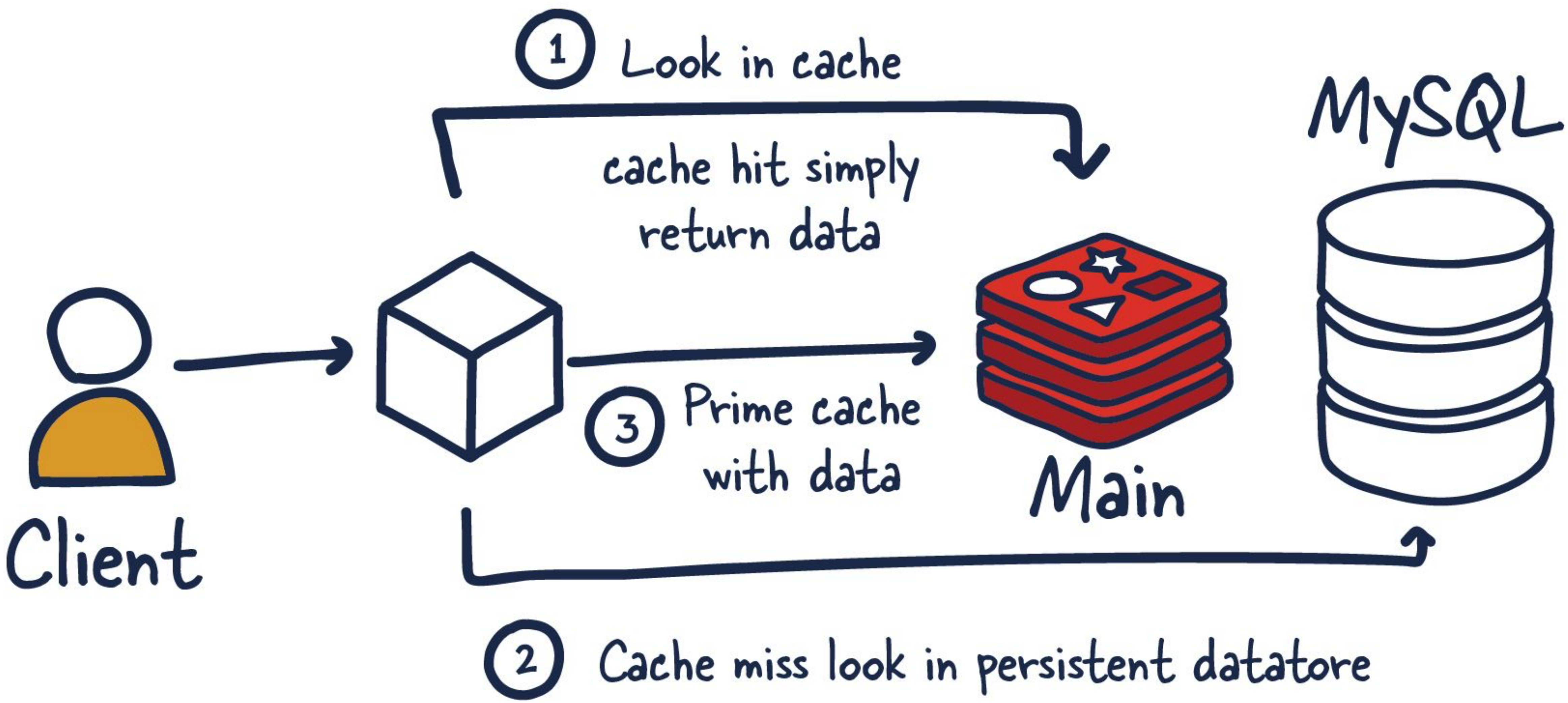
- Support wide-range of data structure with lots of related operations for each structure

- 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.

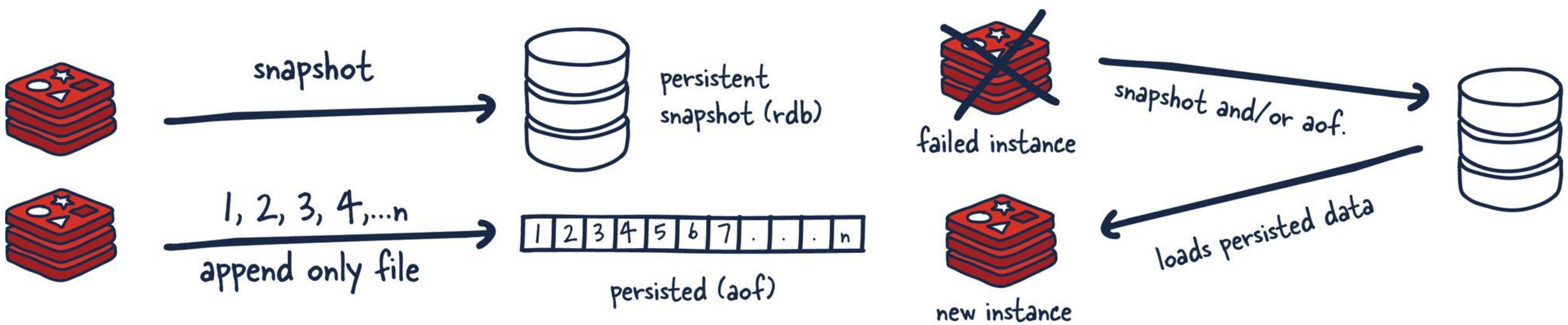
How is redis traditionally used



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

Redis Persistence

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



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

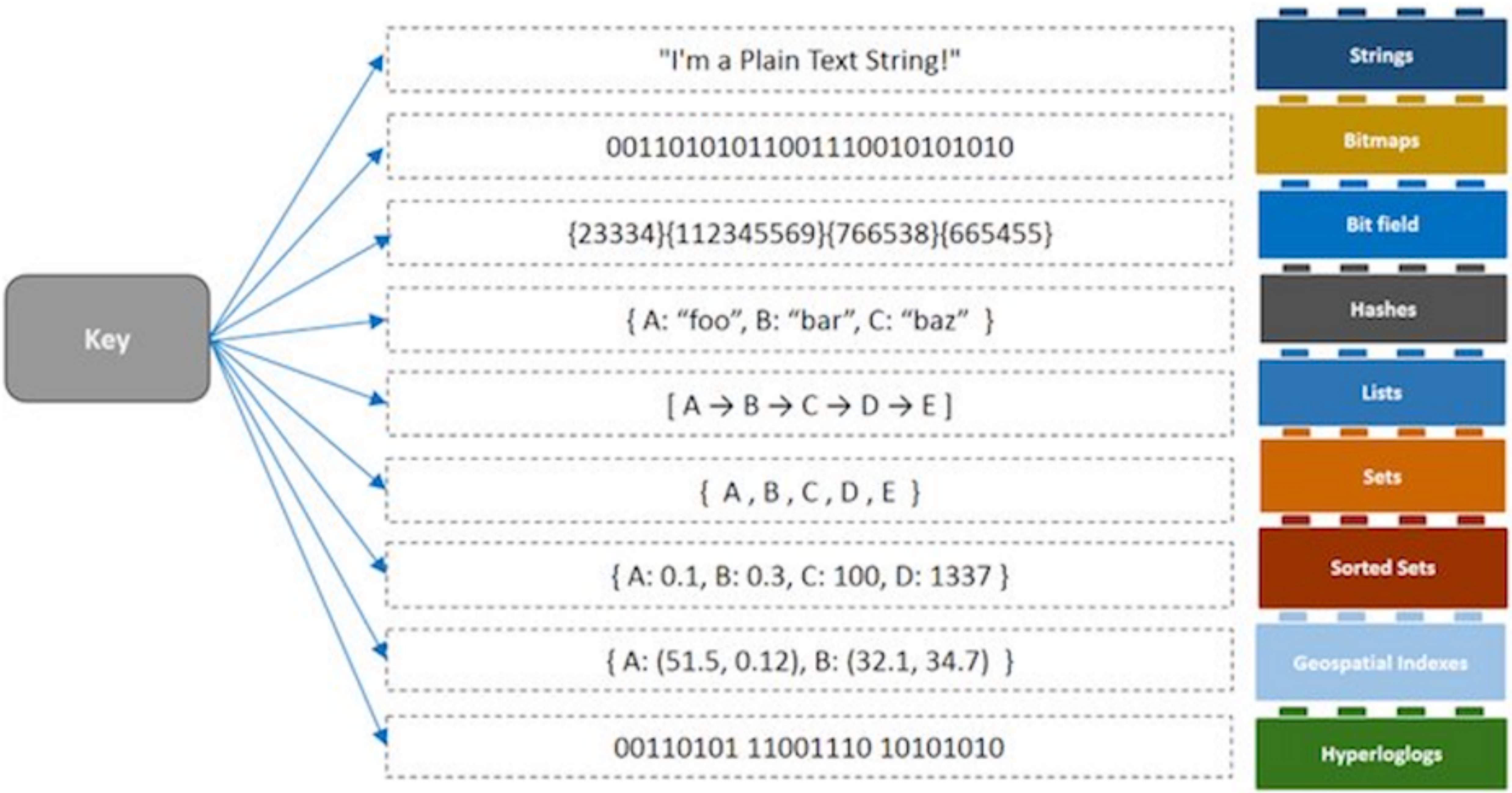
- Redis CLI
  - Standard client program to connect to any redis server
  - Come with any redis installation (see: https://redis.io/docs/getting-started/)

```
redis-cli

redis-cli -h lab.aimet.tech
```

  - You can type in redis command in the CLI input

Redis 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, shopping cart, user profile
  - Queues
  - Activity tracking

Other String Commands

APPEND

DECR

DECRBY

GET

GETDEL

GETEX

GETRANGE

GETSET

INCR

INCRBY

INCRBYFLOAT

LCS

MGET

MSET

MSETNX

PSETEX

SET

SETEX

SETNX

SETRANGE

STRLEN

SUBSTR

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
  - Timelines

List Commands

BLMOVE
BLMPOP
BLPOP
BRPOP
BRPOPLPUSH
LINDEX
LINSERT
LLEN
LMOVE
LMPOP
LPOP
LPOS
LPUSH
LPUSHX
LRANGE
LREM
LSET
LTRIM
RPOP
RPOPLPUSH
RPUSH
RPUSHNX

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

Set Commands

SADD

SCARD

SDIFF

SDIFFSTORE

SINTER

SINTERCARD

SINTERSTORE

SISMEMBER

SMEMBERS

SMISMEMBER

SMOVE

SPOP

SRANDMEMBER

SREM

SSCAN

SUNION

SUNIONSTORE

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

BZPOPMAX

BZPOPMIN

ZADD

ZCARD

ZCOUNT

ZDIFF

ZDIFFSTORE

ZINCRBY

ZINTER

ZINTERCARD

ZINTERSTORE

ZLEXCOUNT

ZMPOP

ZMSCORE

ZPOPMAX

ZPOPMIN

ZRANDMEMBER

ZRANGE

ZRANGEBYLEX

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
HINCRBYprofile:12345 balance 5
```

- Use
    - User profile information
    - Post. Information

Hash Commands

HDEL

HEXISTS

HGET

HGETALL

HINCRBY

HINCRBYFLOAT

HKEYS

HLEN

HMGET

HMSET

HRANDFIELD

HSCAN

HSET

HSETNX

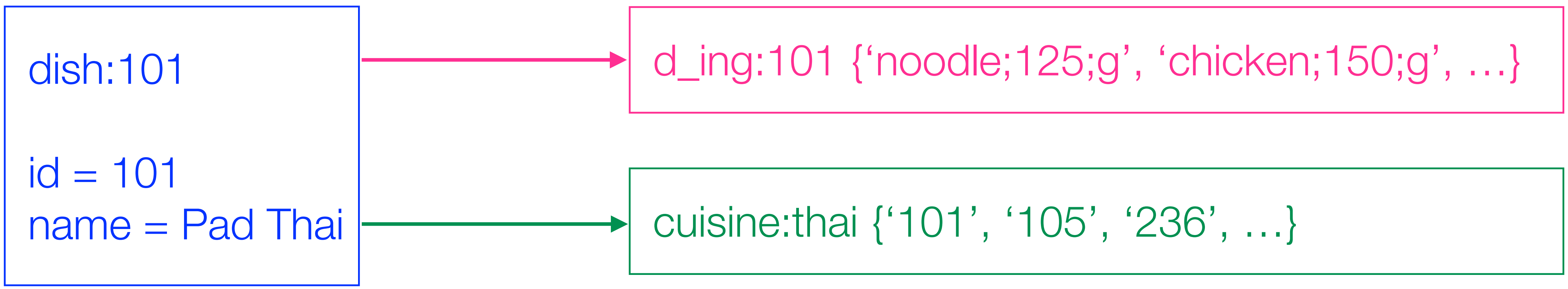
HSTRLEN

HVALS

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



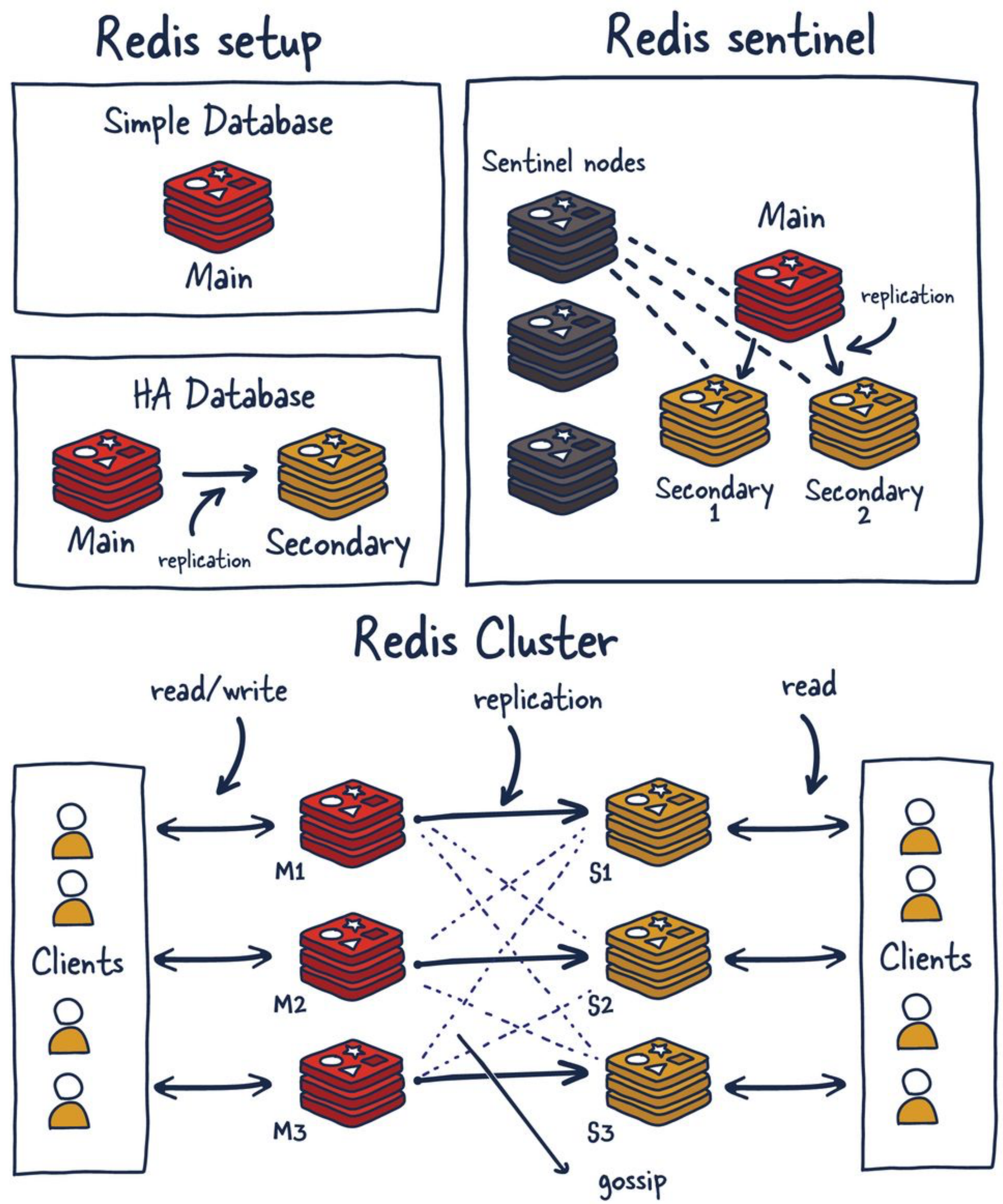

```
HMSET dish:1 id 101 name “Pad Thai” cuisine “thai” category “noodle”

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 Architecture

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



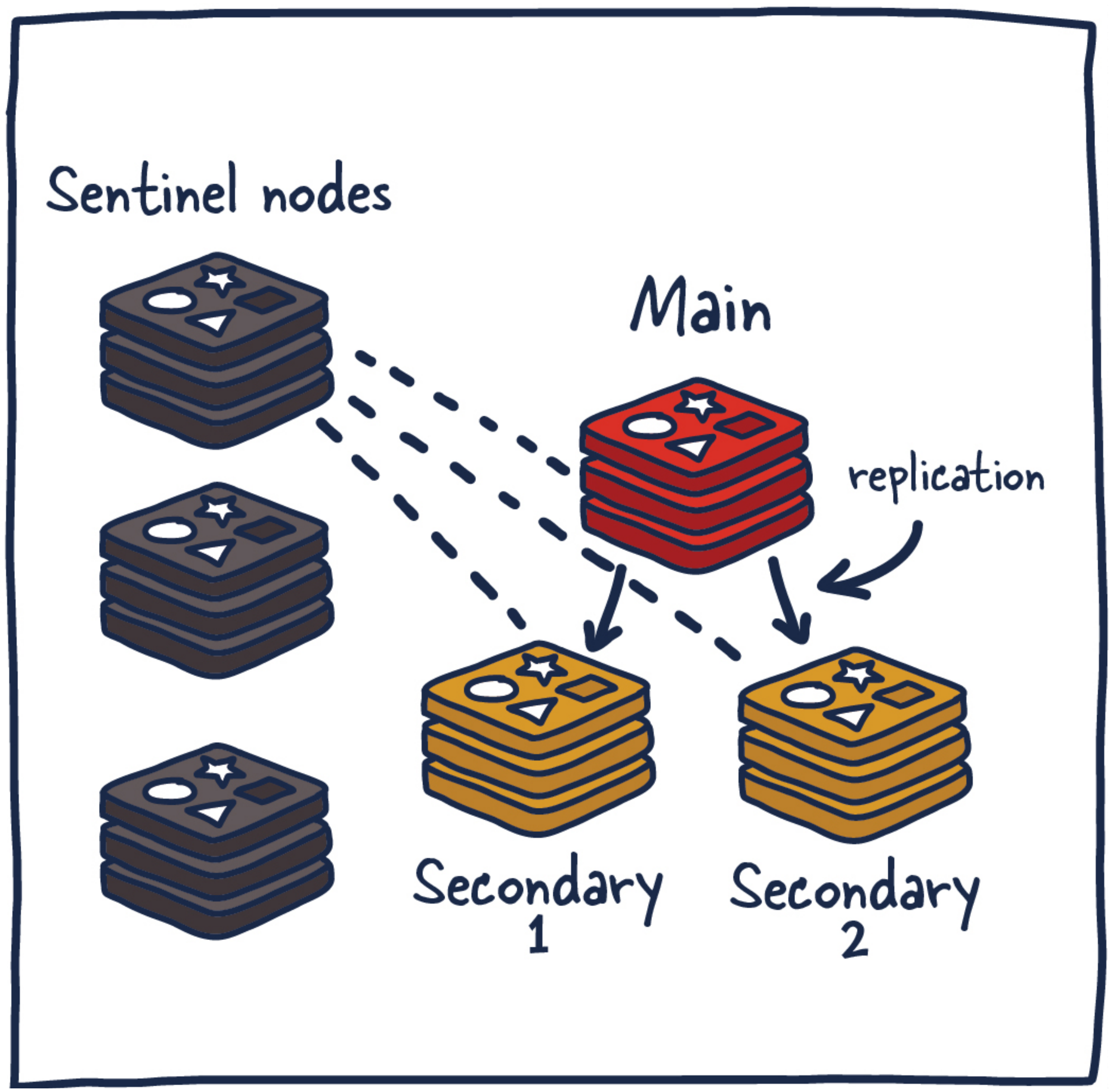
Source: https://architecturenotes.co/redis/

Redis High Availability

- When setup for HA, Redis consists of a master and one or more replicas and follows the following rules
  - Master performs all commands (read/write/etc.), replica performs only read commands
  - When a master and a replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica to replicate the effects on the dataset
  - When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization
  - When a partial resynchronization is not possible, the replica will ask for a full resynchronization
  - Redis uses asynchronous replication, with asynchronous replica-to-master acknowledges of the amount of data processed

Redis High Availability

- To ensure data synchronization, all servers keep data version (replication id, offset); equal data versions mean both servers hold the same data
  - When a server becomes a master, it generates a new replication ID; replica server inherits replication ID from its master once connected
  - When master updates its data, it increments offset by one
- When replicas connect to masters, they use the PSYNC command to send their old master replication ID and the offsets they processed so far
  - If the differences of offsets are not too far behind, partial synchronization is performed
  - If too far or mismatched replication ID, a full resynchronization happens: in this case the replica will get a full copy of the dataset, from scratch



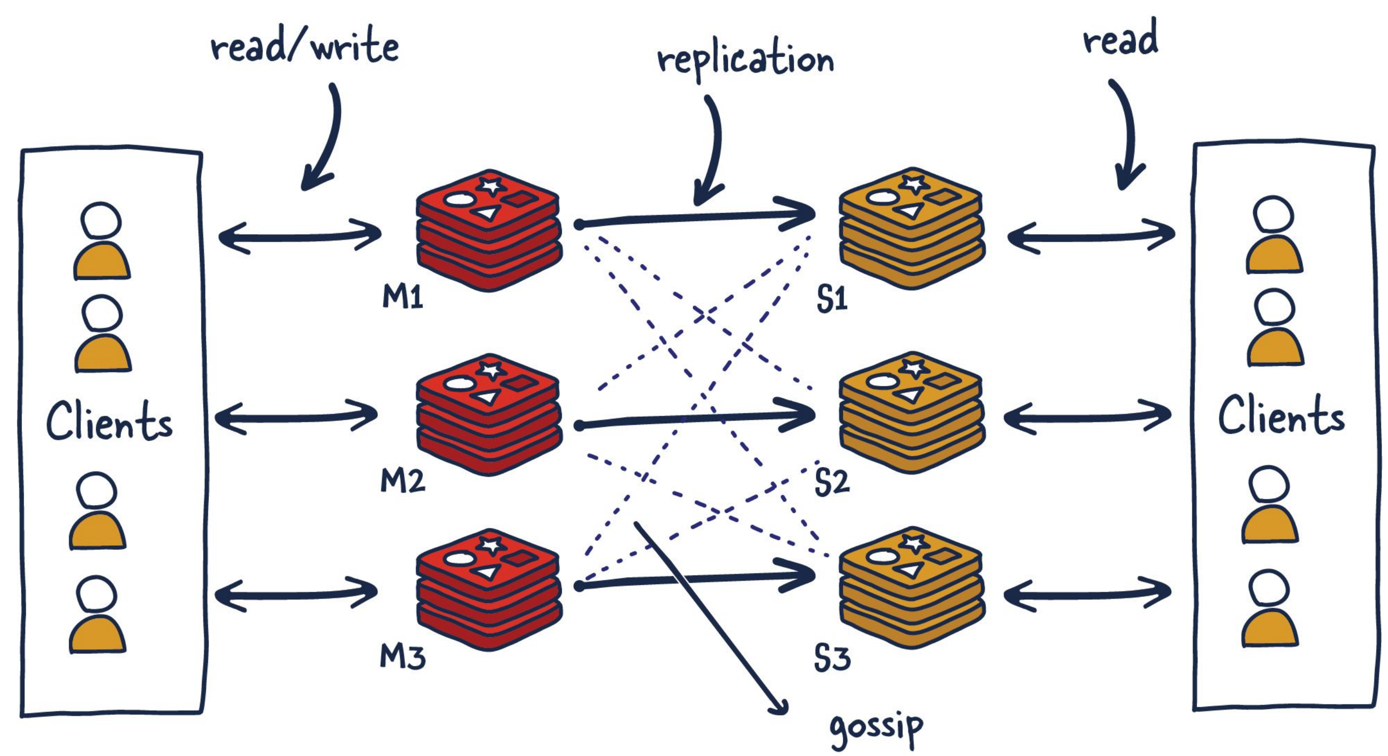
Source: https://architecturenotes.co/redis/

Redis Sentinel

- A cluster of sentinel processes working together to coordinate states of Redis master (main) and replicas (secondaries)
- Focus on HA aspects with minimum number of nodes
- Support single master with multiple replicas with monitoring, notification, auto-failover, configuration providing
- Need at least three Sentinel instances for a robust deployment (quorum or agreement protocol)
- Can be deployed in Redis and client nodes

Redis Cluster

- Support algorithmic sharding for horizontal scaling
    - Incoming data is hashed into a hash slot using deterministic hash function and mod with number of nodes
    - Data with a given key always maps to the same shard
    - Perform resharding when adding new nodes
- There are 16k hashslot; all data in the same hashslot always moves to the new node when resharding
- Nodes communicate with one another using gossiping protocol to determine the entire cluster's health
- Redis Cluster does not guarantee strong consistency



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

Redis in Python

- Redis-Py
  - Standard python package for redis client

```
pip install redis
```

  - Example

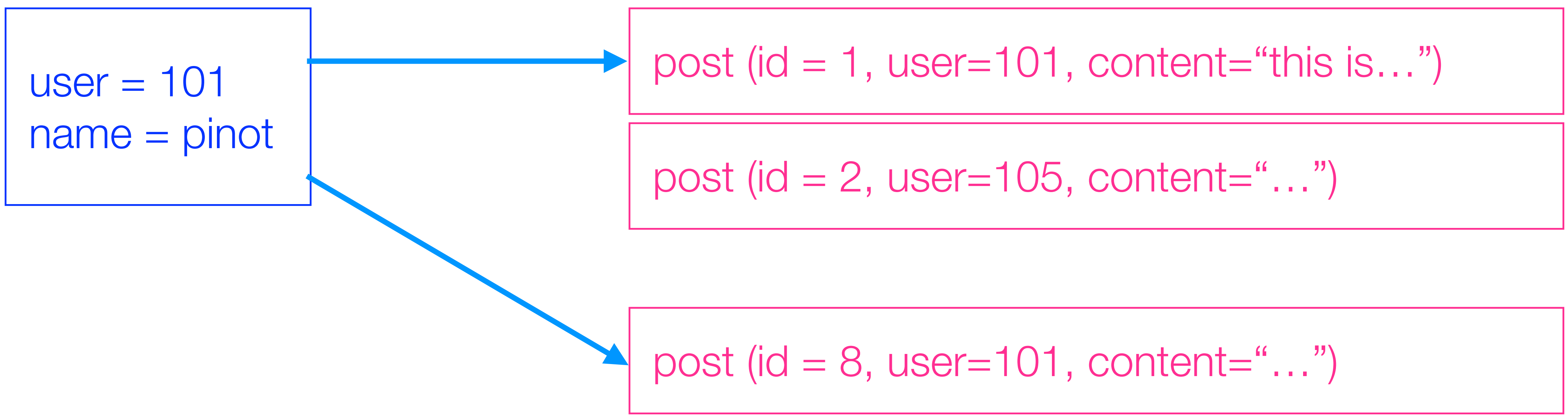
```
import redis

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

Example: Simple Social Network

- Create data model for simple social network
- 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

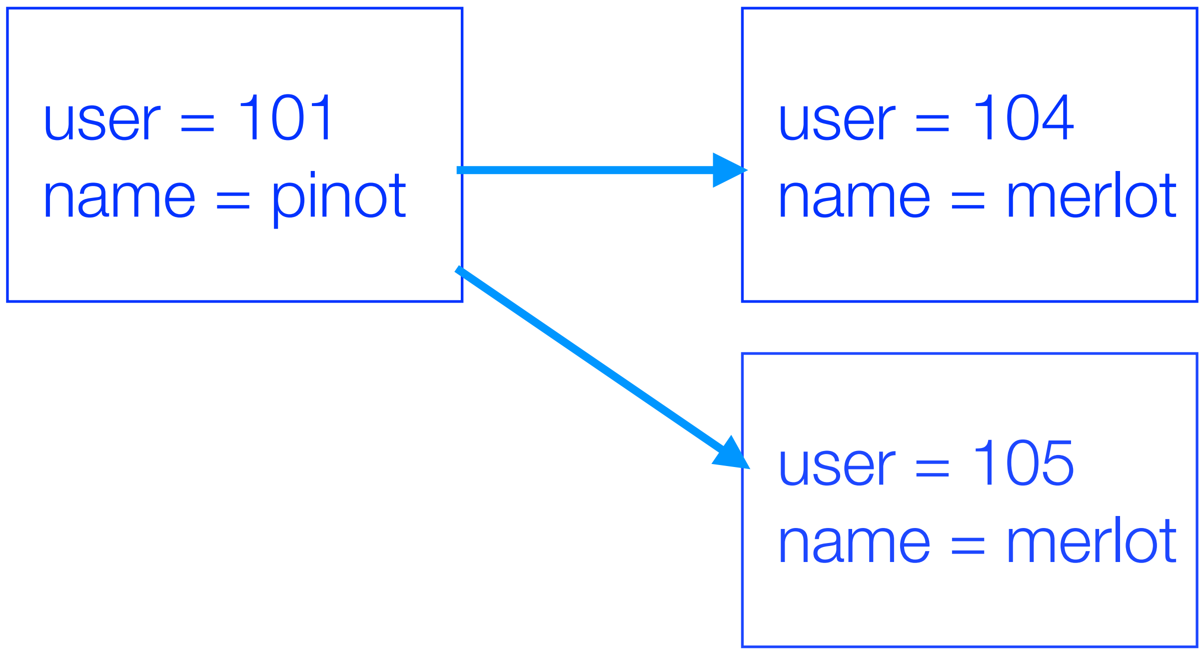



```
SET user:101:name pinot        # user id 101 with name "pinot"
SET username:pinot 101         # refer back to user-id

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

RPUSH user:101:post 1
RPUSH user:101: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

For your local redis only

This notebook contains simple redis python commands.

```
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
```

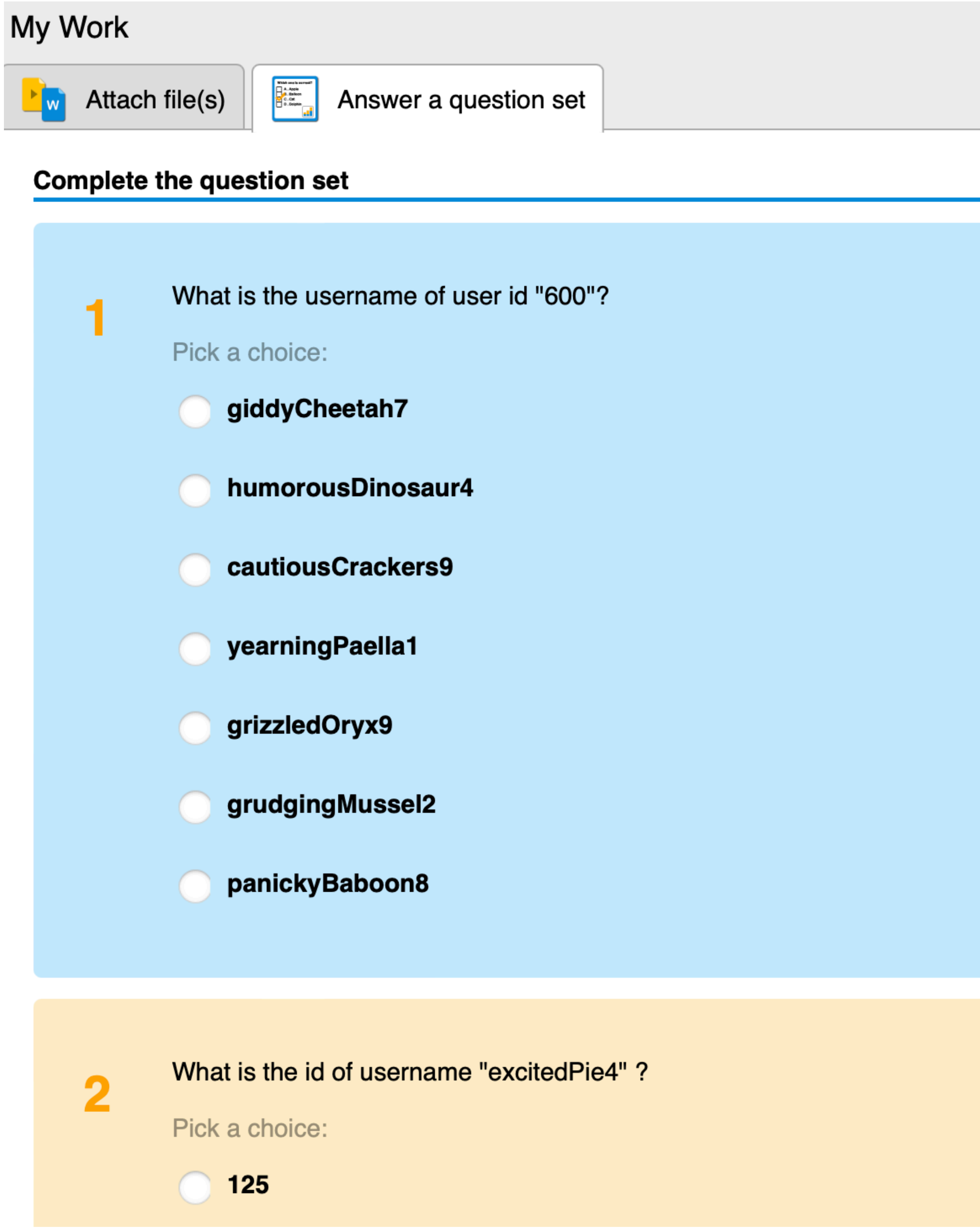
In [12]:
```
rd.smembers('user:101:follows')
```
Out[12]:
```
{'104', '105'}
```

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='username:*')

for key in keys:
    print('found: ', key)
```
```
found:  user:101:follows
found:  user:101:name
found:  user:101:post
```

Assignment: Simple Social Network

- Use the redis assignment notebook in class git repo and provide the following answers:
  - Answer questions in the question set
  - Save your notebook solution in the PDF format similar to the example in class git repo (used File -> Print Preview and then export as PDF or print to PDF with your browser)



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/