

DS 4300

# NoSQL & KV DBs

Mark Fontenot, PhD  
Northeastern University

# Distributed DBs and ACID - Pessimistic Concurrency

- ACID transactions
  - Focuses on “data safety”
  - considered a pessimistic concurrency model because it assumes one transaction has to protect itself from other transactions
    - IOW, it assumes that if something can go wrong, it will.
  - Conflicts are prevented by locking resources until a transaction is complete (there are both read and write locks)
  - Write Lock Analogy → borrowing a book from a library... If you have it, no one else can.

See <https://www.freecodecamp.org/news/how-databases-guarantee-isolation> for more for a deeper dive.

# Optimistic Concurrency

- Transactions do not obtain locks on data when they read or write
- *Optimistic* because it assumes conflicts are unlikely to occur
  - Even if there is a conflict, everything will still be OK.
- But how?
  - Add last update timestamp and version number columns to every table... read them when changing. THEN, check at the end of transaction to see if any other transaction has caused them to be modified.

# Optimistic Concurrency

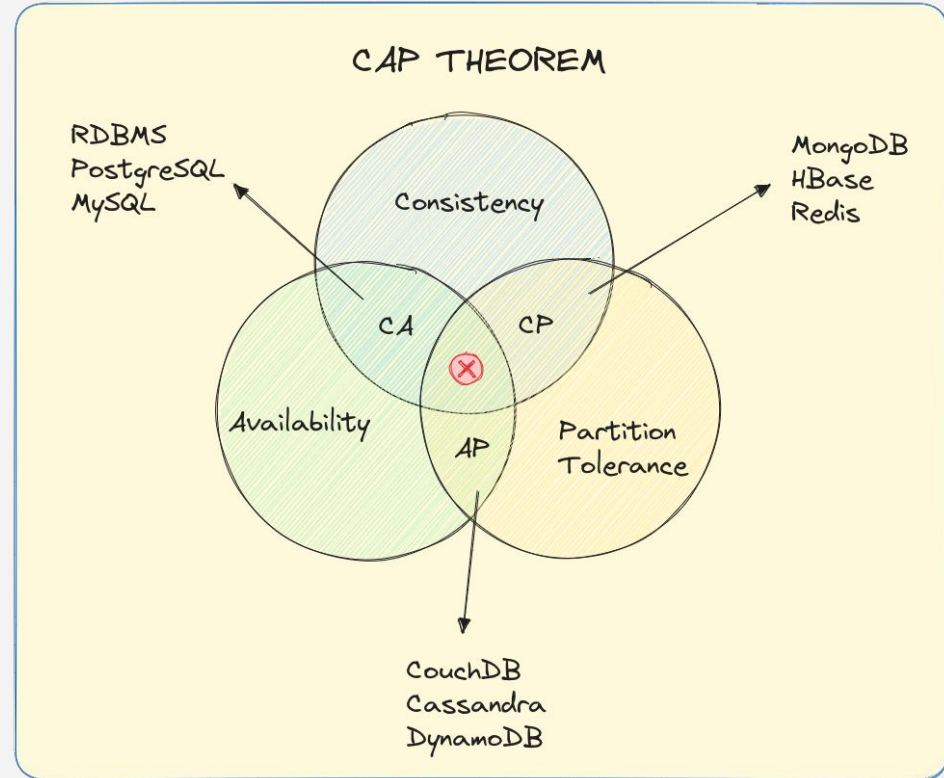
- Low Conflict Systems (backups, analytical dbs, etc.)
  - Read heavy systems
  - the conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict.
  - So, optimistic concurrency works well - allows for higher concurrency \*(read-heavy vs. write-heavy)
- High Conflict Systems
  - rolling back and rerunning transactions that encounter a conflict → less efficient
  - So, a locking scheme (pessimistic model) might be preferable

- “NoSQL” first used in 1998 by Carlo Strozzi to describe his relational database system that *did not use SQL*.
- More common, modern meaning is “Not Only SQL”
- *But*, sometimes thought of as non-relational DBs
- Idea originally developed, in part, as a response to processing unstructured web-based data.

# CAP Theorem Review

You can have 2, but not 3, of the following:

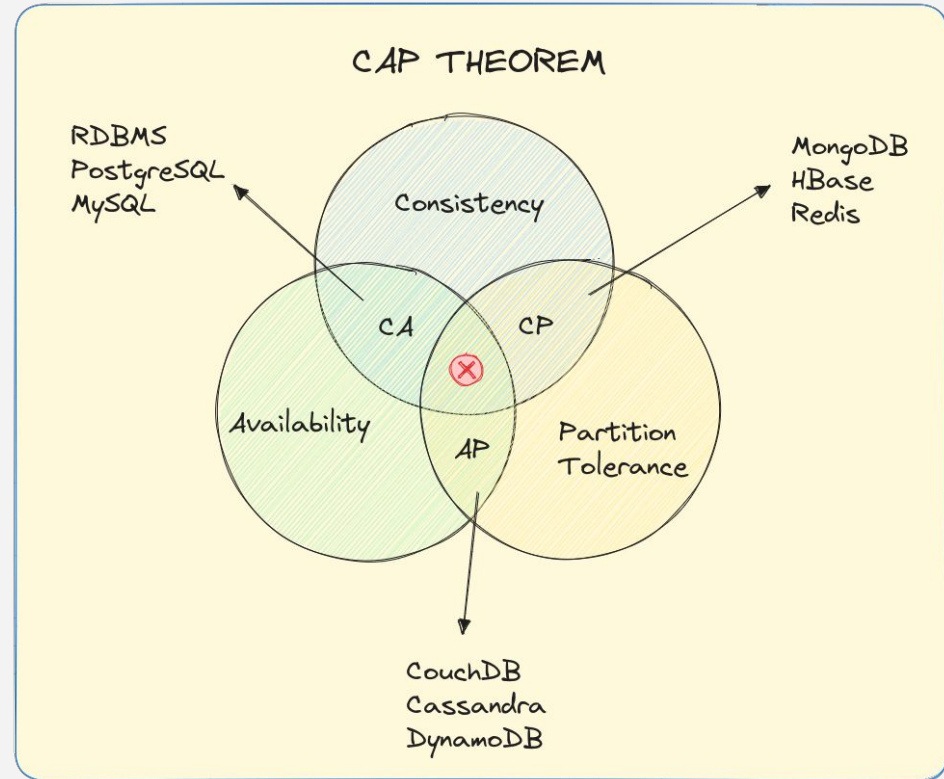
- **Consistency\***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database system remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



\* Note, the definition of Consistency in CAP is different from that of ACID.

# CAP Theorem Review

- **Consistency + Availability:** System always responds with the latest data and every request gets a response, but may not be able to deal with network partitions
- **Consistency + Partition Tolerance:** If system responds with data from the distrib. system, it is always the latest, else data request is dropped.  
\*(not always available)
- **Availability + Partition Tolerance:** System always sends are responds based on distributed store, but may not be the absolute latest data.  
\*(consistency)



# ACID Alternative for Distrib Systems - BASE

BASE stands for:

- **Basically Available**

- Guarantees the availability of the data (per CAP), but response can be “failure”/“unreliable” because the data is in an inconsistent or changing state
- System appears to work most of the time



# ACID Alternative for Distrib Systems - BASE

- **Soft State** - The state of the system could change over time, even w/o input. Changes could be result of *eventual consistency*.
  - Data stores don't have to be write-consistent
  - Replicas don't have to be mutually consistent

# ACID Alternative for Distrib Systems - BASE

- **Eventual Consistency** - The system will eventually become consistent
  - All writes will eventually stop so all nodes/replicas can be updated

# Categories of NoSQL DBs - Review

- Document databases
- Graph databases
- Key-value databases
- Columnar databases
- Vector databases

First Up → Key-Value Databases

# Key Value Stores

*key = value*

- Key-value stores are designed around:
  - *simplicity*
    - the data model is extremely simple
    - comparatively, tables in a RDBMS are very complex.
    - lends itself to simple CRUD ops and API creation

# Key Value Stores

*key = value*

- Key-value stores are designed around:
  - *speed*
    - usually deployed as in-memory DB
    - retrieving a *value* given its *key* is typically a  $O(1)$  op b/c hash tables or similar data structs used under the hood
    - no concept of complex queries or joins... they slow things down

# Key Value Stores

*key = value*

- Key-value stores are designed around:
  - *scalability*
    - Horizontal Scaling is simple - add more nodes
    - Typically concerned with *eventual consistency*, meaning in a distributed environment, the only guarantee is that all nodes will *eventually* converge on the same value.

Key-value data science use cases include (3):

### 1. EDA/Experimentation Results Store

- store intermediate results from data preprocessing and EDA
- store experiment or testing (A/B) results w/o prod db

### 2. Feature Store

- store frequently accessed feature → low-latency retrieval for model training and prediction

### 3. Model Monitoring

- store key metrics about performance of model, for example, in real-time inferencing.



Key-value software engineering use cases include (4):

## 1. Storing Session Information

- everything about the current *session* can be stored via a single PUT or POST and retrieved with a single GET .... VERY Fast

## 2. User Profiles & Preferences

- User info could be obtained with a single GET operation... language, TZ, product or UI preferences

## 3. Shopping Cart Data

- Cart data is tied to the user
- needs to be available across browsers, machines, sessions

## 4. Caching Layer:

- In front of a disk-based database

\*If you want true memory residency only, you need to use a system built for memory-only

- Redis (stands for Remote Directory Server)
  - Open source, in-memory database
  - Sometimes called a data structure store
  - Primarily a KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
- From [db-engines.com Ranking of KV Stores](https://db-engines.com/en/ranking/kv_stores):  
Redis ranks first among KV stores svstems

Rank			DBMS	Database Model	Score		
Oct 2024	Sep 2024	Oct 2023			Oct 2024	Sep 2024	Oct 2023
1.	1.	1.	Redis 	Key-value, Multi-model 	149.63	+0.20	-13.33
2.	2.	2.	Amazon DynamoDB 	Multi-model 	71.85	+1.78	-9.07
3.	3.	3.	Microsoft Azure Cosmos DB 	Multi-model 	24.50	-0.47	-9.80
4.	4.	4.	Memcached	Key-value	17.79	+0.95	-3.05
5.	5.	5.	etcd	Key-value	7.17	+0.12	-1.57
6.	 7.	 8.	Aerospike 	Multi-model 	5.57	+0.41	-0.86
7.	 6.	 6.	Hazelcast	Key-value, Multi-model 	5.57	-0.16	-2.60
8.	8.	 7.	Fhcache	Key-value	4.76	-0.03	-1.79

- It is considered an in-memory database system, but...
  - Supports durability of data by: a) essentially saving snapshots to disk at specific intervals or b) append-only file which is a journal of changes that can be used for roll-forward if there is a failure
- Originally developed in 2009 in C++
- Can be very fast ... > 100,000 SET ops / second
- Rich collection of commands
- Does NOT handle complex data. No secondary indexes.  
Only supports lookup by Key.

# Redis Data Types

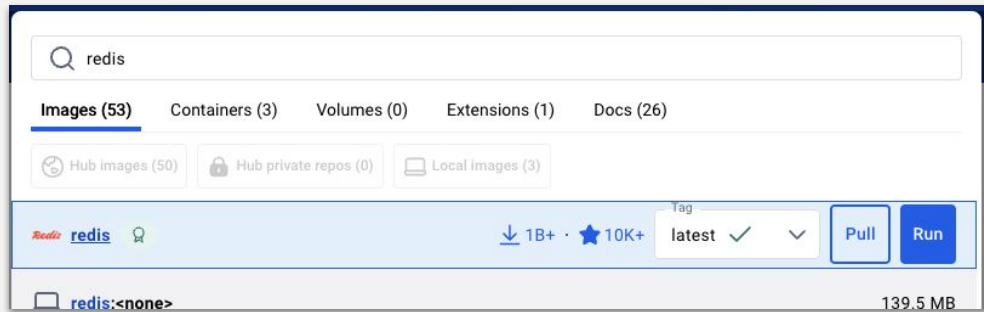
Keys:

- usually strings but can be any binary sequence \*(could be image encoded as a string)

Values:


- Strings
- Lists (linked lists)
- Sets (unique unsorted string elements)
- Sorted Sets
- Hashes (string  $\rightarrow$  string) \*(set of (k, v) pairs)
- Geospatial data

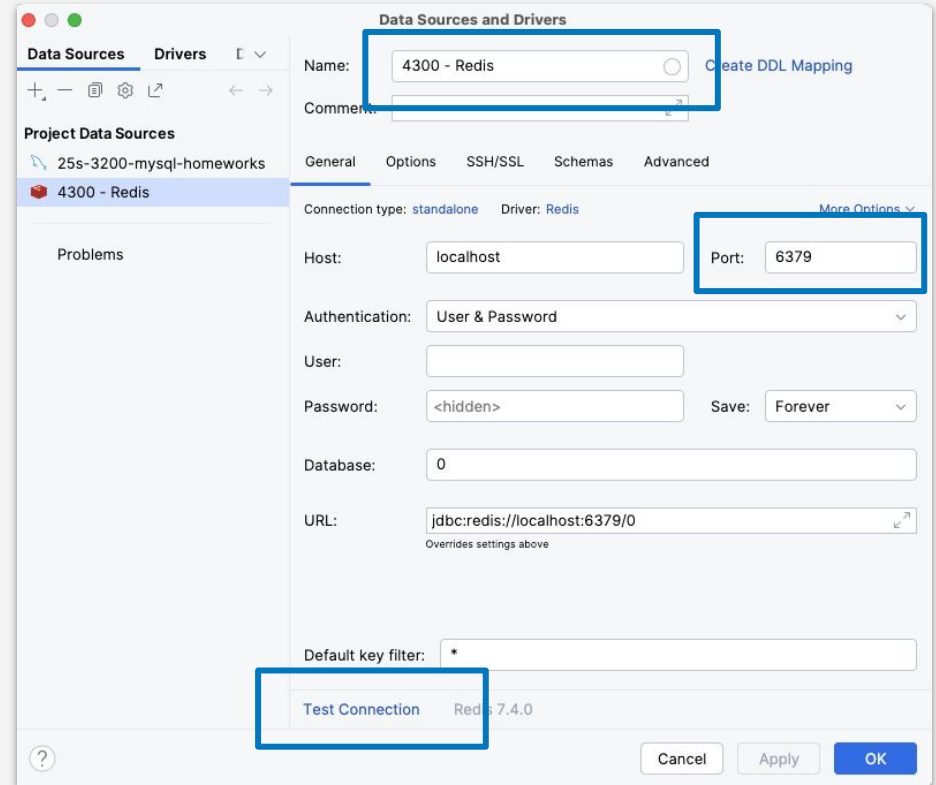
# Setting Up Redis in Docker



- In Docker Desktop, search for Redis.
- Pull/Run the latest image (see above)
  - Optional Settings: add **6379** to Ports to expose that port so we can connect to it.
- Normally, you **would not** expose the Redis port for security reasons
  - If you did this in a prod environment, major security hole.
  - Notice, we didn't set a password...

# Connecting from DataGrip

- File > New > Data Source > Redis
- Give the Data Source a Name
- Make sure the port is **6379**
- Test the connection 



# Redis Database and Interaction

- Redis provides 16 databases by default
  - They are numbered 0 to 15
  - There is no other name associated
- Direct interaction with Redis is through a set of commands related to setting and getting k/v pairs (and variations)
- Many language libraries available as well.

```
SET ds4300 "I Love AVL Trees!"
```

```
SET cs3200 "I Love SQL!"
```

```
KEYS *
```

```
GET cs3200
```

```
DEL ds4300
```

# Foundation Data Type - String

- Sequence of bytes - text, serialized objects \*(ex: Pickle files), bin arrays
- Simplest data type
- Maps a string to another string
- Use Cases:
  - caching frequently accessed HTML/CSS/JS fragments
  - config settings, user settings info, token management
  - counting web page/app screen views OR rate limiting



## Some Initial Basic Commands

- **SET** /path/to/resource 0  
**SET** user:1 "John Doe" \*(setting key, value pair)  
**GET** /path/to/resource \*(cannot directly access by value)  
**EXISTS** user:1  
**DEL** user:1  
**KEYS** user\*
- **SELECT** 5
  - select a different database

# Some Basic Commands

- **SET** someValue 0
- INCR** someValue           #increment by 1
- INCRBY** someValue 10   #increment by 10
- DECR** someValue           #decrement by 1
- DECRBY** someValue 5    #decrement by 5
  - INCR parses the value as int and increments (or adds to value)
- **SETNX** key value
  - only sets value to key if key does not already exist

# Hash Type

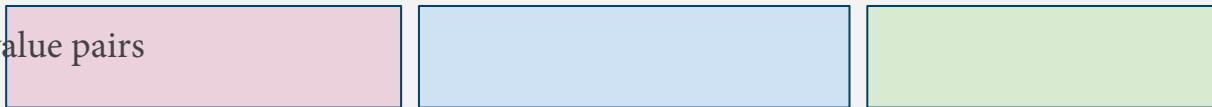
- Value of KV entry is a collection of *field-value* pairs
- Use Cases:
  - Can be used to represent basic objects/structures
    - number of field/value pairs per hash is  $2^{32}-1$
    - practical limit: available system resources (e.g. memory)
  - Session information management
  - User/Event tracking (could include TTL \*Time To Live (how long key exists))
  - Active Session Tracking (all sessions under one hash key)
    - \*ex: what users are doing on your website

# Hash Commands

```
HSET bike:1 model Demios brand Ergonom price 1971
```

\*1 represents the id of the bike (nothing to do with commands, just for book-keeping); could also sell helmets for instance or multiple bikes

\* model Demios is key, value pairs



```
HGET bike:1 model
```

```
HGET bike:1 price
```

\*can get value associated with a particular field associated with a

particular field (ex: what is the model name for bike 1)

```
HGETALL bike:1
```

```
HMGET bike:1 model price weight
```

*\* (will return "null" for weight  
(field name that is not in database)*

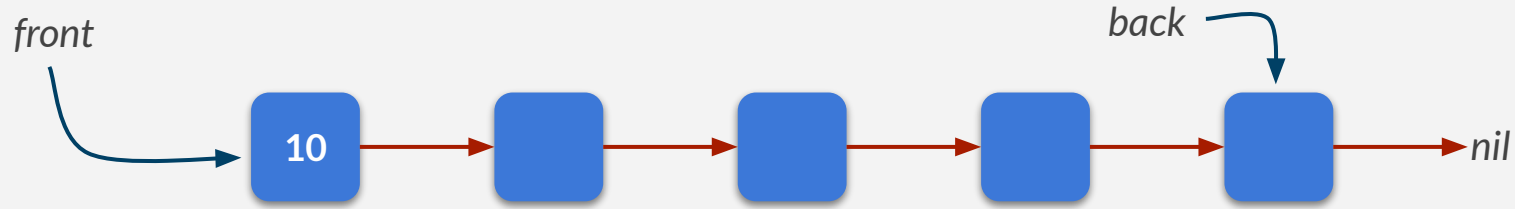
```
HINCRBY bike:1 price 100
```

What is returned?

# List Type

- Value of KV Pair is **linked lists** of string values
- Use Cases:
  - implementation of stacks and queues
  - queue management & message passing queues (producer/consumer model)
  - logging systems (easy to keep in chronological order)
  - build social media streams/feeds
  - message history in a chat application
  - batch processing by queueing up a set of tasks to be executed sequentially at a later time

# Linked Lists Crash Course



- Sequential data structure of linked nodes (instead of contiguously allocated memory)
- Each node points to the next element of the list (except the last one - points to *nil/null*)
- $O(1)$  to insert new value at front or insert new value at end

# List Commands - Queue

## Queue-like Ops

`LPUSH bikes:repairs bike:1`

\*if you push and pop from the same sides/ends, it's a stack. if you push and pop from opposite ends, it's a queue

`LPUSH bikes:repairs bike:2` \*(push to left side of queue)

`RPOP bikes:repairs` \*RPOPUSH pushes to right side

`RPOP bikes:repairs` \*RPOP removes from right side of queue

## List Commands - Stack

**Stack-like Ops** \*pushing and popping from same side so stack (ex: LPUSH, LPOP)

```
LPUSH bikes:repairs bike:1 LPUSH
```

```
bikes:repairs bike:2 LPOP
```

```
bikes:repairs
```

```
LPOP bikes:repairs
```



# List Commands - Others

## Other List Ops

LLen mylist

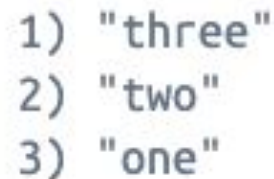
*LPUSH mylist "one"*

*LPUSH mylist "two"*

*LPUSH mylist "three"*

LRANGE <key> <start> <stop>

LRANGE mylist 0 3



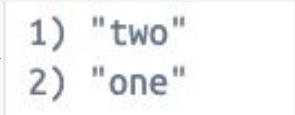
1) "three"  
2) "two"  
3) "one"

LRANGE mylist 0 0



1) "three"

LRANGE mylist -2 -1



1) "two"  
2) "one"

# JSON Type

- Full support of the JSON standard
- Uses JSONPath syntax for parsing/navigating a JSON document
- Internally, stored in binary in a tree-structure → fast access to sub elements

# Set Type

- Unordered collection of unique strings (members)
- Use Cases:
  - track unique items (IP addresses visiting a site, page, screen)
  - primitive relation (set of all students in DS4300)
  - access control lists for users and permission structures
  - social network friends lists and/or group membership
- Supports set operations!!

# Set Commands

```
SADD ds4300 "Mark"
```

```
SADD ds4300 "Sam"
```

```
SADD cs3200 "Nick"
```

```
SADD cs3200 "Sam"
```

```
SISMEMBER ds4300 "Mark"
```

```
SISMEMBER ds4300 "Nick"
```

```
SCARD ds4300
```

# Set Commands

```
SADD ds4300 "Mark" SADD ds4300 "Sam"  
SADD cs3200 "Nick" SADD cs3200 "Sam"
```

\*SADD adds (k, v) pair to set

```
SCARD ds4300
```

\*SCARD takes set cardinality

```
SINTER ds4300 cs3200
```

\*SINTER finds intersection of two sets

```
SDIFF ds4300 cs3200
```

\*SDIFF finds difference between two sets

```
SREM ds4300 "Mark" *SREM removes from a set
```

```
SRANDMEMBER ds4300 *SRANDMEMBER gets  
random member from set
```

??