NoSQL & KV BDs

Created	@February 3, 2025 9:24 AM
Class	DS 4300
Reviewed	☐

ACID transactions:

- Focuses on data safety
- considered a “pessimistic concurrency model” because it assumes one transaction has to protect itself from another
- Conflicts prevented by locking resources until transaction is complete
- Write lock analogy → borrowing a book from a library… If you have it, no one else can

Optimistic Concurrency

- Transactions do not obtain locks on data when they read or write
- Optimistic because it assumes conflicts are unlikely to occur
- 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
- Low conflict systems (backup, 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
- High Conflict System
    - rolling back and rerunning transactions that encounter a conflict → less efficient
    - So a locking scheme might be preferable

NoSQL

- Created by Carlo Strozzi in 1998
- More common, modern meaning is “Not only SQL”
- But, sometimes thought of as non-relational DBs

ACID Alternative for Distributed Systems - BASE

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

Categories of NoSQL DBs

Key Value DBs:

key = value

- Key-value stores are designed around:
  - simplicity
    - the data model is extremely simple
    - comparatively, tables in RDBMS are very complex
    - lends itself to simple CRUD ops and API creation
  - speed
    - usually deployed as in-memory DB
    - retrieving a value given its key is typically O(1)
    - no concept of complex queries or joins... they slow things down
  - scalability
    - horizontal scaling is simple, add more nodes
    - typically concerned with eventual consistency, meaning in a distr. env, the only guarantee is that all nodes will eventually converge on the same value
- Use Cases - DS
  - EDA/Experimentation Results Store
    - store intermediate results from data preprocessing and EDA
    - store experiment or testing(A/B) results w/o prod db
  - Feature Store
    - store frequently accessed feature → low-latency retrieval for model training and prediction
  - Model Monitoring
    - store key metrics about performance of model, for example, in real-time inferencing
- Use Cases - SWE

- Storing session information
  - everything about the current session can be stored via a single PUT or POST and retrieved with a single GET
- User Profiles & Preferences
  - User info could be obtained with a single GET operation...
- Shopping Cart Data
  - Cart data is tied to the user
  - needs to be available across browsers, machines, sessions
- Caching Layer
  - in front of a disk-based database

Redis DB

- Remote Directory Server
  - Open source, in memory database
  - Sometimes called a data structure store
  - Primarily KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
- 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
- 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

    - Values:
        - strings
        - lists (linked lists)
        - sets (unique unsorted string elements)
        - sorted sets
        - hashes (string → string)
        - geospatial data
- 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
- Max Key size 512 mb
- Foundation Data Type - String
    - Sequence of bytes - text, serialized objects, 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
- Basic Commands

```
SET /path/to/resource 0
SET user:1 "John Doe"
GET /path/to/resource
```

```
EXISTS user:1
DEL user:1
KEYS user*
SELECT 5
   - select a different database

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)
    - Active Session Tracking (all sessions under one hash key)
- Hash Commands

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

```
HMGET bike:1 model price weight #returns null for weight
HINCRBY bike:1 price 100
```

- 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 stream/feeds
    - message history in a chat application
    - batch processing by queuing up a set of tasks to be executed sequentially ay a later time

  ▼ Linked List 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

▼ Queue-like Ops

```
LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
RPOP bikes:repairs
RPOP biles:repairs
```

▼ Stack-like Ops

```
LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
LPOP bikes:repairs
LPOP biles:repairs
```

▼ Other List Ops

```
LPUSH myLIST "one"
LPUSH myLIST "two"
LPUSH myLIST "three"

LLEN myList

LRANGE <key> <start> <stop>

LRANGE myList 0 3
LRANGE myList 0 0
LRANGE myList -2 -1
```

- 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 “Same”

SADD ds4300 “Nick”

SISMEMBER ds4300 “Mark”

SISMEMBER ds4300 “Nick”

SCARD ds4300 #cardinality
SINTER ds4300 cs3200 #intersection
SDIFF ds4300 cs3200 #set difference
SREM ds4300 "Mark" #removes value from set
SRANDMEMBER ds4300 #gets random member of set
```

Redis + Python

- Redis-py is the standard client for Python
- Maintained by the Redis Company itself
- In your 4300 Conda Environment
- Connecting to Server

```
import redis
redis_client = redis.Redis(host='localhost',
                                        port=6379,
```

```
                                    db=2,
                                    decode_responses=True)
```

- For Docker deployment, host could be localhost or 127.0.0.1
- Port is the port mapping given when you created the container (default 6379)
- db is the database 0-15
- decode_responses → data comes back form the server as bytes. Setting this true converts them to strings
- Redis Commands https://redis.io/docs/latest/commands/

▼ String Commands

```
# r represents the Redis client object
r.set('clickCount:/abc', 0)
val = r.get('clickCount:/abc')
r.incr('clickCount:/abc')
ret_val = r.get('clickCount:/abc')

redis_client.mset({'key1': 'val1',
                                'key2': 'val2',
                                ...})
redis_client.mget('key1', 'key2', ...) # Returns a list

# set(), mset(), setex(), msetnx(), setnx()
# get(), mget(), getex(), getdel()
# incr(), decr(), incrby(), decrby()
# strlen(), append()
```

▼ List Commands

```
# create list: key = 'names'
# values = ['mark', 'sam', 'nick']
redis_client.rpush('names', 'mark', 'sam', 'nick')
```

```
# print mark, sam, nick
# lpush(), lpop(), lset(), lrem()
# rpush(), rpop()
# lrange(), llen(), lpos()
```

▼ Hash Commands

```
redis_client.hset('user-session:123',
        mapping={'first': 'Sam',
                         'last': 'Uelle',
                         ...})
# hset(), hget(), hgetall()
# hkeys()
#hdel(), hexists(), hlen(), hstrlen()
```

- Redis pipelines
  - Helps avoid multiple related calls to the server → less network overhead
  - Not same as transaction
- SQLite → local RDB for embedded systems
  - Means that data is stored in memory

Dogs name is Winston