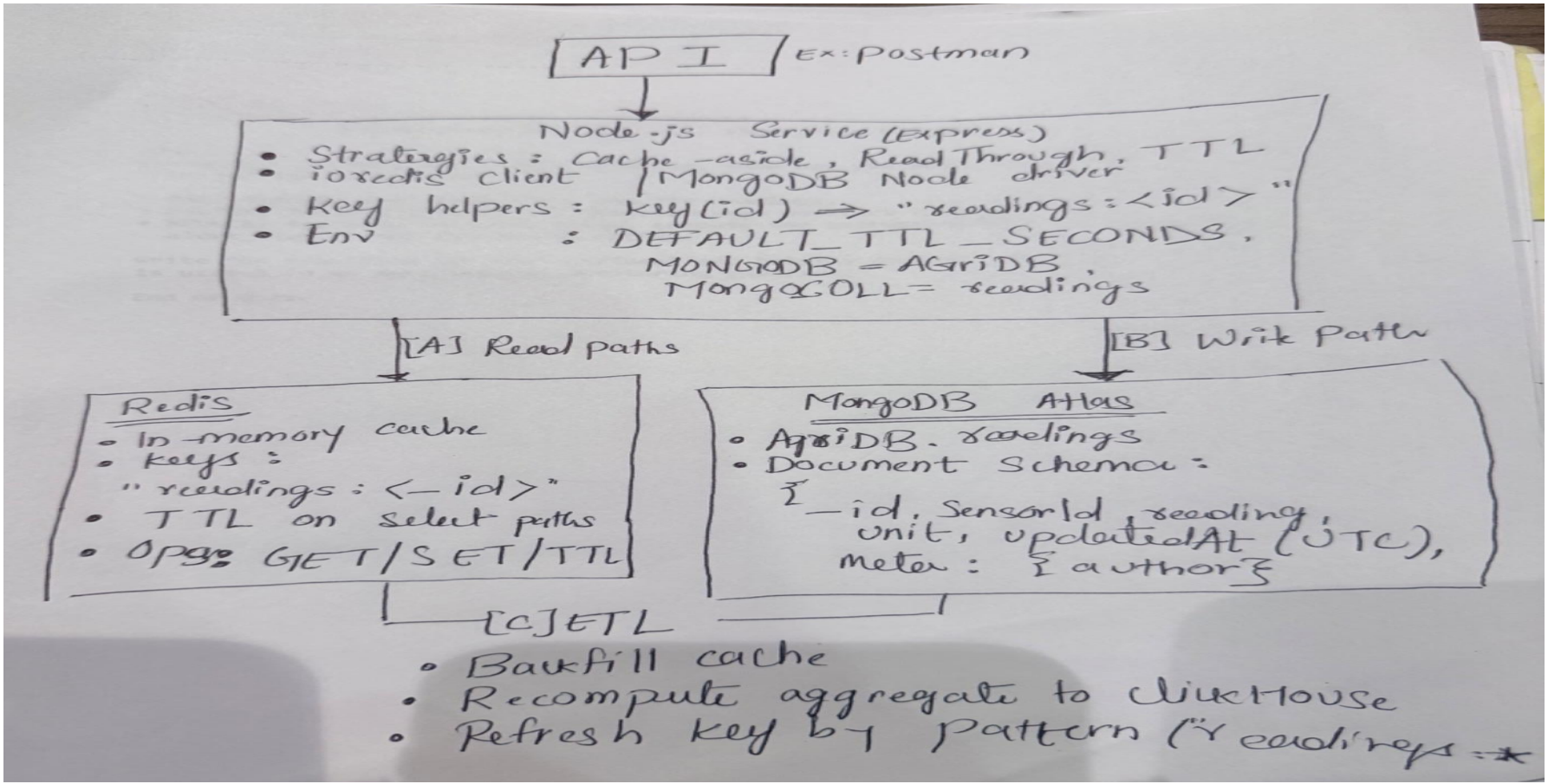
Lab4: Question 5

Farheen Shaikh-989500432

Architecture & ETL Data Flow (*Redis + MongoDB Cache Pipeline)*



Lessons Learned

1. Redis dramatically accelerates repeated reads

Our benchmarking confirmed a dramatic latency difference between cold (database-only) and warm (cached) reads. In the **Cache-Aside cold path**, requests averaged **96.4 ms** when data had to be fetched directly from MongoDB. However, once cached, the same request returned in just **0.6 ms**, making Redis nearly **160x faster** for repeated reads. Under **TTL-based caching**, this pattern remained consistent: requests made before expiration were fast (**~5 ms**), while those made after TTL expiry reverted to slower database reads (**~60 ms**) until recached. These tests highlight that caching not only improves performance but also keeps response times predictable in read-heavy workflows.

2. Different caching strategies serve different use cases

- **Cache-Aside** is simple to implement and works well when the application logic can tolerate occasional cache misses.
- **Read-Through** centralizes cache population logic and simplifies read paths.
- **Write-Through** and **Write-Behind** are more suited for maintaining cache–DB consistency in write-heavy scenarios.
  This lab emphasized that **no single pattern fits all cases**—it depends on access patterns, latency tolerance, and consistency needs.

3. Different caching strategies serve different use cases

- **Cache-Aside** is simple to implement and works well when the application logic can tolerate occasional cache misses.
- **Read-Through** centralizes cache population logic and simplifies read paths.
- **Write-Through** and **Write-Behind** are more suited for maintaining cache–DB consistency in write-heavy scenarios.
  This lab emphasized that **no single pattern fits all cases**—it depends on access patterns, latency tolerance, and consistency needs.

4. Manual invalidation is still necessary in some patterns

Both **Cache-Aside** and **Read-Through** require **manual intervention** to clear or refresh cached items when underlying data changes in MongoDB. This can lead to stale data issues if not planned for. It reinforced the importance of having a **cache invalidation strategy**, especially in distributed systems.

Challenges

What worked well:

- Redis integration with Node (ioredis) and MongoDB Atlas connection was straightforward.
- Cache-aside and TTL strategies were easy to observe with performance benchmarking.

What didn’t work well:

- Handling ObjectId vs custom sensorId required code changes for fallback logic.
- Port conflicts, TTL resets, and MongoDB URI errors were common during setup.
- Measuring true “DB only” behavior required explicit cache deletes each iteration.