**Output:**

Requirement Analysis (Step-by-step breakdown)

1. \*\*Social platform for millions of users\*\*

→ \*Implications:\* Needs horizontal scaling (sharding), efficient distribution of data across nodes, and fault tolerance. Vertical scaling alone won't suffice.

2. \*\*Data types: user profiles, posts, user connections\*\*

→ \*Implications:\*

- Profiles/posts: Semi-structured data (JSON-like) suits document storage

- Connections: Graph-like relationships requiring efficient traversal (e.g., "friends-of-friends")

→ Hybrid data nature demands flexible schema.

3. \*\*High read speed essential\*\*

→ \*Implications:\* Requires indexing optimization, read replicas, in-memory caching, and low-latency query patterns. Prioritize read-optimized architectures.

4. \*\*Read:Write ratio = 80%:20%\*\*

→ \*Implications:\* Read-heavy workload favors:

- Denormalization for fewer joins

- Asynchronous replication

- Caching layers (e.g., Redis)

- Write operations must not block reads.

5. \*\*Must scale efficiently with user growth\*\*

→ \*Implications:\* Need linear scalability via:

- Automatic sharding/partitioning

- Distributed architecture

- No single-point bottlenecks

- Cloud-native elasticity.

### Database Comparison

| Feature | SQL (e.g., PostgreSQL) | Document DB (e.g., MongoDB) | Graph DB (e.g., Neo4j) | Column DB (e.g., Cassandra) |
| --- | --- | --- | --- | --- |
| Data Model | Rigid schema (tables) | Flexible JSON docs | Nodes/relationships | Wide-column families |
| Relationships | JOINs (slow at scale) | Manual refs/denormalization | Native O(1) traversal\* | Limited |
| Read Speed | Good (with indexes) | Excellent (denormalized docs) | Best for deep queries | Excellent (columnar) |
| Write Speed | ACID-safe but slower | Fast (eventual consistency) | Moderate | Blazing writes |
| Scalability | Vertical + complex sharding | Auto-sharding | Limited horizontal scale | Linear scale-out |
| 80% Read / 20% Write | Suboptimal for heavy reads | deal\*\* (read-optimized) | Good for relationship reads| | Excellent for reads |
| Connection Handling | Complex recursive JOINs | App-layer handling | Native graph queries | Manual indexing needed |

### Decision & Justification

\*\*Selected Database Type: Document Database (e.g., MongoDB)\*\*

\*\*Secondary Recommendation: Graph Database (e.g., Neo4j) for connections\*\*

| Requirement | How Document DB Satisfies | Support Level |
| --- | --- | --- |
| Millions of users | Auto-sharding distributes users across clusters; horizontal scaling handles growth |  |
| Profiles/posts storage | JSON-like documents natively store semi-structured data (e.g., profiles with dynamic fields) |  |
| User connections | Denormalized embedding (e.g., friend lists in profiles) + secondary graph DB for deep traversals |  |
| High read speed | Denormalized documents serve reads in single queries; read replicas; optional Redis caching |  |
| 80% reads / 20% writes | Write operations don’t block reads; eventual consistency optimized for read-heavy workloads |  |
| Scalability with growth | Linear scale-out via sharding; cloud-native (e.g., MongoDB Atlas) |  |

### Why Not Others?

- \*\*SQL\*\*: JOINs degrade with deep relationships; sharding complexity hurts scalability.

- \*\*Graph DB\*\*: Struggles with billion-scale document storage; not optimal for profile/post reads.

- \*\*Column DB\*\*: Overkill for document storage; poor graph traversal without extra indexing.

### Recommended Hybrid Architecture

1. \*\*Primary: Document Database (MongoDB)\*\*

- Stores profiles/posts as JSON documents

- Handle 80% read traffic via sharding + read replicas

2. \*\*Secondary: Graph Database (Neo4j)\*\*

- Manages user connections with `O(1)` traversal for "friends-of-friends"

- Sync connection data via change streams

3. \*\*Cache Layer\*\*: Redis for hot data (e.g., trending posts)

\*\*Justification\*\*: This leverages MongoDB's scalability for core data while using Neo4j's relationship superpower where critical – satisfying all requirements with optimal performance.