**Ch 3**

**Storage and Retrieval**

**1. Core Ideas**

* A database must **store data (writes)** and **retrieve data (reads)** efficiently.
* Different workloads require different optimizations:
  + **OLTP (transactions):** many small reads/writes → optimized for disk seeks.
  + **OLAP (analytics):** fewer but very heavy queries → optimized for sequential scans & throughput .
* Storage engines underpin performance, reliability, and scalability.

**2. Data Structures That Power Storage Engines**

**Hash Indexes**

* Keep a **log of key-value pairs** (append-only).
* Use in-memory hash map → maps keys to file offsets.
* Fast lookups if all keys fit in memory .
* Weakness: doesn’t handle range queries well.

**SSTables & LSM-Trees**

* **SSTable (Sorted String Table):** key-value pairs sorted by key, immutable segments .
* Compaction merges old + new → keeps only latest values.
* **LSM-Tree:**
  + Writes: go to in-memory tree (memtable).
  + Periodically flushed to SSTables on disk.
  + Background merge + compaction keeps data efficient .
* Used in: LevelDB, RocksDB, Cassandra, HBase, Lucene .
* Pros: great write throughput, compression-friendly.
* Cons: high read amplification, compaction overhead.

**B-Trees**

* Widely used in relational DBs.
* Page-oriented (4KB blocks), updated in place .
* Strong for **point lookups & range queries**.
* Weakness: random writes = expensive, fragmentation.

**Other Structures**

* **Bloom filters**: avoid unnecessary disk lookups.
* **Column stores**: OLAP → store by column, compress & scan efficiently .
* **Materialized views/cubes**: precomputed aggregates for speed .
* **In-memory DBs**: RAM + periodic logs (VoltDB, Redis, MemSQL) .

**3. Trade-offs**

| **Storage Engine** | **Strengths** | **Weaknesses** | **Use Case** |
| --- | --- | --- | --- |
| Hash Index | Simple, fast KV lookup | No range queries, memory heavy | Caches, logs |
| LSM-Tree | High write throughput, sequential IO | Read amplification, compaction cost | Vector DB (Weaviate, Pinecone), indexing |
| B-Tree | Balanced reads/writes, strong for ranges | Random write overhead, page splits | Relational DB, DynamoDB |
| Column Store | Great for scans, compression | Bad for OLTP | Data warehouses (Redshift, BigQuery) |

**4. Mapping to GenAI Pyramid / LexiFlow**

| **DDIA Concept** | **Pyramid Layer** | **LexiFlow Application** |
| --- | --- | --- |
| Hash Index | Infra (fast cache) | Embedding cache in Redis |
| LSM-Trees (SSTable) | RAG layer | Weaviate / FAISS ANN indexing for chunks |
| B-Trees | Infra (metadata) | DynamoDB for doc metadata / routing |
| Column store | Eval layer | Aggregated retrieval logs for analytics |
| Compaction | Orchestration | Background doc reindexing in LexiFlow |

**5. LexiFlow Example**

* **Chunk storage**: S3 holds raw chunks (append-only log).
* **Vector DB**: Weaviate uses LSM-based index (ANN search).
* **Metadata**: DynamoDB uses B-tree like structures for doc\_id lookups.
* **Query optimization**: Hybrid retrieval = ANN (LSM) + BM25 (inverted index).
* **Compaction**: periodic reindexing to merge duplicates / stale embeddings.

**6. Interview-Style Q&A**

* **Q:** Compare B-Trees vs LSM-Trees.  
  **A:** B-Trees = faster reads, LSM = faster writes. LSM used in high-ingest vector DBs, B-Trees in relational DBs.
* **Q:** How would you store embeddings for RAG?  
  **A:** Use LSM-based vector DB (Weaviate, Pinecone) for scalability; add Redis cache for hot embeddings.
* **Q:** How to handle compaction overhead in LexiFlow?  
  **A:** Async compaction, backpressure with SQS, staggered shard reindexing.

**7. Key Takeaways**

* **Logs + compaction = heart of modern DBs.**
* **LSM-Trees dominate write-heavy workloads** (vector DBs, search).
* **B-Trees dominate traditional relational workloads.**
* **Column stores power analytics.**
* For GenAI pipelines → you almost always need a **hybrid**:
  + LSM for vector retrieval.
  + B-tree for metadata.
  + Column store for analytics/eval.