

Fundamentals of Database Engineering

Transaction

- Collection of queries
- Single unit of work

Atomicity

- All queries in a transaction should execute successfully
- Lack of atomicity leads to inconsistencies
- Cleanups / rollbacks take time

Isolation - Read Phenomena

- Dirty read - another transaction wrote but did not commit yet
- Non-repeatable reads
 - read the same attribute twice; can get different value
 - the value was not repeatable in the same transaction
- Phantom reads
 - don't exist yet
 - first select query - row absent
 - second select query - new row present
 - **MYSQL, ORACLE CAN'T GET RID OF THEM (LOCKS yes)**
- Lost updates
 - other transaction can change what you wrote
 - if you read after write, and before you read, the other transaction changes what you wrote

Note: Postgres versions its updates while MySQL and Oracle keep an Undo stack log where all previous values are kept.

Isolation Levels

- Read uncommitted - no isolation, any outside change is visible to transaction; committed or not
- Read committed - each query in a transaction only sees committed changes by other transactions
- Repeatable read - when a query reads a row, it will remain unchanged throughout the transaction for all subsequent reads
- Snapshot - each query in a transaction only sees changes committed up to start of transaction - like a snapshot of database at that moment

(postgres repeatable read = snapshot => no phantom reads)

- Serializable - transactions run as if they serialized one after the other

Can be adjusted when you begin transaction

Database Implementation of Isolation

- Pessimistic - row level locks, table locks, page locks to avoid lost updates; lock mgmt expensive
- Optimistic - no locks, track if things changed and fail transaction if so
- Repeatable read locks the rows it reads; expensive if many rows read
- Serializable - usually optimistic; can also be pessimistic (SELECT but LOCK it)
 - Failures possible - reader/writer dependencies

Consistency

- Data consistency - defined by DBA
- Read consistency - affects system as a whole - most systems heal by eventual consistency
- Atomicity ensures consistency
- Isolation can cause inconsistency

Durability

- Persist when transaction commits
- Techniques
 - WAL - write ahead log; persists but quite expensive
 - compressed version: WAL segments
 - Asynchronous snapshot
 - Redis uses both
 - AOF
- OS Cache
 - Write request -> OS cache -> OS crashes -> m/c restart => data loss
 - Fsync OS forces writes to disk - expensive, slows down commits

Notes

- Primary key is usually a clustered index (except Postgres)
- Sometimes heap table organized around a single index - clustered index / Index Organized Table

- MySQL, InnoDB: other indexes point to the primary key
- Postgres has only secondary indexes and all indexes point directly to the row_id which lives in heap
 - Update one key => all indexes get updated
- Index data structure stored as a B-tree
- RDBMS backed by B-tree optimized by tree; updates are expensive
- SELECT * on column database is worst - avoid
- VACUUM - garbage collect a database

LSM-Tree

- writes in batch as they arrive to Memtable in memory
- Memtable ordered by object key, usually a balanced binary tree
- As Memtable reaches certain size, flushed to disk as immutable sorted string table (SSTable)
- These writes are all sequential I/O - fast on any storage media
- Update to object table => no change in previous SSTable, but appended as new entry in latest SSTable, superseding all previous SSTables
- Deletes take extra space: add tombstone marker to most recent SST for object key
- High disk space consumption by tombstones and old entries
- Merging and compaction of SSTables similar to merge sort to remove invalidated entries
- Compaction Strategies
 - Size Tiered Compaction
 - optimized for write throughput
 - Leveled Compaction
 - optimized for read throughput
- Most systems maintain a summary table containing min/max range of each disk block of every level
- Bloom filter used to reduce random I/O - gives definite no / probable yes

Scan Types

- Sequential Table Scan
 - Full table scan on table instead of index
- Index Scan
 - Find indexes and corresponding IDs of rows, jump to those rows and fetch
- Bitmap Index Scan (Postgres)
 - Condition for primary index given id < 100 say
 - Create bitmap; set bit for pages that satisfy this condition
 - If multiple conditions, create multiple bitmaps and AND or OR
 - Now using final bitmap, fetch from heap (bitmap heap scan) and perform recheck again since those pages may have rows that do not satisfy the conditions

Indexes

- Creation takes time, blocks inserts/changes
- CREATE INDEX CONCURRENTLY is async, preempts on insert/update. Takes longer

Column Indexes

- Key
- Non-Key - index for search but non-index for return (e.g. id for search but always returned with name)
 - CREATE INDEX index_name ON table_name (key) INCLUDE (non-key)

No hit to table => index only scan

Composite Index

- For index created on a, b: since 'a' is on the left, queries with just 'a' will use index in Postgres
- Queries with just 'b' will not use index. Table Read
- In case of logical OR, indexes used only if both attributes have effectively individual indexes

M-Way Search Trees

- Extension of Binary Search Tree (2-Way Search Tree)
- Each node has M - 1 keys, and each node has at most M children (<, >)
- B-Tree is special case of M-Way search tree
- Limitation: no rules for creation; it is random

B-Trees

- M-Way Search Trees with some rules
- If degree is M, every node must have ceil(M / 2) children. Then only create a new node
- Root can have minimum 2 children
- All leaf nodes at same level

- Bottom up creation process
- Each key has pointer to record and pointer to child node

B+ Trees

- Exactly like B-tree but only keys in leaf nodes have pointers to records
- Every key in tree present in union of leaf nodes
- Duplicates may be present in non-leaf nodes
- Secondary index values either point directly to tuple (Postgres) or to primary key (MySQL)

B-Tree Limitations

- Elements in all nodes store both key and value
- Key is used to filter, if key matches, then value considered. This is not space efficient then; slows down traversal too
- Range queries are slow because of random access
- B+ Tree solves both limitations

Partitioning

- Types
 - Horizontal = split big table into multiple tables in same DB
 - Vertical
- By range: CREATE TABLE table_name (attributes) PARTITION BY RANGE(attribute);
 - Now create ranges yourself
 - CREATE TABLE attribute0035 (LIKE table_name INCLUDING INDEXES);
 - CREATE TABLE attribute3560 (LIKE table_name INCLUDING INDEXES);
 - CREATE TABLE attribute6080 (LIKE table_name INCLUDING INDEXES);
 - CREATE TABLE attribute80100 (LIKE table_name INCLUDING INDEXES);
 - ALTER TABLE table_name ATTACH PARTITION attribute0035 FOR VALUES FROM (0) to (35);
 - ALTER TABLE table_name ATTACH PARTITION attribute3560 FOR VALUES FROM (35) to (60);
 - ALTER TABLE table_name ATTACH PARTITION attribute6080 FOR VALUES FROM (60) to (80);
 - ALTER TABLE table_name ATTACH PARTITION attribute80100 FOR VALUES FROM (80) to (100);
- On INSERT, database decides which partition to go to
- CREATE INDEX index_name ON table_name(attribute);
 - Attribute on which you create partition = attribute to index; very powerful
- SET enable_partition_pruning = off
 - Hits all partitions despite partitioning; renders partitioning useless
- SET enable_partition_pruning = on
 - Hits only required partition
- Automate Partitioning - for loop + DB API in any language

Disadvantages of Partitioning

- Updates that move rows from a partition to another - slow / fail
- Inefficient query could accidentally scan all partitions
- Schema changes can be challenging

Sharding

- Sharding = Partition table and send partitions to different database servers / instances
- Consistent Sharding = every time you have an input, you know which database to hit
- Essentially, partitioning = multiple tables; sharding = multiple databases
- NPM: 'hashring' module does consistent hashing

Advantages of Sharding

- Data, Memory Scalability
- Security - users can access certain shards
- Optimal and Smaller index size

Disadvantages of Sharding

- Complex client
- Transactions across shards
- Rollbacks
- Hard schema changes
- JOINS
- No ACID, ROLLBACK, Transaction, COMMIT, etc.
- Vitess

Concurrency Control

- Exclusive Lock
 - When reading a value, nobody is attempting to read this value
 - No reads allowed when this lock is acquired
 - Someone has exclusive privileges on the value
- Shared (Read) Lock
 - When reading a value, acquire a shared lock and make sure nobody changes it
- Two Phased Locking
 - Phase 1: Acquire all locks
 - Phase 2: Release all locks
 - e.g.: SELECT * FROM table_name WHERE id=13 FOR UPDATE;
 - // FOR UPDATE enters phase 1; row-level lock
 - COMMIT; enters phase 2

Pagination - Use WHERE < or WHERE > with LIMIT instead of OFFSET; OFFSET is very slow

Replication

- Make two Postgres instances and map Postgres data to on disk data
- In backup data folder, edit postgresql.conf
 - primary_conninfo = 'application_name=standby1 host=saarth port=5432 user=postgres password=postgres'
- In standby data folder, touch standby.signal to make it read-only
- Now, in master server postgresql.conf
 - synchronous_standby_names = 'first 1 (standby1,standby2)'
 - Meaning: allow the commit when at least 1 server is written to

Database Engines

- library that takes care of on-disk storage and CRUD; DBMSes use DB engine and build features on top (aka storage engine / embedded database)

- MyISAM
 - indexes sequential access method
 - B-tree indexes point to rows directly
 - No transaction support
 - Updates, deletes slow because indexes stored by offsets
 - row numbers change on update/delete
 - Table level locks
 - Supported by MySQL, MariaDB, Percona
- InnoDB
 - B+trees - indexes point to PK and PK points to row
 - ACID compliant transactions support
 - Foreign keys
 - Row level locking
 - Tablespace - arranging storage
- XtraDB
 - Fork of InnoDB
 - System tables in MariaDB are all Aria
- SQLite
 - B-tree (LSM as extension)
 - Full ACID and table locking
 - Concurrent R/W
 - Web SQL in browsers use it
- Aria
 - Similar to MyISAM
 - Crash-safe unlike MyISAM
 - Designed specifically for MariaDB
- BerkeleyDB
 - Key-value
 - Supports ACID transactions, locks, replication, etc
 - Used to be in Bitcoin core, now switched to LevelDB
 - Used Memcached
- LevelDB
 - no transactions
 - Uses LSMs - great for high insert
 - Supports WAL
 - Levels
 - Memtable
 - Level 0 (young level)
 - Level 1-6
 - As files grow larger, levels are merged
- RocksDB
 - Transactional
 - Multithreaded compaction
 - MyRocks as a DB engine for MySQL, MariaDB, Percona
 - MongoRocks as a DB engine for MongoDB

Note: SSDs and B-trees are a bad combination - SSDs hate changing the bit on place

Security

- SSL insecure and outdated
- TLS is an improved version of SSL and used in HTTPS
- TLS = transport layer security
- HTTP (80) is stateless while TCP is stateful (has information from both client and server)
 - TCP needed just to transfer stuff
- HTTPS (443)
 - handshake: both client and server agree on symmetric key for encryption and decryption
 - client takes key, locks (encrypts) the GET request and transported by TCP
 - server decrypts and processes request
 - HTTP/1 vs HTTP/2 - persistence
- TLS 1.2 (RSA)
 - Client sends request to server with information on supported encryption algorithms
 - Server returns certificate with public key of server
 - Client uses server key to encrypt its private key and sends it to the server
 - Server says OK, let's begin
- TLS 1.3 (ephemeral Diffie Hellman)
 - Client hello: sends (P)
 - own public key (P) and
 - own private (B) + public combined key that is public (merging is unbreakable)
 - Server generates private key (R) and adds the combined key received from client to it (G)
 - Server hello: sends R + P merged
 - Client extracts R, combines with preexisting (B + P) to make G
 - Two-way handshake

Certificates

- Server public key is signed by certificate authority
- Signature is encrypted by another certificate public key which is verified by certificate private key
- Goes on until root that is installed on client machine
- Root is self-signed
- In case of error, untrusted-root

Encryption

- Symmetric - same key encrypts, decrypts
- Asymmetric
- Public Key Encryption (PKE)
 - server has two keys - private, public
 - one encrypts, other decrypts
- Why can't we always encrypt
 - DB queries can only be performed on plain text
 - Analysis, indexing
 - TLS termination L7 reverse proxies and load balancing