DAM / Concept

DataBase

DataBase Types

There are several types of databases, each designed to handle different data storage and retrieval needs efficiently. Here are some common types of databases:

Relational Databases (RDBMS)

- Examples: MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server
- **Description:** Organizes data into structured tables with rows and columns. Uses SQL for querying and managing data. Suitable for complex queries and transactions.

NoSQL Databases

- Examples: MongoDB, Cassandra, Redis
- **Description:** Stores and retrieves data in formats other than traditional tables, often using JSON-like documents, key-value pairs, or wide-column stores. Designed for scalability, flexibility, and handling large volumes of unstructured or semi-structured data.

Key-Value Stores

- Examples: Redis, DynamoDB
- **Description:** Stores data as key-value pairs, where each key is unique and associated with a value. Enables fast retrieval of data based on keys, suitable for caching and real-time applications.

Document Databases

- Examples: MongoDB, Couchbase
- **Description:** Stores semi-structured data as documents (e.g., JSON, XML), allowing nested structures and flexible schema. Ideal for content management, catalogs, and real-time applications.

Columnar Databases

- Examples: Cassandra, HBase
- **Description:** Organizes data by columns rather than rows, enabling efficient data retrieval and analytics on large datasets. Suited for analytical and operational workloads requiring high availability and scalability.

Graph Databases

- Examples: Neo4j, Amazon Neptune
- **Description:** Stores data in nodes, edges, and properties to represent and navigate relationships between data entities. Used for complex relationship-centric queries, such as social networks, fraud detection, and recommendation engines.

Time-Series Databases

- Examples: InfluxDB, Prometheus
- **Description:** Optimized for storing and retrieving time-stamped data points or measurements over time. Ideal for IoT, monitoring, and analytics applications that require handling large volumes of time-series data efficiently.

In-Memory Databases

- Examples: Redis (with in-memory option), Memcached
- **Description:** Stores data primarily in RAM for faster data access and retrieval compared to disk-based storage. Used for caching, session management, and real-time data processing.

Data redundancy

Data redundancy refers to the situation where the same piece of data is stored in multiple locations within a database system.

This can happen intentionally for specific purposes, or unintentionally due to inefficiencies in data management.

1. Types of Data Redundancy

Duplicated Data

This is the most straightforward case, where identical data exists in multiple tables or columns without a clear reason.

Partially Duplicated Data

Here, a core piece of data might be the same across multiple locations, but additional details might differ.

2. Causes of Data Redundancy

Normalization Issues

Databases that are not properly normalized (organized) can lead to data redundancy.

Application Silos

Separate applications within an organization might store redundant copies of data specific to their needs.

Manual Processes

Manually entering the same data into different systems can create redundancy.

3. Problems with Data Redundancy:

Wasted Storage Space

Duplicating data unnecessarily consumes storage resources.

Data Inconsistency

When the same data exists in multiple places, updates in one location might not be reflected in others, leading to inconsistencies.

Maintenance Overhead

Keeping redundant data synchronized requires additional effort and can be error-prone.

Reduced Data Integrity

Data redundancy can make it harder to maintain the overall accuracy and integrity of the data.

4. Data redundancy isn't always entirely negative.

Here are some scenarios where it might be intentional:

Performance Optimization

In some cases, strategically duplicating data can improve query performance by reducing the need for complex joins across tables.

Data Availability

Duplicating data in geographically dispersed locations can improve disaster recovery and ensure data accessibility during outages.

Distributed Id

首先，分布式 ID，是因为随着业务的发展，数据或者 ID 的生成会在不同的地方，比如
常见
的数据的分库分表，在分库分表的情况下，假如不同的表生成的 ID 不能保证唯一性，
那么
就会导致一个 ID 不能保证一条数据的唯一性，就会带来问题。
所以分布式 ID 的主要目的：不论何时何地，生成的 ID 都是唯一的。
实现的方式有很多：
1.Mysql 的全局 ID 生成表
2.UUID（有些版本会重复）
3.Redis 的 incr 自增 ID
4.zk 的有序节点序号
5.mongoDB 的 objectId、

Distributed Transaction

Distributed Transaction Protocol

TCC (Try, Confirm, Cancel)

TCC (Try, Confirm, Cancel) is a distributed transaction protocol that divides the transaction process into three steps:

Try

Reserve necessary resources and ensure that the transaction can be executed successfully.

Confirm

Execute the actual transaction. This step is performed only if the Try phase is successful for all participants.

Cancel

Roll back the transaction and release any reserved resources if the Try phase fails for any participant.

Local Message Table

The **Local Message Table** pattern is used to ensure data consistency between services in a distributed system by using a message table in the local database:

Local Transaction

Write the business logic and the message to be sent in the same local transaction.

Message Relay

A separate process or thread reads messages from the local message table and sends them to a message broker or another service.

Acknowledge and Cleanup

Once the message is successfully processed, it is acknowledged, and the corresponding entry in the local message table is cleaned up.

Best Effort Notification

Best Effort Notification is a pattern where the sender makes a best effort to notify the receiver of a change or event:

Send Notification

The sender attempts to notify the receiver of the event.

Retries and Acknowledgment

If the notification fails, the sender retries a few times. The receiver acknowledges the notification upon successful processing.

No Guaranteed Delivery

There is no guarantee that the notification will be received, only that best efforts will be made to deliver it.

Creating an index in a database

When creating an index in a database, it's essential to carefully consider several factors to ensure that the index improves query performance without introducing unnecessary overhead.

Here are key considerations to keep in mind:

Query Patterns:

Frequent Queries: Identify the most frequent and performance-critical queries. Create indexes that align with the columns used in the WHERE, JOIN, ORDER BY, and GROUP BY clauses of these queries.

Column Usage: Focus on columns that are frequently used for filtering, sorting, or joining. Indexes on these columns can significantly speed up query execution.

Index Type:

Single-Column vs. Composite Index:

Decide whether a single-column index or a composite (multi-column) index is more appropriate based on your query patterns. Composite indexes are beneficial for queries that filter or sort on multiple columns in combination.

Unique Index: Consider creating a unique index if you need to enforce uniqueness on a column or a combination of columns. This prevents duplicate values in the indexed columns.

Full-Text Index: For text-heavy columns where you need to perform search operations, such as in a search engine, consider using a full-text index.

Index Selectivity:

High Selectivity:

Create indexes on columns with high selectivity, meaning they have many unique values.

High selectivity indexes are more likely to be used by the query optimizer and provide better performance improvements.

Low Selectivity:

Indexes on columns with low selectivity (e.g., binary columns like gender) might not be as useful and can even be ignored by the query optimizer in some cases.

Index Maintenance and Overhead:

Insert/Update/Delete Performance:

Indexes can slow down INSERT, UPDATE, and DELETE operations because the database must maintain the index when data is modified. Consider the trade-off between read performance and write performance.

Index Size:

Indexes consume disk space. The more indexes you create, the more storage is required, and the more memory is needed to keep the indexes in memory for quick access.

Be mindful of the storage and memory usage of your indexes.

Column Order in Composite Index:

Leading Columns:

In composite indexes, the order of columns matters. The index is most effective when queries use the leading (leftmost) columns.

For example, an index on (A, B, C) is useful for queries on A or A and B, but not as much for queries on B alone.

Equality and Range Conditions:

Place columns with equality conditions (e.g., =) before columns with range conditions (e.g., <, >, BETWEEN) in a composite index. This allows the database to efficiently use the index for range scans.

Data Distribution:

Skewed Data: If a column has skewed data distribution (e.g., many rows have the same value), the effectiveness of an index on that column might be reduced.

Consider this when deciding to index columns with uneven data distributions.

Cardinality: The cardinality of a column (the number of distinct values) impacts the index's effectiveness. High-cardinality columns often benefit more from indexing than low-cardinality columns.

Database Workload:

Read vs. Write Intensive: In read-heavy workloads, indexes can greatly improve performance. In write-heavy workloads, the overhead of maintaining indexes might outweigh the benefits, so careful index design is required.

Concurrent Access: In environments with high levels of concurrent access, the locking mechanisms and transaction isolation levels can impact index performance. Choose indexes that minimize locking contention.

Database-Specific Features:

Partitioning: If your table is partitioned, consider how indexes will work with the partitioning scheme. Some databases allow partitioned indexes, which can be more efficient for certain queries.

Indexing Options: Some databases offer specialized indexing options like partial indexes (indexing only a subset of rows) or function-based indexes (indexing the result of a function). Use these features when they align with your query requirements.

Testing and Monitoring:

Performance Testing: Before deploying an index to production, test it with real or simulated workloads to ensure it improves performance. Monitor the impact on both query execution times and write operations.

Query Plan Analysis: Use tools like EXPLAIN in MySQL or EXPLAIN PLAN in Oracle to analyze query execution plans. This helps you understand how the optimizer uses indexes and whether adjustments are needed.

Database migration

Checksum

A checksum is a value used to verify the integrity of data.
It is calculated from a data set (such as a file, a block of data, or a database row) using a specific algorithm.
If the data changes even slightly, the checksum value will change, indicating that the data has been altered.
Checksums are widely used in various fields, including data transmission, storage, and database replication, to detect errors or corruption.

How Checksums Work

- Data Input:
  - The original data is taken as input.
- Checksum Algorithm:
  - An algorithm (such as MD5, SHA-256, or CRC32) processes the data and produces a fixed-size string or number, known as the checksum.
- Verification
  - The checksum is stored or transmitted along with the data. When the data needs to be verified, the same checksum algorithm is applied to the data again.
  - The resulting checksum is compared to the original checksum:
    - If the checksums match, the data is assumed to be unchanged and thus valid.
    - If the checksums do not match, the data has been altered or corrupted.

Process

Pre-Migration Preparations

- Backup
  - Always start with a complete backup of your database to ensure you can restore data in case something goes wrong.
- Schema Comparison
  - Ensure that the schema of the source and destination databases are identical.
  - Tools like pt-schema-compare for MySQL or dbms_metadata.get_ddl for Oracle can be used to compare schemas.
  - This involves checking the definitions of tables, columns, indexes, constraints, views, triggers, and other database objects.
  - The goal is to verify that the databases have the same schema structure.

Data Migration Process

- Chunk-Based Migration
  - Migrate data in smaller chunks or batches rather than all at once. This makes it easier to track and verify data consistency.
- Dual Write Strategy
  - Slave Database (Optional)
    - Configure the new database as a slave database of the source database to synchronize data.
  - Asynchronous Write
    - Write data to the main database.
    - Simultaneously write data to the migration database.
  - Retry Mechanism and Error Logging
    - Implement a retry logic, such as retrying up to three times if the write operation to the migration database fails.
    - Log any errors encountered during the write operations to analyze and resolve issues.
    - Continue writing to both databases until no failed write operations remain, ensuring data consistency between them.

Grayscale release

Gradually switch traffic to the new database.

- ETL Tools

  Use Extract, Transform, Load (ETL) tools that have built-in data validation features. Examples include Apache NiFi, Talend, Oracle GoldenGate, AWS Database Migration Service (DMS), or custom scripts.

Real-Time Verification:

- Checksum Comparison

  Compute checksums for tables or data chunks on both the source and destination databases and compare them. Tools like pt-table-sync can help with this.

- Row Count Comparison

  Ensure that the number of rows in each table in the source and destination databases match after each chunk is migrated.

- Trigger-Based Verification

  Set up database triggers to log changes during migration and ensure they are reflected on both sides.

Post-Migration Verification:

- Full Data Comparison

  Perform a full data comparison between the source and destination databases. This can be resource-intensive but is essential for verifying data integrity.

- Application Testing

  Run your application against the destination database to ensure that it behaves as expected and that no data inconsistencies affect application functionality.

- Data Quality Checks

  Implement data quality checks to ensure data integrity. This includes checking for duplicates, null values where not allowed, and data type mismatches.

SQL Optimization

Index

Create Indexes

Create indexes on columns used frequently in WHERE, ON, ORDER BY, and GROUP BY clauses.

Composite Indexes

Use composite indexes for columns that are frequently used together in queries.

Avoid Over-Indexing

Too many indexes can slow down INSERT, UPDATE, and DELETE operations.

- INSERT Operations

  When a new row is inserted into a table, the database not only writes the new data to the table but also updates all relevant indexes.

  Each index on the table must be modified to include the new row.

- UPDATE Operations

  Updating a row can be even more complex because it might involve multiple steps:

  Index Key Changes

  If the update modifies columns that are part of an index, the database must update the index entries for these columns.

  Index Maintenance

  Even if the update doesn't directly affect indexed columns, the database still needs to verify that the integrity and structure of all indexes associated with the table being updated remain intact.

- DELETE Operations

  When a row is deleted, the database must remove the corresponding entries from all indexes.

  Similar to insert operations, the presence of multiple indexes means that the database has to perform more operations to delete all relevant index entries.

Avoid Full Table Scans

No Index on the Column

When there is no index on the column used in the `WHERE` clause, the database must scan the entire table.

```
SELECT * FROM employees WHERE department_id = 10;
```

Reason: If department_id is not indexed, a full table scan is required.

Non-Selective Index

When the indexed column has low cardinality or the query condition matches a large percentage of rows, the index is not helpful, and a full table scan may be chosen.

```
SELECT * FROM employees WHERE last_name = 'Smith';
```

Reason: If last_name has many distinct values or 'Smith' is common, the index may not reduce the scan cost.

Using Functions or Expressions

When functions or expressions are used in the `WHERE` clause, indexes on the column are often bypassed.

```
SELECT * FROM employees WHERE UPPER(first_name) = 'JOHN';
SELECT * FROM products WHERE SUBSTRING(product_code, 1, 3) = 'ABC';
SELECT * FROM sales WHERE total_price / quantity > 100;
SELECT * FROM transactions WHERE amount + 0.5 = 100;
```

Query Hints for Full Table Scan

Using query hints that explicitly instruct the database to perform a full table scan.

```
SELECT /*+ FULL(employees) */ * FROM employees WHERE salary > 50000;
```

Reason: The hint FULL(employees) forces a full table scan.

Using Wildcards in LIKE Clauses

When LIKE patterns start with a wildcard, indexes are not used, leading to a full table scan.

```
SELECT * FROM employees WHERE first_name LIKE '%John';
```

Reason: The pattern `%John` requires a full table scan as the wildcard at the beginning prevents index use.

OR Conditions Without Indexes

Using OR conditions with columns that are not indexed can lead to a full table scan.

```
SELECT * FROM employees WHERE department_id = 10 OR salary > 50000;
```

Reason: If neither department_id nor salary has an index, a full table scan is necessary.

If one side of the OR condition has an index,

the database optimizer may use that index to filter rows efficiently based on that part of the condition.

If both sides of the OR condition have indexes,

the optimizer may choose to use one or both indexes, depending on the query plan it deems most efficient.

Sql Tricks

UNION ALL vs UNION

UNION ALL

Generally more efficient than UNION because it simply concatenates the result sets without additional sorting or removing duplicates.

```
SELECT column1 FROM table1
UNION ALL
SELECT column1 FROM table2;
```

UNION

Less efficient than UNION ALL due to the additional step of sorting and removing duplicates.

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

WHERE vs HAVING

WHERE

Filters individual rows before grouping or aggregating, it is more efficient and can leverage indexes.

```
SELECT 1
FROM table2
```

```
    WHERE table2.column2 = table1.column1;
```

HAVING

Used to filter the results after the `GROUP BY` clause has aggregated them. This is useful for filtering based on aggregate values.

Less efficient for row-level filtering because it cannot take advantage of indexes as effectively as the `WHERE` clause.

```
SELECT 1
FROM table2
GROUP BY table2.column2
HAVING table2.column2 = table1.column1;
```

EXISTS vs IN

EXISTS

Checks the existence of rows that satisfy a certain condition in a subquery.

Generally more efficient for large datasets because it stops searching once a matching row is found.

```
SELECT column1
FROM table1
WHERE EXISTS (SELECT 1 FROM table2 WHERE table2.column2 = table1.column1);
```

IN

Checks if a value exists within a set of values returned by a subquery or a static list.

Can be less efficient than EXISTS for large subqueries, especially if the subquery returns a large result set.

```
SELECT column1
FROM table1
WHERE column1 IN (SELECT column2 FROM table2);
```

JOINs vs Subqueries

JOINs

Combines rows from two or more tables based on a related column between them.

Generally more efficient than subqueries as databases can optimize JOIN operations better.

```
SELECT t1.column1, t2.column2
FROM table1 t1
JOIN table2 t2 ON t1.column3 = t2.column3;
```

Subqueries

Filters or transforms data on the fly within the main query.

Can be less efficient due to nested query execution but useful for readability and complex filtering.

```
SELECT column1
FROM table1
WHERE column2 = (SELECT MAX(column2) FROM table2);
```

Avoid SELECT*

Avoid SELECT * in production queries to fetch all columns from a table.

Fetching only required columns reduces the amount of data transferred and processed.

```
SELECT column1, column2
FROM table1;
```

Use COUNT(*)

Use COUNT(*) to count rows in a table or result set.

Efficient in most databases and avoids issues with NULL values in specific columns.

```
SELECT COUNT(*)
FROM table1;
```

Avoid Aggregates with DISTINCT

Avoid using DISTINCT inside aggregate functions like COUNT, SUM, AVG when possible, as it can be less efficient.

Prefer filtering duplicates before aggregation if needed.

```
SELECT COUNT(DISTINCT column1)
FROM table1;
```

Paginate with LIMIT and OFFSET

Use LIMIT and OFFSET to paginate results in queries.

Efficient for limiting the number of rows returned, though OFFSET can become less efficient with large values.

```
SELECT column1
```

```
FROM table1
ORDER BY column1
LIMIT 10 OFFSET 20;
```

Batch or Defer Updates

Batch or defer updates in a transaction to reduce the number of writes.

Improves performance by minimizing the overhead of multiple individual update operations.

```
BEGIN TRANSACTION;
UPDATE table1 SET column1 = value1 WHERE condition1;
UPDATE table1 SET column2 = value2 WHERE condition2;
COMMIT;
```

Cache

CORE

Cache Inconsistency

Inconsistency occurs when the cache and the data store are not in sync, leading to stale or outdated data being served.

Cause

This can happen due to asynchronous updates, failures in writing through or writing back to the data store, or improper cache invalidation.

Solution

Implement robust cache invalidation policies and consider using strong consistency mechanisms like write-through or read-through caching.

Strong Consistency

Strong consistency ensures that any update to the data is immediately visible to all subsequent read requests.

In other words, once a data update is confirmed to be successful, all future reads will reflect that update.

Weak Consistency

Weak consistency allows for updates to be visible after some unspecified time, meaning there is no guarantee that an update will be immediately visible to all read requests.

This model can tolerate the situation where updated data might not be immediately accessible.

Eventual Consistency

Eventual consistency guarantees that, given enough time, all replicas of the data will converge to the same value.

This means that all updates will eventually propagate to all replicas, but there is no guarantee about the timing of when the updates will be visible.

Cache Strategies

Read-Through Cache

How It Works

- Read Operations
  - Automatic Cache Population:
    - The cache itself automatically loads data from the database when there is a cache miss.
  - Simplifies Read Logic:
    - The application interacts with the cache directly, without worrying about database access.
- Write Operations
  - Write-Behind Approach:
    - Read-through caching often pairs with a write-behind approach, where updates to the cache are propagated to the database asynchronously.
  - Cache Consistency:

The cache handles consistency and synchronization with the database, simplifying application logic.

Use Cases:

- Useful for applications that require simplified caching logic and automatic cache population.
- Effective for applications with read-heavy workloads.

Issues

- Cache Invalidation

  Managing cache invalidation or updates needs to be handled separately, and it may add complexity.
- Overhead

  The cache must implement logic for fetching data from the backing store, which may introduce some overhead.

Write-Through Cache

How It Works

- Read Operations

  When a read operation is performed, the application first checks the cache.

  If the data is not in the cache (cache miss), the application loads the data from the backing store, stores it in the cache, and then returns it.
- Write Operations

  Synchronous Writes to Both Cache and Database:

  Ensures that data is always consistent between the cache and the database.

  Immediate Consistency:

  Data is always written to both the cache and the database in a single transaction.

  No Cache Invalidation Required:

  Since the cache is always up-to-date with the database, there is no need for explicit invalidation or refresh logic.

Use Cases:

- Ideal for applications where data consistency is critical and write performance can be traded off for consistency.
- Suitable for write-heavy applications where changes need to be immediately reflected in the cache.

Issues

- Write Latency

  Can introduce latency for write operations since data needs to be written to both the cache and the backing store before the operation is considered complete.
- Scalability

  May not be suitable for high write throughput applications, as the synchronous nature of the writes can become a bottleneck.

Cache Aside (Lazy Loading)

How It Works

- Read Operations:

  When a read operation is performed, the application first checks the cache.

  If the data is not in the cache (cache miss), the application loads the data from the backing store, stores it in the cache, and then returns it.
- Write Operations:

  Write Directly to Database:

  The application writes directly to the database.

  Cache Invalidation:

  After writing to the database, the application usually invalidates or updates the relevant cache entry to ensure consistency. This is done synchronously by the application.

  Consistency Management:

The application must ensure that cache and database remain consistent, which can introduce complexity.

Use Cases:

- Suitable for read-heavy applications where data changes are infrequent.
- Provides flexibility in controlling when and how data is cached.

Issues

- Concurrency Issues

  If thread A finishes updating the database and is preparing to update the cache, but another thread B performs a concurrent update to both the database and the cache,
  thread A may later update the cache with old data. This can lead to stale or inconsistent data in the cache.
- Cache Miss Penalty

  Cache misses result in higher latency as data is fetched from the backing store and then loaded into the cache.
- Implementation Complexity

  Requires careful management of cache invalidation and updating logic.

Write-Behind (Write-Back) Cache

How It Works

- Read Operations

  When a read operation is performed, the application checks the cache.
  If the data is not in the cache (cache miss), the application loads the data from the backing store, stores it in the cache, and then returns it.
- Write Operations

  Write to Cache First:

  Data is initially written to the cache, allowing for quick write operations.

  Deferred Database Writes:

  The database is updated asynchronously, either individually or in batches, based on predefined policies.

  Failure Handling Required:

  Mechanisms are needed to ensure that cached data changes are not lost before being written to the database, maintaining data integrity.

Use Cases:

- Write-Heavy Applications:

  Suitable for applications with a high volume of write operations that can tolerate eventual consistency.
- Batch Processing Workloads:

  Ideal for scenarios where updates can be efficiently batched to reduce database load.
- Reduced Database Load:

  Effective in environments where it is crucial to minimize the immediate impact on the database by smoothing out spikes in write operations.

Advantages

- Improved Write Performance

  By allowing quick acknowledgments, write performance is improved because the backend data store is updated asynchronously.
- Reduced Latency

  Immediate acknowledgment of write operations reduces latency for write-heavy applications.
- Efficient Resource Usage

  Asynchronous writes can batch updates, reducing the load on the backend data store.

Issues

- Data Consistency

  If the cache is updated successfully but the database update fails, this results in dirty cache data.
- Complexity

Implementing asynchronous writes and handling potential failures can be more complex than synchronous methods.

- Durability

  If the cache fails before the data is written to the backend store, there could be data loss.

Cache Preloading (Cache Warming)

The cache is pre-populated with data before it is actually needed, often during application startup or scheduled intervals.

How It Works

- Initial Load:

  At system startup or at scheduled intervals, the cache is pre-populated with data from the backing store.

  (Event Listener or Scheduled Tasks)

  The application loads a set of data through scripts, batch processes, or scheduled tasks into the cache in advance, before it is needed by the application.

- Read Operations:

  After preloading, read operations can access the data directly from the cache, reducing the latency of fetching data from the backing store.

- Write Operations:

  Write operations are handled as usual, with updates going to the backing store. The cache may be updated or invalidated as necessary to keep it in sync with the backing store.

Advantages

- Reduced Latency

  By preloading frequently accessed data into the cache, read operations benefit from faster access times, as data is available in memory.

- Improved Performance

  Reduces the likelihood of cache misses during peak usage times, leading to more consistent performance.

- Predictable Performance

  Ensures that critical data is available in the cache ahead of time, providing more predictable performance under load.

Issues

- Cache Warming Overhead

  The process of preloading data can be resource-intensive and may impact the performance of the backing store during the initial load.

- Stale Data

  Preloaded data may become outdated if the backing store is updated frequently. Cache invalidation or refreshing strategies need to be in place to address this.

- Cache Size Management

  Preloading a large amount of data may require significant memory resources. Careful management of the cache size and data selection is needed to avoid excessive memory usage.

Refresh-Ahead Cache

How It Works:

The cache preemptively refreshes items before they expire.

Ensures that data is always up-to-date when accessed.

- Cache Expiry:

  The cache is configured with a Time-to-Live (TTL) or expiration time for each cache entry.

- Refreshing Data:

  Before a cache entry expires, a background process or scheduled task proactively refreshes the data in the cache.

  The refreshed data is loaded from the backing store and replaces the old data in the cache before the TTL

expires, ensuring that fresh data is available as soon as the old data is invalidated.

- Read Operations:
  When a read operation is performed, the application retrieves data from the cache.
  If the data is present, it is returned immediately.
  The background refresh process ensures that the data in the cache is kept up-to-date, reducing the likelihood of cache misses due to expired data.
- Write Operations:
  Write operations update the backing store directly and may update or invalidate the cache as necessary.

Advantages

- Reduced Latency
  Ensures that data is always available in the cache and up-to-date, reducing latency for end users.
- Improved User Experience
  Provides a smoother user experience by avoiding cache misses and reducing the likelihood of serving stale data.
- Predictable Performance
  Keeps cache data fresh without waiting for the data to expire, leading to more predictable performance.

Delayed Double Deletion

How It Works

- Read Operations
  When a read operation is performed, the application first checks the cache. If the data is not in the cache (cache miss),
  the application retrieves the data from the backing store, stores it in the cache, and then returns the data to the requester.
- Write Operations
  When data is updated, the application performs the following steps:
    Deletes the cache entry associated with the data.
    Writes the updated data to the backing store.
    Waits for a short period (delayed) to allow for any concurrent updates to complete.
    Deletes the cache entry again to ensure that it is removed and to prevent stale data from being served.

Advantages

- Data Consistency
  Reduces the risk of serving stale data by ensuring that the cache entry is eventually removed, even if there are concurrent updates or deletions.
- Mitigates Race Conditions
  By performing the delete operation twice with a delay, the strategy helps handle race conditions where the cache might be accessed or modified concurrently.
- Reliability
  Enhances the reliability of cache invalidation, ensuring that outdated or inconsistent data is not served from the cache.

Issues

- Complexity
  Introduces additional complexity compared to simple cache invalidation strategies due to the need for managing delays and handling potential race conditions.
- Latency
  The delay introduced in the deletion process can add latency to cache operations, which might affect performance, especially in systems with high update frequencies.
- Overhead

The strategy can result in additional overhead due to the extra delete operations and the need to manage delays, which can impact overall system performance.

In-Memory Caching

How It Works:

In-memory caching stores data in the RAM (Random Access Memory) of a server or cluster of servers. This enables rapid data retrieval compared to disk-based storage.

- Read Operation

  When an application needs data, it first checks the in-memory cache. If the data is found (cache hit), it is retrieved directly from the cache.

  If the data is not found (cache miss), the application fetches it from the primary data store (e.g., a database), places it in the cache, and then returns it.
- Write Operation: When data is updated or added, the cache is updated along with the primary data store.

  This can be done synchronously (write-through) or asynchronously (write-behind).

Advantages:

- Speed

  Accessing data in-memory is significantly faster than fetching data from disk, leading to improved application performance.
- Reduced Load on Data Stores

  By serving data from the cache, the load on the primary data store is reduced, which can enhance scalability and reduce response times.

Browser Cache

Local Storage

Local storage is a web storage mechanism that stores data on a user's computer indefinitely unless explicitly deleted.
Controlled entirely by frontend JavaScript.

How It Works:

- Persistence

  Data persists even after the browser is closed and reopened.
- Scope

  Data is accessible across all tabs and windows from the same origin.
- Storage Capacity

  Typically around 5-10MB per origin, depending on the browser.
- Usage Examples

  Ideal for storing user preferences, settings , state, and other long-term data.

Advantages:

- Persistence

  Long-term data storage.
- Accessibility

  Shared across multiple tabs and windows of the same origin.

Issues:

- Security

  Data is stored in plaintext and can be accessed via JavaScript, posing a risk if sensitive data is stored.
- Size Limitation

  Limited storage capacity.

Session Storage

Session storage is a web storage mechanism that stores data for the duration of the page session.
Controlled entirely by frontend JavaScript.

How It Works:

- Persistence
  Data is available only for the duration of the page session (until the browser tab or window is closed).
- Scope
  Data is accessible only within the same tab or window.
- Storage Capacity
  Typically around 5-10MB per origin, depending on the browser.
- Usage Examples
  Suitable for temporary data like session tokens or transient application state.

Advantages:

- Session-based Persistence
  Ideal for temporary data storage.
- Isolation
  Data is isolated to the tab or window where it was set.

Issues:

- Scope Limitation
  Data is not shared across tabs or windows.
- Security
  Similar security risks as local storage.

IndexedDB Storage

IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files and blobs. IndexedDB is accessed and manipulated through the JavaScript API in the browser. This API allows web applications to create, read, update, and delete data within IndexedDB.

How It Works:

- Persistence
  Data persists indefinitely unless explicitly deleted.
- Scope
  Data is accessible across all tabs and windows from the same origin.
- Storage Capacity
  Much larger capacity compared to local and session storage, often measured in hundreds of megabytes or more.
- Usage Examples
  Suitable for complex data storage, such as offline applications, large datasets, and blob storage.

Advantages:

- High Capacity
  Suitable for storing large amounts of structured data.
- Complex Queries
  Supports transactions and complex queries.

Issues:

- Complexity
  More complex API compared to local and session storage.
- Browser Support
  Not all features are supported in all browsers.

Cookies

Cookies are small pieces of data stored by the browser and sent to the server with every request to the same domain.
Controlled by Both UI Logic and Backend Service

How It Works:

- Persistence
  Can be set to expire at the end of the session (session cookies) or at a specific date (persistent cookies).
- Scope
  Data is accessible across all tabs and windows from the same origin and is sent to the server with every request.
- Storage Capacity
  Typically limited to about 4KB per cookie.
- Usage Examples
  Suitable for storing user authentication tokens, session identifiers, and user preferences.

Advantages:

- Server Communication
  Automatically sent with every request, useful for session management.
- Expiration Control
  Can be set to expire at specific times.

Issues:

- Size Limitation
  Limited storage capacity.
- Performance Impact
  Sent with every request, potentially impacting performance.
- Security
  Subject to cross-site scripting (XSS) attacks if not properly secured.

Time-to-Live (TTL) Based Caching

How It Works

- Cache Expiry:
  Each cache entry is assigned a Time-to-Live (TTL) value, which specifies the duration (in seconds) for which the entry should remain in the cache.
  After the TTL expires, the cache entry is automatically invalidated and removed from the cache.
- Read Operations:
  When a read operation is performed, the application checks the cache for the requested data.
  If the data is present and has not expired, it is retrieved from the cache.
  If the data is not present or has expired (cache miss), the application fetches the data from the backing store, stores it in the cache with a new TTL, and then returns the data to the user.
- Write Operations:
  When data is written to the backing store, the cache may either be updated with a new TTL or invalidated.
  The TTL-based cache ensures that data is refreshed or evicted based on the TTL settings.

Advantages

- Automatic Expiry
  Automatically manages the lifecycle of cache entries, reducing the need for manual cache invalidation.
- Reduced Stale Data
  Helps prevent stale data from lingering in the cache, as entries are periodically removed after their TTL expires.
- Predictable Cache Size
  By setting appropriate TTL values, you can control the size and turnover rate of the cache, avoiding excessive memory usage.

Issues

- TTL Misconfiguration
  Incorrect TTL settings can lead to either excessive cache evictions (if TTL is too short) or stale data (if TTL is too long).

- Cache Misses
  Data that expires frequently can lead to a higher number of cache misses and increased load on the backing store.
- No Immediate Invalidation
  TTL-based caching does not immediately reflect updates made to the backing store unless the cache entry is explicitly invalidated.

Content Delivery Network (CDN) Caching

How It Works

- Content Distribution:
  A Content Delivery Network (CDN) caches static assets such as images, videos, CSS files, and JavaScript files across a distributed network of edge servers.
  These edge servers are strategically located in various geographic regions to reduce latency and improve access speeds for users.
- Request Handling:
  When a user requests content, the request is routed to the nearest edge server based on the user's geographic location.
  If the requested content is available in the CDN cache (cache hit), it is served directly from the edge server.
  If the content is not available in the cache (cache miss), the edge server retrieves it from the origin server, caches it, and serves it to the user.
- Cache Control:
  CDN providers use cache control headers (e.g., Cache-Control, Expires) to determine how long content should be cached at edge servers.
  These headers help manage the caching duration and control when content should be refreshed or invalidated.

Advantages

- Improved Performance
  Reduces latency by serving content from geographically closer edge servers, improving load times for end-users.
- Reduced Load on Origin Server
  Offloads the content delivery from the origin server to the CDN, reducing the load and improving scalability.
- Scalability
  Handles large volumes of traffic and spikes in demand by distributing the load across multiple edge servers.
- High Availability
  Enhances content availability and reliability by serving cached content even if the origin server experiences issues.

Issues

- Cache Invalidation
  Managing cache invalidation can be complex, especially for dynamic content that changes frequently. Stale content may be served if cache invalidation policies are not properly configured.
- Cost
  CDN services can incur additional costs based on data transfer and storage, which may impact overall budget.
- Complexity
  Integrating and configuring a CDN can add complexity to the infrastructure and require additional setup and maintenance.

Strong Consistency Models

Strong consistency models guarantee that every read operation receives the most recent write operation's value or an error.

This ensures that all nodes in a distributed system have a consistent view of the data at all times, regardless of where and how data is accessed or modified.

Key Characteristics

Immediate Consistency

: Updates to the data are immediately visible to all clients once committed.

Synchronization

: All replicas or nodes are updated synchronously, ensuring that no stale data is served to clients.

Serializability

: Transactions appear to execute sequentially, even if they occur concurrently, to maintain a consistent order of operations.

Use Cases

Strong consistency models are essential in applications where data integrity and immediate consistency are critical, such as financial transactions, e-commerce platforms, and real-time collaborative editing tools.

Eventual Consistency with Reconciliation

Eventual consistency acknowledges that achieving immediate consistency in distributed systems can be challenging due to network partitions, latency, and the CAP theorem.

It allows for temporary inconsistencies that are resolved over time as updates propagate across distributed nodes.

Key Characteristics

Asynchronous Updates

: Updates are propagated asynchronously across distributed nodes, allowing local operations to proceed without waiting for global consensus.

Eventual Convergence

: Over time, all replicas or nodes converge to a consistent state through periodic reconciliation processes.

Availability and Partition Tolerance

: Ensures that the system remains available and operational, even in the presence of network partitions or temporary failures.

Use Cases

Eventual consistency with reconciliation is suitable for scalable distributed systems where high availability and partition tolerance are priorities,

such as social media platforms, content delivery networks (CDNs), and distributed content management systems.

Cache Eviction Policies

LFU (Least Frequently Used)

LFU evicts the least frequently accessed items from the cache.

It keeps track of how often each item is accessed and removes the item with the lowest access frequency when the cache is full.

Usage

: Useful when there are long periods between accesses to some items, ensuring that frequently accessed items remain in cache.

MRU (Most Recently Used)

MRU evicts the most recently accessed items from the cache when it needs to make space for new items.

This means that when the cache is full and a new item needs to be added, MRU will remove the item that was accessed most recently.

Usage

: MRU prioritizes retaining items that were accessed further in the past over items that were accessed more recently.

This can be useful in scenarios where recently accessed items are less likely to be accessed again soon, or when there is a need to prioritize items that have not been accessed recently.

FIFO (First In, First Out)

FIFO evicts the oldest items in the cache based on the order they were added. It operates like a queue where the first item added is the first to be removed when the cache is full.

Usage

Simple to implement and suitable for scenarios where access patterns do not affect eviction decisions.

LIRS (Low Inter-reference Recency Set)

LIRS is an advanced variant of LRU that distinguishes between recently accessed items and items accessed again after a longer period. It aims to improve performance by maintaining two distinct lists for recent and less recent accesses.

Usage

Designed for systems with complex access patterns where distinguishing between recent and less recent accesses can optimize cache performance.

ARC (Adaptive Replacement Cache)

ARC dynamically adjusts between LRU and LFU based on recent access history. It uses both recent access and frequency of access to determine cache eviction decisions.

Usage

Adaptive to changing workload patterns and balances between the benefits of LRU and LFU.

CLOCK (Approximate LRU)

CLOCK maintains a circular list of cache entries and uses a clock hand to mark recently accessed items.
It combines aspects of FIFO with LRU by replacing items based on a "use bit" rather than exact access timestamps.

Usage

Provides a balance between simplicity (like FIFO) and performance optimization (like LRU).

Common Caching Issues

Cache Stampede

This occurs when multiple requests for the same data result in repeated cache misses and simultaneous fetches from the data store, overloading the backend.

Cause

High concurrency of requests for the same key when the cache entry expires.

Solution

Implement mechanisms like request coalescing (batching multiple requests into one), using locks or semaphores, and employing techniques like "early recomputation" or "refresh-ahead".

- Staggered Expirations

  Implement varied expiration times for cache entries to reduce the likelihood of multiple entries expiring simultaneously.

- Cache Preloading

  Proactively load or periodically refresh cache entries before they expire, based on access patterns or anticipated usage, to minimize the impact of simultaneous requests.

- Mutex or Locking Mechanisms

  Use mutex or locking mechanisms to ensure only one client or process fetches the data from the backend while others wait for the result. This reduces the load on the backend by preventing simultaneous fetches.

- Fallback Mechanisms

  Implement fallback strategies such as serving stale data temporarily or using a secondary cache to handle

requests while the primary cache is being repopulated.

- Rate Limiting

  Control the rate of requests to the backend during cache rebuilds or high traffic periods to avoid overwhelming the data store and mitigate the effects of cache stampede.

Cache Avalanche

When a large number of cache entries expire simultaneously, it leads to a sudden spike in cache misses.
This results in a significant increase in requests to the backend data store, which can overwhelm the system, causing degraded performance or even system downtime.

Cause

Simultaneous expiration of a large number of cache entries, often due to having the same TTL (Time-to-Live) for many items.

Solution

- Stagger Expiration Times

  Ensure that cache entries have varied expiration times to prevent them from expiring all at once.

- Cache Preloading

  Preload or warm up the cache with frequently accessed data during off-peak hours or upon startup to avoid a sudden load on the data store.

- Fallback Mechanisms

  Implement fallback strategies such as serving stale data temporarily or using a secondary cache to handle requests while the primary cache is being repopulated.

- Rate Limiting

  Control the rate of requests to the backend during cache rebuilds to avoid overwhelming the data store.

Cache Penetration (Cache Miss Storm)

Repeated requests for non-existent keys cause continuous cache misses, leading to unnecessary load on the data store.

Cause

Querying for keys that do not exist in the cache or the data store.

Solution

- Caching Negative Results

  Store a "null" or "not found" entry in the cache with a short Time-to-Live (TTL) to prevent repeated cache misses for the same non-existent keys.

- Input Validation

  Validate user input and requests to filter out invalid or malicious queries before they reach the cache or data store.

- Pattern-Based Filtering

  Analyze access patterns and apply filtering or caching strategies to block or mitigate potential cache penetration attacks.

- Monitoring and Alerts

  Monitor cache access patterns and set up alerts for unusual spikes in requests for non-existent keys, which may indicate a cache penetration attack.

- Rate Limiting

  Implement rate limiting mechanisms to control the number of requests from a single client or IP address within a specific timeframe, preventing them from causing excessive cache misses.

Cold Cache Latency

Cold cache at application startup can lead to high latency and poor performance until the cache is populated.

A cold cache refers to a cache that is initially empty when an application or system starts.
Over time, as the application runs and data is requested, the cache becomes populated with data, transitioning from a "cold" state to a "warm" or even "hot" state.

Cause

Lack of preloading or warming strategies.

Solution

Implement cache warming strategies to prepopulate the cache during off-peak hours or at application startup.

Cache Size Management

Overly large caches can lead to increased memory usage, while undersized caches might lead to frequent cache misses.

Cause

Poor estimation of required cache size.

Solution

Monitor cache hit/miss ratios and adjust the cache size accordingly. Implement dynamic scaling if necessary.

Eviction Policies

Cache Pollution

Infrequently accessed data occupies cache space, displacing more frequently accessed data.

Cause

Poor cache eviction policies or lack of proper cache key management.

Solution

Use appropriate eviction policies like Least Recently Used (LRU), Least Frequently Used (LFU), or a combination thereof, and monitor cache usage patterns to adjust strategies as needed.

Cache Eviction Issues

Important or frequently accessed data is prematurely evicted from the cache.

Cause

Inadequate eviction policies or insufficient cache size.

Solution

Tune eviction policies based on access patterns and consider increasing the cache size if necessary.

Data Staleness

Serving outdated data from the cache that does not reflect the latest state in the data store.

Cause

Ineffective cache invalidation or update policies.

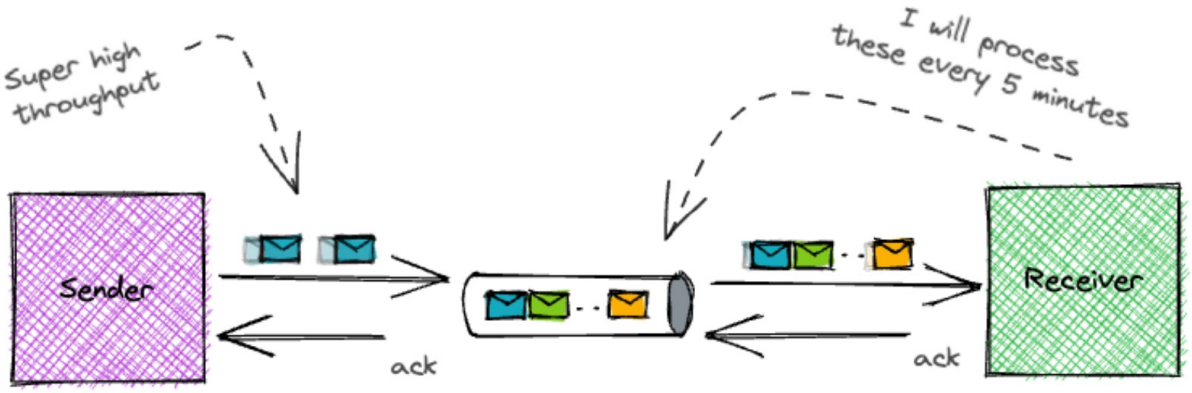
Solution

Ensure proper invalidation, use TTL appropriately, and consider more dynamic update mechanisms.

Message Broker

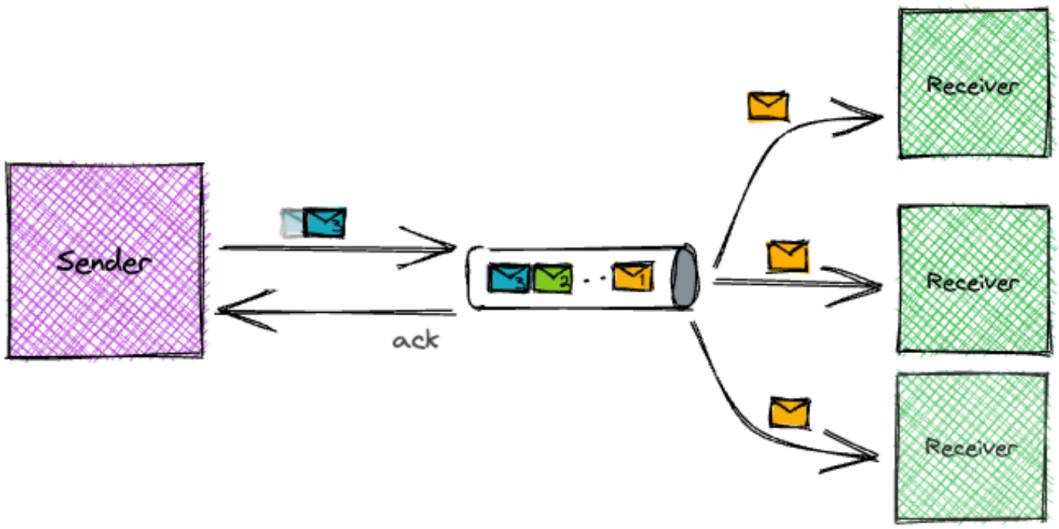
Usage Scenarios

Reduce Pressure off Downstream consumers



Many people use message brokers to reduce back pressure from downstream services.
The sender puts messages on the queue, and consumers can pick up these messages and process them in the batch size they want and the time they like.
You can use message brokers to make sure that downstream services do not get too overloaded, which can improve system reliability.
Example of this could be to use an SQS queue, to handle many messages, and a consumer to process these messages and delivery information to a third-party API.

Parallel Processing



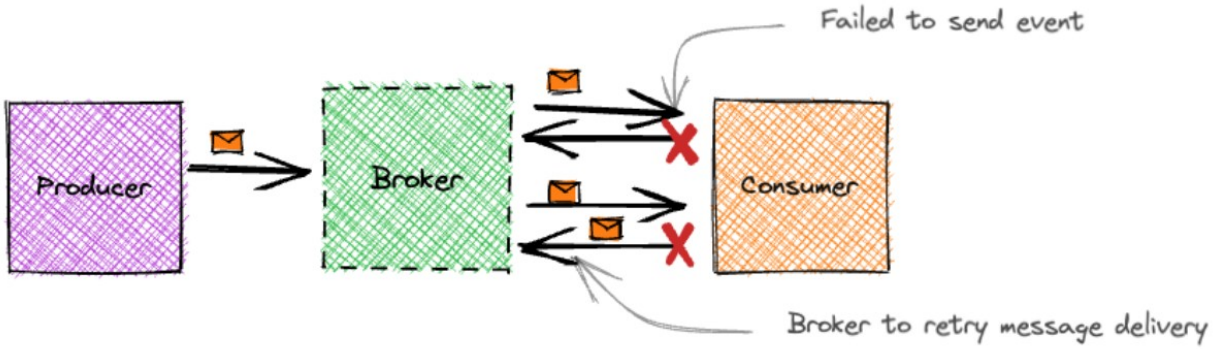
Publish-Subscribe Model

- Some brokers offer pub/sub patterns allowing you to send events and notify downstream consumers.
- This pattern allows you to notify downstream processes that something has happened. Many downstream systems can listen to these events.
- If you are using message queues, you can still have many consumers processing messages from the queue, this allows you to scale processing downstream.

Asynchronous tasks

- Offload some tasks that don't need to be executed synchronizely
- Separate some tasks that do not need to be executed synchronously

Prevent Messages/Data being Lost



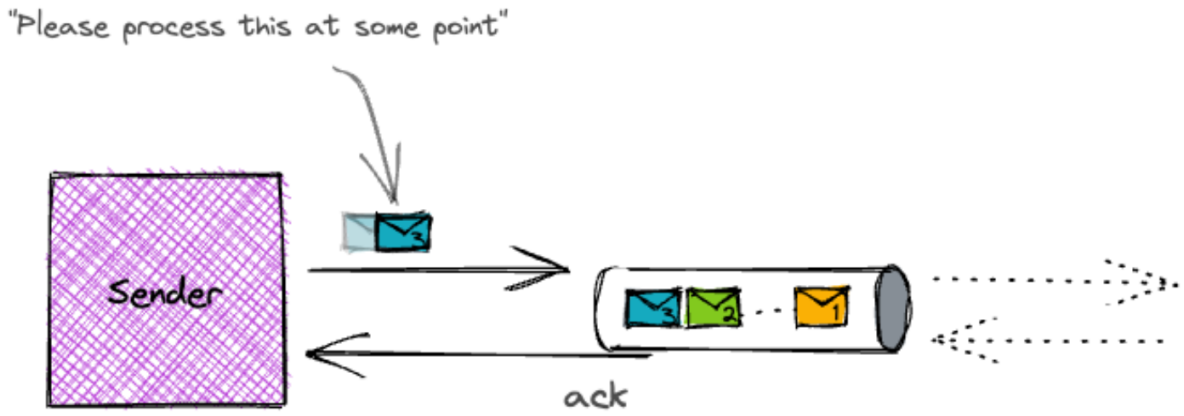
Retry mechanism

Many brokers offer the ability to retry messages/events if they fail to get processed or delivered. This can help prevent any information being lost.

If your broker does offer retry and replay, you want to consider idempotency as your consumer may be triggered more than once, and you don't want any unwanted side effects.

This could be a great pattern, as you broker may do the heavy lifting for you (retry/reprocess), many are configurable and if all fails you can decide what you want to do with the messages after the retry period (example drop them or store them for later processing)

Reducing Knowledge of Systems



Using messages/events to communicate between systems/services can help us decouple our architecture.
Producers may not need to know about downstream consumers, this gives the producers the ability to isolate and focus within its own domain/boundary.

Log Collection

Message Brokers can be used to collect logs from various services across an organization. These logs can then be processed or analyzed by different consumers.
Example:

Centralized log aggregation where logs from web servers, databases, and applications are published to Kafka topics. These logs can then be consumed by systems like Hadoop, HBase, or Solr for further processing, analysis, and storage.

Queue

Dead Letter Queue

Reasons for Dead-Lettering:

- Message Expiration
  If a message has a Time-To-Live (TTL) and it expires before being processed.
- Message Rejection
  If a consumer explicitly rejects a message (e.g., using basic.reject or basic.nack).
- Queue Length Limit Exceeded
  If the queue has a length limit and it is exceeded.

Dead Letter Queue in Amazon SQS

Many hosted messaging systems, including Amazon's Simple Queuing Service (SQS) include a Dead Letter Channel.
Amazon's Dead Letter Queue behavior is closely related to the way SQS delivers messages.
After a consumer fetches a message from a queue, the messages remains on the queue, but hidden from other consumers to avoid fetching and processing the same message multiple times.
After processing a message, the consumer is responsible for explicitly deleting the message.
If the consumer doesn't delete the message, for example because because it crashed while processing the message, the message becomes visible again after the message's Visibility Timeout expires.
Each time this happens, the message's receive count is increased. When this count reaches a configured limit, the message is considered undeliverable and placed in a designated Dead Letter Queue.
Other AWS services like Amazon EventBridge also feature a Dead Letter Channel.

Deferred Delivery Queue

A more professional name for a delay queue in RabbitMQ is a Deferred Delivery Queue or Scheduled Message Delivery Queue.
These queues are used to defer the delivery of messages to consumers for a specified period. This feature is particularly useful for implementing features such as retry mechanisms, delayed processing, or scheduled tasks.

Key Concepts of Deferred Delivery Queue

- Message Delay
  Messages are temporarily held in the queue and delivered to consumers only after a specified delay period.
- TTL (Time-To-Live)
  The TTL value on messages defines how long they should be held in the queue before being processed.
- Dead-Letter Exchange (DLX)
  Messages are initially routed to an intermediate queue with a TTL. After the TTL expires, they are forwarded to a DLX, which routes them to the intended queue for processing.

Programmatic Delay Queues

org.springframework.scheduling.annotation.Scheduled
java.util.concurrent.DelayQueue

Redis

org.springframework.data.redis.listener.KeyspaceEventMessageListener
org.redisson.api.RDelayedQueue

RabbitMQ

org.springframework.amqp.core.MessageBuilder
x-message-ttl
rabbitmq_delayed_message_exchange

MQ Comparison

Key differences between RabbitMQ, ActiveMQ, RocketMQ, and Kafka

Feature	RabbitMQ	ActiveMQ	RocketMQ	Kafka
Type	Message Broker	Message Broker	Distributed Messaging/Streaming Platform	Distributed Streaming Platform
Protocol	AMQP, MQTT, STOMP (via plugins)	AMQP, MQTT, OpenWire, STOMP	Proprietary	Proprietary
Features	Reliability, Flexible Routing	Persistence, Clustering	High Throughput, Low Latency, Transactional Messaging, Batch Processing	Distributed, Fault-Tolerant, Real-time Processing, Log Aggregation
Use Cases	Microservices, Reliable Messaging	Enterprise Messaging, Pub-Sub	Real-time Messaging, Streaming, Big Data Analytics	Real-time Analytics, Log Aggregation, Event Sourcing
Scalability	Good	Good	Excellent	Excellent
Latency	Low to Medium	Low to Medium	Low	Low
Persistence	Yes	Yes	Yes	Yes
Message Model	Queues, Topics	Queues, Topics	Messaging, Streaming	Streaming

Use Optimistic Locking

YourEntity

To manage concurrent updates, use optimistic locking in your JPA entities:

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Version;

@Entity
public class YourEntity {

    @Id
    private Long id;

    private String data;

    @Version
    private Long version;     // Optimistic lock version

    // Getters and setters
}
```

Ensure Idempotency

Idempotency ensures that processing the same message more than once does not produce different results. Implement idempotent operations in your message processing logic:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class YourService {

    @Autowired
    private YourEntityRepository repository;
```

```
    @Transactional
    public void processMessage(String messageId, String data) {
        // Check if the message was processed before
        if (repository.existsById(messageId)) {
            // Message already processed, skip
            return;
        }

        YourEntity entity = new YourEntity();
        entity.setId(messageId);
        entity.setData(data);

        // Save the entity (Optimistic locking ensures data consistency)
        repository.save(entity);
    }
}
```

Configure RabbitMQ Acknowledgment

Ensure that your RabbitMQ listener is configured to acknowledge messages only after successful processing. This can be done manually or automatically:

```
 spring:
   rabbitmq:
     listener:
       simple:
         auto-ack: false
```

Then, in your listener, acknowledge the message manually only after processing:

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Service;
import com.rabbitmq.client.Channel;

@Service
public class MessageListener {

    @RabbitListener(queues = "yourQueueName")
    public void receiveMessage(String message, Channel channel, Message rabbitMessage) throws IOException {
        try {
            // Process the message
            processMessage(rabbitMessage.getMessageProperties().getMessageId(), message);

            // Manually acknowledge message after successful processing
            channel.basicAck(rabbitMessage.getMessageProperties().getDeliveryTag(), false);
        } catch (Exception e) {
            // Handle exceptions and possibly reject or requeue the message
            channel.basicNack(rabbitMessage.getMessageProperties().getDeliveryTag(), false, true);
        }
    }
}
```

System

Metrics

QPS

QPS stands for Queries Per Second.

It is a measure of how many queries or requests a system can handle per second.

QPS is a key performance metric for web services, databases, and any other system that processes requests.

hakari db connection numbers and db configuration loading order

UI Optimization

Web Page

- Reduce the size of downloaded files.
- Reduce the number of UI file requests to reduce the number of connections estabished and avoid DNS resoltion between different domains. (Multiple Resource Request)
- Avoid refreshing web pages on pages with high triffic.

Exponential Backoff Algorithm

How Exponential Backoff Works

1) **Initial Retry Delay**

   Start with an initial retry delay, for example, 1 second.

2) **Exponential Increase**

   After each failed attempt, the delay is doubled.

   For example, after the first failure, wait 1 second before retrying; after the second failure, wait 2 seconds; after the third failure, wait 4 seconds, and so on.

3) **Maximum Delay**

   Optionally, set a maximum delay to prevent the retry interval from growing indefinitely.

4) **Jitter**

   Add randomness (jitter) to the delays to avoid the thundering herd problem, where many clients retry simultaneously.

Cache

- Add Browser Cache to save Non-sensitive User Data.

Dynamic Data

- Get dynamic data from the proxy server instead of the client and import it into the web page to return it to the user.
- Asynchronous JavaScript requests.

Server Optimization
Idempotence

Token

Token (or token-based mechanism) is used to ensure idempotency or uniqueness in operations:

Generate Token

Create a unique token with an expiration time for the first reqeust and store It in a database or cache.

Verify Token

Before process the incomming requests, verify that the token is already in th database or cache.

Consume Token

If the token is valid and unused, perform the operation and mark the token as used to prevent re-use.

Select + Insert + Primary Key / Unique Index Conflict

Select:

Check if the record already exists by querying the database.

Insert:

If the record does not exist, insert the new record.

Primary Key / Unique Index Conflict

If the record exists (based on primary key or unique index constraints), handle the conflict (e.g., retry or handle as a duplicate).

Direct Insert + Primary Key / Unique Index Conflict

Insert

Directly attempt to insert the new record into the database.

Primary Key / Unique Index Conflict:

If the record already exists (based on primary key or unique index constraints), handle the conflict, typically catching an exception or error returned by the database.

Idempotency with State Machines

Idempotency ensures that multiple identical requests have the same effect as a single request. When using state machines:

State Machine:

Model the workflow as a series of states and transitions.

Idempotency:

Ensure that repeated operations that transition between states do not cause unintended side effects or duplicate state changes.

Extracting Duplicate Prevention Table

Table Extraction: Create a dedicated table to store unique keys or identifiers.

Check and Insert: Before performing an operation, check this table to see if the identifier already exists. Insert the identifier if it does not exist to ensure the operation is only performed once.

Optimistic Lock

Optimistic Locking uses versioning to handle concurrency:

Read with Version:

Read a record along with its version number.

Update with Version Check

When updating, check if the version number has changed. If it hasn't, proceed with the update; otherwise, abort or retry the transaction.

Usage

Suitable for scenarios with infrequent conflicts.

Pessimistic Lock

Pessimistic Locking involves locking a database record to prevent concurrent access:

Select for Update:

Lock the selected rows to prevent other transactions from modifying them until the lock is released.

Usage

Rarely used due to performance overhead and potential for deadlocks. Suitable when conflicts are expected to be frequent.

Distributed Lock

Distributed Locking ensures that only one instance of a process runs across a distributed system:

Lock Acquisition:

Use a distributed system (like Redis, Zookeeper, or a database) to acquire a lock.

Exclusive Access:

The process holding the lock can perform the operation exclusively.

Lock Release: Release the lock after the operation is complete.

Usage: Ensures consistency and prevents race conditions in distributed environments.

Data

Simple Response

Make returning data simple.

Request

Avoid external requests

Try to avoid database or other service interactions, and don't request some services that you don't need.

Shorten request routes

Reduce the length of the request route, avoid requesting too many servers.

Asynchronous Processing

Asynchronous Requests

For tasks that don't need to be processed immediately (e.g., sending emails, processing files), use asynchronous processing with @Async or message queues (e.g., RabbitMQ, Kafka).

Batch Processing

Aggregate similar tasks into batches and process them together to reduce the overhead on the system.

Cache

Cache data with large read requests

Cache product details, inventory counts, and user sessions to improve response times during peak load. （Flash Sale System）

Multi-Region Clusters

Multi-region clusters are designed to enhance the availability, reliability, and performance of data services by distributing the workload across multiple geographical regions.

This approach is especially valuable for critical applications where downtime and data loss must be minimized, and where global user bases demand low-latency access to data.

Such as: Membership System

Geographic Distribution:

- Primary and Secondary Regions

  Data and services are distributed across multiple regions.

  Each region can act as a primary or secondary, enhancing disaster recovery and ensuring low-latency access for users in different locations.

Data Partitioning and Replication:

- Data Sharding

  Data is partitioned into shards that are distributed across different regions.

- Replication

  Data is replicated between regions to ensure consistency and availability. Replication can be synchronous (strong consistency) or asynchronous (eventual consistency).

Load Balancing Server:

- Traffic Distribution

  Load balancers distribute incoming traffic across regions based on various criteria, such as load, proximity to the user, or health of the region.

- Failover Mechanism

  In case of a failure in one region, traffic is redirected to the remaining healthy regions to maintain service availability.

Long Connection Server:

Splitting a monolithic system

When do you need to split a microservice system?

Splitting a system into microservices can lead to an increased need for servers or other computing resources.

- Performance Bottlenecks:

  If certain services are experiencing significant performance issues due to high traffic or resource constraints, it might be beneficial to split them to optimize performance and allocate resources more efficiently.

- Scalability Requirements:
  When different parts of the system have different scaling needs, splitting services allows for more granular scaling.
  Services can be scaled independently based on their specific load and resource requirements.
- Business Requirements:
  If different parts of the system need to be managed or monitored differently due to varying business needs or priorities, splitting services can provide the necessary flexibility.

New Service

Create a new service, and configure several new server clusters.

Read service and Write service.

Split the system into a read service and a write service.

- Read service
  The read service handles all the read operations. It can filter out some invalid requests before forwarding them to the write service if necessary, thus reducing the pressure on the write system.
- Write Service
  The write service handles all the write operations. By isolating write operations, you can optimize the system to handle write-intensive tasks more efficiently.

Read database and Write database.

Split the database into a read database and a write database.

- Read Database
  The read database handles all read operations, such as queries to fetch data. It can be optimized for high-speed data retrieval and can scale horizontally to handle a large number of read requests.
- Write Database
  The write database handles all write operations, such as inserts, updates, and deletes. It can be optimized for handling write-intensive operations and ensuring data integrity.

Business

Authentication

Use OAuth or JWT for user authentication and session management.

Bundle Purchase

Implementing a "bundle purchase" feature involves creating a system that allows users to combine multiple items into a single purchase for a discount or special offer.

This is commonly used in e-commerce platforms to encourage customers to buy more items together. Here's how you might approach designing such a feature:

Define Requirements and Use Cases

Types of Bundles:

Fixed Bundles: Pre-defined combinations of products at a fixed price.

Custom Bundles: Allows users to select their own items to create a bundle.

Dynamic Bundles: Based on user behavior or preferences, automatically suggesting bundles.

Pricing and Discounts:

Percentage Discount: Apply a percentage discount to the total price of the bundle.

Fixed Amount Discount: Apply a fixed discount amount when a bundle is purchased.

Buy X Get Y: Offers such as "Buy 2 get 1 free" or "Buy one, get the second at 50% off".

User Experience:

Bundle Creation: Easy selection and addition of items to the bundle.

Display: Clear presentation of bundle offers and discounts.

Validation: Ensuring bundle rules are met before finalizing the purchase.

Database Schema:

Create tables for bundles, bundle items, and relationships between products and bundles.

```
CREATE TABLE bundles (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    description TEXT,
    discount_type ENUM('percentage', 'fixed_amount'),
    discount_value DECIMAL(10, 2)
);

CREATE TABLE bundle_items (
    id INT PRIMARY KEY,
    bundle_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (bundle_id) REFERENCES bundles(id),
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Business Logic Implementation:

Bundle Eligibility: Check if the user's cart items meet the bundle criteria.

Discount Application: Calculate and apply the discount based on the bundle rules.

API Development:

Develop RESTful endpoints for creating, updating, and retrieving bundles.

Implement cart and checkout endpoints to handle bundle operations.

User Interface Development:

Design UI components for bundle selection, cart display, and checkout.

Ensure the UI reflects bundle discounts and pricing accurately.

Comment Platform

Functional Requirements

Add Comment:

Users should be able to post comments on content items.

Comments can be added to any type of content (e.g., articles, products, posts).

Users can provide feedback or ask questions related to the content.

Delete Comment:

Users should be able to delete their own comments.

Moderators or administrators should be able to delete any comment, especially those violating guidelines.

Edit Comment:

Users should be able to edit their own comments within a certain time frame.

Edits should be logged or flagged to maintain comment integrity.

View Comments:

Users should be able to view comments on content items.

Comments should be displayed in a chronological or threaded format, depending on the design.

Reply to Comments:

Users should be able to reply to existing comments, creating a threaded discussion.

Replies should be visible under the original comment, maintaining a hierarchical structure.

Comment Liking and Reporting:

Users should be able to like or upvote comments they find helpful or interesting.

Users should be able to report comments that are inappropriate or violate guidelines.

Moderation:

Comments should be reviewed for inappropriate content or spam.

Administrators should have tools to manage, approve, or reject comments.

Notifications:

Users should receive notifications for replies to their comments or when their comment receives likes.
Notifications can be via email or in-app alerts.
Email Notifications
Sending emails to users to notify them about new comments, replies, or other relevant events.
Push Notifications
Sending notifications to mobile devices or web browsers.
In-App Notifications
Displaying notifications within the application interface itself, such as a notification bell icon or a message center.

Integration with User Profiles:
Each comment should be associated with the user's profile.
Users should be able to view their comment history.

Search and Filtering:
Users should be able to search for comments related to specific content.
Comments can be filtered based on various criteria such as most recent, most liked, or most relevant.

Comment Sorting and Pagination:
Improve the user experience by managing how comments are displayed.
Allow comments to be sorted by various criteria (e.g., most recent, most liked).
Redis
If you need real-time sorting or quick access to frequently updated comments based on likes, Redis can be used.
Use a Redis sorted set to store comments with their likes count.
Use Cases
You need very fast read and write operations with low latency.
You have a dataset that fits in memory or is a subset of a larger dataset.
You require real-time updates and access.
Elasticsearch
Elasticsearch is well-suited for handling and sorting large datasets due to its distributed nature and powerful querying capabilities.
Ensure that each comment document in Elasticsearch includes a likes field

```
{
  "id": "comment1",
  "content": "This is a comment",
  "likes": 123,
  "timestamp": "2024-07-29T12:00:00Z"
}
```

Use Cases
You are dealing with large volumes of data that exceed in-memory capacity.
You need advanced search capabilities, full-text search, and complex querying.
You require distributed storage and processing.
Update Latency
Updating data in Elasticsearch involves indexing overhead, which might introduce some latency compared to in-memory updates in systems like Redis.
For real-time performance, this could be a concern if updates are very frequent.

Implement pagination or infinite scrolling to handle large volumes of comments.
Ensure that sorting and pagination options are user-friendly and intuitive.

Comment Visibility and Privacy Settings:
Allow users to control the visibility of their comments.
Provide options for users to make their comments public, private, or visible to specific groups.

Implement privacy settings in the user's profile to manage comment visibility.

Discovering Hot Data in advance

Pre-Sale Indicators

Early Bird Promotions

Offer early bird promotions or pre-sale registrations to gauge interest in specific products.

Example

If a large number of users register for notifications or early access to a particular product, it is likely to be hot data.

User Behavior Analysis

Track user actions on your site, such as searches, product views, and wishlist additions.

Tools

Google Analytics, Mixpanel, or custom tracking solutions.

Example

Products with a high number of views or additions to wishlists are potential hot data.

Real-Time Monitoring

Use real-time data processing tools to monitor ongoing user activities and predict hot data.

Tools

Apache Kafka, Apache Flink, or AWS Kinesis.

Example

Monitor the number of views, clicks, and add-to-cart actions for products in real-time to identify potential hot data.

Apache Kafka

When a user performs an action such as viewing a page, clicking on something, or adding an item to the cart, you would send a message to notify that this action has occurred.

Historical Analysis

Analyze Past Sale Data

Review data from previous sales to identify patterns, such as frequently accessed products or high-traffic times.

Tools

Use data analytics tools like Apache Spark, Hadoop, or business intelligence tools like Tableau.

Example

If certain products sold out quickly in previous flash sales, they are likely to be hot data again.

leaderboard system

Ranking Calculation

Real-Time Calculation:

Incremental Updates:

Update rankings as new data comes in to avoid recalculating the entire leaderboard from scratch.

Batch Processing:

Periodically recalculate rankings in batches if real-time updates are not feasible.

Ranking Algorithms:

Simple Ranking:

Rank users based on a single metric, such as the number of steps.

Composite Ranking:

Rank users based on multiple metrics (e.g., steps, distance, calories burned).

Caching

In-Memory Caching:

Redis: Use Redis to cache leaderboard data for quick retrieval.

Local Caching: Use local caches (e.g., in-memory stores) on application servers to reduce database load.

Cache Invalidation:

Ensure that the cache is updated when user data changes or when rankings are recalculated.

Data Synchronization

Real-Time Data Streams:

Use message queues or streaming platforms (e.g., Kafka) to handle real-time data and updates.

Event-Driven Architecture:

Process events (e.g., user activity) and trigger updates to the leaderboard.

Implementation Steps

Database Design:

Define schema for user profiles, activity logs, and rankings.

Implement indexing for efficient querying.

Data Collection:

Implement APIs or data collection mechanisms to record user activities.

Ensure data integrity and accuracy.

Ranking Logic:

Implement ranking algorithms to calculate and update user positions.

Handle edge cases such as ties or updates to user data.

Caching Strategy:

Set up caching mechanisms to store and retrieve leaderboard data quickly.

Implement cache invalidation strategies to keep data up-to-date.

Real-Time Updates:

Set up data pipelines or event streams to handle real-time updates and recalculations.

User Interface:

Design and develop the frontend to display rankings effectively.

Optimize for performance and user experience.

Notification Service

Use WebSocket or Server-Sent Events (SSE) to push real-time updates to users.

Process Hot Data (Flash Sale System)

Cache

Content Delivery Network (CDN) Caching

Load product details and images into a CDN or in-memory cache before the sale starts.

Static product details and images can be pre-loaded into a CDN (Content Delivery Network) or an in-memory cache (e.g., Redis).

This ensures fast access and reduces load on the origin servers during the sale.

Cache Preloading (Cache Warming)

Pre-fetch data for products that are expected to be hot based on historical trends and real-time indicators.

In-Memory Caching

Cache product details, inventory counts, and user sessions to improve response times during peak load.

Upon receiving the request, the server assigns a unique identifier to identify this session.

This identifier is often stored in a cookie (session cookie) or appended to URLs (URL rewriting).

Business Design

Filter out bots

Add flash purchase question or captcha to filter robots.

Disable button before activity

Prevent users from clicking a button to request the server before the event starts.

Disable button after clicking

Prevent users from submitting repeatedly.

Queue display page

Delay processing user requests and display the current position of the user in the queue.

Reduce Inventory of goods

Order to reduce inventory

Subtract the quantity of the item purchased by the buyer from the item's inventory.

Problem:

Some people may buy in bulk and intentionally not pay.

But It is the best solution in the Flash Sale System because the users rarely fail to pay, and the sellers limit the inventory quantity of flash sale products.

Payment to reduce inventory

In a high-concurrency scenario, the buyer may not be able to pay because the item may have been purchased by someone else.

Problem:

Since the system only deducts the inventory quantity when the user places an order, the order quantity will be greater than the actual inventory quantity, resulting in the user being unable to place an order.

Pre-deduct the inventory

The inventory is held for buyers for a few minutes, and then released after the time is up, and other buyers can continue to purchase.

Before the buyer pays, the system will check whether the order's inventory is empty.

If the inventory is empty, payment will not be allowed.

If the inventory is not empty and the payment is successful, the payment will be completed and the inventory will be reduced.

Problem:

Some people may buy in bulk and intentionally not pay.

Even if the system releases the inventory, these people can still purchase the goods.

Solution:

- Mark buyers who frequently place orders but do not pay.

  Do not reduce inventory quantity when they place orders, and limit the maximum purchase quantity of specific items,

  Limit the number of repeated orders and non-payment.
- For some sellers who can replenish, we can replenish when the order quantity is greater than the inventory quantity.

  If replenishment is not allowed, we can only prompt that the inventory is insufficient.

Ensure that a monetary deduction occurs only once

Database Transactions

Use database transactions to ensure atomicity of operations.

Transactions allow you to group multiple operations into a single unit of work.

If one part of the transaction fails, the entire transaction can be rolled back.

Steps:

Start a Transaction:

Begin a transaction when starting the order and deduction process.

Place the Order and Deduct Money:

Execute both actions within the transaction.

Commit or Rollback:

Commit the transaction if both actions succeed. Rollback if any action fails to ensure no partial updates occur.

Example with Spring Boot:

```
@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;
```

```
    @Autowired
    private PaymentService paymentService;

    @Transactional
    public void placeOrderAndDeductMoney(Order order, double amount) {
        // Place the order
        orderRepository.save(order);

        // Deduct the money
        paymentService.deductAmount(order.getUserId(), amount);
    }
}
```

Idempotency

Ensure that the operations are idempotent, meaning that performing the operation multiple times will have the same effect as performing it once.

This can be achieved by using unique transaction identifiers.

Steps:

Generate a Unique Transaction ID:

Generate a unique ID for each transaction request.

Check for Existing Transactions:

Before performing the deduction, check if the transaction ID already exists in the database.

Perform Deduction:

Proceed with deduction only if the transaction ID is not found.

Record the Transaction ID:

Save the transaction ID to prevent duplicate deductions.

Example with Unique Transaction ID:

```
public void deductAmount(String userId, double amount, String transactionId) {
    if (transactionRepository.existsById(transactionId)) {
        // Transaction already processed
        return;
    }

    // Deduct money
    paymentProcessor.deduct(userId, amount);

    // Record the transaction ID
    transactionRepository.save(new Transaction(transactionId, userId, amount));
}
```

Distributed Locks

For distributed systems, you can use distributed locks to prevent multiple instances from performing the deduction concurrently.

Steps:

Acquire a Lock:

Before performing the deduction, acquire a distributed lock.

Perform Deduction:

Proceed with deduction if the lock is acquired.

Release the Lock:

Release the lock after the operation is complete.

Example with Redis Distributed Lock:

```
public void deductAmountWithLock(String userId, double amount, String transactionId) {
    RLock lock = redissonClient.getLock("payment-lock:" + userId);
    try {
        if (lock.tryLock(10, TimeUnit.SECONDS)) {
            // Perform deduction if lock is acquired
            deductAmount(userId, amount, transactionId);
        } else {
            // Handle lock acquisition failure
```

```
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }
```

Message Queues

For asynchronous processing, use message queues to ensure that operations are processed exactly once. Message queues can handle retries and ensure that messages are not processed more than once.

Steps:

Send a Message:

Send a message to the queue when placing an order.

Process the Message:

Process the message to perform the deduction.

Ensure Exactly-Once Processing:

Configure the message queue to ensure that each message is processed exactly once.

Example with RabbitMQ:

```
public void sendOrderMessage(Order order) {
    rabbitTemplate.convertAndSend("orderQueue", order);
}

public void processOrderMessage(Order order) {
    // Process the order and deduct money
}
```

Ride-Hailing System

Functional Requirements

User Registration and Authentication:

Users can sign up, log in, and log out.

Support for social login (Google, Facebook, etc.).

Driver Registration and Authentication:

Drivers can sign up, log in, and log out.

Verification process for drivers.

Ride Request and Matching:

Users can request a ride.

The system matches users with available drivers based on proximity and other factors.

Redis Geospatial Indexes

Order Aggregation Pool

After receiving a user's ride order, the system doesn't immediately send it to the dispatch engine. Instead, the order is placed into an "order aggregation pool."

Buckets in the Aggregation Pool

Within this pool, there are multiple "buckets." These buckets are used to temporarily hold and group orders together before sending them to the dispatch engine.

Writing to a Bucket:

- Time-Based Approach

  One way to consider a bucket as "complete" is to fill it with orders for a certain period.

  In this case, the system aggregates orders for a set time interval (e.g., 3 seconds) before considering the bucket full.

- Quantity-Based Approach

  Another way is to consider a bucket full once it reaches a certain number of orders.

- Chosen Method

  Udi chose the time-based approach, where a bucket is written (i.e., filled) every 3 seconds. After 3 seconds, the current bucket is considered complete, and a new bucket starts to collect incoming orders.
- Dispatch Engine Processing

  Once a bucket is considered full (after 3 seconds), all the orders within that bucket are pushed to the dispatch engine.

  The dispatch engine then processes these orders based on the principle of minimizing overall dispatch time.

Importance of the Bucket System

- Efficiency

  Aggregating orders in buckets allows the dispatch engine to process multiple orders at once, potentially improving efficiency and optimizing dispatch times.
- Geographic System Dependency

  The dispatch process depends on geographic system integration for route planning.

  This includes estimating arrival times, calculating fares, navigating routes, and handling order settlements and complaints.

Real-Time Tracking:

Real-time tracking of drivers and rides. (TCP long connection management system)

Notifications:

Real-time notifications for ride status, driver arrival, etc.

Payments:

Integration with payment gateways.

Support for multiple payment methods (credit/debit cards, wallets, etc.).

Rating and Feedback:

Users and drivers can rate each other and provide feedback.

Support and Help:

In-app customer support and helpdesk.

Scan QR Code to log in

Requirements

Functional Requirements:

Generate QR Code

The system should generate a QR code containing a unique login token or URL that users can scan.

Scan QR Code

The user scans the QR code using a mobile app or a web camera.

Authenticate User

After scanning, the system should authenticate the user and establish a session.

Handle Expiration

The QR code should have an expiration time to ensure security.

Display User Information

Once authenticated, the user should be redirected to a logged-in state or page.

Non-Functional Requirements:

Security

Ensure the QR code and login process are secure to prevent unauthorized access.

Performance

The scanning and authentication process should be quick and responsive.

User Experience

Provide clear feedback and instructions to users during the scanning and login process.

Database Structure

Authentication Tokens Table

Store unique tokens generated for login sessions.

Columns:

- id: Unique identifier for the token record.
- token: The unique token value used in the QR code.
- user_id: Foreign key linking to the user trying to log in.
- created_at: Timestamp when the token was created.
- expires_at: Timestamp when the token expires.

Backend Request Design

Generate QR Code

Endpoint: POST /api/auth/generate-qr

Purpose: Generate a QR code with a unique login token.

Request Parameters:

user_id: The ID of the user requesting login.

Response:

token: The unique token associated with the QR code.

qr_code_url: URL where the QR code image can be retrieved.

Backend Logic:

Generate a unique token.

Save the token to the AuthenticationTokens table with an expiration time.

Generate a QR code containing the token and provide the QR code URL.

Controller Method:

```
@PostMapping("/api/auth/generate-qr")
public ResponseEntity<Map<String, String>> generateQrCode(@RequestParam Long userId) {
    String token = UUID.randomUUID().toString();
    LocalDateTime expiresAt = LocalDateTime.now().plusMinutes(5); // 5 minutes expiration
    authenticationTokensRepository.save(new AuthenticationToken(token, userId, expiresAt));

    String qrCodeUrl = "https://example.com/login?token=" + token;
    Map<String, String> response = new HashMap<>();
    response.put("token", token);
    response.put("qr_code_url", qrCodeUrl);

    return ResponseEntity.ok(response);
}
```

Check Status

Endpoint: GET /api/auth/check-status

Purpose: Periodically check if the user has been authenticated.

Request Parameters:

token: The unique token associated with the QR code.

Response:

status: Authentication status (authenticated or pending).

redirect_url: URL to redirect the user upon successful login (if authenticated).

Backend Logic:

Check the authentication_tokens table to see if the token has been used and the user is authenticated.

Return the authentication status and redirect URL if authenticated.

Controller Method:

```
@GetMapping("/api/auth/check-status")
public ResponseEntity<Map<String, Object>> checkStatus(@RequestParam String token) {
    AuthenticationToken authToken = authenticationTokensRepository.findByToken(token);

    Map<String, Object> response = new HashMap<>();
```

```
        if (authToken != null && authToken.isUsed()) {
            response.put("status", "authenticated");
            response.put("redirect_url", "/dashboard"); // Example redirect URL after successful login
        } else {
            response.put("status", "pending");
        }

        return ResponseEntity.ok(response);
    }
```

Scan QR Code

Endpoint: POST /api/auth/scan-qr

Purpose: Process the scanned QR code and authenticate the user.

Request Body:

token: The token retrieved from the scanned QR code.

Response:

status: Success or failure of the authentication process.

user_id: ID of the authenticated user (if successful).

redirect_url: URL to redirect the user upon successful login. ()

Backend Logic:

Verify the token against the AuthenticationTokens table.

Check if the token is valid and not expired.

Authenticate the user and create a session.

Return a success response and provide a redirect URL for the user.

Controller Method

```
@PostMapping("/api/auth/scan-qr")
public ResponseEntity<Map<String, Object>> scanQrCode(@RequestBody Map<String, String> request) {
    String token = request.get("token");
    AuthenticationToken authToken = authenticationTokensRepository.findByToken(token);

    if (authToken == null || authToken.isUsed() ||
authToken.getExpiresAt().isBefore(LocalDateTime.now())) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(null);
    }

    authToken.setUsed(true);
    authenticationTokensRepository.save(authToken);

    // Authenticate user (create session)
    Long userId = authToken.getUserId();
    String redirectUrl = "/dashboard"; // Example redirect URL after successful login

    Map<String, Object> response = new HashMap<>();
    response.put("status", "success");
    response.put("user_id", userId);
    response.put("redirect_url", redirectUrl);

    return ResponseEntity.ok(response);
}
```

Shopping Cart

Functional Requirements

Add Items to Cart

Users should be able to add items to their shopping cart. Each item should include product details and quantity.

Remove Items from Cart

Users should be able to remove items from their cart.

Update Item Quantity

Users should be able to change the quantity of an item in their cart.

View Cart

Users should be able to view the current contents of their shopping cart, including item details and the total price.

Persist Cart

The cart should be saved and retrieved across user sessions. This could be for logged-in users or persistent anonymous carts.

Cache the shopping cart to the browser local storage when the user is not logged in, and sync this information to the server when the user logs in.

Database Structure

Users Table

Store user information.

Columns:

id:	Unique identifier for each user.
username:	User's login name.
password:	User's password (hashed).
email:	User's email address.
created_at:	Timestamp when the account was created.

Products Table

Store product details.

Columns:

id:	Unique identifier for each product.
name:	Name of the product.
description:	Description of the product.
price:	Price of the product.
stock_quantity:	Available quantity in stock.

ShoppingCarts Table

Represent shopping carts, which can be associated with users or anonymous.

Columns:

id:	Unique identifier for each cart.
user_id:	Foreign key referencing the Users table (nullable for anonymous carts).
created_at:	Timestamp when the cart was created.

CartItems Table

Purpose: Store items in a shopping cart.

Columns:

id:	Unique identifier for each cart item.
cart_id:	Foreign key referencing the ShoppingCarts table.
product_id:	Foreign key referencing the Products table.
quantity:	Number of units of the product in the cart.

Backend Request Design

Add Item to Cart

Endpoint: POST /api/carts/{cartId}/items

Purpose: Add a specific product to the shopping cart.

Request Parameters:

cartId:	The identifier of the cart to which the item will be added.

Request Body:

productId:	The identifier of the product being added.
quantity:	The number of units of the product.

Response:

Success or failure message with appropriate HTTP status codes.

Remove Item from Cart

Endpoint: DELETE /api/carts/{cartId}/items/{itemId}

Purpose: Remove a specific item from the shopping cart.

Request Parameters:

cartId: The identifier of the cart.

itemId: The identifier of the item to be removed.

Response:

Success or failure message with appropriate HTTP status codes.

Update Item Quantity

Endpoint: PUT /api/carts/{cartId}/items/{itemId}

Purpose: Update the quantity of a specific item in the cart.

Request Parameters:

cartId: The identifier of the cart. (In RESTful API design, the request body of an update operation typically contains the parameters that need to be modified.)

itemId: The identifier of the item.

Request Body:

quantity: The new quantity of the item.

Response:

Success or failure message with appropriate HTTP status codes.

View Cart

Endpoint: GET /api/carts/{cartId}

Purpose: Retrieve the details of a shopping cart, including items and total price.

Request Parameters:

cartId: The identifier of the cart to be viewed.

Response:

Cart details including item information and total price.

Short URL Generator

Status Code: 302

URL generation algorithm

java.security.MessageDigest

Use a hash algorithm such as MD5 or SHA256 to perform hash calculation on the long URL to obtain a 128-bit or 256-bit hash value.

Then perform Base64 encoding on the hash value to obtain 22 or 43 Base64 characters, and then intercept the first 6 characters.

Disadvantages:

Conflict handling requires multiple lookups of the URL to the store, which can degrade performance.

Auto-incrementing short URL

Maintain a self-increasing binary natural number, then Base64 encode the natural number to obtain a series of short URLs.

Disadvantages:

It's predictable

Pre-generated short URLs

Generate a batch of non-conflicting short URL strings in advance.

The algorithm for pre-generating short URLs can be implemented using random numbers.

Each of the 6 characters is generated using a random number (a Base64 encoded character is generated using a random

number from 0 to 63).

When an external request inputs a long URL and needs to generate a short URL, just get one from the pre-generated short URL string pool (using Bloom filter detection).
Because the pre-generated short URL is offline, there will be no performance issues at this time.

Cache

It is generally believed that more than 80% of requests are concentrated on short URLs generated in the last 6 days, so it is sufficient to cache the short URLs generated in the last 6 days.
The number of short URLs generated in the last 6 days is about 100 million, so the memory space required for the Redis cache server is: 100000000*1kb=100GB

Offset

In addition to the file that records the pre-generated short URL in HDFS (Hadoop Distributed File System), a file that records the offset is also required.
The file access process of the application should be: write open offset file -> read offset -> read open short URL file -> read 60K data from offset -> close short URL file -> modify offset file -> close offset file.

Because multiple servers may load the same short URL at the same time,
it is also necessary to use the offset file to perform mutual exclusion operations on multiple servers,
that is, to use the mutual exclusion of the file system write operation lock to achieve multi-server access mutual exclusion.
linked list:

The 10,000 short URLs loaded into the preloaded short URL server are stored in a linked list.
Each time a short URL is used, the head pointer of the linked list moves back one position and sets the next object of the previous linked list element to null.
In this way, the used short URL objects can be garbage collected. When the remaining linked list length is less than 2000,
an asynchronous thread is triggered to load 10,000 new short URLs from the file and link them to the end of the linked list.

Single Sign-On

Single Sign-On (SSO) is an authentication process that allows a user to access multiple applications or services with a single set of login credentials.

How Single Sign-On Works

Initial Authentication:

User Login

The user logs in to an SSO provider or an identity provider (IdP) using their credentials (username and password).

Authentication Token

Upon successful authentication, the SSO provider issues an authentication token or session.

Accessing Applications:

Application Request

When the user tries to access an application or service (relying party or service provider), the application requests authentication from the SSO provider.

Token Validation

The application redirects the user to the SSO provider, which verifies the user's identity and issues an authentication token or ticket.

Token Exchange

The application receives the token, verifies it, and grants access to the user without requiring them to log in again.

Session Management:

- Session Duration
  - The user remains logged in to all associated applications as long as the session is valid.
- Session Expiry
  - If the session expires or the user logs out, they are logged out of all applications that use the same SSO provider.

Steps to Implement SSO with Spring Cloud Components

Set Up Authorization Server:

- Use an existing Identity Provider (IdP) like Keycloak, Okta, or Auth0, or implement your own using Spring Authorization Server.
- Configure the authorization server to issue tokens and manage user authentication.

Configure the Authorization Server:

- Define clients (your microservices) that will use the authorization server for authentication.
- Set up the required scopes and grant types.

Set Up Resource Servers (Microservices):

- Each microservice acts as a resource server that validates tokens issued by the authorization server.

Configure Spring Security in Microservices:

- Use spring-boot-starter-oauth2-resource-server to set up resource servers.
- Configure OAuth 2.0 properties in application.yml or application.properties.

Stock Keeping Units

To implement a feature where logged-in users can store up to 1000 SKUs (Stock Keeping Units) while non-logged-in users can store up to 200 SKUs,
you can use a combination of user authentication, session management, and storage limits.

Design the System

- **Define Storage Limits**:
  - **Logged-In Users**: Can store up to 1000 SKUs.
  - **Non-Logged-In Users**: Can store up to 200 SKUs.
- **Identify User States**:
  - **Authenticated**: User is logged in and has an active session.
  - **Unauthenticated**: User is not logged in and is using the application as a guest.

Design Storage Management

- **Data Structure**:
  - **For Logged-In Users**: Use a data structure that allows storing up to 1000 SKUs, such as a List or Set with a size limit.
  - **For Non-Logged-In Users**: Use a similar data structure with a limit of 200 SKUs.
- **Implementation**:
  - **For Logged-In Users**:
    - Check if the user is authenticated.
    - Implement logic to store and manage up to 1000 SKUs.
  - **For Non-Logged-In Users**:
    - If the user is not authenticated, use session or temporary storage to manage up to 200 SKUs.

DAM / SQL

Data Definition Language (DDL)

CREATE TABLE

```
CREATE TABLE employees (
```

```
    id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    hire_date DATE
);
```

DROP TABLE

```
DROP TABLE employees;
```

ALTER TABLE

```
ALTER TABLE employees ADD COLUMN phone_number VARCHAR(15);
```

CREATE INDEX

```
CREATE INDEX idx_last_name ON employees (last_name);
```

Data Query Language (DQL)

WITH

The WITH clause (Common Table Expressions or CTEs) is used to create a temporary result set that can be referenced within the same query.
It is primarily used for improving query readability and avoiding repeated subqueries.

Example:

```
    WITH ValidIDs AS (
        SELECT ID
        FROM TEMP
    )
```

```
    WITH category AS (
        SELECT 'Low Salary' AS category
        UNION
        SELECT 'Average Salary'
        UNION
        SELECT 'High Salary'
    ),
    cnt AS (
        SELECT
            CASE
                WHEN income < 20000 THEN 'Low Salary'
                WHEN income BETWEEN 20000 AND 50000 THEN 'Average Salary'
                WHEN income > 50000 THEN 'High Salary'
            END AS category,
            COUNT(*) AS accounts_count
        FROM accounts
        GROUP BY category
    )
```

> MySQL does not support multiple CTEs (WITH clauses) without using a comma to separate them.
> In MySQL, multiple CTEs must be declared in a single WITH clause, separated by commas.

Data Manipulation Language (DML)

SELECT

SELECT age || '-' || name as result AS n FROM person p;

SELECT DISTINCT stu.name AS sname stu, IFNULL(age, 0)+10 FROM student stu Normal Query (DISTINCT is used for deduplication)

age || '-' || name Splicing symbol '| |'

SYSDATE + INTERVAL '1' DAY Add or subtract time (DAY | HOUR | MINUTE | SECOND).

`<field-or-table-name>` Reverse quotation marks prevent keyword conflicts or Chinese names

WHERE id = 9 and id !=9 and id > 9 and id < 9 and id >=9 and id<=9

WHERE name NOT 'abc%'; Negative condition

WHERE name IS NOT NULL Non empty condition

WHERE name LIKE 'abc%'; Fuzzy conditions (like queries starting with%, such as'% abc ', cannot use indexes, like queries starting with%, such as' abc%', are equivalent to range queries and will use indexes)

WHERE id NOT IN (1,2,3) Range condition

// (Employee.DepartmentId , Salary) IN (SELECT DepartmentId, MAX(Salary) FROM Employee GROUP BY DepartmentId) Use multiple fields to query

WHERE price BETWEEN 30 AND 60; Range condition

ORDER BY piece DESC, sort, createtime ASC Sort, when multiple conditions are met, only return the partial data that satisfies both conditions simultaneously

LIMIT 2 OFFSET 3 Skip 3 items, take 2 items==>[3, 5] (Limit cannot be used in Oracle)

LIMIT 2, 3 Skip 2 items, take 3 items==>[3, 6] (Limit cannot be used in Oracle)

// LiMIT 3 Only take three items

// LIMIT 2 -1 Skip 2 items and take the remaining records.

GROUP BY

SELECT MIN(stock) FROM product;

Aggregate query (taking a column of data as a whole and performing vertical calculations)

SELECT brand, SUM(price) FROM product WHERE price > 4000 GROUP BY brand;

Group query by field brand (displaying the total price based on brand grouping)

SELECT brand, SUM(price) getSum FROM product WHERE price > 4000 GROUP BY brand HAVING getSum >7000;

Group query by field brand (display the total price based on brand grouping, and only display the total price greater than 7000)

WHERE is used to filter individual rows based on specific criteria before grouping.

HAVING will filter the content of each group separately after grouping.

you cannot use columns in the HAVING clause that are not part of the aggregated result.

```
SELECT COUNT(DISTINCT user_id) AS user_cnt FROM Purchases WHERE time_stamp BETWEEN startDate AND endDate GROUP BY user_id HAVING SUM(amount) > minAmount
```

Field Aliases

Because the GROUP BY clause is processed before the SELECT clause, the alias defined in the SELECT clause isn't yet available for grouping.

SQL Order

FROM The table or tables from which to retrieve data.

JOIN Combine data from multiple tables based on conditions.

ON Specify the join condition.

WHERE	Filter rows based on specified conditions.
GROUP BY	Group rows sharing a property so aggregate functions can be applied.
HAVING	Filter groups based on aggregate conditions.
SELECT	Choose the columns or expressions to return.
DISTINCT	Remove duplicate rows from the result set.
ORDER BY	Sort the result set based on specified columns.
LIMIT/OFFSET	Limit the number of returned rows or skip rows.

No Matching Rows

If no rows meet the WHERE clause criteria (time_stamp BETWEEN startDate AND endDate), then the subquery will yield no results(SUM funciton used by GROUP BY).

This would cause the RETURN statement to output NULL.

INNER JOIN

SELECT * FROM student u

JOIN person p ON u.pid = person.id;

Merge columns from two different tables using one or more common columns.

After any JOIN statement, the tables have already been merged, but a WHERE filter can be added to remove data that does not meet the WHERE conditions between the merged tables.

[INNER] JOIN person p ON u.pid = person.id;

Explicit internal connection, returns the left and right tables that meet the criteria (special where)

(WHERE can be used behind JOIN directly)

(If each join clause matches multiple left or right table data, the number of returned results will increase.)

(For HANA database, The same table cannot be joined multiple times, but it is allowed if it is renamed.)

If you select 2 records from the right table, and there is no association between the two tables in the on clause, you will get twice the number of records from the left table;

```
John    P1
Alice       P1
John    P2
Alice       P2
```

LEFT [OUTER] JOIN person p ON u.pid = person.id;

Left external connection, returns all left tables and right tables that meet the criteria

(If there are no records in the right table that meet the criteria, the fields in the right table will be displayed as NULL)

(WHERE can be used behind JOIN directly)

RIGHT [OUTER] JOIN person p ON u.pid = person.id;

Right external connection, returns all right tables and left tables that meet the criteria

(If there are no records in the left table that meet the criteria, the fields in the left table will be displayed as NULL)

(WHERE can be used behind JOIN directly)

Integration

SELECT stu1.name, stu2.name FROM student stu1 LEFT JOIN student stu2 ON stu1.mgr=stu2.id;

Self association query (data in the same table has correlation)

SELECT * FROM student u, person p WHERE u.name=p.name; Implicit internal connection

UNION

SELECT * FROM person UNION SELECT age FROM person

After merging, duplicate elements will be removed (the resulting set will be sorted, duplicate records will be deleted, and the result will be returned)

SELECT * FROM person UNION ALL SELECT age FROM person

Merge (unsorted)

Nested

```
SELECT stu.name, per.number FROM student stu, (SELECT * FROM person WHERE id>4 ) per
    WHERE stu.pid=per.id;
SELECT name, age FROM student
    WHERE age = (SELECT MAX(age) FROM student);
SELECT * FROM student
    WHERE pid IN (SELECT id FROM person WHERE name IN ('stu1','stu2'));
```

PARTITION BY

```
SELECT <window_function>() OVER (PARTITION BY <partition_expression> ORDER BY <order_expression>)
```

RANK()

The window function applied over the window of rows (e.g., ROW_NUMBER(), RANK(), SUM(), etc.).

ROW_NUMBER()

Assigns a sequential number to each row.

RANK()

Assigns a rank to each row, allowing ties with gaps in ranks.

DENSE_RANK()

Assigns a rank like RANK(), but without gaps for ties.

PARTITION BY

Optionally divides the result set into partitions (similar to grouping). The function is applied to each partition.

ORDER BY

Optionally orders rows within each partition for the calculation of window functions.

Example:

```
SELECT department, employee_name, COUNT(*) OVER (PARTITION BY department) AS total_employees
FROM employees;
```

Table `employees`:

department	employee_name
HR	Alice
HR	Bob
HR	Charlie
IT	David
Sales	Hannah
Sales	Ian

Output:

department	employee_name	total_employees
HR	Alice	3
HR	Bob	3
HR	Charlie	3
IT	David	1
Sales	Hannah	2
Sales	Ian	2

```
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

Output:

department	total_employees
HR	3

IT 1
Sales 2

Compatibility

CONSTANT

Mysql

SELECT 'Average Salary', ...
SELECT 'Low Salary' AS category

Oracle

SELECT 'Average Salary' from DUAL
SELECT 'Low Salary' AS category, ...

GROUP BY 1

Oracle does not support GROUP BY 1

```
GROUP BY
    CASE
        WHEN income < 20000 THEN 'Low Salary'
        WHEN income BETWEEN 20000 AND 50000 THEN 'Average Salary'
        WHEN income > 50000 THEN 'High Salary'
    END
```

INSERT INTO

INSERT INTO person (age, name) VALUES(18, '张三'), (20, '李四'); 增（不指定字段默认为一行的顺序，null 为使用自动生成的 id，兼容写法 value 或 values）

INSERT INTO person VALUES(null, 18), (null, 20);

ON DUPLICATE KEY UPDATE msyql 语法，同 REPLACE INTO，根据唯一键删除

REPLACE INTO person (age, name) VALUES (18, '王五'); msyql 语法。根据 唯一键 删除原有的一条记录， 并且插入一条新的记录来替换原记录。

如果插入列没有唯一键 或 插入的数据唯一键没有重复的，那么会新增一条记录。
// REPLACE INTO test(age,name) VALUES ('138', '王五');
此时会根据唯一键 name 更新上一条数据

UPDATE

UPDATE table_name
SET column1 = value1,
column2 = value2,
column3 = value3

CASE WHEN

In this example, the two UPDATE statements are part of a single transaction.
If one of the updates fails, the entire transaction can be rolled back, ensuring that either both updates occur, or neither does. This ensures atomicity.

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE account_id = 2;
```

```
COMMIT;
```

The CASE statement is used to add conditional logic to SQL queries. It allows you to perform different actions based on different conditions.

In this example, the `CASE` statement is used to categorize employees' salaries into 'Low', 'Medium', and 'High' ranges.

```
SELECT
    employee_id,
    first_name,
    last_name,
    CASE
        WHEN salary < 3000 THEN 'Low'
        WHEN salary BETWEEN 3000 AND 7000 THEN 'Medium'
        ELSE 'High'
    END AS salary_range
FROM employees;
```

In this example, the CASE statement is used in the WHERE clause to filter orders based on their status and shipped date.

```
SELECT
    order_id,
    customer_id,
    order_date,
    total_amount
FROM orders
WHERE
    CASE
        WHEN status = 'Completed' THEN shipped_date IS NOT NULL
        WHEN status = 'Pending' THEN shipped_date IS NULL
        ELSE TRUE
    END;
```

DELETE FROM

DELETE p FROM person p, user u;

The DELETE operation is transactional, meaning it can be rolled back if performed within a transaction, ensuring data integrity in case of errors.

UNION ALL

The UNION ALL operator in SQL is used to combine the results of two or more SELECT statements into a single result set. Unlike UNION, which eliminates duplicate rows, UNION ALL includes all rows, even if they are duplicates.

Example1:

```
SELECT id, name, role FROM employees
UNION ALL
SELECT id, name, role FROM contractors;
```

Data Control Language (DCL)

GRANT

Provides users with access privileges.

```
-- Grant SELECT and INSERT privileges to a user
GRANT SELECT, INSERT ON employees TO user_name;
```

REVOKE

Removes access privileges from users.

```
-- Revoke INSERT privilege from a user
REVOKE INSERT ON employees FROM user_name;
```

Transaction Control Language (TCL)

COMMIT

Saves all changes made during the current transaction permanently to the database.

```
-- Begin a transaction
BEGIN TRANSACTION;

-- Update data
UPDATE employees SET salary = salary + 1000 WHERE department = 'IT';
DELETE FROM employees WHERE status = 'inactive';

-- Commit changes
COMMIT;
```

-- Begin a transaction

BEGIN;

ROLLBACK

Undoes changes made in the current transaction back to the last COMMIT, SAVEPOINT, or the beginning of the transaction.

```
-- Begin a transaction
BEGIN TRANSACTION;

-- Update data
UPDATE employees SET salary = salary + 1000 WHERE department = 'IT';
DELETE FROM employees WHERE status = 'inactive';

-- Rollback changes (undo all changes made in this transaction)
ROLLBACK;
```

SAVEPOINT

Sets a named point within a transaction to which you can later roll back.

```
-- Begin a transaction
BEGIN TRANSACTION;

-- Update data
UPDATE employees SET salary = salary + 1000 WHERE department = 'IT';

-- Set a savepoint
SAVEPOINT before_delete;

-- Delete data
DELETE FROM employees WHERE status = 'inactive';

-- Rollback to the savepoint (undo only the delete operation)
ROLLBACK TO SAVEPOINT before_delete;

-- Commit or rollback the transaction
COMMIT;
```

Operators

Arithmetic Operators

SELECT salary + bonus AS total_income FROM employees;

SELECT salary - deductions AS net_salary FROM employees;

SELECT salary * 1.1 AS adjusted_salary FROM employees;

SELECT salary / 2 AS half_salary FROM employees;

Equality and Inequality

SELECT * FROM employees WHERE referee_id = 2;

SELECT * FROM employees WHERE referee_id != 2;

IS NULL / IS NOT NULL

```
SELECT * FROM employees WHERE referee_id IS NULL;
SELECT * FROM employees WHERE referee_id IS NOT NULL;
```

Logical Operators

```
SELECT * FROM employees WHERE referee_id = 2 AND hire_date > '2020-01-01';
SELECT * FROM employees WHERE referee_id = 2 OR referee_id IS NULL;
SELECT * FROM employees WHERE NOT (referee_id = 2);
```

Comparison Operators

```
SELECT * FROM employees WHERE hire_date > '2020-01-01';
SELECT * FROM employees WHERE hire_date >= '2020-01-01';
SELECT * FROM employees WHERE hire_date < '2020-01-01';
SELECT * FROM employees WHERE hire_date <= '2020-01-01';
SELECT * FROM employees WHERE hire_date BETWEEN '2020-01-01' AND '2021-01-01';
SELECT * FROM employees WHERE referee_id IN (1, 2, 3);
SELECT * FROM employees WHERE first_name LIKE 'John%';
SELECT * FROM employees WHERE first_name NOT LIKE 'John%';
```

String and Pattern Matching

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
SELECT TRIM(first_name) FROM employees;
```

Aggregation Operators

```
SELECT COUNT(*) FROM employees;
SELECT SUM(salary) FROM employees;
SELECT AVG(salary) FROM employees;
SELECT MAX(salary) FROM employees;
SELECT MIN(salary) FROM employees;
```

mysql

```
10 div 3        Take the divisible number
10 mod 3        Take remainder
```

Functions

```
COALESCE(expression1, expression2, ..., expressionN)
```

The COALESCE function in SQL returns the first non-NULL value from a list of expressions.

It is useful when you need to handle NULL values and provide a default or alternative value.

Oracle

```
NVL(expression, replacement_value)
```

The NVL function is used to replace NULL values with a specified replacement value.

If the first argument is NULL, it returns the second argument. Otherwise, it returns the first argument.

```
CONCAT(s1, s2)                          Splicing strings
TO_DATE('2022-11-27', 'yyyy.mm.dd')     Convert string to date
TO_CHAR (ISRT_DT, 'yyyy.mm.dd')         Convert date to string
UPPER ('str')                   Capitalize string
LENGTH('str')                   Get the string length
ROUND(123,456, 2)               Round to 2 decimal places.
MAX (AGE)                       Calculate the maximum value.
```

MIN (AGE) Calculate the minimum value.
FLOOR(AGE) Round a number down to the nearest integer

LISTAGG(ID, ',') Concatenates the query results using a comma.
SUBSTR ('xxx', 1, 2) Extracts a substring, specifying the starting position and length (can be used in WHERE conditions).
BITAND(x, y)

The BITAND function performs a bitwise AND operation between two integer values.

Mysql

Number

COUNT(1)
MAX(price)
MIN(price)
SUM(price)
AVG(price)
FLOOR(2.33)
CEIL(4.2)

This function returns the smallest integer that is greater than or equal to a given number. It rounds numbers up to the next whole number.

```
SELECT CEIL(4.2);    The output will be 5, as it rounds 4.2 up to the nearest integer.
```

ROUND(12345678.90, 2)
IFNULL(sum(price), 0)
IF(T.STATUS = 'completed', 1, 0)
REPLACE(name,'detail','description')

Replace string with another string

```
update table set name=replace(name,'detail','description')
```

TRUNCATE(12345678.90, 2)

Remove numbers after 2 decimal places directly

FORMAT(12345678.90, 2)
SIGN(255.5)

Return the symbol of the number (if the number is greater than 0, return 1; if it is equal to 0, return 0; if it is less than zero, return -1)

String

CHAR_LENGTH("str")

Get the string length

SUBSTRING("str", 2,3)

Extract a string of length 3 from index 2

GROUP_CONCAT(price)

Connect the query results with commas

```
GROUP_CONCAT(table_name SEPARATOR '&')   Specify delimiter
```

Date

YEAR(date)

The YEAR() function extracts the year from a given date or datetime expression.

The BETWEEN operator is useful for checking if a date falls within a specific range:

```
SELECT YEAR('2024-10-11');            2024
```

```
SELECT * FROM orders WHERE order_date BETWEEN '2024-10-01' AND '2024-10-14';
```

MONTH(date)

The MONTH() function extracts the month from a given date or datetime expression.

```
SELECT MONTH('2024-10-11');            10
```

DAY(date)

The DAY() function in MySQL is used to extract the day of the month from a date.

```
SELECT DAY('2024-10-12')               12
```

WEEK(date, mode)

The WEEK() function in MySQL is used to get the week number from a date value.

```
SELECT WEEK('2024-10-12')              41
```

date

This is the date from which you want to extract the week number. It can be a date, datetime, or timestamp value.

mode

This is an optional parameter that specifies the mode for calculating the week number.

The mode can affect which day is considered the start of the week and how to determine the first week of the year.

If omitted, the default mode is 0.

0 Week starts on Sunday, and the first week of the year is the week with January 1 (default).

1 Week starts on Monday, and the first week of the year is the week with January 1.

2 Week starts on Sunday, and the first week of the year is the week with the first Sunday in January.

3 Week starts on Monday, and the first week of the year is the week with the first Monday in January.

4 Week starts on Sunday, and the first week of the year is the week with the first day of the year.

5 Week starts on Monday, and the first week of the year is the week with the first day of the year.

6 Week starts on Sunday, and the first week of the year is the week with the first Thursday in January (ISO 8601).

7 Week starts on Monday, and the first week of the year is the week with the first Thursday in January (ISO 8601).

DATE_ADD(date, INTERVAL value unit)

This function adds a specified time interval to a date. It allows you to manipulate dates by adding days, months, years, or other time intervals.

```
SELECT DATE_ADD('2023-11-01', INTERVAL 10 DAY);        This will return '2023-11-11'.
```

DAYOFWEEK(date)

This function returns the weekday index of a given date, where Sunday is 1 and Saturday is 7. It helps identify the day of the week for date calculations.

```
SELECT DAYOFWEEK('2023-11-01');       If November 1st, 2023 is a Wednesday, the function will return
4.
```

CURDATE()

DATE_FORMAT(create_time,'%Y-%m-%d %H:%i:%s')

%Y	Year, four digits
%y	Year, two digits
%m	Month, two digits
%c	Month, without leading zeros
%d	What day of the month, two digits
%e	What day of the month, without leading zeros
%H	Hour, 24-hour format, two digits
%h	hour, 12-hour format, two digits
%i	minute, two digits
%s	seconds, two digits
%P	AM or PM

```
SELECT * FROM orders WHERE DATE_FORMAT(order_date, '%Y-%m') = '2024-10';
If you need to compare dates based on a specific format, you can use DATE_FORMAT:
```

Functional

COALESCE(value1, value2, ..., valueN)

The COALESCE() function in MySQL returns the first non-null value from a list of arguments.

If all the arguments are NULL, it returns NULL.

It is useful for handling null values and providing a fallback or default value.

```
SELECT COALESCE(NULL, 'default_value', 'another_value');       This will return 'default_value', as it is the first non-null value in the list.
```

BIN_TO_UUID(binary_uuid [, swap_flag])

convert a binary UUID stored as a BINARY(16) data type into its standard textual representation.

This function is particularly useful when dealing with UUIDs in a binary format for storage efficiency and you need to retrieve them in a human-readable form.

The BIN_TO_UUID() function itself does not provide ordering. It simply converts the binary representation to a textual format.

```
BIN_TO_UUID(binary_uuid [, swap_flag])
```

Parameters:

binary_uuid: The binary UUID value that you want to convert.

swap_flag (optional): A boolean value that, if set to true, swaps the time and node parts of the UUID. This is useful if the UUIDs were stored using the UUID_TO_BIN function with the swap_flag set to true.

FIND_IN_SET('b', 'a,b,c,d')

Find 'b' in the set 'a,b,c,d', return 0 or 1.

Statement

CASE WHEN

CASE score WHEN 'A' THEN 'Excellent' WHEN 'B' THEN 'Poor' ELSE 'Fail' END

switch statement

CASE WHEN score>60 THEN ''Excellent' ' WHEN score<60 THEN ''Poor' ' ELSE ''Fail' ' END

if statement

Example:

```
WITH salary_categories AS (
    SELECT
        CASE
            WHEN salary < 20000 THEN 'Low Salary'
            WHEN salary BETWEEN 20000 AND 50000 THEN 'Average Salary'
            ELSE 'High Salary'
        END AS category
    FROM accounts
)
```

The CTE, Common Table Expression (salary_categories) assigns a category label (Low Salary, Average Salary, or High Salary) to each salary in the accounts table.

WITH

Create a temporary table to store the result set

```
WITH temporary_table_name (column1, column2, ...) AS (
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition
)
SELECT * FROM temporary_table_name;
```

WITH ... AS

The WITH ... AS clause defines a non-recursive CTE, which is a temporary result set that is calculated once and can be referred to multiple times in a query.

```
WITH sales_summary AS (
  SELECT customer_id, SUM(amount) AS total_sales
  FROM sales
  WHERE sale_date >= '2024-01-01'
  GROUP BY customer_id
)
SELECT * FROM sales_summary
WHERE total_sales > 1000;
```

In this example, sales_summary is a CTE that calculates the total sales for each customer from January 1, 2024. The main query retrieves customers whose total sales exceed 1,000.

WITH RECURSIVE ... AS

This operator is used to define a Common Table Expression (CTE) that can refer to itself.
It allows for recursive queries, which are useful for tasks like generating sequential dates or traversing hierarchical data.
In a recursive Common Table Expression (CTE) like the one shown, the UNION ALL is necessary to combine the results from two different parts of the CTE:
Anchor member:

The initial query that provides the starting point for the recursion.

`SELECT 1 AS number`

Recursive member:

The part that references the CTE itself to generate additional rows. In this example:

`SELECT number + 1 FROM sequence WHERE number < 10`

This generates a sequence of numbers from 1 to 10.

```
WITH RECURSIVE sequence AS (
    SELECT 1 AS number
    UNION ALL
    SELECT number + 1 FROM sequence WHERE number < 10
)
SELECT number FROM sequence;
```

INTERVAL ... DAY

This operator is used to specify a time interval, typically in conjunction with date functions like DATE_ADD.
The interval specifies the unit (such as DAY, MONTH, YEAR) and the amount to add or subtract from a date.

```
SELECT DATE_ADD('2023-11-01', INTERVAL 7 DAY);
```

This adds 7 days to the date '2023-11-01', resulting in '2023-11-08'.

PARTITION BY

Order

The grouping and sorting in over() are executed later than where and group by, but not later than order by.

GROUP BY and PARTION BY

PARTION BY

PARTION BY does not change the number of rows in the result set,
but adds calculation columns to the original result set for calculating aggregate values or rankings within a specified window.

GROUP BY

GROUP BY will change the number of rows in the result set, returning only one row per group.

It is suitable for grouping and aggregating query results and is commonly used in statistical analysis, report generation, and other scenarios.

Window Function

ROW_NUMBER()

Assigns a unique, sequential number to each row within a partition of the result set.

RANK()

Assigns a unique rank to each row within a partition of the result set, with the possibility of assigning the same rank to rows that have equal values in the ordering column.

DENSE_RANK()

Similar to RANK(), but it does not leave gaps between ranks when there are ties.

ROW_NUMBER() OVER(PARTITION BY <group-column> ORDER BY <sort-column> DESC) as rowno

Sorting number for rows: not repeated

The grouping and sorting in over() are executed later than where and group by, but not later than order by.

RANK() OVER(PARTITION BY <group-column> ORDER BY <sort-column> DESC) as rowno

Sorting number for rows: will be repeated, total number remains unchanged

DENSE_RANK() OVER(PARTITION BY <group-column> ORDER BY <sort-column> DESC) as rowno

Sorting number for rows: will be repeated, reducing the total number

```
SELECT
    employee_id,
    department,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as rownum,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) as rank,
    DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dense_rank
FROM
    employees;

Dataset:
employee_id  department   salary
1            A            1000
2            A            1000
3            A            900
4            B            1200
5            B            1100

Output:
employee_id  department   salary   rownum   rank     dense_rank
1            A            1000     1        1        1
2            A            1000     2        1        1
3            A            900      3        3        2
4            B            1200     1        1        1
5            B            1100     2        2        2


SELECT
    artical_id,
     SUM(tag) OVER (PARTITION BY artical_id ORDER BY dt,tag DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
ROW) AS uv
FROM main
SELECT
    user_id,
    gender
FROM Genders
ORDER BY (
```

```
   RANK() OVER( PARTITION BY gender ORDER BY user_id ASC) ) * 3 + IF(gender = 'female', 0, IF(gender =
'other', 1, 2)
)
```

ROWS BETWEEN

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Indicates from the starting point to the current row

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

Indicates that the current line has reached the endpoint

ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING

Indicate from 2 rows forward to 1 row backward

ROWS BETWEEN 2 PRECEDING AND 1 CURRENT ROW

Indicate from 2 lines ahead to the current line

```
SELECT
    uname,
    create_time,
    pv,
    first_value(pv) over (PARTITION BY uname ORDER BY create_time ROWS BETWEEN UNBOUNDED PRECEDING AND
    CURRENT ROW) as first_value_pv,
    last_value(pv) over (PARTITION BY uname ORDER BY create_time ROWS BETWEEN UNBOUNDED PRECEDING AND
    CURRENT ROW) as last_value_pv
FROM dw_tmp.window_function_temp;
```

DECLARE

DECLARE a INT;

The DECLARE statement must come before other SQL statements in a precedural block.

mysql

@prev = Num	Determine whether the custom field Prev is equal to Num
@prev := Num	Assign num to the custom field Prev

DAM / Mysql

Core

MySQL, a popular relational database management system (RDBMS), has a comprehensive internal structure that includes several components working together to manage and process data efficiently.

Data Types

Numeric Data Types

TINYINT

A very small integer. Range: -128 to 127 or 0 to 255 (unsigned).

SMALLINT

A small integer. Range: -32,768 to 32,767 or 0 to 65,535 (unsigned).

MEDIUMINT

A medium-sized integer. Range: -8,388,608 to 8,388,607 or 0 to 16,777,215 (unsigned).

INT or INTEGER

A standard integer. Range: -2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295 (unsigned).

The M in INT(M) specifies the display width, but it does not affect the range or storage.

```
    CREATE TABLE example_int (
        id INT,
```

```
    unsigned_id INT UNSIGNED,
    display_width INT(5)
);
```

BIGINT

A large integer. Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 0 to 18,446,744,073,709,551,615 (unsigned).

Bit Types

BIT(M)

Bit-field type. M represents the number of bits (from 1 to 64).

Fixed-Point Types

DECIMAL(M, D) or NUMERIC(M, D)

Fixed-point number. M is the maximum number of digits (precision), and D is the number of digits to the right of the decimal point (scale).

```
CREATE TABLE example_decimal (
    price DECIMAL(10, 2),
    rate NUMERIC(5, 2)
);
```

Floating-Point Types

FLOAT(M, D)

Single-precision floating-point number. M is the total number of digits, and D is the number of digits after the decimal point.

DOUBLE(M, D)

Double-precision floating-point number. M is the total number of digits, and D is the number of digits after the decimal point.

String (Character) Data Types

CHAR(M)

A fixed-length string. M represents the length.

VARCHAR(M)

A variable-length string. M represents the maximum length.

BINARY(M)

A fixed-length binary string. M represents the length of the string in bytes.

Unlike CHAR(M), which stores character data, BINARY(M) stores raw binary data.

This means the data is not interpreted as text but as a sequence of bytes.

TEXT

A text column with a maximum length of 65,535 characters.

TINYTEXT

A text column with a maximum length of 255 characters.

MEDIUMTEXT

A text column with a maximum length of 16,777,215 characters.

LONGTEXT

A text column with a maximum length of 4,294,967,295 characters.

BLOB

A binary large object with a maximum length of 65,535 bytes.

TINYBLOB

A binary large object with a maximum length of 255 bytes.

MEDIUMBLOB

A binary large object with a maximum length of 16,777,215 bytes.

LONGBLOB

A binary large object with a maximum length of 4,294,967,295 bytes.

ENUM

An enumeration, which is a string object with a value chosen from a list of permitted values.

```
CREATE TABLE example_enum (
    status ENUM('active', 'inactive', 'pending')
);
```

SET

A set, which is a string object that can have zero or more values, each chosen from a list of permitted values.

```
CREATE TABLE example_set (
    permissions SET('read', 'write', 'execute')
);
```

Date and Time Data Types

DATE

A date value in 'YYYY-MM-DD' format.

DATETIME

A date and time value in 'YYYY-MM-DD HH:MM' format.

TIMESTAMP

A timestamp value in 'YYYY-MM-DD HH:MM' format. Automatically updates when a row is modified.

TIME

A time value in 'HH:MM' format.

YEAR

A year value in 'YYYY' format.

Json

JSON

Stores JSON-formatted data. This type is useful for storing JSON objects and arrays directly in a column.

```
INSERT INTO example_json (settings) VALUES ('{"theme": "dark", "notifications": true}');

SELECT settings->>"$.theme" AS theme FROM example_json;
```

Page Data

An **index page** is a unit of storage used by database management systems (DBMS) to store index data.

In the context of MySQL and its InnoDB storage engine, an index page contains index entries, which are references to the actual data rows in a table.

Index pages are part of the overall structure that allows databases to quickly locate and retrieve rows without having to perform a full table scan.

The number of pages in a B+ tree depends on the branching factor of the tree (i.e., the number of children each internal node can have) and the number of keys stored in each node.

Contents of a Page

Index Pages: These contain index entries. Each index page in a B+ tree stores multiple index entries (key values and pointers to child nodes or data pages).

Data Pages: These contain actual row data. Each data page stores multiple rows of a table.

Visualization

Internal Node (Index Page):

```
+-------------------------------------------+
| Key 1 | Pointer 1 | Key 2 | Pointer 2 | ... |
+-------------------------------------------+
```

Each key-pointer pair is an entry in the internal node (index page).

In InnoDB, pointers (or row IDs) in indexes are generally 6 bytes.

Key Size Details

Fixed-Length Data Types:

For fixed-length data types like CHAR or INT, the size is consistent. For instance, an INT key is always 4 bytes.

Variable-Length Data Types:

For variable-length data types like VARCHAR, the size depends on the length of the string and the character set used.

For example, VARCHAR(100) might use up to 100 bytes for the characters plus additional bytes for length information.

Leaf Node (Data Page):

```
+-----------------------------------+
| Row 1 | Row 2 | Row 3 | ... | Row N |
+-----------------------------------+
```

Each row is an entry in the leaf node (data page).

Index Type

Clustered Index

The table data is physically organized according to the order of the clustered index. This means the rows are stored in sorted order based on the index key.

There can be only one clustered index per table because the table's rows can be sorted in only one way.

Sparse Index

The index is stored separately from the actual table data. The index contains the index key values and pointers to the corresponding rows in the table.

Unlike clustered indexes, a table can have multiple sparse indexes. Each sparse index provides an additional way to look up data based on different columns.

```
-- Creating a table with a primary key, which is a clustered index in InnoDB
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50)
) ENGINE=InnoDB;

-- Creating a secondary (sparse) index on the 'department' column
CREATE INDEX idx_department ON employees(department);
```

Steps to Use a Sparse Index to Find Actual Table Data

Index Lookup:

The database engine first searches the sparse index for the specified key value(s).

In MySQL, the structure of both clustered and sparse (secondary) indexes is typically a B+Tree rather than a B-Tree.

This applies to the InnoDB storage engine, which is the most commonly used storage engine in MySQL.

which allow for efficient searching, insertion, and deletion operations.

Retrieve the Primary Key:

Each entry in a sparse index contains the index key and a pointer to the actual table data. In InnoDB, this pointer is actually the primary key value of the row in the clustered index.

This means that the sparse index entry maps the indexed column to the primary key of the row.

Clustered Index Lookup:

After finding the primary key value in the sparse index, the database engine uses this primary key to perform a lookup in the clustered index (which is the primary key index in InnoDB).

Retrieve the Row Data:

The clustered index contains the actual data rows in the order of the primary key.

By using Bloom filters, MySQL can reduce the number of disk I/O operations.

Since Bloom filters reside in memory and are compact, they allow the database engine to quickly eliminate non-relevant data without having to read it from disk.

Using the primary key found from the sparse index, the engine can directly locate and retrieve the complete row data from the clustered index.

Bloom Filter Assisted Index

A Bloom filter is initialized with a bit array of a fixed size. All bits are initially set to 0.

In MySQL, Bloom filters can be used to filter out non-matching rows in queries, especially in cases where full table scans would be costly.

Before accessing the actual data, the query processor can check the Bloom filter to quickly rule out rows that do not match the query conditions.

Index Pages

Structure:

An index page is structured to facilitate quick searches, insertions, deletions, and updates of index entries.

Typically, these pages are organized in a B-tree or B+ tree structure, which allows for efficient indexing and searching operations.

B-Tree and B+ Tree:

InnoDB uses a B+ tree structure for its indexes. Each node in the tree is a page that contains a set of index entries.

The leaf nodes (index pages at the lowest level of the tree) contain pointers to the actual data rows, while the internal nodes contain pointers to other index pages.

Contents of an Index Page:

Each index page contains a number of index entries. An index entry consists of the indexed column values and a pointer (or a reference) to the corresponding data row in the table.

This reference is typically a physical location or a primary key value.

Page Size:

The size of an index page is usually the same as the database page size, which is often 16KB in InnoDB.

This size can be configured in MySQL using the innodb_page_size parameter.

Efficient Searches:

Index pages allow for efficient binary search operations.

Since index pages are organized in a tree structure, the database can quickly traverse the tree from the root to the leaf nodes to find the required index entry.

Caching in Buffer Pool:

Index pages, like data pages, are cached in the InnoDB buffer pool.

Caching index pages in memory significantly improves the performance of index-based queries because it reduces the need for disk I/O.

B+ tree structure

The number of pages in a B+ tree depends on the branching factor of the tree (i.e., the number of children each internal node can have) and the number of keys stored in each node.

Record Pointers in Indexes:

In InnoDB, pointers (or row IDs) in indexes are generally 6 bytes.

These pointers reference the actual rows in the clustered index, which is the primary index containing all the data rows.

Assuming a default page size of 16KB and ignoring the specifics of different column types for simplicity, let's outline the calculation steps:

Entries per Leaf Node:

If each index entry is approximately 1KB in size, a 16KB page can hold about 16 entries (this is a simplification, as the actual size might be smaller, allowing more entries).

Fan-out of Internal Nodes:

If each internal node entry (including a pointer) is approximately 100 bytes, a 16KB page can hold about 160 pointers to child nodes.

Query optimizer selects the wrong index

The query optimizer in MySQL is a crucial component that determines the most efficient way to execute a SQL query.
It analyzes various possible query execution plans and chooses the one that is expected to be the fastest or most resource-efficient.

When the query optimizer in a database like MySQL selects the wrong index, it can lead to inefficient query execution, resulting in slower performance.
If you encounter this situation, there are several strategies you can use to guide the optimizer or improve query performance:

Analyze and Update Statistics

Update Statistics:

Ensure that the statistics used by the optimizer are up-to-date. You can do this by running the ANALYZE TABLE command, which updates the index statistics.

```
ANALYZE TABLE your_table_name;
```

Types of Statistics

- Table Statistics:
  - Row Count:
    - The number of rows in a table.
  - Table Size:
    - The amount of space used by the table.
- Index Statistics:
  - Index Cardinality:
    - The number of unique values in an index. This helps the optimizer understand the selectivity of the index (i.e., how effectively it can filter out rows).
  - Index Size:
    - The amount of space used by the index.
- Column Statistics:
  - Column Data Distribution:
    - Information about the distribution of values in a column (e.g., histograms of value ranges or frequencies).
  - Column Cardinality:
    - The number of distinct values in a column.

Use Index Hints

Force the Use of a Specific Index:

You can explicitly instruct the optimizer to use a particular index by using index hints in your query.

```
SELECT * FROM your_table_name USE INDEX (index_name) WHERE condition;
```

Ignore an Index:

Alternatively, you can tell the optimizer to ignore certain indexes if you believe it's picking the wrong one.

```
SELECT * FROM your_table_name IGNORE INDEX (index_name) WHERE condition;
```

Adjust the Query Cost Model

Adjust System Variables:

MySQL allows you to fine-tune the query optimizer through system variables like optimizer_switch. You can modify these settings to change how the optimizer behaves.

```
SET optimizer_switch = 'index_merge=off';
```

Review and Optimize the Query

Rewrite the Query:

Sometimes, small changes in how the query is written can influence the optimizer's index choice.

For example, restructuring joins, or conditions, or even using EXPLAIN to better understand how the query is executed can help.

Simplify the WHERE Clause:

If possible, simplify the conditions in the WHERE clause to make it easier for the optimizer to choose the best index.

Create or Drop Indexes

Create More Suitable Indexes:

If the current indexes aren't effective for your query, consider creating a new composite index that better matches the query's conditions.

```
CREATE INDEX new_index_name ON your_table_name(column1, column2);
```

Drop Unnecessary Indexes:

In some cases, dropping redundant or ineffective indexes can improve performance and make it easier for the optimizer to choose the right index.

Optimize SQL Queries in MySQL

Optimizing SQL queries in MySQL involves several strategies to enhance performance and reduce execution time. Here are some key techniques:

Create Indexes:

Indexes on columns frequently used in WHERE, JOIN, ORDER BY, and GROUP BY clauses can significantly speed up queries.

Use composite (multi-column) indexes when queries filter on multiple columns. The order of columns in the index should match the query's filter order.

Avoid Full Table Scans:

Ensure that your queries use indexes effectively to avoid full table scans, which are slower and consume more resources.

Use Proper Data Types:

Optimize Data Types:

Choose the most efficient data types for your columns. For example, use INT for numeric data rather than VARCHAR.

Avoid Nulls if Possible:

Try to avoid NULL columns as they can complicate indexing and comparisons.

Partition Large Tables:

If you have a very large table, consider partitioning it. Partitioning can help by dividing the table into smaller, more manageable pieces, which can speed up queries.

Optimize JOIN Operations:

Place the table with the smallest result set first in the JOIN sequence.

Use EXPLAIN to Analyze Queries:

Use EXPLAIN to see how MySQL executes your query, including which indexes are used, the order of tables, and the estimated number of rows scanned. This can help identify bottlenecks.

Optimize SELECT Statements:

Select Only Necessary Columns:

Avoid SELECT *. Instead, specify only the columns you need.

Use LIMIT:

When you don't need all results, use LIMIT to fetch only a subset of rows.

Avoid Subqueries When Possible:

Use JOIN Instead of Subqueries:

MySQL can often optimize JOIN operations better than subqueries.

Rewrite Subqueries as Joins:

If possible, rewrite subqueries in the FROM or WHERE clause as JOIN operations.

Handle over 50,000 transactions per day

For a MySQL database handling over 50,000 transactions per day and expected to be maintained for three years, optimization should focus on performance, scalability, and maintenance.

Here's a comprehensive approach to optimize and ensure the reliability of your database:

Sharding to Distribute Data Across Multiple Tables:

- Sharding Strategy:
  - Implement a sharding strategy to divide large tables into smaller, more manageable pieces.
  - This reduces the amount of data each query needs to process, significantly speeding up query execution.

Using Caching Middleware to Reduce Database Load:

- Implementing Cache Layers:
  - Use caching mechanisms (e.g., Redis, Memcached) to store frequently accessed data in memory, reducing the need for repetitive database queries and decreasing response times.

Choosing Appropriate Data Types and Storage Engines:

- Optimal Field Data Types:
  - Select the smallest data types that can hold your data (e.g., TINYINT instead of INT where possible) to save space and improve speed.
- Choosing the Right Storage Engine:
  - Use the appropriate storage engine for your tables.
  - For example, InnoDB is generally preferred for transactional tables due to its support for ACID compliance and foreign keys, while MyISAM might be used for read-heavy workloads.

Designing a Well-Structured Database:

- Allow Partial Data Redundancy:
  - To improve query performance, allow some data redundancy to minimize the need for complex joins, which can slow down query execution.
- Avoiding Join Queries:
  - When possible, structure your database to reduce the need for join queries. This can involve denormalizing certain data to keep related information together, improving access times.

Implementing Master-Slave Replication for Read-Write Separation:

- Setting Up Master-Slave Databases:
  - Split your database into a master for handling writes and slaves for handling reads.
  - This separation reduces the load on any single database instance, improving overall performance and availability.

Generating Static Pages for Rarely Changed Content:

- Static Content Generation:
  - For pages that don't change often, generate static HTML pages to serve content without querying the database.
  - This can greatly reduce the load on your database and improve page load times.

Writing Efficient and Optimized SQL:

- Focus on SQL Optimization:
  - Write SQL queries that are optimized for performance, using indexes, avoiding unnecessary columns in SELECT statements, and using efficient algorithms for data manipulation.

Storage Engines

Storage engines are modules that handle the actual reading and writing of data to the disk. MySQL supports multiple storage engines, each with unique features and optimizations.

特性	MyISAM	InnoDB	MEMORY
存储限制	有(平台对文件系统大小的限制)	64TB	有(平台的内存限制)
事务安全	**不支持**	**支持**	**不支持**
锁机制	**表锁**	**表锁/行锁**	**表锁**
B+Tree索引	支持	支持	支持
哈希索引	不支持	不支持	支持
全文索引	支持	支持	不支持
集群索引	**不支持**	**支持**	**不支持**
数据索引	不支持	支持	支持
数据缓存	不支持	支持	N/A
索引缓存	支持	支持	N/A
数据可压缩	支持	不支持	不支持
空间使用	低	高	N/A
内存使用	低	高	中等
批量插入速度	高	低	高
外键	**不支持**	**支持**	**不支持**

InnoDB

InnoDB is the default storage engine in MySQL starting from version 5.5.

Features

- ACID Transactions
  Supports Atomicity, Consistency, Isolation, Durability (ACID) properties, ensuring reliable transactions.
- Row-level Locking
  Provides fine-grained locking, reducing contention in concurrent environments.
- Foreign Keys
  Supports foreign key constraints for referential integrity.
- Crash Recovery
  Uses a write-ahead logging mechanism to ensure data integrity and recovery after crashes.
- Data Integrity
  Ensures data consistency through checksums and transaction logs.

Indexes Supported:

- Primary Key Index (automatically created for primary key constraints)
- Unique Index
- Foreign Key Constraint (supports enforcing referential integrity)
- Full-text Index (supports full-text search)

Constraints Supported:

- Primary Key Constraint
- Unique Constraint
- Foreign Key Constraint
- NOT NULL Constraint

Use Cases

General-purpose applications requiring robust transaction support, data integrity, and concurrency handling.

MyISAM

Legacy storage engine, optimized for read-heavy operations.

Features

- Table-level Locking
  Locks entire tables, which can be a bottleneck in write-heavy applications.
- Non-transactional
  Does not support transactions, making it less suitable for applications requiring ACID compliance.
- High Read Performance
  Optimized for fast read operations, making it suitable for read-heavy workloads.
- Compact Storage
  Efficient storage format with lower disk space usage.

Indexes Supported:

- Primary Key Index (used for primary key constraints)
- Unique Index

Constraints Supported:

- Primary Key Constraint
- Unique Constraint

Use Cases

Applications with heavy read operations, logging systems, and data warehousing where transactions are not critical.

Memory

Stores data in RAM for fast access.

Features

- Volatile Storage
  Data is lost when the MySQL server is restarted.
- Fast Access
  Extremely fast read and write operations due to in-memory storage.
- Hash Indexes
  Uses hash indexes for quick data retrieval.

Indexes Supported:

- Primary Key Index
- Unique Index

Constraints Supported:

- Primary Key Constraint
- Unique Constraint

Use Cases

Temporary data storage, session management, caching frequently accessed data.

NDB (Cluster)

Used for MySQL Cluster, providing high availability and scalability.

Features:

- Distributed Storage
  Data is distributed across multiple nodes for high availability.
- Real-time Performance
  Designed for real-time applications with low-latency requirements.
- Failover
  Automatic failover and self-healing capabilities.
- Scalability

Can scale out by adding more nodes to the cluster.

Indexes Supported:

- Primary Key Index
- Unique Index
- Hash Index (for non-unique indexes)

Constraints Supported:

- Primary Key Constraint
- Unique Constraint
- Foreign Key Constraint (limited support compared to InnoDB)

Use Cases

Telecommunications, real-time applications, high-availability systems.

Merge

Allows multiple MyISAM tables to be used as a single table.

Features:

- Combines Tables

  Merges identical MyISAM tables into a virtual table for querying.
- Scalability

  Enables scaling by splitting data into multiple tables.
- Read-only

  Suitable for read-heavy operations.

Indexes Supported:

Acts as a wrapper for multiple MyISAM tables.

Inherits support for indexes from underlying MyISAM tables.

Does not introduce its own index types beyond what MyISAM supports.

- Primary Key Index
- Unique Index

Constraints Supported:

- Primary Key Constraint
- Unique Constraint

Use Cases

Data warehousing, large datasets that need to be queried as a single unit.

CSV

Stores data in comma-separated values format.

Features:

- Human-readable

  Data is stored in a format that can be easily read and edited with text editors.
- Portability

  Easy to move data between different systems and applications.
- No Indexes

  Does not support indexing, making it slow for large datasets.

Indexes Supported:

- Primary Key Index
- Unique Index

Constraints Supported:

- Primary Key Constraint
- Unique Constraint

Use Cases

Data exchange between MySQL and other systems, simple data logging.

Archive

Optimized for high-compression storage and retrieval of large volumes of data.

Features:

- Compression

  High compression ratios to save disk space.
- Insert-only

  Supports only INSERT and SELECT operations, not UPDATE or DELETE.
- Efficient Storage

  Suitable for archiving historical data.

Indexes Supported:

- None (does not support indexes)

Constraints Supported:

- None (does not support constraints)

Use Cases

Logging, audit trails, archival storage of infrequently accessed data.

Buffer Pool

The MySQL Buffer Pool is a critical component of the InnoDB storage engine.

It is an in-memory area where data and index pages are cached to improve performance by reducing disk I/O.

By keeping frequently accessed data in memory, the buffer pool significantly speeds up read and write operations.

Key Features

Caching Data Pages:

The buffer pool stores pages that have been read from disk.

When a query requests data, InnoDB first checks if the required data is in the buffer pool.

If it is, the data is retrieved directly from memory, which is much faster than reading from disk.

Caching Index Pages:

Similarly, index pages are also cached in the buffer pool.

This improves the speed of index lookups, which are critical for query performance.

Adaptive Hash Indexing:

InnoDB can create a hash index on the most frequently accessed pages in the buffer pool.

This adaptive hash index provides fast lookups for frequently accessed values.

Doublewrite Buffer:

To protect against data corruption, InnoDB uses a doublewrite buffer. Before pages are written to the data files, they are first written to the doublewrite buffer in the buffer pool.

This helps to ensure data integrity in case of a crash during a write operation.

Page Replacement Policy:

InnoDB uses a modified LRU (Least Recently Used) algorithm to manage the buffer pool. Pages that have not been used recently are candidates for eviction to make room for new pages.

However, the algorithm is modified to give preference to pages that are expected to be used again soon.

Configuration

my.cnf or my.ini

```
[mysqld]
default-storage-engine=INNODB
character-set-server = utf8mb4                    #server 级别字符集
socket=/usr/local/mysql-8.0.27/sdk/mysql.sock
datadir=/usr/local/mysql-8.0.27/data              # 数据目录
pid-file=/usr/local/mysql-8.0.27/sdk/mysqld.pid   # pid 文件
user= mysql                                       # 指定用户启动
port=13306                                        # 设置端口
log-error=/usr/local/mysql-8.0.27/log/mysqld-err.log   #日志目录（默认：/var/log/mysqld.log）
lower_case_table_names=1                  小写表名
tmpdir =/usr/local/mysql-8.0.27/tmp               #此目录被 MySQL 用来保存临时文件.例如,它被用来处理基于磁盘的大型排序,和内部排序一样，以及简单的临时表.


bind-address = 0.0.0.0                            #监听的 ip 地址 【主从复制配置】
server_id=1                                       # 配置 mysql replication 需要定义，不能和 canal 的 slaveId 重复【主从复制配置】
log-bin = /var/log/mysql/mysql-bin.log            # 打开二进制日志功能.在复制（replication）配置中【主从复制配置】

binlog-format=ROW                 # binlog 模式
log-bin-index = mysql-bin.index   #二进制的索引文件名
default-time-zone = system        # 服务器时区
log_timestamps=system             # mysql5.7 参数

default_authentication_plugin=mysql_native_password    老版本的密码认证方式
slave-load-tmpdir =/usr/local/mysql-8.0.27/tmp/        #当 slave 执行 load data infile 时使用
wait-timeout = 28800                                   #等待关闭连接的时间
net_read_timeout = 30                                  #从服务器读取信息的超时


collation-server = utf8mb4_general_ci
skip-character-set-client-handshake
secure_file_priv=''
[client]
default-character-set=utf8mb4
[mysql]
default-character-set=utf8mb4
```

Optimizing MySQL from a configuration standpoint

Engine [A]

innodb_buffer_pool_size 8G (for a system with 12GB of RAM)

Set this to 70-80% of your system's available memory. This is crucial because it stores the indexes and data of InnoDB tables in memory, reducing disk I/O.

innodb_log_file_size 2G (for a buffer pool size of 8GB)

Increase the size of InnoDB log files to improve performance for write-heavy operations. A good starting point is 25% of innodb_buffer_pool_size.

innodb_flush_log_at_trx_commit 1

Set this to 1 for full ACID compliance, 2 for better performance with some data loss risk in case of a crash, or 0 for the highest performance with higher risk.

innodb_file_per_table 1

Enable this to store each InnoDB table and its indexes in separate files, making management and backup easier.

innodb_flush_method O_DIRECT

Set this to O_DIRECT to avoid double buffering and reduce I/O overhead.

key_buffer_size 256M (for a system with 12GB of RAM)

For MyISAM tables, this should be set to 25% of available memory.

it is crucial for caching index blocks.

The key_buffer_size is not as critical for InnoDB tables since InnoDB uses the InnoDB buffer pool (innodb_buffer_pool_size) to manage its data and indexes.

Cache

sort_buffer_size 2M

Increase this if your queries involve large sorts. However, setting this too high can cause excessive memory usage.

join_buffer_size 2M

Increase this to optimize join operations, especially if you're dealing with large joins that cannot be handled by indexes alone.

query_cache_size 0 (recommended for write-heavy workloads)

Allocate memory for the query cache if you have many identical SELECT queries.

If your workload involves a lot of writes, consider disabling it (query_cache_type=0) as it may cause contention.

query_cache_limit 1M

Set a maximum result size that can be cached. This prevents large queries from consuming too much cache space.

table_open_cache 2000

Increase this to keep more tables open at once, reducing the overhead of opening and closing tables.

Performance

optimizer_switch 'index_merge=on,block_nested_loop=on,batched_key_access=on'

Adjust optimizer behavior, such as index usage, to better suit your queries.

You can fine-tune options like `index_merge`, `block_nested_loop`, and `batched_key_access` depending on your workload.

performance_schema 1

Enabling the Performance Schema allows you to monitor detailed server performance metrics. However, it can add overhead, so consider enabling it only when needed.

Temporary Table [B]

tmp_table_size 64M

max_heap_table_size 64M

Increase these if your queries frequently create temporary tables.

This allows larger temporary tables to be kept in memory rather than being written to disk.

Log Configuration

slow_query_log 1

Enable the slow query log to identify queries that are taking longer than expected. Analyze and optimize these queries.

log_queries_not_using_indexes 1

Enable this to log queries that are not using indexes, which can help identify where indexing may be needed.

long_query_time = 1

This parameter sets the threshold (in seconds) that determines which queries are considered "slow" and should be logged.

slow_query_log_file = /var/log/mysql/slow-query.log

This parameter specifies the file path where the slow query log will be stored.

Connection [D]

max_connections 200

Increase this if your application needs to handle many simultaneous connections.
Be cautious not to set it too high, as each connection consumes memory.

thread_cache_size 16

Set this to reduce the overhead of creating new threads by reusing existing ones.
A good starting point is setting it to 8 or using the formula max_connections / 2.

Replication

sync_binlog 1

Set this to 1 to ensure that the binary log is flushed to disk after every transaction,
which is crucial for data consistency in replication setups. Setting this to 0 can improve performance but risks losing data on a crash.

binlog_cache_size 32M

Increase this for workloads with large transactions to avoid using temporary files for binary logging.

General Logs

[mysqld]
general_log = 1
general_log_file = /path/to/general_query_log.log

Example

The General Log records all SQL queries received by the MySQL server, including both successful and unsuccessful ones.
It essentially logs every action taken by the server, making it useful for debugging and auditing purposes.

Contents:

All SQL statements executed by the server.
Client connections and disconnections.

Usage:

Debugging: Since it logs every query, the General Log is useful for tracking what exactly is being executed on the server.
Auditing: You can use it to audit user activity, including login attempts and the exact queries being run.

Storage:

It can be stored as a file or in a table within the MySQL database (mysql.general_log).
The log can grow very large quickly because it records all queries, so it's often only enabled temporarily for troubleshooting.

In the mysql.general_log table, the entries might look like this:

```
SELECT * FROM mysql.general_log LIMIT 10;
```

Output:

```
+----------------------------+------+-----------------+-----------+---------+-------+-----------------
---+
| event_time                 | user | host            | thread_id | command | arg   |    |
+----------------------------+------+-----------------+-----------+---------+-------+-----------------
---+
| 2024-07-25 14:22:56.123456 | root | localhost       |         3 | Connect |       |    |
| 2024-07-25 14:22:56.123456 | root | localhost       |         3 | Query   | SELECT * FROM my_table |
| 2024-07-25 14:22:56.123456 | root | localhost       |         3 | Quit    |       |    |
+----------------------------+------+-----------------+-----------+---------+-------+-----------------
---+
```

Entries in the general query log file might look like this:

```
2024-07-25T14:22:56.123456Z    3 Connect   root@localhost on
2024-07-25T14:22:56.123456Z    3 Query     SELECT * FROM my_table
2024-07-25T14:22:56.123456Z    3 Quit
```

Error Logs

The primary purpose of Error Logs is to record any errors, warnings, and other important messages related to the MySQL server's operation.

This includes startup and shutdown messages, critical errors, warnings, and notifications that can help in diagnosing issues.

Contents:

Startup and Shutdown Messages: Logs messages when the MySQL server starts up or shuts down.

Error Messages: Logs any errors that occur during the MySQL server's operation, such as syntax errors in queries, connection issues, or problems accessing the database files.

Warnings: Logs warnings that may not be critical errors but indicate potential issues or configuration problems.

Critical Events: Logs events like crashes, out-of-memory errors, and hardware-related issues that impact the server's operation.

Information Messages: May also include informational messages about the server's operation, such as successful startup or shutdown, or important configuration changes.

Usage:

Troubleshooting: Error Logs are indispensable for troubleshooting issues with the MySQL server. They provide detailed information about what went wrong, which can be used to diagnose and fix problems.

Monitoring: Regularly monitoring Error Logs can help you detect and address issues before they become critical. Automated monitoring tools can be set up to alert you when specific errors or warnings appear in the logs.

Security: Error Logs can also help in identifying unauthorized access attempts or potential security breaches by logging failed connection attempts or suspicious activity.

Storage:

The Error Log is typically stored as a text file. The location of this file is determined by the log_error system variable.

The default location and name of the Error Log file might vary depending on the operating system and MySQL configuration. For example, it could be named mysqld.err or simply error.log.

The log file is usually located in the data directory, but it can be configured to be placed elsewhere.

Configuration:

You can configure various aspects of the Error Log, such as its location and verbosity, through the MySQL configuration file (my.cnf or my.ini).

The log_error variable is used to specify the path of the Error Log file.

You can also direct error output to the system's logging service, such as syslog on Unix-like systems, instead of or in addition to a file.

Performance Impact:

The Error Log typically has a minimal impact on performance because it only logs events when something noteworthy occurs, rather than logging every transaction or query.

However, if the server encounters a large number of errors or warnings, the Error Log can grow quickly, which might need to be managed to prevent disk space issues.

[mysqld]

[mysqld]

log_error = /path/to/mysql_error.log # Enable error logging

log_error_verbosity = 3 # Optional: Set the error log verbosity

innodb_status_output = ON # Enable periodic output of InnoDB status reports to the MySQL error log.
innodb_status_output_locks = ON # Enable periodic output of InnoDB status reports, including detailed lock information, to the MySQL error log.

innodb_monitor Table

The innodb_monitor table is a special table used for diagnostics and performance monitoring in MySQL's InnoDB storage engine.
When you create and use this table, it triggers the InnoDB Monitor to produce a status report, which includes detailed information about the internal workings of InnoDB.
The generated status report is then written to the MySQL error log file. You can locate the error log file based on your MySQL server configuration.

Creating the Table:

```
CREATE TABLE innodb_monitor (a INT) ENGINE=INNODB;
```

This creates a table named innodb_monitor using the InnoDB storage engine.
The structure of this table is minimal and does not affect InnoDB's functionality.

Inserting Data:

```
INSERT INTO innodb_monitor VALUES (1);
```

Performing this INSERT operation triggers the InnoDB Monitor to produce a status report.

Status Output in the Error Log

```
=====================================
2024-07-25 14:30:01 0x7f3b7c3e4700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 2 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 1 srv_active, 0 srv_shutdown, 1 srv_idle
srv_master_thread log flush and writes: 1
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 1
OS WAIT ARRAY INFO: signal count 1
RW-shared spins 0, rounds 0, OS waits 0
RW-excl spins 0, rounds 0, OS waits 0
SPINS PER WAIT: RW-shared 0.00 RW-excl 0.00
--------
FILE I/O
--------
I/O thread 0 state: waiting for completed aio requests (insert buffer thread)
I/O thread 1 state: waiting for completed aio requests (log thread)
I/O thread 2 state: waiting for completed aio requests (read thread)
I/O thread 3 state: waiting for completed aio requests (write thread)
Pending normal aio reads: 0, aio writes: 0,
ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
```

Status Output with Lock Information

```
=====================================
2024-07-25 14:30:01 0x7f3b7c3e4700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 2 seconds
------------------------
LATEST DETECTED DEADLOCK
------------------------
2024-07-25 12:34:56 0x7f3cbe9b4700
*** (1) TRANSACTION:      # The first transaction associated with the dead lock.
```

```
TRANSACTION 40928956, ACTIVE 0 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1136, 3 row lock(s)
MySQL thread id 1234, OS thread handle 139897635575680, query id 98765432 localhost user_name
SELECT * FROM table_name WHERE id = 123 FOR UPDATE
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 12345 page no 123 n bits 72 index `PRIMARY` of table `db_name`.`table_name` trx id
40928956 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
...
*** (2) TRANSACTION:
TRANSACTION 40928955, ACTIVE 0 sec fetching rows
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1136, 2 row lock(s)
MySQL thread id 1235, OS thread handle 139897635574144, query id 98765433 localhost user_name
UPDATE table_name SET column_name = 'value' WHERE id = 123
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 12345 page no 123 n bits 72 index `PRIMARY` of table `db_name`.`table_name` trx id
40928955 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
...
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 12345 page no 123 n bits 72 index `PRIMARY` of table `db_name`.`table_name` trx id
40928955 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
...
*** WE ROLL BACK TRANSACTION (1)
------------------
BACKGROUND THREAD
------------------
srv_master_thread loops: 1 srv_active, 0 srv_shutdown, 1 srv_idle
srv_master_thread log flush and writes: 1
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 1
OS WAIT ARRAY INFO: signal count 1
RW-shared spins 0, rounds 0, OS waits 0
RW-excl spins 0, rounds 0, OS waits 0
SPINS PER WAIT: RW-shared 0.00 RW-excl 0.00
--------
FILE I/O
--------
I/O thread 0 state: waiting for completed aio requests (insert buffer thread)
I/O thread 1 state: waiting for completed aio requests (log thread)
I/O thread 2 state: waiting for completed aio requests (read thread)
I/O thread 3 state: waiting for completed aio requests (write thread)
Pending normal aio reads: 0, aio writes: 0,
ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
```

Analyzing the Deadlock Log

- Transaction Information

  Each deadlock entry starts with a timestamp and contains details about the transactions involved.

- Transaction Details

  The log includes the SQL statement being executed, the state of the transaction, and the locks held and requested.

- Locks

  Look for the HOLDS THE LOCK(S) and WAITING FOR THIS LOCK TO BE GRANTED sections to see which locks are causing the deadlock.

- Resolution

  MySQL decides which transaction to roll back to resolve the deadlock (*** WE ROLL BACK TRANSACTION).

Binary Logs

The Binary Log is primarily used for replication and data recovery. It records all changes made to the database's data, such as updates, inserts, and deletes, in a binary format.

Contents:

All data-modifying SQL statements (e.g., INSERT, UPDATE, DELETE).

Data definition language (DDL) statements like CREATE, ALTER, DROP.

Information required for replication (e.g., statements that would be run on slave servers).

Usage:

Replication:

Binary Logs are used to replicate databases across multiple servers.

The master server writes changes to the Binary Log, and these changes are sent to slave servers to apply them, ensuring consistency across replicas.

Point-in-Time Recovery:

Binary Logs can be used to restore a database to a specific point in time.

By replaying the Binary Logs after restoring a backup, you can recover lost data up to a specific moment.

Storage:

Stored in binary format, which is more compact than the General Log.

Can be configured to be stored on disk and is typically managed by the server in rotating files.

```
[mysqld]
log-bin=mysql-bin            # Enable binary logging
binlog-format=ROW            # Recommended format (ROW, STATEMENT, or MIXED)
```

STATEMENT: Logs each SQL statement that modifies data.

ROW: Logs each row that is modified.

MIXED: Uses statement-based logging by default, but switches to row-based logging in certain cases.

Because MySQL uses binlog to synchronize data in the master-slave replication process,
if the isolation level is set to read committed (RC), when transactions are out of order, the results of the standby database will be inconsistent with the content of the master database after SQL playback.
MySQL prohibits using READ COMMITTED as the transaction isolation level when using statement format binlog.

Example Scenario

Consider the following scenario where the issues might arise:

Primary Server:

Transaction T1 starts and runs under READ COMMITTED.

Transaction T2 updates some rows and commits.

Transaction T1 executes a SELECT statement and sees the changes made by T2.

Transaction T1 executes an UPDATE statement based on the results of the SELECT.

Replica Server:

The replica replays the SQL statements from the binary log.

Because the replica might have a different view of the data (due to different query snapshots or concurrent transactions), the results of the SELECT statement might differ.

As a result, the UPDATE statement might affect different rows or have different effects.

```
server-id=1                  # Unique server ID
expire-logs-days=7           # Automatically remove logs older than 7 days
max_binlog_size=100M         # Maximum size of each binary log file
                               # If the file exceeds this size, it will be renamed with an extension of -old and a new file will
```

be created

Example

The binary log is used for replication and data recovery purposes. It records all changes to the database, including data modifications and DDL operations.
Essential for replication and recovery, focusing on changes to the data.

```
# at 4                                                          # The position in the log file.
#210725 10:30:00 server id 1  end_log_pos 1234                  # The timestamp of the event, server ID, and end
position of the event in the log file.
# Query   thread_id=1   exec_time=0   error_code=0              # Indicates the type of event and other metadata,
such as the thread ID and execution time.
SET TIMESTAMP=1627225800;                                       # The timestamp set for the event.
INSERT INTO mytable (id, name) VALUES (1, 'John Doe');          # The actual SQL statement executed.

# at 1234
#210725 10:31:00 server id 1  end_log_pos 2345
# Query   thread_id=1   exec_time=0   error_code=0
SET TIMESTAMP=1627225860;
UPDATE mytable SET name='Jane Doe' WHERE id=1;

# at 2345
#210725 10:32:00 server id 1  end_log_pos 3456
# Query   thread_id=1   exec_time=0   error_code=0
SET TIMESTAMP=1627225920;
DELETE FROM mytable WHERE id=1;
```

Redo Logs

Redo Logs are used for crash recovery.
They help ensure that all committed transactions are durable and can be recovered in case of a crash or system failure.
Redo Logs are crucial for maintaining the durability aspect of the ACID properties (Atomicity, Consistency, Isolation, Durability).

Contents:

Redo Logs store a record of all changes made to the data in the database. Specifically, they log changes to data pages before those changes are written to disk.

They contain physical changes (like updates to pages) rather than logical changes (like SQL statements).

Usage:

Crash Recovery:

If the MySQL server crashes, the Redo Logs are used to replay any committed transactions that were not yet written to the data files.

This ensures that the database can be brought back to a consistent state.

Durability:

By writing changes to Redo Logs before applying them to data files, MySQL ensures that committed transactions are not lost, even if the server crashes before the data is fully written to disk.

Storage:

Redo Logs are typically stored in a circular buffer. Once the buffer is full, old log records are overwritten, but only after the corresponding data pages have been safely written to disk.

They are stored in the ib_logfile0 and ib_logfile1 files in the InnoDB directory.

```
[mysqld]
innodb_log_file_size=512M              # Size of each redo log file
innodb_log_files_in_group=2            # Number of redo log files in the log group
innodb_flush_log_at_trx_commit=1    # Flush redo logs to disk at each transaction commit
```

Undo Logs

Undo Logs are used to maintain transaction isolation and to roll back transactions when necessary. They support the atomicity and isolation aspects of the ACID properties.

Contents:

Undo Logs contain the old versions of data before any changes were made by a transaction. If a transaction is rolled back, the Undo Logs are used to revert the changes to the data.

They store the previous values of rows that were modified by a transaction, allowing those changes to be undone.

Usage:

Transaction Rollback: If a transaction is aborted or rolled back, the changes made by that transaction can be undone using the information in the Undo Logs.

MVCC (Multi-Version Concurrency Control): Undo Logs are used to support MVCC by keeping track of older versions of data. This allows MySQL to provide consistent reads without locking, as it can access previous versions of rows for other transactions that need to read them.

Storage:

Undo Logs are stored within the InnoDB tablespace and are managed internally by MySQL.

Unlike Redo Logs, Undo Logs are more closely associated with the transaction lifecycle, and their contents are kept until the associated transaction is fully committed and no longer needed for MVCC.

Performance Impact:

Undo Logs can impact performance, particularly with large or long-running transactions, as they require additional storage and processing to maintain multiple versions of data.

[mysqld]
innodb_undo_tablespaces=2 # Number of undo tablespaces
innodb_undo_log_truncate=1 # Enable automatic truncation of undo logs
innodb_max_undo_log_size=2G # Maximum size of each undo log tablespace
innodb_undo_directory=/var/lib/mysql # Directory for undo tablespaces

Check Deadklock

Enable the General Query Log (Optional)

If you want to log all queries, including the deadlock information, you can enable the general query log. However, this might generate a large amount of data and should be used with caution.

```
SET GLOBAL general_log = 'ON';
SET GLOBAL general_log_file = '/path/to/general_log_file.log';
```

Enable the InnoDB Monitor

The InnoDB monitor can provide detailed information about the state of InnoDB, including deadlocks.

```
CREATE TABLE innodb_monitor (a INT) ENGINE=INNODB;
INSERT INTO innodb_monitor VALUES (1);
```

To disable the monitor:

```
DROP TABLE innodb_monitor;
```

Enable the InnoDB Status Output

The parameters innodb_status_output and innodb_status_output_locks are indeed related to the InnoDB storage engine in MySQL,
and they provide additional information about the status of InnoDB, including detailed lock information.

```
SET GLOBAL innodb_status_output = ON;
SET GLOBAL innodb_status_output_locks = ON;
```

You can get the current status of InnoDB, including information about deadlocks, by using the SHOW ENGINE INNODB STATUS command.

```
SHOW ENGINE INNODB STATUS;
```

This command outputs a lot of information. Look for the LATEST DETECTED DEADLOCK section to find details about the most recent deadlock.

Replication

MySQL master-slave replication is a technique used to replicate data from one MySQL database server (the master) to one or more MySQL database servers (the slaves).

This setup is commonly used for improving database performance, scalability, and redundancy.

Master Server

This is the primary MySQL server where all write operations (INSERT, UPDATE, DELETE) occur. The master server logs these changes into its binary log.

Slave Server(s)

These are MySQL servers configured to replicate data from the master. They connect to the master server, read the binary log, and apply changes locally to replicate the same data.

Replication Configuration

1. Initial Setup
   1) Enable Binary Logging
   2) Restart MySQL to apply the changes
   3) Create a Replication User:

```
CREATE USER 'repl'@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
FLUSH PRIVILEGES;
```

2. Obtain Master Log File and Position
   1) Lock the tables to ensure consistency during the initial data dump:

```
FLUSH TABLES WITH READ LOCK;
```

   2) Get the current binary log file name and position:

```
SHOW MASTER STATUS;
```

   3) Export the Database

      Use mysqldump to create a dump of the database:

```
mysqldump -u root -p --all-databases --master-data > data_dump.sql
```

      Unlock the tables after the dump

```
UNLOCK TABLES;
```

3. Configure the Slave Server(s):
   1) Import the Database Dump to the slave server

```
mysql -u root -p < data_dump.sql
```

   2) Configure Slave to Point to the Master

      Add a unique server-id for the slave

```
[mysqld]
server-id=2
```

      Restart MySQL to apply the changes

4. Set Up the Slave
   1) Log in to the MySQL server on the slave.
   2) Configure the slave to connect to the master

```
CHANGE MASTER TO
  MASTER_HOST='master_host',
  MASTER_USER='repl',
  MASTER_PASSWORD='password',
  MASTER_LOG_FILE='mysql-bin.000001',
  MASTER_LOG_POS=120;
```

5. Start the Slave:
   1) Start the replication process

```
START SLAVE;
```

   2) Check the slave status to ensure it's running correctly:

```
SHOW SLAVE STATUS\G;
```

Ongoing Replication

On the Master Server

Binary Log Dump Thread

This thread continuously sends the binary log events to connected slave servers.

All changes (write operations) to the database are recorded in the binary log.

On the Slave Server(s)

Replication SQL Thread

Reads events from the relay log.

Executes these events to apply changes to the slave's data, ensuring it mirrors the master.

Replication I/O Thread

Connects to the master server and reads binary log events.

Writes the events to a local relay log.

Cluster

Uses NDB (Network Database) storage engine for distributing data across multiple nodes with automatic sharding and replication.

Automatically partitions data across nodes for balanced load and redundancy.

Key Concepts of Data Sharding in MySQL NDB

Partitioning

The table is split into partitions based on a partitioning key. Each partition is stored in a different Data Node.

Partitioning Key

This is typically the primary key or a unique key in the table. The partitioning key determines how the data is distributed across the Data Nodes.

Distribution

The partitions are distributed across the available Data Nodes using a hashing algorithm. The hashing algorithm ensures an even distribution of data across the nodes.

Redundancy and Replication

Each partition is replicated to ensure data redundancy and high availability. The number of replicas is configurable.

How Partitioning Works

Hash-Based Partitioning

The most common method, where the value of the partitioning key is hashed to determine the partition. For example, if you have a table with a primary key id, the hash of id determines the partition.

Range-Based Partitioning

Less common in MySQL NDB, but involves dividing the table into ranges based on the partitioning key.

Cluster Configuration

Download and Install MySQL NDB Cluster

1. **Download the MySQL NDB Cluster software** from the official MySQL website or your package manager if available.
2. **Install the software** on each node that will participate in the cluster.

Set Up Configuration Files

MySQL NDB Cluster consists of three types of nodes:

1. **Management Nodes (ndb_mgmd)**
2. **Data Nodes (ndbd or ndbmtd)**
3. **SQL Nodes (mysqld)**

Management Node Configuration (config.ini)

Create a configuration file (config.ini) for the management node. This file contains the configuration settings for the entire cluster.

Place the config.ini file in the data directory of the management node.

```
[ndb_mgmd]
hostname=management-node-hostname
datadir=/var/lib/mysql-cluster

[ndbd default]
noofreplicas=2
datadir=/usr/local/mysql/data

[ndbd]
hostname=data-node1-hostname
datadir=/usr/local/mysql/data

[ndbd]
hostname=data-node2-hostname
datadir=/usr/local/mysql/data

[mysqld]
hostname=sql-node1-hostname

[mysqld]
hostname=sql-node2-hostname
```

Data Node Configuration

Create a configuration file for each data node.

Place this file in the MySQL configuration directory (e.g., /etc/my.cnf or /etc/mysql/my.cnf).

```
[mysql_cluster]
ndb-connectstring=management-node-hostname
```

SQL Node Configuration

Add the following lines to the MySQL configuration file (my.cnf) on each SQL node.

```
[mysqld]
ndbcluster
ndb-connectstring=management-node-hostname
```

Start the Cluster Components

Start the management node:

```
ndb_mgmd -f /path/to/config.ini
```

Start each data node:

```
ndbd
```

Start each SQL node (MySQL server):

```
mysqld_safe --ndbcluster &
```

Verify the Cluster

Connect to the management node to verify the status of the cluster.

```
ndb_mgm
```

Within the NDB management client, you can use commands like SHOW to display the status of the nodes.

```
ndb_mgm> SHOW
```

Configure SQL Nodes

Create the NDB Cluster metadata:

```
CREATE TABLE test_table (id INT PRIMARY KEY, data VARCHAR(255)) ENGINE=NDB;
```

Ensure that tables are created with the NDB storage engine to take advantage of the NDB cluster capabilities.

Monitoring and Management

Monitor the cluster using the ndb_mgm management client.

Check logs located in the data directories for each node to diagnose issues.

Example Configuration

config.ini for Management Node:

```
[ndb_mgmd]
hostname=mgm1
```

```
datadir=/var/lib/mysql-cluster

[ndbd default]
noofreplicas=2
datadir=/usr/local/mysql/data

[ndbd]
hostname=ndbd1
datadir=/usr/local/mysql/data

[ndbd]
hostname=ndbd2
datadir=/usr/local/mysql/data

[mysqld]
hostname=mysqld1

[mysqld]
hostname=mysqld2
```

my.cnf for SQL Nodes:

```
[mysqld]
ndbcluster
ndb-connectstring=mgm1
```

my.cnf for Data Nodes:

```
[mysql_cluster]
ndb-connectstring=mgm1
```

Partition Example

Suppose you have a MySQL Cluster with three Data Nodes and you configure the cluster to have one replica.

This setup will have three partitions, each stored in a different Data Node, and each partition will be replicated to another Data Node.

```
[ndbd]
HostName=192.168.0.2
DataDir=/usr/local/mysql/data

[ndbd]
HostName=192.168.0.3
DataDir=/usr/local/mysql/data

[ndbd]
HostName=192.168.0.4
DataDir=/usr/local/mysql/data
```

Data Node Distribution

Data Node 1: Stores Partition 1 and Replica of Partition 2.

Data Node 2: Stores Partition 2 and Replica of Partition 3.

Data Node 3: Stores Partition 3 and Replica of Partition 1.

Query Execution

When a query is executed:

Query Received:	The SQL Node receives the query.
Partition Lookup:	Based on the partitioning key, the SQL Node determines which Data Node holds the relevant partition.
Data Retrieval:	The SQL Node retrieves the data from the appropriate Data Node(s).
Result Return:	The SQL Node returns the result to the client.

Percona Toolkit

Ensure that you have Percona Toolkit installed on your system. You can download and install it from the Percona Toolkit website.

pt-schema-compare

Percona Toolkit includes pt-schema-compare, a command-line tool specifically for comparing and synchronizing database schemas.

pt-schema-compare --host=<source_host> --user=<user> --password=<password> --databases=<database_name> --tables=<table_name> h=<target_host>,D=<target_database>

--host=<source_host>: Hostname of the source database.
--user=<user>: Username for authentication.
--password=<password>: Password for authentication.
--databases=<database_name>: Name of the source database.
--tables=<table_name>: Name of the source table (optional, for comparing specific tables).
h=<target_host>: Hostname of the target database.
D=<target_database>: Name of the target database.

--charset=utf8: Specify the character set (optional).
--verbose: Provide detailed output (optional).
--ignore-tables: Ignore specific tables during the comparison (optional).
--alter-foreign-keys-method: Specify how to alter foreign keys (optional, e.g., drop_swap).

Example Commands

```
pt-schema-compare --host=source_host --user=root --password=your_password --databases=source_db --tables=table_name h=target_host,D=target_db
```

This command compares the schema of table_name in source_db on source_host with the schema of the same table in target_db on target_host.

```
pt-schema-compare --host=source_host --user=root --password=your_password --databases=source_db --tables=table_name h=target_host,D=target_db --execute
```

This command compares and synchronizes the schema of table_name in source_db on source_host with the schema of the same table in target_db on target_host.
The --execute option applies the necessary changes to make the target schema match the source schema.

Example Outputs

Schema Differences

```
$ pt-schema-compare h=source_host,u=user,p=pass,D=source_db h=target_host,u=user,p=pass,D=target_db

Schema differences:
--- `source_db`.`users`
+++ `target_db`.`users`
@@ -1,6 +1,6 @@
 CREATE TABLE `users` (
   `id` int(11) NOT NULL AUTO_INCREMENT,
-  `name` varchar(100) NOT NULL,
+  `username` varchar(150) NOT NULL,
   `email` varchar(255) DEFAULT NULL,
   `created_at` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
   PRIMARY KEY (`id`)
@@ -8,7 +8,7 @@

--- `source_db`.`orders`
+++ `target_db`.`orders`
@@ -1,5 +1,5 @@
 CREATE TABLE `orders` (
   `order_id` int(11) NOT NULL AUTO_INCREMENT,
-  `amount` decimal(10,2) NOT NULL,
+  `amount` decimal(12,2) NOT NULL,
   `status` enum('pending','shipped','delivered') DEFAULT 'pending',
   PRIMARY KEY (`order_id`)
 );
```

pt-table-checksum

pt-table-checksum calculates checksums for table data by reading the data from the tables and computing checksums at the row level.
This is particularly useful for verifying data consistency between a master and its replicas in a MySQL database setup.

Detection of New Data:

If new data is added to the target database after the initial checksums are calculated, pt-table-checksum will not automatically detect this new data in the subsequent comparison runs.
The tool primarily focuses on comparing the existing data between source and target based on the initial checksums calculated.

Basic Checksum Command:

```
pt-table-checksum --host=<source_host> --user=<user> --password=<password> --databases=<database_name>
```

- `<source_host>:` The hostname or IP address of the source database.
- `<user>:` The username with necessary privileges to access the database.
- `<password>:` The password for the user.
- `<database_name>:` The name of the database to check.

Checksum Specific Tables:

```
pt-table-checksum --host=<source_host> --user=<user> --password=<password> --databases=<database_name> --tables=<table_name>
```

- `<table_name>:` The name of the specific table to check.

Checksum with Replica Comparison:

```
pt-table-checksum --host=<source_host> --user=<user> --password=<password> --databases=<database_name> --replicate=percona.checksums --recursion-method=processlist
```

- `--replicate=percona.checksums:` Store the checksums in a table named checksums in the percona database.
- `--recursion-method=processlist:` Use the processlist method to detect replicas.

Example Output:

The command will output a summary of the checksum differences. If differences are found, it will list the tables and rows that are different.

pt-table-sync

pt-table-sync is a tool from the Percona Toolkit used to synchronize data between tables in different databases or between master and replica databases. This tool is helpful for resolving data inconsistencies by syncing the data.

pt-table-sync from Percona Toolkit does not simply copy all data from the source database to the target database.
Instead, it is designed to synchronize data between a master and its replicas or between two different databases by resolving inconsistencies.

How pt-table-sync Works

Identify Differences

pt-table-sync calculates checksums (or hashes) of rows from both source and target tables.
It identifies discrepancies between these checksums to find out which rows are different or missing.

Synchronize Data

After identifying discrepancies, pt-table-sync generates and executes the necessary SQL statements to update the target database so that its data matches the source database.
This involves inserting missing rows, updating existing rows, or deleting rows that are no longer present in the

source database.

Selective Synchronization

It does not always copy all data. It synchronizes only those rows that are different between the source and target tables. This makes the process efficient and avoids unnecessary data transfer.

Chunked Operations

pt-table-sync operates in chunks to avoid locking large tables and to improve performance. It divides the table data into smaller chunks, processes each chunk, and synchronizes it incrementally.

Error Handling and Logging

The tool logs the changes it makes and can handle various errors that might occur during synchronization.

```
pt-table-sync --host=<source_host> --user=<user> --password=<password> --databases=<database_name> --
tables=<table_name> --execute
```

--host=<source_host>: Hostname of the source database.

--user=<user>: Username for authentication.

--password=<password>: Password for authentication.

--databases=<database_name>: Name of the source database.

--tables=<table_name>: Name of the source table (optional, for syncing specific tables).

--execute: Apply the changes to the target database. Without this option, pt-table-sync only shows the changes it would make without applying them.

--replicate=database.sync_log: Specify the table where synchronization logs will be stored.

--where='condition': Only sync rows that match the given condition.

--check-charset: Check character set compatibility between source and target tables.

--no-check-binlog-format: Skip checking whether the binlog format is Row.

--ignore-databases=REGEX: Ignore databases that match this Perl regex.

--ignore-tables=REGEX: Ignore tables that match this Perl regex.

Example Commands

Synchronizing Tables

```
pt-table-sync --host=source_host --user=root --password=your_password --databases=source_db --
tables=table_name --execute
```

This command synchronizes the data in table_name in source_db on source_host with the corresponding table in the replica or target database.

Example Outputs

Dry-Run Mode

```
$ pt-table-sync --dry-run --print --sync-to-master h=replica1,P=3306,u=username,p=password

REPLACE INTO `mydb`.`users`(`id`, `name`, `email`) VALUES (42, 'Alice', 'alice@example.com');
DELETE FROM `mydb`.`orders` WHERE `order_id`=17;
INSERT INTO `mydb`.`products`(`id`, `name`, `price`) VALUES (99, 'New Product', 19.99);
UPDATE `mydb`.`orders` SET `status`='shipped' WHERE `order_id`=5;
```

Execution Mode

```
$ pt-table-sync --execute --sync-to-master h=replica1,P=3306,u=username,p=password

# Syncing `mydb`.`users`:
REPLACE INTO `mydb`.`users`(`id`, `name`, `email`) VALUES (42, 'Alice', 'alice@example.com');
# Syncing `mydb`.`orders`:
DELETE FROM `mydb`.`orders` WHERE `order_id`=17;
UPDATE `mydb`.`orders` SET `status`='shipped' WHERE `order_id`=5;
# Syncing `mydb`.`products`:
INSERT INTO `mydb`.`products`(`id`, `name`, `price`) VALUES (99, 'New Product', 19.99);

Changes applied successfully.
```

Master-slave synchronization

master：主从配置项在 my.ini 中添加

```
grant replication slave on *.* to 'slave1'@'192.168.33.33';      为 slave1 赋予 REPLICATION SLAVE 权限
FLUSH TABLES WITH READ LOCK;          为 mysql 加入读锁，使其变为只读。
SHOW MASTER STATUS;
mysqldump -u root -p --all-databases --master-data > dbdump.sql
UNLOCK TABLES;
```

slave：主从配置项在 my.ini 中添加

```
mysql -u root -p < /home/ubuntu/dbdump.sql        #导入 master 的 dump 文件
STOP SLAVE;
CHANGE MASTER TO
 -> MASTER_HOST='192.168.33.22',
 -> MASTER_USER='slave1',
 -> MASTER_PASSWORD='slavepass',
 -> MASTER_LOG_FILE='mysql-bin.000001',        #从上面的 SHOW MASTER STATUS 得到的
 -> MASTER_LOG_POS=613;
START SLAVE;
```

linux yum installation

rpm -ivh https://repo.mysql.com/mysql80-community-release-el7-1.noarch.rpm　下载 yum 仓库（https://repo.mysql.com/ http://mirrors.163.com/mysql/Downloads/MySQL-8.0/　查看镜像源，el7 对应 centos7）

/etc/yum.repos.d/mysql-community.repo，/etc/yum.repos.d/mysql-community-source.repo。　安装 repo.mysql.com 包后，会获得两个 mysql 的 yum repo 源。

setenforce 0　　临时关闭 selinux（0 permissive 模式　1 enforcing 模式）

vi /etc/selinux/config　重启永久关闭 selinux（SELINUX=disabled）

```
rpm -ivh mysql-community-common-8.0.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-client-plugins-8.0.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-libs-8.0.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-client-8.0.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-server-8.0.26-1.el7.x86_64.rpm

useradd labuladuo                    添加用户
passwd labuladuo                     重置密码

vim /etc/my.cnf                                  修改配置    l:a/av*AQ13a
chown -R labuladuo: labuladuo /var/lib/mysql                 设置 mysql 存储目录权限
chown -R  labuladuo:labuladuo /var/log/mysql                 设置 mysqld 日志 mysqld.log 权限
chown -R  labuladuo:labuladuo /var/run/mysqld                设置 mysqld pid 存储文件 mysqld.pid 权限

mysqld --initialize --console          初始化（获取密码）
vi /lib/systemd/system/mysqld.service  修改服务文件启动用户为 labuladuo 用户

systemctl start mysqld                 启动 mysqld
systemctl status mysqld                查看 mysqld 状态
```

yum 安装的 mysql 目录结构

/var/lib/mysql　　mysql 数据文件存放路径，可自定义

/etc/my.cnf mysql 配置文件路径
/usr/lib64/mysql mysql 库文件路径
/usr/bin/mysql* mysql 二进制可执行文件路径
/etc/rc.d/init.d/mysqld mysql 服务管理脚本地址
/var/log/mysqld.log mysql 日志文件路径

linux local 安装

wget https://cdn.mysql.com/archives/mysql-8.0/mysql-8.0.26-linux-glibc2.12-x86_64.tar.xz

tar -xJvf mysql-8.0.26-linux-glibc2.12-x86_64.tar.xz -C /usr/local

useradd -m labuladuo 创建用户

chown -R mysql:mysql /usr/local/mysql-8.0.26 修改权限

mkdir data mysql 下创建 data 文件夹

chmod 750 /usr/local/mysql/data -R

mkdir /var/lib/mysql

chown -R mysql:mysql /var/lib/mysql/

/usr/local/mysql/bin/mysql_install_db --user=mysql --basedir=/usr/local/mysql/ --datadir=/usr/local/mysql/data 初始化数据库 oLOxT>gLg7h+

/usr/local/mysql/bin/mysqld --user=mysql --basedir=/usr/local/mysql/ --datadir=/usr/local/mysql/data --initialize 5.7 版本的初始化命令后面要加 --initialize

./mysqld --user=mysql --basedir=/usr/local/mysql/mysql-8.0 --datadir=/usr/local/mysql/data/ --initialize

cp /usr/local/mysql/support-files/mysql.server /etc/init.d/mysql 加入服务

service mysql start 启动

export MYSQL8_HOME=$PATH:/usr/local/mysql 添加环境变量

export PATH=$PATH:$MYSQL8_HOME/bin:$MYSQL8_HOME/lib

vi /etc/my.cnf 修改配置文件

Installation Package Type	
mysql-server	MySQL 服务器。你需要该选项，除非你只想连接运行在另一台机器上的 MySQL 服务器。
mysql-client	MySQL 客户端程序，用于连接并操作 Mysql 服务器。
mysql-devel	库和包含文件，如果你想要编译其它 MySQL 客户端，例如 Perl 模块，则需要安装该 RPM 包。
mysql-shared	该软件包包含某些语言和应用程序需要动态装载的共享库(libmysqlclient.so*)，使用 MySQL。
mysql-bench	MySQL 数据库服务器的基准和性能测试工具。

Service	
systemctl start mysqld	启动 mysqld
systemctl status mysqld	查看 mysqld 状态

./bin/mysqld --defaults-file="/usr/local/mysql-8.0.26/my.cnf" 测试配置正常

vi ./support-files/mysql.server 修改启动器，配置 datadir，basedir，添加 start 启动参数--defaults-file="$baseconf"

tail -f /var/log/mariadb/mariadb.log 查看 mysql_safe 启动日志

error while loading shared libraries: libtinfo.so.5: ln -s /usr/lib64/libtinfo.so.6.1 /usr/lib64/libtinfo.so.5

Command
mysqld

mysqld -V 查看版本

mysqld --defaults-file="/usr/local/mysql-8.0.26/my.cnf" --daemonize --pid-file=/data/mysql/data/mysqld.pid 启动
mysqld --initialize --console 获取初始密码

mysql -hxxhost -uxxuser -pxxpwd -P3306 xxdbname < dbname.sql 连接 mysql，导入 sql（-h 使用 TCP/IP 协议连接 mysql，不会使用套接字 sock 连接）【不同版本的 mysql 命令密码加密方式会不同】
mysql -V 查看版本

mysqldump -hxxhost -uxxuser -pxxpwd -P3306 -databases xxdb --tables xxtb2 xxtb2 --add-drop-table > dbname.sql
默认不带参数的导出，导出文本内容大概如下：创建数据库判断语句-删除表-创建表-锁表-禁用索引-插入数据-启用索引-解锁表

【--databases xxdb 指定数据库 --all-databases 所有数据库 --set-gtid-purged=OFF 关闭 gtid】

【 --add-drop-database 建库语句前删库 --add-drop-table 建表语句前删表】
update mysql.user set password=password('123456') where user='root' 修改密码，5.7 之前
update mysql.user set authentication_string=password('123456') where user='root' 修改密码，5.7 开始（5.7 前，密码列为 password，5.7 后，密码列为 authentication_string）
alter user 'root'@'localhost' identified by '123456'; 修改密码，8.0 【% 表示所有主机，例如'pig'@'192.168.1.101'】

create user 'saidake'@'%' identified by '$#$#ff '; 添加远程登录用户，并授予权限。
grant all privileges on *.* to 'saidake'@'%'; 用户对所有数据库和表的相应操作权限 【 *.* 授予所有表的权限，可指定为 on xxdbname.xxtbname to ....】
update `user` set `Host` = '113.128.*.*' where `User` = 'labuladuo'; Host 和 User 修改

flush privileges; 刷新配置

mysqlbinlog

View Binary Log File

mysqlbinlog <binlog-file-path1> <binlog-file-path2> ...

Processes and displays MySQL binary logs. The binary logs record all changes to the MySQL database, including data modifications and DDL operations.

Parameters:

--start-position <position>: The position in the binary log to start reading from.
--stop-position <position>: The position in the binary log to stop reading at.
--start-datetime <datetime>: The start date and time for filtering logs.
--stop-datetime <datetime>: The end date and time for filtering logs.
--database <database>: The database name to filter logs for.

mysqlbinlog /path/to/binlog-file | mysql -u username -p

Restore Data from Binary Logs

Core SQL

Database

CREATE DATABASE IF NOT EXISTS mydb CHARACTER SET gbk;

Create a database

DROP DATABASE IF EXISTS mydb;

Drop a database (irreversible)

ALTER DATABASE mydb CHARACTER SET utf8;

Modify the character set of the xxdb database to utf8

```
SHOW CREATE DATABASE xxdb;
```
Show the SQL statement used to create the database

```
SHOW DATABASES;
```
List all databases

```
SELECT DATABASE();
```
Show the current selected database

```
USE mydatabase;
```
Switch to the specified database

```
SHOW ENGINES;
```
Show all supported storage engines

```
SHOW ENGINE INNODB STATUS;
```
Provides a snapshot of the InnoDB engine's internal state, including details about deadlocks, transactions, locks, buffer pool usage, and performance metrics.

Table

```
CREATE TABLE IF NOT EXISTS student (
    id INT PRIMARY KEY,
```
Primary key constraint (implicitly includes NOT NULL and UNIQUE)

```
        UNIQUE (id),
```
Unique constraint (automatically creates an index)

```
        NOT NULL,
```
Not NULL constraint

```
        AUTO_INCREMENT,
```
Auto-increment constraint (auto generates values for new rows)

```
        DEFAULT 29,
```
Default value constraint

```
        CHECK (id > 0 AND name = 'lala'),
```
Check constraint (enforces specific conditions on column values)

```
        COMMENT 'Student table',
```
Table comment

```
        UNSIGNED,
```
Unsigned values (for non-negative numbers)

```
        CHARACTER SET utf8,
```
Set character set for the column

```
        COLLATE utf8_general_ci,
```
Set collation for the column

```
    pid INT,
    CONSTRAINT fk_name FOREIGN KEY (pid) REFERENCES person(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
```
Foreign key constraint (cascade updates and deletes)

```
    KEY `shop_code_index` (`shop_code`) USING BTREE
```
Create index for `shop_code` using BTREE type

```
) ENGINE = MYISAM
```
Specify storage engine (MYISAM in this case)

AUTO_INCREMENT = 268
    Set starting value for auto-increment
DEFAULT CHARSET = utf8mb4
    Set default character set for the table
COLLATE = utf8mb4_general_ci
    Set collation for the table
ROW_FORMAT = DYNAMIC
    Specify row format for storage
COMMENT = 'Platform message table';
    Table comment

CREATE TABLE new_table_name AS SELECT * FROM old_table_name;
    Create a new table based on the structure and data of another table (temporary table)
    MySQL doesn't support SELECT INTO for both structure and data
DROP TABLE IF EXISTS student;
    Drop a table (irreversible)

ALTER TABLE student RENAME TO newstudent;
    Rename the table
ALTER TABLE student CHARACTER SET utf8;
    Modify the table's character set
ALTER TABLE student ENGINE = INNODB;
    Modify the table's storage engine

SHOW CREATE TABLE student;
    Show the table creation SQL
SHOW TABLES;
    Show all tables in the current database
SHOW TABLE STATUS FROM xxdb WHERE name = 'xxtab';
    Query detailed information about a specific table in a database
SHOW TABLE STATUS FROM mysql LIKE 'student';
    Query detailed information about a table using LIKE (wildcard search)
DESC student;
    Describe the structure of a table

Indexes

Indexes are essential components of databases that significantly enhance the performance of queries by allowing for faster retrieval of records.
In MySQL, various types of indexes can be used, each suited for different scenarios and types of queries.
Here's an in-depth look at the different types of indexes in MySQL and how they are used:

Primary Key Index

Ensures that each record in a table is unique and not null.

Details:
    Every table can have only one primary key.
    Automatically creates a unique index on the primary key column(s).

Often used for identifying records uniquely.

Create a table:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    -- Other columns
);
```

Alter table:

```
CREATE TABLE employees (
    employee_id INT NOT NULL,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    hire_date DATE
);
ALTER TABLE employees ADD CONSTRAINT pk_employee_id PRIMARY KEY (employee_id);
```

Unique Index

Ensures that all values in the indexed column(s) are unique.

Details:

Allows null values unless specified otherwise.

Can be applied to one or multiple columns.

Prevents duplicate values in the indexed columns.

Create a table:

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(255) NOT NULL,
    UNIQUE (email)
);
```

Assume you want to ensure that the combination of first_name and last_name is unique in the persons table.

```
CREATE TABLE persons (
    person_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE,
    UNIQUE (first_name, last_name)
);
```

Create Index

```
CREATE UNIQUE INDEX index_name ON table_name (column_name);
```

Alter table:

```
ALTER TABLE users ADD UNIQUE INDEX idx_firstname_lastname (first_name, last_name);
```

Secondary Index (or Non-Clustered Index)

In MySQL and other database systems, a "secondary index" is often referred to as a "regular index" or simply an "index."

Improves the speed of data retrieval operations.

Details:

Can be applied to one or more columns.

Does not enforce uniqueness.

Suitable for columns frequently used in WHERE clauses.

Example:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
```

```
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    department_id INT,
    INDEX idx_department (department_id)
);
CREATE INDEX index_name ON table_name (column_name);
```

Composite Index

Indexes multiple columns together.

Details:

Useful for queries involving multiple columns in WHERE clauses.

The order of columns in the index is crucial and affects query performance.

Can speed up queries that filter on multiple columns.

Match where clause

MySQL can use the composite index from left to right. This means that for the index to be used efficiently, the leftmost columns of the composite index must appear in the query's WHERE, ORDER BY, or GROUP BY clauses.

- Example Scenario

```
CREATE INDEX idx_comp ON table_name (column1, column2, column3);
SELECT * FROM table_name
WHERE column1 = 'value1'
  AND column2 > 'value2'
  AND column3 = 'value3';
```

Exact Match on column1:

MySQL uses the composite index idx_comp to quickly locate rows where column1 matches 'value1'.

Range Condition on column2:

After narrowing down to rows where column1 = 'value1', MySQL further filters based on the range condition column2 > 'value2'.

The index helps in efficiently accessing rows that satisfy this range condition.

Exact Match on column3:

Once rows matching column1 = 'value1' and column2 > 'value2' are identified, MySQL then applies the exact match condition on column3 = 'value3'.

The composite index facilitates quick lookup and retrieval of rows that meet all three conditions specified in the query.

- Prefix Searches and Sorting

In a composite index, MySQL requires that queries start with the leading column(s) of the index to effectively utilize the index.

Queries that use the leading columns of the composite index (column1, column2) as prefixes or involve sorting by these columns benefit from the index's ordered structure.

```
SELECT * FROM table_name WHERE column1 = 'value1' ORDER BY column2 DESC;
```

MySQL can efficiently use the composite index idx_comp to both filter and sort the results:

It uses the index to quickly locate and retrieve rows where column1 = 'value1'.

The ordered storage of column2 within the index allows MySQL to directly return the rows in the desired sorted order without additional sorting operations.

- Covering Index

If the composite index (column1, column2, column3) covers all columns required by a query (both in SELECT and WHERE clauses), it can act as a covering index.

This means MySQL can fulfill the query entirely using the index without needing to access the actual table data rows, thereby optimizing performance.

JSON Indexes

Json Indexes allows indexing on JSON (JavaScript Object Notation) data stored in JSON columns.

Details:

Introduced in MySQL 5.7, JSON indexes are used for indexing JSON data stored in JSON columns.

They enable efficient querying and searching of JSON documents within MySQL.

Example:

```
CREATE TABLE products (
    id INT PRIMARY KEY,
    product_info JSON,
    INDEX idx_product_info ((product_info->>'$.name'))
);
```

Full-Text Index

Optimizes full-text searches for columns with large text data.

Details:

Used for searching text within large datasets.

Supports natural language and Boolean text searches.

Only available for CHAR, VARCHAR, and TEXT columns.

Example:

```
CREATE TABLE articles (
    article_id INT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    body TEXT,
    FULLTEXT idx_fulltext (title, body)
);
CREATE FULLTEXT INDEX index_name ON table_name (column_name);
```

Spatial Index

Optimizes queries for spatial data types.

Details:

Used with spatial data types like GEOMETRY, POINT, LINESTRING, etc.

Enhances performance of spatial queries (e.g., location-based queries).

Example:

```
CREATE TABLE spatial_data (
    id INT PRIMARY KEY,
    location GEOMETRY,
    SPATIAL INDEX idx_location (location)
);

CREATE SPATIAL INDEX index_name ON table_name (spatial_column);
```

Structure

Clustered Index

Structure

A clustered index determines the physical order of data in a table.

The table rows themselves are stored in the order of the clustered index. There can only be one clustered index per table because the data rows can only be sorted in one order.

In MySQL, the storage engine that supports clustered indexes is primarily InnoDB.

Other storage engines in MySQL, such as MyISAM, MEMORY (HEAP), NDB Cluster (NDB), Archive, CSV, and others, do not support clustered indexes as implemented in InnoDB.

These engines have different indexing and storage mechanisms tailored to specific use cases, and they may not organize data in the same clustered manner as InnoDB.

Storage Engines

InnoDB

It supports clustered indexes where the primary key or a unique key determines the physical order of rows on disk.

Data rows in InnoDB tables are stored in primary key order within the clustered index, which can significantly improve query performance for operations that use the primary key or other indexed columns.

MyISAM

MyISAM is an older storage engine that does not support clustered indexes.

Instead, MyISAM uses separate files for data storage and index storage.

Data rows in MyISAM tables are stored in insertion order, and indexes are stored separately, leading to different performance characteristics compared to InnoDB.

MEMORY (HEAP):

The MEMORY storage engine stores all table data in memory.

It does not support clustered indexes because it does not persist data to disk.

Indexes in MEMORY tables are typically implemented as hash indexes or BTREE indexes, but they do not affect the physical storage order of rows because the data resides only in memory.

Characteristics

- B-tree Structure

  Clustered indexes are typically implemented using a B-tree (balanced tree) structure for efficient searching, insertion, and deletion.

- Physical Order

  Data rows are stored in the table in the order of the clustered index.

- Primary Key

  By default, InnoDB uses the primary key as the clustered index. If a table does not have a primary key, InnoDB will use the first unique key that does not contain NULL values.

  If no such key exists, InnoDB will create an internal clustered index.

- One Per Table

  Each table can have only one clustered index because the data rows can only be stored in one order.

B-Tree Indexes

Supported Storage Engines

InnoDB (Default in recent MySQL versions)

MyISAM (Deprecated but still supported)

NDB (Cluster)

Structure

B-trees organize index data hierarchically, allowing for efficient search, insert, and delete operations.

Unlike hash indexes, B-tree indexes support range scans (e.g., `WHERE col BETWEEN value1 AND value2`) and sorting operations efficiently.

Usage

Suitable for primary keys, unique keys, and non-unique indexes.

Hash Indexes

Supported Storage Engines:

MEMORY (HEAP)

Structure

Hash indexes use hash tables to directly map keys to their associated values, providing fast exact-match lookups.

They are particularly efficient for equality comparisons (e.g., WHERE col = value) but do not support range queries or sorting.

Limitations

Hash indexes do not support range queries, making them suitable for equality comparisons only.

Spatial Indexes

Structure

Spatial indexes are used for spatial data types such as points, lines, and polygons.

They support efficient querying of spatial relationships and are available in InnoDB and MyISAM storage engines.

Full-Text Indexes

Supported Storage Engines:

InnoDB

Usage

Designed for efficient searching of large text fields, enabling users to perform natural language queries against text data.

Constraints

Primary Key Constraint

Ensures uniqueness for each row in a table and provides a unique identifier for each record.

Usage: Typically used to uniquely identify rows and enforce entity integrity.

Example:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    -- Other columns
);
```

Unique Constraint

Ensures that all values in a column (or combination of columns) are unique across the table.

Usage: Prevents duplicate values in specific columns that require uniqueness but allows NULL values unless specified as NOT NULL.

Example:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    sku VARCHAR(50) UNIQUE,
    -- Other columns
);
```

Foreign Key Constraint

Maintains referential integrity by ensuring that values in a column (or group of columns) match values in another table's referenced column(s).

Usage: Establishes relationships between tables and enforces data consistency across related tables.

Example:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE NOT NULL,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

Check Constraint

Specifies a condition that must be true for each row in a table.

Usage: Validates the values entered into a column based on a specific condition or expression.

Example:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
```

```
    salary DECIMAL(10, 2),
    CONSTRAINT chk_salary CHECK (salary >= 25000)
);
```

NOT NULL Constraint

Specifies that a column cannot contain NULL values.

Usage: Ensures that every row must have a value for that column, preventing NULL entries in the specified column.

```
CREATE TABLE books (
    book_id INT PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    author VARCHAR(100) NOT NULL,
    publication_date DATE,
    CHECK (publication_date >= '2000-01-01')
);
```

Lock

Lock Types in InnoDB

Table Locks

Intended for administration purposes rather than for regular transaction control.

Example: `LOCK TABLES table_name READ/WRITE;`

READ LOCK

```
LOCK TABLES table_name READ;

-- The following query will be allowed
SELECT * FROM table_name WHERE column1 = 'value';

-- The following query will be blocked until the READ LOCK is released
INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');

-- To release the lock
UNLOCK TABLES;
```

WRITE LOCK

```
LOCK TABLES table_name WRITE;

-- The following query will be allowed
SELECT * FROM table_name WHERE column1 = 'value';

-- The following query will also be allowed
INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');

-- Any other session will be blocked from reading or writing to the table until the WRITE LOCK is
released

-- To release the lock
UNLOCK TABLES;
```

Row Locks

Shared (S) Lock

Multiple transactions can hold a shared lock on the same row, allowing them to read the row concurrently.

```
-- Shared lock on specific rows
SELECT * FROM table_name WHERE column1 = 'value1' LOCK IN SHARE MODE;

-- Other sessions can also acquire a shared lock on these rows
-- They can read but cannot write to these rows until the lock is released
```

Exclusive (X) Lock

Prevents other transactions from acquiring any lock on the row. Only the transaction holding the exclusive lock can read or write to the row.

```
-- Exclusive lock on specific rows
SELECT * FROM table_name WHERE column1 = 'value1' FOR UPDATE;

-- Other sessions will be blocked from reading or writing to these rows until the lock is released
```

Gap Locks

Prevent phantom reads by locking the gaps between rows.

Used primarily in REPEATABLE READ and SERIALIZABLE isolation levels.

```
-- Gap lock to prevent insertion into the gap between selected rows
SELECT * FROM table_name WHERE column1 BETWEEN 'value1' AND 'value2' FOR UPDATE;

-- Other sessions cannot insert rows with column1 values between 'value1' and 'value2' until the lock is
released
```

Next-Key Locks

Combination of row lock and gap lock.

Prevents other transactions from inserting rows into the gap covered by the lock.

```
-- Next-key lock prevents insertion of new rows in the gap covered by the lock
SELECT * FROM employees WHERE salary > 55000 FOR UPDATE;

-- This locks the row where column1 = 'value1' and the gap around it
```

Intent Locks

Intent Shared (IS) Lock

Indicates a transaction intends to acquire shared locks on specific rows.

```
-- Acquiring an intent shared lock
LOCK TABLES table_name READ;

-- Any row-level shared locks can be acquired within this table
SELECT * FROM table_name WHERE column1 = 'value1' LOCK IN SHARE MODE;

-- To release the lock
UNLOCK TABLES;
```

Intent Exclusive (IX) Lock

Indicates a transaction intends to acquire exclusive locks on specific rows.

```
-- Acquiring an intent exclusive lock
LOCK TABLES table_name WRITE;

-- Any row-level exclusive locks can be acquired within this table
SELECT * FROM table_name WHERE column1 = 'value1' FOR UPDATE;

-- To release the lock
UNLOCK TABLES;
```

Lock Types in MyISAM

Table Locks

Automatic Table Locking: MyISAM automatically acquires the necessary table-level locks based on the SQL operations being performed.

Concurrent Inserts: MyISAM supports concurrent inserts, allowing new rows to be inserted at the end of the table while reads are happening.

READ LOCK

Allows multiple transactions to read from the table concurrently but prevents any transaction from writing to the table.

```
LOCK TABLES table_name READ;
```

WRITE LOCK

Prevents other transactions from reading or writing to the table. Only the transaction holding the write lock can read from or write to the table.

```
LOCK TABLES table_name WRITE;
```

Lock Types in MEMORY

Automatic Table Locking: The MEMORY engine uses table-level locking like MyISAM.

High Performance for Reads/Writes: Due to data being stored in RAM, reads and writes are very fast, but concurrency is still managed via table locks.

Table Locks

READ LOCK: Allows multiple transactions to read from the table concurrently but prevents any transaction from writing to the table.

WRITE LOCK: Prevents other transactions from reading or writing to the table. Only the transaction holding the write lock can read from or write to the table.

Lock Types in NDB

Row Locks

Primary Key Locking:

NDB primarily uses row-level locking, especially on primary keys.

Explicit Row Locks:

Applications can request explicit locks on rows.

Transaction

Key Concepts of Transactions

Atomicity

Ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction is rolled back, and no changes are applied.

Consistency

Ensures that a transaction brings the database from one valid state to another, maintaining database invariants.

Isolation

Ensures that the operations of one transaction are isolated from others. Intermediate states of a transaction are invisible to other transactions.

Durability

Ensures that the results of a completed transaction are permanently stored in the database, even in the case of a system failure.

Isolation Levels

READ UNCOMMITTED

Transactions can see changes made by other transactions before they are committed.

Locks Used:

No Locking on Read:

At this isolation level, transactions do not use locks for reading data. As a result, transactions can see uncommitted changes made by other transactions, leading to dirty reads.

Row-Level Locks for Writes:

When performing write operations (e.g., INSERT, UPDATE, DELETE), MySQL uses row-level locks to prevent other transactions from modifying the same rows until the transaction commits.

Behavior:

Dirty Reads: Cannot be avoided. Transactions can see uncommitted changes made by other transactions.

Non-Repeatable Reads: Cannot be avoided. A transaction may see different values for the same row if it is updated by another transaction before the first transaction commits.

Phantom Reads: Cannot be avoided. A transaction may see rows that are inserted or deleted by other transactions before the first transaction commits.

READ COMMITTED

Transactions can only see changes made by other transactions after they are committed.

Adavantage

Improve concurrency.

In the process of locking, RC does not need to add GapLock and Next-KeyLock.

It only needs to add row-level locks to the records to be modified. Therefore, the concurrency supported by RC is

much higher than that of RR.

Reduce deadlocks.

Because the RR isolation level adds GapLock and Next-KeyLock locks, it is more likely to cause deadlocks than RC.

Locks Used:

Row-Level Locks on Read:

MySQL uses row-level locks for reading to ensure that a transaction only sees committed changes. This prevents dirty reads.

Row-Level Locks for Writes:

Similar to READ UNCOMMITTED, row-level locks are used during write operations.

Behavior:

Dirty Reads: Avoided. Transactions only see committed changes, preventing them from reading uncommitted changes from other transactions.

Non-Repeatable Reads: Possible. A transaction may see different values for the same row if it is updated by another transaction between reads within the same transaction.

Phantom Reads: Possible. A transaction may see different sets of rows if another transaction inserts or deletes rows that satisfy the query's WHERE clause between reads within the same transaction.

REPEATABLE READ (Default)

Ensures that if a transaction reads a row, it will see the same data throughout the transaction, even if other transactions modify the data.

Locks Used:

Row-Level Locks on Read:

MySQL uses row-level locks to ensure that once a row is read, the data remains consistent for the duration of the transaction.

Gap Locks:

MySQL also employs gap locks to prevent new rows from being inserted into ranges that are being read. This helps avoid non-repeatable reads.

Behavior:

Dirty Reads: Avoided. Transactions cannot read uncommitted changes from other transactions.

Non-Repeatable Reads: Avoided. Once a row is read within a transaction, it will see the same value throughout the transaction, even if the row is updated by another transaction.

Phantom Reads: Possible. A transaction may see new rows inserted by other transactions that satisfy the query's WHERE clause between reads within the same transaction.

SERIALIZABLE

Ensures complete isolation by serializing all transactions, making them appear to run sequentially.

Locks Used:

Row-Level Locks on Read:

Similar to REPEATABLE READ, row-level locks are used to ensure data consistency.

Gap Locks:

MySQL uses gap locks and next-key locks (a combination of row-level and gap locks) to prevent new rows from being inserted or existing rows from being modified within ranges being read.

Behavior:

Dirty Reads: Avoided. Transactions cannot read uncommitted changes from other transactions.

Non-Repeatable Reads: Avoided. Once a row is read within a transaction, it will see the same value throughout the transaction, even if the row is updated by another transaction.

Phantom Reads: Avoided. Serializable isolation ensures that no new rows that satisfy a query's WHERE clause can be inserted by other transactions between reads within the same transaction.

In Mysql, the default isolation level is **REPEATABLE READ**.

In Oracle Database, the default isolation level is **READ COMMITTED**.

Transaction Problems

Dirty Reads

: Occurs when a transaction reads data that has been modified by another transaction but not yet committed. This can lead to inconsistent data.

Non-Repeatable Reads

: Happens when a transaction reads the same row twice and gets different results each time because another transaction has modified the data in between.

Phantom Reads

: Occurs when a transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that also satisfy the condition, causing the first transaction to see a different set of rows upon re-execution of the query.

Change Isolation Level for the Current Session

```
-- Set the isolation level to READ UNCOMMITTED for the current session
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- Set the isolation level to READ COMMITTED for the current session
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Set the isolation level to REPEATABLE READ for the current session
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Set the isolation level to SERIALIZABLE for the current session
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Usage

```
-- Set isolation level to READ COMMITTED for the current session
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Start a transaction
START TRANSACTION;

-- Perform some operations
SELECT * FROM accounts WHERE balance > 1000;

-- Commit the transaction
COMMIT;
```

Change Isolation Level Globally

```
-- Set the global isolation level to READ UNCOMMITTED
SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- Set the global isolation level to READ COMMITTED
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Set the global isolation level to REPEATABLE READ
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Set the global isolation level to SERIALIZABLE
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Usage

```
-- Set the global isolation level to REPEATABLE READ
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Verify the change
SELECT @@GLOBAL.transaction_isolation;
```

Check the Current Isolation Level

For the Current Session

```
SELECT @@SESSION.tx_isolation;
SELECT @@SESSION.transaction_isolation;
```

For the Global Scope

```
SELECT @@GLOBAL.tx_isolation;
SELECT @@GLOBAL.transaction_isolation;
```

Column Operations

ALTER TABLE student ADD COLUMN price INT AFTER 'phone'; 添加一列， AFTER 表示添加在 phone 后方（oracle 没有 COLUMN）

ALTER TABLE student ADD COLUMN outname VARCHAR(20) GENERATED ALWAYS AS(xxjson->'$.name'); 拿出 data 数据的 name 创建新虚拟字段 user_name（再为这个字段添加索引）

ALTER TABLE student DROP COLUMN price; 删除一列

ALTER TABLE student MODIFY price INT AUTO_INCREMENT; 修改列 price 的类型为 INT，并可以添加约束

ALTER TABLE student CHANGE price address VARCHAR(200) 修改列 price 为新列名 address

Foreign Key

ALTER TABLE student ADD CONSTRAINT fk_name FOREIGN KEY (pid) REFERENCES person (id) ON UPDATE CASCADE ON DELETE CASCADE; 添加外键，同时添加级联更新和级联删除

ALTER TABLE student DROP CONSTRAINT fk_name; 删除外键（同时还需要删除关联的外键索引）

```
SELECT TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME, REFERENCED_TABLE_NAME, REFERENCED_COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE  -- 查询外键
    WHERE  CONSTRAINT_SCHEMA = xxdb'
           AND TABLE_NAME = 'student'
           AND COLUMN_NAME = 'origin_fk';
           AND REFERENCED_TABLE_NAME = 'person_fk';          -- 指向表名称
```

Performance Analysis

Commands

EXPLAIN

```
EXPLAIN SELECT * FROM table_name;
```

Analyzes the execution plan of a query, helping to understand how MySQL processes the query, including index usage and table scan methods.

Importance: Identifies performance bottlenecks in query execution, allowing for informed optimization decisions.

SHOW PROFILE

SELECT @@profiling

Check profiling support

SET profiling = 1

Enable profiling

SHOW PROFILES

View recent query performance

Provides detailed performance statistics for a specific query or session, including time and resource usage for each query stage.

Offers in-depth analysis data, critical for pinpointing performance issues

SHOW PROFILE FOR QUERY query_id

View detailed stats for a specific query

SHOW STATUS

SHOW STATUS LIKE 'Connections';

Displays various server status metrics, such as the number of connections, queries processed, and cache hit rates.

Provides real-time server performance data, essential for ongoing performance tuning and monitoring.

SHOW VARIABLES

SHOW VARIABLES LIKE 'max_connections';

Retrieves the current configuration parameters of the MySQL server, including performance-related settings. Understanding server configuration is vital for optimizing performance, and this command helps identify settings that might need adjustment.

SHOW PROCESSLIST

Lists all current connections and executing queries, useful for spotting long-running queries or lock issues.

SHOW CREATE TABLE

Displays the CREATE TABLE statement for a table, helping to understand its structure and indexes.

SHOW INDEXES

Lists all indexes on a table, allowing you to evaluate index usage and optimize indexing strategies.

Functional SQL
PREPARE

PREPARE my_statement FROM 'select * from user where id=?'; mysql 语法，定义一个预编译语句

SET @id=1;

EXECUTE my_statement USING @id; mysql 语法，执行预编译

Lock

INNODB 锁

SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE; 行级读锁/表级读锁 （INNODB 共享锁，可以同时被其他事务加共享锁。其他事务可以查询，但是在共享锁提交前不能修改）

INNODB 默认加行锁，如果是不带索引的列加锁，加的就是表锁。

SELECT * FROM student WHERE id=1 FOR UPDATE; 等待行锁释放之后，返回查询结果（INNODB 排他锁，与共享锁和排他锁都不兼容。其他事务不能查询或修改）

FOR UPDATE NOWAIT 不等待行锁释放，提示锁冲突，不返回结果

FOR UPDATE WAIT 5 等待 5 秒，若行锁仍未释放，则提示锁冲突，不返回结果

FOR UPDATE SKIP LOCKED 忽略有行锁的记录，直接返回查询结果

锁查询

SELECT * FROM t for update 会等待行锁释放之后，返回查询结果。

SELECT * FROM t for update nowait 不等待行锁释放，提示锁冲突，不返回结果

SELECT * FROM t for update wait 5 等待 5 秒，若行锁仍未释放，则提示锁冲突，不返回结果

SELECT * FROM t for update skip locked 查询返回查询结果，但忽略有行锁的记录

MYISAM 锁

LOCK TABLE student READ; 添加 MYISAM 读锁（当前事务查询数据，其他事务只能查询数据，不能修改）

UNLOCK TABLES; 解除 MYISAM 读锁

LOCK TABLE sdudent WRITE; 添加 MYISAM 写锁（当前事务修改数据，其他事务不能查询和修改数据）

UNLOCK TABLES; 解除 MYISAM 写锁

查询

SELECT @@AUTOCOMMIT; 查看事务提交方式（默认自动提交，可以修改为手动提交）

SET @@AUTOCOMMIT=1; 修改事务提交方式（0 为手动，1 为自动）

SELECT @@TX_ISOLATION; 查看数据库隔离级别
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED; 修改数据库隔离级别（需要重新连接）
READ UNCOMMITTED 读未提交 READ COMMITTED 读已提交 REPEATABLE READ 可重复读 SERIALIZABLE 串行化

SELECT * FROM information_schema.INNODB_TRX; 查询 正在执行的事务（trx_mysql_thread_id 事务线程 id）
SELECT * FROM information_schema.INNODB_LOCKS; 查询死锁表 【mysql<8.0】
SELECT * FROM information_schema.INNODB_LOCK_waits; 查询死锁等待时间 【mysql<8.0】
SELECT * FROM performance_schema.data_locks; 查询死锁表 【mysql>=8.0】
SELECT * FROM performance_schema.data_lock_waits; 查询死锁等待时间 【mysql>=8.0】

SHOW OPEN TABLES WHERE in_use>0; 查询是否锁表
SHOW PROCESSLIST; 查看进程

SELECT @@global.sql_mode; 设置 sql 查询模式
SET
@@global.sql_mode='STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';

PROCEDURE

Creating a Stored Procedure

To create a stored procedure in MySQL, you use the CREATE PROCEDURE statement.

DELIMITER: Changes the default delimiter (semicolon) to another string to allow for the definition of the procedure without executing it prematurely.
IN: Input parameter.
OUT: Output parameter.
INOUT: Input-output parameter.
BEGIN ... END: Block that contains the SQL statements for the procedure.

Definition

```
DELIMITER $$

CREATE PROCEDURE procedure_name (IN parameter_name datatype, OUT parameter_name datatype, INOUT
parameter_name datatype)
BEGIN
    -- SQL statements
END$$
DELIMITER ;
```

Usage

```
DELIMITER $$

CREATE PROCEDURE GetEmployeeName (IN emp_id INT, OUT emp_name VARCHAR(100))
BEGIN
    SELECT name INTO emp_name
    FROM employees
    WHERE id = emp_id;

    INSERT INTO TableName (Column1, Column2)
    SELECT Value1, Value2
    FROM Dual; -- In some databases like MySQL, use 'Dual' as a placeholder table.
END$$
```

```
DELIMITER ;
```

Executing a Stored Procedure

To execute a stored procedure, you use the CALL statement:

Usage

```
CALL procedure_name(parameters);
```

```
SET @emp_id = 1;
CALL GetEmployeeName(@emp_id, @emp_name);
SELECT @emp_name;
```

Managing Stored Procedures

To list all stored procedures in your database:

```
SHOW PROCEDURE STATUS WHERE Db = 'your_database_name';
```

To view the code of a stored procedure:

```
SHOW CREATE PROCEDURE procedure_name;
```

To drop (delete) a stored procedure:

```
DROP PROCEDURE IF EXISTS procedure_name;
```

CURSOR

In MySQL, a cursor is a database object used to retrieve and manipulate rows from a result set one row at a time.
This is particularly useful for processing each row individually within a stored procedure.

Key Points about Cursors

- Declaration — Cursors must be declared within a stored procedure.
- Opening — A cursor must be opened before it can be used to fetch rows.
- Fetching — Rows can be fetched one at a time from the cursor.
- Closing — A cursor must be closed when it is no longer needed to free up resources.

Performance Concerns

- Row-by-Row Processing
  - Cursors process each row individually, which can be significantly slower than set-based operations that process all rows in a single statement.
- Increased Resource Usage
  - Cursors can consume more memory and CPU resources, especially when handling large result sets.
  - This can lead to resource contention and degraded performance.
- Locking Issues
  - Cursors can hold locks on rows, impacting concurrency and potentially leading to deadlocks or blocking other transactions.

Usage

```
DELIMITER $$

CREATE PROCEDURE ProcessEmployees()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE emp_id INT;
    DECLARE emp_name VARCHAR(100);
    DECLARE emp_salary DECIMAL(10,2);

    -- Declare the cursor for employee data
    DECLARE emp_cursor CURSOR FOR
    SELECT id, name, salary FROM employees;

    -- Declare a handler to set the done variable to TRUE when there are no more rows to fetch
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
```

```
    -- Open the cursor
    OPEN emp_cursor;

    -- Fetch rows from the cursor
    read_loop: LOOP
        FETCH emp_cursor INTO emp_id, emp_name, emp_salary;

        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Process each row
        -- For example, just printing the employee details (this would typically be more complex logic)
        SELECT CONCAT('Employee ID: ', emp_id, ', Name: ', emp_name, ', Salary: ', emp_salary) AS
EmployeeDetails;
    END LOOP;

    -- Close the cursor
    CLOSE emp_cursor;
END$$

DELIMITER ;
```

FUNCTION

```
DELIMITER $
CREATE FUNCTION func(xxa INT, xxb VARCHAR(10))
RETURNS INT
    The number of return values must equal the number of return values required by the RETURN operator.
        SELECT IFNULL(COUNT(DISTINCT user_id), 0) AS user_cnt
BEGIN
    DECLARE count INT;
    SELECT COUNT(*) INTO count FROM student WHERE score > 75;
    RETURN count;
END$
DELIMITER;


SELECT func(3, 'xxx');          Using the Function
DROP FUNCTION func;             Dropping a Function

SHOW CREATE FUNCTION func;
SHOW FUNCTION STATUS LIKE 'func' ;
```

TRIGGER

```
DELIMITER $
CREATE TRIGGER account_insert          创建触发器（可以在 insert、update、delete 之前或之后触发并执行触发器中定义的
SQL 语句.)
AFTER INSERT                插入后触发
    BEFORE INSERT           插入前触发（old 无,                    new 表示将要或已经新增的数据）
    BEFORE UPDATE           更新前触发（old 表示修改前的数据       new 表示将要或已经修改后的数据）
    BEFORE DELETE           删除前触发（old 表示将要或已经删除的数据   new 无）
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL,'INSERT', NOW(), new.id, CONCAT('xx{id=',new.id, ',name=', new.name, '}'))
```

触发器要执行的功能
END$

SHOW TRIGGERS; 查看触发器
DROP TRIGGER account_insert; 删除触发器

VIEW

CREATE VIEW city_country (city_id, city_name, country_name) AS 创建视图，是一种虚拟存在的数据表,这个虚拟表并不在数据库中实际存在。
SELECT id, name, age FROM student WHERE id >4; 作用：将一些较为复杂的查询语句的结果,封装到一个虚拟表中,后期再有相同需求时,直接查询该虚拟表即可。

DROP VIEW IF EXISTS city_country; 删除视图
UPDATE city_country SET city_name WHERE city_name='北京'; 修改视图数据语法
ALTER VIEW city_country(city_id, city_name, name) AS 修改视图结构
SELECT id, name, age FROM student WHERE id >4;
SELECT * FROM city_country; 查询视图

Build-in tables

数据库中所有表 SELECT `table_name`, table_comment FROM information_schema.tables WHERE table_schema=(select database());
表中所有列 SELECT `column_name`, is_nullable, data_type, column_comment FROM information_schema.columns WHERE table_name='t_user' AND table_schema=(select database());

查询存在的数据库名 SELECT schema_name FROM information_schema.schemata;（返回一条字段schema_name）
show master logs; 显示最新的 binlog 日志信息
show master status; 最新的一条记录状态
show binlog events in 'row.000004' ; 查看 position 的事件

Permissions

mysql.user
This table contains information about MySQL user accounts and their global privileges.
Columns:
Host: The host from which the user can connect.
User: The username.
Select_priv, Insert_priv, Update_priv, etc.: Columns that store the global privileges for the user.
mysql.db
This table holds information about database-level privileges for users.
Columns:
Host: The host from which the user can connect.
Db: The database name.
User: The username.
Select_priv, Insert_priv, Update_priv, etc.: Columns that store privileges for the specific database.
mysql.roles_mapping
This table maps roles to users.
Columns:
Host: The host from which the user can connect.

User: The username.
Role_host: The host from which the role is granted.
Role_user: The role assigned to the user.

mysql.tables_priv

This table contains table-level privileges for users.

Columns:

Host: The host from which the user can connect.
Db: The database name.
User: The username.
Table_name: The name of the table.
Column_priv, Select_priv, Insert_priv, etc.: Columns that store privileges for the specific table.

mysql.columns_priv

This table holds column-level privileges for users.

Columns:

Host: The host from which the user can connect.
Db: The database name.
User: The username.
Table_name: The name of the table.
Column_name: The name of the column.
Column_priv: Columns that store privileges for the specific column.

mysql.procs_priv

This table stores privileges related to stored procedures and functions.

Columns:

Host: The host from which the user can connect.
Db: The database name.
User: The username.
Proc_priv: Privileges related to stored procedures.

DAM / Oracle

Core

Oracle Database is a multi-model database management system produced and marketed by Oracle Corporation.
It is widely used for running online transaction processing (OLTP), data warehousing (DW), and mixed (OLTP & DW) database workloads.

Components

An Oracle instance is a combination of the background processes and memory structures required to manage database files.
Each running Oracle database is associated with an Oracle instance.

Oracle SID

The Oracle SID (System Identifier) is a unique identifier for an Oracle database instance. It distinguishes each database instance on a single machine, allowing multiple Oracle databases to run on the same server without conflict.

The SID (System Identifier) of an Oracle database instance cannot be changed directly after the database has been created.

CDB and PDB

CDB

A Container Database (CDB) is a core component of Oracle's multitenant architecture, introduced in Oracle 12c.
It serves as the root database that can house multiple Pluggable Databases (PDBs).

The CDB is the root database that holds the system-level data and metadata.
It contains the root container, which includes the data dictionary and other system-level information, as well as one or more pluggable databases (PDBs).

PDB

A Pluggable Database (PDB) is a key concept in Oracle's multitenant architecture introduced in Oracle 12c.
It allows a single Oracle database (referred to as a Container Database (CDB)) to host multiple databases in an isolated, pluggable manner.
A PDB is a portable, self-contained database that can be plugged into or unplugged from a CDB.
It contains user data, schema objects, and application data. PDBs are fully functional databases but share common infrastructure with other PDBs in the same CDB.

Global Database Name (GDN):

The Global Database Name is a unique name for the entire Container Database (CDB). It consists of two parts:

Database Name: This is the base name of your database (e.g., orcl).
Domain Name: This typically represents the network domain and is usually the same as your hostname or domain (e.g., example.com).

Example:

<Database_Name>.<Domain_Name>
orcl.example.com

Pluggable Database Name (PDB Name)

In Oracle Multitenant Architecture, the Pluggable Database (PDB) is a self-contained database within the Container Database (CDB).

Example:

pdb1, sales_pdb, hr_pdb

Structure

Memory Structures

The main memory structures are the System Global Area (SGA) and Program Global Area (PGA).

- SGA

  Shared by all server and background processes, it contains data and control information for the Oracle instance.
  Key components include:

  - Database Buffer Cache

    Stores copies of data blocks read from the data files.
  - Redo Log Buffer

    Holds log entries before they are written to the redo log files.
  - Shared Pool

    Caches various types of data, including SQL, PL/SQL, and data dictionary information.
  - Large Pool

    Optional memory area that relieves the burden on the shared pool by allocating memory for large operations.
  - Java Pool

    Used for all session-specific Java code and data within the JVM.
  - Streams Pool

    Used by Oracle Streams.

- PGA
  Memory allocated to a server process or background process, used for session-specific information.

Background Processes

Oracle instance uses several background processes, including:

- DBWn (Database Writer) Writes modified blocks from the database buffer cache to the data files.
- LGWR (Log Writer) Writes redo log entries from the redo log buffer to the redo log files.
- CKPT (Checkpoint Process) Updates the data file headers and control files to indicate the checkpoint.
- SMON (System Monitor) Performs instance recovery at startup.
- PMON (Process Monitor) Cleans up failed user processes.
- ARCn (Archiver Processes) Copies redo log files to a designated storage location.
- RECO (Recoverer Process) Resolves distributed transactions that are pending due to a failure.

Index Data Structures

Oracle uses different index data structures for different types of data to optimize query performance.
The choice of index data structure depends on the characteristics of the data and the types of queries that will be executed.

- B-tree indexes
  Used for most data types, including numbers, strings, and dates. They are efficient for both sequential and random access.
- Bitmap indexes
  Ideal for columns with a limited number of distinct values, such as Boolean columns or columns representing categories.
- Function-based indexes
  Used for indexing calculated or derived values, such as expressions or functions.
- Domain indexes
  Used for custom data types that require specialized indexing.
- Text indexes
  Used for full-text search on textual data.
- Spatial indexes
  Used for spatial data, such as geographic locations.
- XML indexes
  Used for XML data.

Oracle's optimizer analyzes queries and selects the most appropriate index data structure based on factors such as the number of rows, data distribution, and query complexity.
By using the right index data structure, Oracle can significantly improve query performance and reduce the overall database load.
It's important to note that creating too many indexes can also have a negative impact on performance, as it can increase the overhead of database operations.

Database Files

Oracle Database consists of three types of files:
Data Files:
Store the user and system data. These files are organized into tablespaces.
Redo Log Files:
Record all changes made to the database. They are crucial for recovery.
Control Files:
Contain metadata about the database and are essential for database startup.

Logical Structures

Tablespaces: Logical storage units that group related data. Each tablespace consists of one or more data files.

Schema Objects: The logical structures that directly refer to the database's data, such as tables, indexes, views, sequences, and stored procedures.

Physical Structures

Data Files: Physical files on disk that store the database's data.

Redo Log Files: Physical files on disk that log changes made to the database.

Control Files: Physical files that record the structure of the database.

Table

Data Types

Numeric Data Types

NUMBER(P, S) Stores fixed and floating-point numbers. P is the precision (total number of digits), and S is the scale (number of digits to the right of the decimal point).

```
CREATE TABLE example_number (
    id NUMBER(5),       -- Fixed-point
    salary NUMBER(7, 2) -- Floating-point
);
```

BINARY_FLOAT 32-bit single-precision floating-point number.

BINARY_DOUBLE 64-bit double-precision floating-point number.

Character Data Types

CHAR(N) Fixed-length character data. Maximum size is 2000 bytes. N represents the length of the string.

VARCHAR2(N) Variable-length character data. Maximum size is 4000 bytes. N represents the maximum length of the string.

NCHAR(N) Fixed-length national character set data. Maximum size is 2000 bytes. N represents the length of the string.

NVARCHAR2(N) Variable-length national character set data. Maximum size is 4000 bytes. N represents the maximum length of the string.

Date and Time Data Types

DATE Stores date and time values (year, month, day, hour, minute, second).

TIMESTAMP Stores date and time values with fractional seconds. Optional time zone.

```
CREATE TABLE example_timestamp (
    event_time TIMESTAMP,
    event_time_tz TIMESTAMP WITH TIME ZONE,
    event_time_ltz TIMESTAMP WITH LOCAL TIME ZONE
);
```

INTERVAL YEAR TO MONTH Stores a period of time in years and months.

```
CREATE TABLE example_interval_ym (
    duration INTERVAL YEAR TO MONTH
);
```

INTERVAL DAY TO SECOND Stores a period of time in days, hours, minutes, and seconds.

```
CREATE TABLE example_interval_ds (
    duration INTERVAL DAY TO SECOND
);
```

Large Object (LOB) Data Types

CLOB Character large object. Stores large amounts of character data.

NCLOB National character set large object. Stores large amounts of national character set data.

BLOB Binary large object. Stores large amounts of binary data.

BFILE External binary file. Stores a locator to a binary file stored outside the database.

Raw Data Types

RAW(N) Stores variable-length binary data. Maximum size is 2000 bytes. N represents the length in bytes.
LONG RAW Stores variable-length binary data. Maximum size is 2GB.
Row Identifier Data Types
ROWID Stores the unique address (rowid) of a row in a table.
UROWID Stores the logical address of a row (universal rowid).
XML Data Types
XMLTYPE Stores and manipulates XML data.

Command

```
CREATE TABLE STUDENT(
    ID NUMBER CONSTRAINT FK_XXX
        PRIMARY KEY                          Primary key constraint, default includes two functions: non null and
        unique. A table can only have one primary key
        REFERENCES PERSON(PER_ID),           Foreign key constraint
        UNIQUE,                              Unique constraint
        NOT NULL                             NOT NULL constraint

        CHECK (id>0 AND name="lala")         True value constraint, limiting the values of some columns
        DEFAULT 29                           Default value constraint (remove constraint by modifying column to
        DEFAULT NULL, DEFAULT SYSTIMESTAMP default time)

    PID NUMBER,
    CONSTRAINT constraint_name PRIMARY KEY (column1, column2, ... columnN),
    CONSTRAINT constraint_name FOREIGN KEY (column1, column2, ... columnN) REFERENCES parent_table (column1,
    column2, ... columnN)
);

DROP TABLE STUDENT;                                  删除表（不可逆）
ALTER TABLE STUDENT RENAME TO TEST_STUDENT;          修改表名

select dbms_metadata.get_ddl('TABLE','xxtable') from dual;      查看建表语句

alter table TEST_PERSON drop primary key;                       删除主键，自动删除主键约束
alter table TEST_PERSON add primary key (PER_ID,NAME);          重建主键，自动添加约束

alter table 'TEST_PERSON' drop constraint 'SYS_C008483';        删除约束


- 定义存储过程
create or replace procedure proc_droptable(
    p_table in varchar2
) is
    v_count number(10);
begin
   select count(*) into v_count from user_tables
```

```
    where table_name = upper(p_table);

    if v_count > 0 then
        execute immediate 'drop table ' || p_table ||' purge';
    end if;

end proc_droptable;

-- 调用执行过程
exec proc_droptable('tableName');
```

Column

```
ALTER TABLE STUDENT ADD ( REMARK VARCHAR2(20), OUTDATE DATE )    添加列
ALTER TABLE STUDENT MODIFY ( REMARK CHAR(20), OUTDATE DATE )     修改列
ALTER TABLE STUDENT RENAME COLUMN REMARK TO NEW_REMARD           修改列名
ALTER TABLE STUDENT DROP COLUMN REMARK, OUTDATE                  删除列
```

操作外键

```
ALTER TABLE student ADD CONSTRAINT fk_name FOREIGN KEY (pid)  REFERENCES person (id) ON UPDATE CASCADE ON DELETE CASCADE;  添加外键，同时添加级联更新和级联删除
ALTER TABLE student DROP CONSTRAINT fk_name;                                                              删除外键（同时还需要删除关联的外键索引）
```

操作索引

```
DROP INDEX indexname;

CREATE INDEX indx_user_info ON user_info(user_id, user_name)                default index
        tablespace TBS_MY_INDEX
        pctfree 10
        initrans 2
        maxtrans 255;


CREATE INDEX indx_user_info ON user_info(user_id) online                    online index
        tablespace TBS_MY_INDEX;
```

Lock

Update

Row-Level Locks

Oracle uses row-level locks to prevent concurrent transactions from modifying the same row simultaneously. This is crucial for maintaining data integrity.

Table-Level Locks

While Oracle predominantly uses row-level locking, it also uses table-level locks in specific scenarios, such as when altering table structures or in certain DDL operations.

Configuration

Deadlock Handling

Oracle provides initialization parameters that can influence deadlock detection and resolution behavior.
These parameters are typically set in the INIT.ORA file or dynamically using ALTER SYSTEM commands. Some relevant parameters include:

deadlock_detection_time

Specifies the interval in seconds between deadlock detection checks.

To set the deadlock detection interval to 10 seconds:

```
ALTER SYSTEM SET deadlock_detection_time = 10;
```

transactions_per_rollback_segment

Controls the number of transactions that can concurrently use a rollback segment, affecting rollback behavior during deadlock resolution.

To set the number of transactions per rollback segment to 16:

```
ALTER SYSTEM SET transactions_per_rollback_segment = 16;
```

undo_management

Determines how undo (rollback) segments are managed, which can impact transaction rollback operations during deadlock handling.

To set undo management to automatic:

```
ALTER SYSTEM SET undo_management = AUTO;
```

To set undo management to manual:

```
ALTER SYSTEM SET undo_management = MANUAL;
```

Types of Locks

Exclusive Lock (X)

Prevents other transactions from modifying the locked resource until the lock is released.
Can be applied to either the entire table or specific rows, depending on the need.
Example: Acquired by INSERT, UPDATE, DELETE statements.

```
-- Example of acquiring an exclusive lock
UPDATE employees SET salary = salary + 1000 WHERE employee_id = 101 FOR UPDATE;
```

Share Lock (S)

Allows multiple transactions to read the locked resource concurrently but prevents modifications.
Always locks the entire table.
Example: Acquired by SELECT statements in FOR SHARE or LOCK IN SHARE MODE clauses.

```
-- Example of acquiring a shared lock
SELECT * FROM employees WHERE department_id = 30 FOR SHARE;
```

Row-Level Lock

Locks specific rows of a table to prevent conflicting operations on the same rows by concurrent transactions.
Example: Implicitly acquired by UPDATE or DELETE statements when modifying specific rows.

```
-- Example of row-level locking
DELETE FROM employees WHERE employee_id = 102;
```

Table-Level Lock

Locks entire tables to prevent conflicting operations that affect the entire table.
Example: Acquired explicitly using LOCK TABLE statement for schema changes or bulk operations.

```
-- Example of table-level locking
LOCK TABLE employees IN EXCLUSIVE MODE;
```

Monitoring Locks

Lock Monitoring Views

Oracle provides views such as V$LOCK and V$LOCKED_OBJECT to monitor active locks and locked objects in the database.

```
-- Query to monitor active locks
```

```
SELECT session_id, type, mode_held, mode_requested, lock_id1 FROM v$lock WHERE block = 1;
```

Deadlock Handling

Oracle automatically detects deadlocks and resolves them by rolling back one of the transactions involved.

Detecting Deadlocks

Oracle uses a detection algorithm to identify deadlock situations.

A deadlock occurs when two or more transactions are waiting for resources held by each other, forming a circular dependency.

The deadlock detection mechanism in Oracle continuously monitors resource requests and detects potential deadlocks based on wait-for graphs.

Resolving Deadlocks

Selecting Victim Transaction:

Oracle selects one of the involved transactions as the victim to break the deadlock cycle.

The selection is typically based on criteria such as transaction complexity or elapsed time since the transaction started.

Rolling Back the Victim:

Oracle automatically rolls back the victim transaction to release the resources it holds. This action breaks the circular dependency and allows the remaining transactions to proceed.

Error Messaging:

Upon detecting and resolving a deadlock, Oracle raises an error (ORA-00060) indicating the deadlock condition.

This error message is logged in the database alert log and can be captured programmatically by applications for handling and retrying transactions if necessary.

Transaction

Key Concepts of Transactions

Atomicity

Ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction is rolled back, and no changes are applied.

Consistency

Ensures that a transaction brings the database from one valid state to another, maintaining database invariants.

Isolation

Ensures that the operations of one transaction are isolated from others. Intermediate states of a transaction are invisible to other transactions.

Durability

Ensures that the results of a completed transaction are permanently stored in the database, even in the case of a system failure.

Isolation Levels

READ COMMITTED (Default)

Behavior:

Each query within a transaction sees only the data that was committed before the query began.

Uncommitted changes made by other transactions are not visible.

Locks are released as soon as the data is read or written.

Use Case:

Suitable for most applications where consistency is required but where complete isolation is not necessary.

Dirty Reads:	Avoided. Transactions cannot read uncommitted changes from other transactions.
Non-Repeatable Reads:	Possible. A row read in one part of the transaction can change if another transaction updates and commits the row before the current transaction reads it again.
Phantom Reads:	Possible. A transaction may see new rows inserted by other transactions that satisfy the query's

WHERE clause between reads within the same transaction.

Example:

```
-- Set the isolation level for the session
ALTER SESSION SET ISOLATION_LEVEL READ COMMITTED;

-- Perform a transaction
BEGIN
  UPDATE employees
  SET salary = salary + 1000
  WHERE employee_id = 101;
  COMMIT;
END;
```

Read-Only

Behavior:

The transaction is not allowed to perform any data modification operations (INSERT, UPDATE, DELETE).

Queries within the transaction see a consistent snapshot of the database as of the start of the transaction.

Use Case:

Suitable for reporting and analytical queries where data modifications are not needed.

Dirty Reads: Avoided. Transactions cannot read uncommitted changes from other transactions.

Non-Repeatable Reads: Avoided. Once a row is read within a transaction, it will see the same value throughout the transaction, even if the row is updated by another transaction.

Phantom Reads: Avoided. Transactions see a consistent set of rows that match the query conditions throughout the transaction, even if other transactions insert or delete rows.

Example:

```
-- Set the isolation level for the session
ALTER SESSION SET ISOLATION_LEVEL READ ONLY;

-- Perform a read-only transaction
BEGIN
  SELECT employee_id, salary
  FROM employees
  WHERE department_id = 10;
END;
```

SERIALIZABLE

Ensures complete isolation by serializing all transactions, making them appear to run sequentially.

Behavior:

Transactions are completely isolated from each other.

Queries within a transaction see a consistent snapshot of the database as of the start of the transaction.

Ensures that transactions execute as if they were serialized, one after the other, rather than concurrently.

Use Case:

Suitable for applications that require high consistency and can tolerate reduced concurrency.

Dirty Reads: Avoided. Transactions cannot read uncommitted changes from other transactions.

Non-Repeatable Reads: Avoided. Once a row is read within a transaction, it will see the same value throughout the transaction, even if the row is updated by another transaction.

Phantom Reads: Avoided. Transactions see a consistent set of rows that match the query conditions throughout the transaction, even if other transactions insert or delete rows.

Example:

```
-- Set the isolation level for the session
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;
```

```
-- Perform a transaction
BEGIN
  UPDATE employees
  SET salary = salary + 1000
  WHERE employee_id = 101;
  COMMIT;
END;
```

Transaction Problems

Dirty Reads

Occurs when a transaction reads data that has been modified by another transaction but not yet committed. This can lead to inconsistent data.

Non-Repeatable Reads

Happens when a transaction reads the same row twice and gets different results each time because another transaction has modified the data in between.

Phantom Reads

Occurs when a transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that also satisfy the condition, causing the first transaction to see a different set of rows upon re-execution of the query.

Table Space

A tablespace in Oracle is a logical container for segments (such as tables, indexes, and large objects) and is used to manage how data is physically stored on disk.

Types of Tablespaces

Oracle provides several types of tablespaces to cater to different storage needs:

SYSTEM and SYSAUX Tablespaces

SYSTEM Tablespace

This is a mandatory tablespace that contains the data dictionary, which is a set of tables and views that Oracle uses to manage the database.

SYSAUX Tablespace

This auxiliary tablespace supports the SYSTEM tablespace and contains various Oracle components and features, reducing the load on the SYSTEM tablespace.

Bigfile and Smallfile Tablespaces

Bigfile Tablespace

Contains a single, very large data file (up to 8 exabytes), making management of very large databases easier.

Single Large Data File:

A Bigfile tablespace consists of a single, large data file, allowing it to grow up to 4 exabytes (EB) in size.

Utilizes the maximum capacity supported by the underlying file system, reducing the need for managing multiple data files.

Management:

Simplifies administrative tasks by reducing the number of data files per tablespace, which can streamline backup, recovery, and storage management operations.

Reduces potential I/O contention that may occur with multiple data files in Smallfile tablespaces.

Performance:

Can offer better performance for large-scale operations due to reduced administrative and I/O overheads.

Ideal for very large databases or applications requiring extensive storage capabilities within a single tablespace.

Example:

```
CREATE BIGFILE TABLESPACE bigfile_tbs
DATAFILE '/path/to/bigfile_tbs01.dbf' SIZE 1T
AUTOEXTEND ON NEXT 100M MAXSIZE UNLIMITED
```

```
EXTENT MANAGEMENT LOCAL;
```

Smallfile Tablespace

Contains multiple data files, each of which can be up to 32 terabytes in size.

File Size Limitations:

Smallfile tablespaces consist of multiple data files, each limited in size based on the database block size and operating system file size limits.

Typical maximum size per data file is several gigabytes (e.g., up to 32GB on some operating systems).

Number of Data Files:

Each smallfile tablespace can accommodate up to 1022 data files, providing flexibility in spreading storage across multiple files.

Management:

Requires managing multiple data files per tablespace, which can increase administrative overhead compared to Bigfile tablespaces.

Suitable for general-purpose databases where individual data files do not need to be excessively large.

Example

```
CREATE TABLESPACE smallfile_tbs
DATAFILE '/path/to/smallfile_tbs01.dbf' SIZE 100M
AUTOEXTEND ON NEXT 10M MAXSIZE 1000M
EXTENT MANAGEMENT LOCAL;
```

User Tablespaces

Default Tablespace:

When users or applications create objects without specifying a tablespace, they are stored in this designated default tablespace.

- Purpose

  To provide a default location for database objects like tables and indexes.

- Configuration

  Can be set at the database level or user level.

Command to Set Default Tablespace

```
ALTER DATABASE DEFAULT TABLESPACE default_tbs;
ALTER USER username DEFAULT TABLESPACE user_default_tbs;
```

Example

```
CREATE USER example_user IDENTIFIED BY password
DEFAULT TABLESPACE default_tbs
TEMPORARY TABLESPACE temp_tbs;
```

Creating a Default Tablespace

```
CREATE TABLESPACE user_tbs
DATAFILE 'user_tbs01.dbf' SIZE 100M
AUTOEXTEND ON NEXT 10M MAXSIZE 1000M
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
ALTER DATABASE DEFAULT TABLESPACE user_tbs;
```

Permanent Tablespaces:

Used to store persistent schema objects like tables and indexes.

- Types

  Can be either smallfile or bigfile tablespaces.

- Extent Management

  Typically locally managed with automatic segment space management (ASSM) being the preferred option.

- Storage

  Contains one or more data files that store the actual data.

Command to Create

```
CREATE TABLESPACE perm_tbs
```

```
DATAFILE 'perm_tbs01.dbf' SIZE 100M
AUTOEXTEND ON NEXT 10M MAXSIZE 1000M
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
```

Temporary Tablespaces

Used for temporary segments created during operations such as sorting and joining data.

- Purpose

  To store temporary segments for operations like sorting, hashing, and temporary table storage.
- Configuration

  A user can be assigned a specific temporary tablespace.
- Extent Management

  Usually locally managed.

Creating a Temporary Tablespace

```
CREATE TEMPORARY TABLESPACE temp_tbs
TEMPFILE 'temp_tbs01.dbf' SIZE 50M
AUTOEXTEND ON NEXT 5M MAXSIZE 500M
EXTENT MANAGEMENT LOCAL;
ALTER DATABASE DEFAULT TEMPORARY TABLESPACE temp_tbs;
```

Undo Tablespaces

Used to store undo information, which is essential for read consistency, rollback operations, and recovery purposes.

- Purpose

  To store undo records for transaction management and read consistency.
- Automatic Undo Management (AUM)

  Oracle recommends using AUM to manage undo tablespaces.

Setting Default Undo Tablespace

```
ALTER SYSTEM SET UNDO_TABLESPACE = undo_tbs;
```

Creating an Undo Tablespace

```
CREATE UNDO TABLESPACE undo_tbs
DATAFILE 'undo_tbs01.dbf' SIZE 100M
AUTOEXTEND ON NEXT 10M MAXSIZE 1000M;
ALTER SYSTEM SET UNDO_TABLESPACE = undo_tbs;
```

Command

Creating and Managing Tablespaces

```
CREATE TABLESPACE users
DATAFILE 'users01.dbf' SIZE 50M
AUTOEXTEND ON
NEXT 10M MAXSIZE 500M
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
```

Adding a Data File

```
ALTER TABLESPACE users ADD DATAFILE 'users02.dbf' SIZE 50M;
```

Resizing a Data File

```
ALTER DATABASE DATAFILE 'users01.dbf' RESIZE 100M;
```

Making a Tablespace Read-Only

```
ALTER TABLESPACE users READ ONLY;
```

Other

```
create tablespace <tablespace-name>                    Create tablespace.
    logging
    datafile '<tablespace-name>.dbf'                   Specify a dbf file path or use default path.
    size 100M                                          Data file capacity size
    autoextend on
    next 10M                                           Size of automatic growth at once
    maxsize unlimited;

    You can create identical tablespaces in both CDB and PDB
    // create tablespace UFO logging datafile 'C:\app\UFO\oradata\UFO.DBF' size 1000M autoextend on next 1000M
    maxsize unlimited;


select tablespace_name, status, contents from user_tablespaces;                    Show all table spaces;


drop tablespace <tablespace-name> including contents and datafiles;                Delete tablestace


select tablespace_name, sum(bytes)/1024/1024 from dba_data_files group by tablespace_name;    View all tablespaces;
    // select Segment_Name,Sum(bytes)/1024/1024 From User_Extents Group By Segment_Name

create user C##<username> identified by <password> default tablespace <tablespace-name>;      Create a new user.
    CDB：Normal user COMMON USERS is established in the CDB level, with uername starting with C## or c##.
        (The C## user established by CDB can log in to PDB level, but they need authorized separately by PDB, along with
        the corresponding tns of PDB)
    PDB： LOCAL USERS are only established in the PDB level, and a CONTAINER must be specified when establishing
    them, PDB cannot directly delete C### users.

drop user <username>

select * from dba_users where username =' [C##]<username>';        Ensure that the user has been successfully created.

grant connect, select, insert, update, delete to [C##]<username>;      Grant permissions.
        connect
        dba
        select insert update delete


sqlplus system/xxpass  /nolog          密码登录 system 账号（system，as sysdba）

select * from tabs;                    查询所有表
```

```
select userenv('language') from dual;        查询服务器编码
sqlplus OT@PDBORCL                                    将 OT 用户连接到位于 PDBORCL 可插拔数据库中的示例数据库

alter session set container= PDBORDL;       将当前数据库设置为可插入数据库【POCP1，PDBORDL】
show pdbs;
show con_name;
select username,created from dba_users where created>sysdate-1;


comment on column table.column_name is 'comments_on_col_information';          添加注释


alter table 表名 rename column 原列名 to 新列名;              修改列名
alter system kill session
```

User

```
create user c##test                          创建用户 test = 123456【default tablespace 指定用户默认所属的表空间】
    identified by 123456
    default tablespace waterboss
    container=all;                    The containers clause can only be used by a common user in the root contaienr.
```

CDB 层：普通用户 COMMON USERS 建立在 CDB 层，用户名以 C##或 c##开头。（CDB 建立的 C##用户可以在 PDB 登录，但是要 PDB 单独授权，加上 PDB 对应的 tns）

PDB 层：本地用户 LOCAL USERS 仅建立在 PDB 层，建立的时候必须指定 CONTAINER，PDB 无法直接删除 C##用户。

The local environment should be switched to PDB to create users

```
alter user test identified by 123456;              更改用户
alter user <username> quota unlimited on <tablespace-name>;

drop user test [cascade];                          删除用户【cascade 可删除用户所有的对象，然后再删除用户】

grant connect, resource to test;                   授权 role 给用户【connect, resouce, dba】
revoke connect, resource from test;                撤销 role 授权
create role testRole;                              创建 role
grant select on xxtab to testRole;                 授权角色，拥有 testRole 角色的所有用户都具有对 xxtab 表的 select 查询权限
                    // grant s connect,elect, insert, update, delete on testdb.* to common_user@'%'
grant unlimited tablespace to <user>;          Grant unlimited tablespace priviledges;
drop role testRole;
```

Connection

```
show pdbs;
show con_name;            显示数据库名
select pdb_name, status from cdb_pdbs;
select sys_context('userenv','instance_name')from dual;      Get oracle SID

alter session set container = ORCLPDB;       Recommend to connect to pdb
```

show parameter name　查看当前登录数据库的参数配置

Predefined Variables

DB_CREATE_FILE_DEST　这个参数指定了数据库创建新 DBF 文件时使用的目录。其默认值通常是一个绝对路径，例如“/u01/app/oracle/oradata”。

DB_CREATE_ONLINE_LOG_DEST_n　这个参数指定了在线日志文件的位置，其中 n 是一个数字，表示日志文件的组号。默认情况下，这些组的路径形式是“/u01/app/oracle/fast_recovery_area”。

CONTROL_FILES　这个参数指定了控制文件的位置，它是 Oracle 数据库的元数据文件。默认情况下，这些文件的路径形式是“/u01/app/oracle/oradata/数据库名/”。

show parameter DB_CREATE_FILE_DEST

alter SYSTEM set <parameter-name>=<value> SCOPE = spfile;

Configuration

Enverionment

These environment variables must be defined before running "dbca"

ORACLE_HOME=/path/to/oracle/home

ORACLE_SID=<custom-sid-name>

ORACLE_SID uniquely identifies the Oracle instance you want to interact with.

In environments with multiple Oracle instances running on the same server, ORACLE_SID helps direct the client or administrative tools to the right database.

NLS_LANG=SIMPLIFIED CHINESE_CHINA.ZHS16GBK　(作用：防止中文乱码)

ORACLE_HOME= <db home directory>

TNS_ADMIN= <database installations directory>

ERROR　[INS-30014]无法检查指定的位置是否位于 CFS 上　修改 hosts 文件　192.168.1.101 orcl

Replication

Using dbms_metadata.get_ddl in Oracle

dbms_metadata.get_ddl is an Oracle built-in package that generates the DDL (Data Definition Language) statements for database objects such as tables, indexes, views, etc.

You can use a text comparison tool to compare the DDL statements retrieved from the source and target databases. Tools like `diff`, `Beyond Compare`, or any text comparison tool can be used for this purpose.

Get DDL for a Table:

```
SET LONG 10000;
SELECT dbms_metadata.get_ddl('TABLE', 'your_table_name', 'your_schema_name') FROM dual;
```

`'TABLE'`: The type of object you want to generate DDL for.

`'your_table_name'`: The name of the table.

`'your_schema_name'`: The schema where the table is located.

Get DDL for an Index:

```
SET LONG 10000;
SELECT dbms_metadata.get_ddl('INDEX', 'your_index_name', 'your_schema_name') FROM dual;
```

`'INDEX'`: The type of object you want to generate DDL for.

`'your_index_name'`: The name of the index.

`'your_schema_name'`: The schema where the index is located.

Get DDL for All Tables in a Schema:

```
SET LONG 10000;
SELECT dbms_metadata.get_ddl('TABLE', table_name, 'your_schema_name')
FROM all_tables
WHERE owner = 'your_schema_name';
```

Get DDL for All Objects in a Schema:

```
SET LONG 10000;
BEGIN
    FOR rec IN (SELECT object_type, object_name
                FROM all_objects
                WHERE owner = 'your_schema_name')
    LOOP
        dbms_output.put_line(dbms_metadata.get_ddl(rec.object_type, rec.object_name,
'your_schema_name'));
    END LOOP;
END;
/
```

Cluster

Oracle RAC

Oracle Real Application Clusters (RAC) is an advanced feature of Oracle Database that allows multiple computers to run Oracle RDBMS software simultaneously while accessing a single database,
thus providing a highly available and scalable database solution.

Key Features

- High Availability
  - Oracle RAC provides fault tolerance and redundancy, ensuring that if one node fails, the others continue to process database requests. This minimizes downtime and maximizes database availability.
- Scalability
  - With Oracle RAC, you can add or remove nodes as needed without downtime. This allows the database to scale horizontally, handling increased workloads by adding more nodes.
- Load Balancing:
  - RAC automatically balances the load across all available nodes, ensuring optimal resource utilization and performance.
- Failover:
  - If one node in the RAC cluster fails, the workload is automatically redistributed to the remaining nodes, ensuring continuous availability of the database.
- Cluster Interconnect:
  - A high-speed, low-latency network connects the nodes in a RAC cluster, allowing them to communicate and coordinate access to the shared database efficiently.

Components

- Cluster Nodes:
  - Multiple servers (nodes) running Oracle Database instances that form the RAC environment.
- Oracle Clusterware:
  - Provides infrastructure to bind multiple servers together to form a cluster. It manages node membership and communication between nodes.
- Shared Storage:
  - All RAC nodes share the same physical database files. This can be on a SAN, NAS, or any other shared storage solution.
- Global Cache Service (GCS) and Global Enqueue Service (GES):
  - Ensure data consistency and coordinate access to shared resources across nodes.

Configuration and Architecture

- Grid Infrastructure:
  - The foundation of Oracle RAC, which includes Oracle Clusterware and Automatic Storage Management (ASM).

Oracle Clusterware:

Manages the cluster resources and node membership. It includes components like the Cluster Synchronization Service (CSS), Oracle Cluster Registry (OCR), and Voting Disk.

Automatic Storage Management (ASM):

Provides a unified storage solution for Oracle databases, simplifying storage management and improving performance.

Cluster Configuration

Prerequisites

Hardware Requirements:

Multiple servers (nodes) with shared storage.

Network interfaces for the private interconnect and public network.

Software Requirements:

Oracle Database software.

Oracle Grid Infrastructure (for Oracle Clusterware and ASM).

Operating System Configuration:

Ensure the operating system and kernel parameters are configured correctly.

Shared Storage Configuration:

Configure shared storage using ASM, NFS, or SAN.

Install and Configure the Operating System

Install the OS

Install the same version and patch level of the operating system on all nodes.

Configure Network Interfaces

Set up public and private network interfaces on all nodes.

Set Kernel Parameters

Configure kernel parameters as per Oracle's recommendations.

Install Oracle Grid Infrastructure

Download Software

Download the Oracle Grid Infrastructure software.

Run Installer

Run the runInstaller script to start the installation.

```
./runInstaller
```

Select Installation Option

Choose to install and configure Oracle Grid Infrastructure for a Cluster.

Specify Cluster Configuration

Define the cluster name and SCAN (Single Client Access Name).

Configure Network Interfaces

Assign interfaces for the public network and private interconnect.

Shared Storage Configuration

Choose ASM and configure disk groups.

Complete Installation

Follow the prompts to complete the installation.

Install Oracle Database Software

Download Software

Download the Oracle Database software.

Run Installer

Run the runInstaller script to start the installation.

```
./runInstaller
```

Select Installation Type

Choose to install the database software only.

Install Software

Follow the prompts to install the software on all nodes.

Create the Database

Run Database Configuration Assistant (DBCA):

```
dbca
```

Create a Database

Choose to create a new RAC database.

Specify Database Configuration

Define the database name, configuration options, and storage locations.

Specify Cluster Nodes

Select the nodes that will participate in the RAC database.

Complete Database Creation

Follow the prompts to create the RAC database.

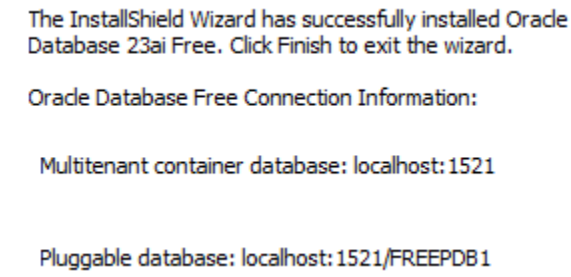
Windows Installation

Get oracle zip file

Reference:

https://www.oracle.com/cn/database/technologies/oracle-database-software-downloads.html

Destination Folder:	E:\UserDev\oracle23ai\
Oracle Home:	E:\UserDev\oracle23ai\dbhomeFree\
Oracle Base:	E:\UserDev\oracle23ai\



Verify and Clean Up Central Inventory

C:\Program Files\Oracle\Inventory\ContentsXML\inventory.xml

Check for entries related to the existing Oracle Home (E:\UserDev\Oracle23ai\dbhomeFree) and remove them if the Oracle Home is no longer valid. **Backup the file before making changes.**

Uninstall Oracle

Stop Oracle Services:

- Open a Command Prompt or PowerShell window with Administrator privileges.
- Stop all Oracle-related services using the following command:
  ```
  net stop OracleServiceXE
  ```

Replace OracleServiceXE with the correct service name if it's different in your setup.

Remove Oracle from Windows:

- Go to the **Control Panel** > **Programs** > **Programs and Features**.
- Find **Oracle Database 21c** (or the version you installed) in the list, and select **Uninstall**.
- This will begin the uninstallation process. Follow the prompts to remove Oracle 21c from your system.

Remove Oracle Directories:

- Manually delete the following directories if they are not removed during the uninstallation:
  - C:\app\oracle\

- C:\oracle\
- C:\ORACLE_HOME\ (or wherever your Oracle installation was located)
- Any other directories where Oracle might have been installed.

Remove Registry Entries (Optional but recommended):

- Open the **Registry Editor** (regedit) and carefully remove Oracle-related keys under:
  - HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE
  - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Oracle
- Be careful when modifying the registry, as incorrect changes can affect your system.

Remove Oracle Environment Variables:

- Right-click on **My Computer** and go to **Properties** > **Advanced system settings** > **Environment Variables**.
- Remove any Oracle-related environment variables such as ORACLE_HOME, ORACLE_SID, and TNS_ADMIN.

Reboot the System:

- After all the steps are completed, reboot your system to ensure that all Oracle-related services and configurations are completely removed.

Management page

"Oracle Enterprise Manager Database Express" web page path：https://localhost:5500/em　　sys = 口令

Error

ORA-01033: ORACLE initialization or shutdown in progress

linux Installation

创建 oracle 用户：

useradd oracle

创建 oracle 目录：

mkdir /usr/local/oracle
mkdir /usr/local/oracle/oracle
mkdir /usr/local/oracle/oradata
mkdir /usr/local/oracle/oracle/product

chown -R oracle:oracle /usr/local/oracle/app

添加 oracle 用户环境变量：

su oracle
export ORACLE_BASE=/usr/local/oracle
export ORACLE_HOME=$ORACLE_BASE/oracle/product/11.2.0/dbhome_1
export ORACLE_SID=orcl
export PATH=$PATH:$HOME/bin:$ORACLE_HOME/bin
export LD_LIBRARY_PATH=$ORACLE_HOME/lib:/usr/lib

/etc/selinux/config >>　重启永久关闭 selinux（reboot 生效）

SELINUX=disabled

/etc/sysctl.conf >>　　调整系统参数的工具（/sbin/sysctl -p 配置生效）

```
fs.file-max = 6815744
fs.aio-max-nr = 1048576
kernel.shmall = 2097152
kernel.shmmax = 2147483648
kernel.shmmni = 4096
kernel.sem = 250 32000 100 128
net.ipv4.ip_local_port_range = 9000 65500
net.core.rmem_default = 4194304
net.core.rmem_max = 4194304
net.core.wmem_default = 262144
net.core.wmem_max = 1048576
```

/etc/security/limits.conf >> 修改操作系统核心参数

```
oracle soft nproc 2047
oracle hard nproc 16384
oracle soft nofile 1024
oracle hard nofile 65536
```

/etc/pam.d/login >>

```
#session required /lib/security/pam_limits.so      会导致开机 module is unknown
session required pam_limits.so
```

/etc/profile >> 用户配置（source /etc/profile 配置生效）

```
if [ $USER = "oracle" ]; then
        if [ $SHELL = "/bin/ksh" ]; then
                ulimit -p 16384
                ulimit -n 65536
        else
                ulimit -u 16384 -n 65536
        fi
fi
```

/usr/local/oracle/install/response/db_install.rsp >> 无图形化安装应答设置

```
oracle.install.option=INSTALL_DB_SWONLY          //29 行 安装类型
ORACLE_HOSTNAME=chances                          //37 行 主机名称
UNIX_GROUP_NAME=oracle                           //42 行 安装组
INVENTORY_LOCATION=/usr/local/oracle/inventory                  //47 行 INVENTORY 目录
ORACLE_HOME=/usr/local/oracle                                    //oracle runInstall 脚本 解压文件所在位置
ORACLE_BASE=/usr/local/oracle-19.3/app                           //安装目录
oracle.install.db.InstallEdition=EE                        //99 行 oracle 版本
oracle.install.db.DBA_GROUP=dba                           //142 行 dba 用户组
oracle.install.db.OPER_GROUP=oracle                       //147 行 oper 用户组
oracle.install.db.config.starterdb.type=GENERAL_PURPOSE          //160 行 数据库类型
oracle.install.db.config.starterdb.globalDBName=orcl             //165 行 globalDBName
oracle.install.db.config.starterdb.SID=orcl                      //170 行 SID
oracle.install.db.config.starterdb.memoryLimit=800               //192 行 自动管理内存的最小内存(M)
oracle.install.db.config.starterdb.password.ALL=oracle           //233 行 设定所有数据库用户使用同一个密码
```

DECLINE_SECURITY_UPDATES=true　　//385 行 设置安全更新

/usr/local/oracle/cv/admin >>　supportedOSCheck 安装错误解决方法
CV_ASSUME_DISTID=OEL7.8

安装依赖软件包：
yum install binutils compat-libstdc++ elfutils-libelf elfutils-libelf-devel elfutils-libelf-devel-static gcc gcc-c++ glibc glibc-common glibc-devel glibc-headers kernel-headers ksh libaio libaio-devel libgcc libgomp libstdc++ libstdc++-devel make sysstat unixODBC unixODBC-devel
yum -y install elfutils-libelf elfutils-libelf-devel elfutils-libelf-devel-static*
yum install compat-libstdc++-33　（没有 compat-libstdc++时替换）
yum install libnsl.x86_64

安装到 ORACLE_BASE 目录：
./runInstaller -silent -responseFile /opt/oracle/response/db_install.rsp -ignorePrereq　　无图形化安装

安装成功后操作（切换到 root 用户执行）：
/usr/local/oracle-19.3/inventory/orainstRoot.sh
/usr/local/oracle/root.sh

静默配置监听（切换到 root 用户执行）：
chmod -R 775 /usr/local/oracle　　root 用户赋予权限
su - oracle
/path/oracle/11.2.0/bin/netca /silent /responsefile /path/oracle/assistants/netca/netca.rsp

/path/oracle/assistants/dbca/dbca.rsp >> 修改建库配置
RESPONSEFILE_VERSION = "11.2.0" //不能更改
OPERATION_TYPE = "createDatabase"//默认也是这个
GDBNAME = "orcl" //数据库的名字,重要,建议设置为 orcl 与 sid 相同
SID = "orcl" //对应的实例名字 => orcl 是默认的,一般不建议用默认的,如同 123456
TEMPLATENAME = "General_Purpose.dbc" //建库用的模板文件,默认也是这个
SYSPASSWORD = "SYS" //SYS 管理员密码
SYSTEMPASSWORD = "SYSTEMP" //SYSTEM 管理员密码
DATAFILEDESTINATION = /path/oracle/oradata //数据文件存放目录(此时是没有创建的)
RECOVERYAREADESTINATION=/path/oracle/flash_recovery_area //恢复数据存放目录
CHARACTERSET = "ZHS16GBK" //字符集，重要!!! 建库后一般不能更改，所以建库前要确定清楚。
TOTALMEMORY = "1048" //oracle 内存 5120MB

/usr/local/oracle/bin/dbca -silent -executePrereqs -responseFile /usr/local/oracle/assistants/dbca/dbca.rsp -databaseConfigType SINGLE （oracle 用户执行，创建数据库）
/usr/local/oracle/bin/dbca -silent -force -noconfig -ignorePrereq -responseFile /usr/local/oracle/assistants/dbca/dbca.rsp -databaseConfigType
/usr/local/oracle/bin/dbca -silent -ignorePrereq -responseFile /usr/local/oracle/assistants/dbca/dbca.rsp

Sqlplus

sqlplus

Enter the username and password to enter the user interface.

-v Check sqlplus version

/nolog Non password login

```
sqlplus OT@PDBORCL
```

Connect OT user to the sample database located in the PDBORCL pluggable database.

```
sqlplus / as sysdba
```

Connect to the database as a privileged user (e.g., SYSDBA) and check the status of the database.

Sqlplus terminal

Database

```
SELECT STATUS FROM V$INSTANCE;
```

First, check the current status of the database:

STARTED: Database is in the initial startup phase.

MOUNTED: Database is partially open, but not ready for connections.

OPEN: Database is fully operational and ready for connections.

SHUTDOWN: Database is fully shut down.

```
SELECT INSTANCE_NAME FROM V$INSTANCE;
```

Check the current SID

```
SHUTDOWN IMMEDIATE;
```

Shutdown the Database

```
STARTUP MOUNT;
STARTUP OPEN;
```

Start the Database

```
SELECT NAME FROM V$DATABASE;
SHOW PARAMETER control_files;

CREATE DATABASE simi
  USER SYS IDENTIFIED BY "Simi110120%"
  USER SYSTEM IDENTIFIED BY "Simi110120%"
  LOGFILE GROUP 1 ('E:\UserDev\Oracle_23ai\oradata\simi\redo01.log') SIZE 50M,
          GROUP 2 ('E:\UserDev\Oracle_23ai\oradata\simi\redo02.log') SIZE 50M
  DATAFILE 'E:\UserDev\Oracle_23ai\oradata\simi\simi.dbf' SIZE 500M
  CHARACTER SET AL32UTF8
  NATIONAL CHARACTER SET AL16UTF16
  EXTENT MANAGEMENT LOCAL
  DEFAULT TEMPORARY TABLESPACE temp
  TEMPFILE 'E:\UserDev\Oracle_23ai\oradata\simi\temp01.dbf' SIZE 50M
  UNDO TABLESPACE undots
  DATAFILE 'E:\UserDev\Oracle_23ai\oradata\simi\undots01.dbf' SIZE 100M
  SYSAUX DATAFILE 'E:\UserDev\Oracle_23ai\oradata\simi\sysaux01.dbf' SIZE 100M
  ENABLE PLUGGABLE DATABASE
  SEED
  FILE_NAME_CONVERT=('E:\UserDev\Oracle_23ai\oradata\simi\','E:\UserDev\Oracle_23ai\oradata\simi\pdb\')
```

CONTROLFILE REUSE;

Create a new Container Database (CDB) with a Pluggable Database (PDB).

Change undo tablespace

```
CREATE UNDO TABLESPACE undotbs2
  DATAFILE 'E:\UserDev\Oracle_23ai\oradata\simi\undotbs2.dbf' SIZE 100M;
SHOW PARAMETER undo
ALTER SYSTEM SET UNDO_TABLESPACE = 'UNDOTBS2';
```

If there's a database running and you want to create a new one, you'll need to shut down the existing database instance:

```
SHUTDOWN IMMEDIATE;
```

Once the database is shut down, start the Oracle instance in nomount mode (Connect to an **existing database instance**, but not to an existing database itself). This will allow you to create a new database.

```
STARTUP NOMOUNT;
```

You need to ensure that the DB_NAME initialization parameter is set correctly for the new database you're trying to create.

```
SHOW PARAMETER DB_NAME;
ALTER SYSTEM SET DB_NAME = 'SIMI' SCOPE = SPFILE;
```

After changing the DB_NAME, you will need to restart the database for the changes to take effect.

```
SHUTDOWN IMMEDIATE;
STARTUP NOMOUNT;
```

```
ERROR at line 1:
ORA-00603: ORACLE server session terminated by irrecoverable error
ORA-01092: ORACLE instance terminated. Disconnection forced
ORA-01501: CREATE DATABASE failed
ORA-01519: error while processing file '%ORACLE_HOME%\RDBMS\ADMIN\dtxnspc.bsq'
near line 7
ORA-00604: Error occurred at recursive SQL level 1. Check subsequent errors.
ORA-30012: undo tablespace 'UNDOTBS1' does not exist or of wrong type
Process ID: 5312
Session ID: 162 Serial number: 46751
Help: https://docs.oracle.com/error-help/db/ora-00603/
```

```
CREATE TABLESPACE simi_ts
  DATAFILE 'E:\UserDev\oracle23ai\oradata\simi\simi_ts01.dbf' SIZE 100M
  EXTENT MANAGEMENT LOCAL
  SEGMENT SPACE MANAGEMENT AUTO;
```

```
ALTER DATABASE OPEN;
```

Open the Database (if in MOUNTED state)

```
ALTER DATABASE CLEAR LOGFILE GROUP 2;
```

Clear the Corrupt Log File

```
ALTER DATABASE DROP LOGFILE GROUP 2;
```

Oracle prevents dropping the current log group, as it's in active use for the transaction processing.

```
SELECT NAME, OPEN_MODE FROM V$PDBS;
```

Query to List All Databases (PDBs)

```
NAME            OPEN_MODE
--------------- ------------
PDB1            READ WRITE
PDB2            MOUNTED
PDB3            READ ONLY
```

ALTER SESSION SET CONTAINER = PDB1;

Switch to a Specific PDB

select name, open_mode from v$pdbs;

View all container databases in PDB. (READ WRITE indicates that it is turned on. If it is not turned on, it needs to be manually turned on.)

select name from v$database;

View all databases.

show pdbs;

View all databases.

show con_name;

Show the current connected database container.

alter pluggable database <container-name> open;

Open PDB container database.

alter session set container= orclpdb;

Switch container database.

orclpdb

cdb$root

select name, open_mode from v$pdbs;

View all container databases in PDB. (READ WRITE indicates that it is turned on. If it is not turned on, it needs to be manually turned on.)

select name from v$database;

View all databases.

show pdbs;

View all databases.

show con_name;

Show the current connected database container.

alter pluggable database <container-name> open;

Open PDB container database.

alter session set container= orclpdb;

Switch container database.

orclpdb

cdb$root

User

CREATE USER C##<username> IDENTIFIED BY "<password>" DEFAULT TABLESPACE <tablespace-name>;

Create a new user.

By default, Oracle assigns the USERS tablespace for new users.

However, you can specify a different default tablespace when creating a user.

CDB：This is a COMMON USER, which is established in the CDB level, with uername starting with C## or c##.

(The C## user established by CDB can log in to PDB level, but they need authorized separately by PDB, along with the corresponding tns of PDB)

PDB： LOCAL USERS are only established in the PDB level, and a CONTAINER must be specified when establishing them, PDB cannot directly delete C### users.

Example:

CREATE USER C##simi IDENTIFIED BY "Simi110120%";

DROP USER <username>

GRANT connect, select, insert, update, delete TO [C##]<username>;

Grant permissions.

connect

dba

select insert update delete

Example:

GRANT DBA TO C##simi;

GRANT CONNECT, RESOURCE TO C##simi;

SELECT * FROM dba_users WHERE username =' [C##]<username>';

Ensure that the user has been successfully created.

Example:

SELECT username, default_tablespace FROM dba_users WHERE username = 'C##SIMI';

GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW TO new_user;

Grant privileges to the new user:

Tablespace

drop tablespace <tablespace-name> including contents and datafiles;

Delete tablestace

select tablespace_name, sum(bytes)/1024/1024 from dba_data_files group by tablespace_name;

View all tablespaces;

// select Segment_Name,Sum(bytes)/1024/1024 From User_Extents Group By Segment_Name

dbca

Create a New Database with DBCA

Global Database Name

<database_name>.<domain_name>

simiorcl.simi.com

password

Simi110120%

Pdb

simipdb

lsnrctl (Listener)

lsnrctl status

Check the listener's status

lsnrctl stop

Example:

```
LSNRCTL for 64-bit Windows: Version 23.0.0.0.0 - Production on 22-DEC-2024 18:21:42

Copyright (c) 1991, 2024, Oracle.  All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=windows10.microdone.cn)(PORT=1521)))
The command completed successfully
```

lsnrctl start

Functional SQL

锁，事务

锁

select sid, type, id1, id2, lmode, request, block from v$lock where type in ('TM','TX') order by 1,2;　　查询所有的锁

select distinct sid from v$mystat;

select sid,event from v$session_wait where sid in (1, 3);

事务

start transaction　　开启事务

rollback　　回滚事务　（将数据恢复到事务开始时状态）

commit　　提交事务　（对事务中进行操作，进行确认操作，事务在提交后，数据就不可恢复）

SID	TYPE	ID1	ID2	LMODE	REQUEST	BLOCK
251	TM	92404	0	3	0	0
251	TX	655361	30249	0	6	0
373	TM	92404	0	3	0	0
373	TX	655361	30249	6	0	1

事务 373：执行了写操作，但是未 commit，此时没有被 block

事务 251：新增事务 251，前方的事务 373 被 block

存储过程

存储函数

触发器

select * from ALL_TRIGGERS;　　查看所有触发器

视图

语句

变量

declare num **number**;　　**声明变量**

select count(1) **into** num from user_tables where table_name = upper('SH_PLACARD_INFO') ;　　写入变量

判断

```
begin       声明逻辑块
    if num>0 then
        execute immediate 'drop table SH_PLACARD_INFO' ;    立即执行字符串中的 sql
    end if ;
end;
/                   代码块结束，下方才可以正常执行 sql
```

Build-in tables

Constraint

user_constraints

OWNER	CONSTRAINT_NAME	CONSTRAINT_TYPE	TABLE_NAME	SEARCH_CONDITION
CBCM_OWNER	SYS_C0010451	C	TEST_CCB	"ID" IS NOT NUL
CBCM_OWNER	OPPTY_FORM_CONTACT_FK	R	OPPTY_FORM_CONTACT	(null)

CONSTRAINT_NAME 指定表的 约束名称

CONSTRAINT_TYPE 指定表的 约束类型

- C 检查表上的约束
- P 主键
- U 唯一键
- R 外键
- V 带有检查选项，在视图上
- O 只读，在视图上
- H 哈希表达式
- F 包含 REF 列的约束
- S 补充记录

TABLE_NAME 指定表的 名称

R_CONSTRAINT_NAME 关联表的 外键名称

select * from user_constraints where table_name='TEST_PERSON';

user_cons_columns

OWNER	CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME	POSITION
CCBCM_OWNER	BIN$6K9tALbNJ7jgU5DXsanAKA==$0	BIN$6K9tALbZJ7jgU5DXsanAKA==$0	REQUEST_ID	(null)
CCBCM_OWNER	BIN$6K9tALbOJ7jgU5DXsanAKA==$0	BIN$6K9tALbZJ7jgU5DXsanAKA==$0	FORM_ID	(null)
CCBCM_OWNER	BIN$6K9tALbPJ7jgU5DXsanAKA==$0	BIN$6K9tALbZJ7jgU5DXsanAKA==$0	ENTITY_NO	(null)
CCBCM_OWNER	BIN$6K9tALbQJ7jgU5DXsanAKA==$0	BIN$6K9tALbZJ7jgU5DXsanAKA==$0	EVENT	(null)

CONSTRAINT_NAME 指定表的 外键名称

TABLE_NAME 指定表的 名称

COLUMN_NAME 指定表 列名

select * from user_cons_columns where table_name='TEST_PERSON'; show COLUMN_NAME

Index

user_indexes

INDEX_NAME	INDEX_TYPE	TABLE_OWNER	TABLE_NAME	TABLE_TYPE	UNIQ
ACCOUNTS_PK	NORMAL	CCBCM_OWNER	ACCOUNTS	TABLE	UNIQU
ACCT_SETUP_AUTH_PK	NORMAL	CCBCM_OWNER	ACCT_SETUP_AUTH	TABLE	UNIQU
PK_API_CONFIG	NORMAL	CCBCM_OWNER	API_CONFIG	TABLE	UNIQU
SYS_IL0000097384C00009$$	LOB	CCBCM_OWNER	API_ERROR_TRACKER	TABLE	UNIQU
PK_API_ERROR_TRACKER	NORMAL	CCBCM_OWNER	API_ERROR_TRACKER	TABLE	UNIQU
APPS_PK	NORMAL	CCBCM_OWNER	APPS	TABLE	UNIQU

TABLE_NAME 指定表的 名称

INDEX_NAME 指定表的 索引名称

select * from user_indexes where table_name='TEST_PERSON';　　show COLUMN_NAME

user_ind_columns

INDEX_NAME	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	COLUM
ACCOUNTS_PK	ACCOUNTS	ACCT_NUM	1	
ACCT_SETUP_AUTH_PK	ACCT_SETUP_AUTH	APP_USER_ID	1	
ACCT_SETUP_AUTH_PK	ACCT_SETUP_AUTH	USER_ROLE_ID	2	
ACCT_SETUP_AUTH_PK	ACCT_SETUP_AUTH	ACCT_NUM	3	

Table

user_tables

TABLE_NAME　　指定表的 名称

TABLESPACE_NAME　　表空间 名称

user_tab_columns

TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_TYPE_MOD	DATA_TYPE_OWNER	DATA_LENGTH
FORM_MGMT_PRD	DOC_STATUS	VARCHAR2	(null)	(null)	20
FORM_MGMT_PRD	BATCH_ID	NUMBER	(null)	(null)	22
CASE_ONBOARD	VALID_CASE_ACK_RCPT_MS_CRM_MSG	VARCHAR2	(null)	(null)	255

user_col_comments

TABLE_NAME	COLUMN_NAME	COMMENTS
SYS_LOG	REQUEST_ID	Request id in response header
SYS_LOG	REQUEST_URI	Request uri
SYS_LOG	REQUEST_METHOD	Request method
SYS_LOG	STATUS_CODE	Status code of request

Advance

dual

功能操作

select user from dual;　　查看当前用户

select 7*9*10-10 from dual;　　用做计算器

调用系统函数

select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') from dual;　　获得当前系统时间

select sys_context('userenv','terminal') from dual;　　获得主机名

select sys_context('userenv','language') from dual;　　获得当前 locale

select DBMS_RANDOM.random from dual;　　获得一个随机数

查看序列值

create sequence aaa increment by 1 start with 1;　　创建序列 aaa 以 1 开始，每次加 1

select aaa.nextval from dual;　　获得序列 aaa 的下一个序列值

select aaa.currval from dual;　　获得序列 aaa 的当前序列值

Permission

dba_users

USERNAME	USER_ID	PASSWORD	ACCOUNT_STATUS	LOCK_DATE	EXPIRY_DATE	DEFAULT_TABLESPA
ORDDATA	90	(Null)	LOCKED	2022-09-10 00:55:56	(Null)	USERS
ORACLE_OCM	41	(Null)	LOCKED	2022-09-10 00:55:56	(Null)	USERS
SYSDG	2147483618	(Null)	LOCKED	2022-09-10 00:55:56	(Null)	USERS
ORDSYS	89	(Null)	LOCKED	2022-09-10 00:55:56	(Null)	USERS
SDK	108	(Null)	OPEN	(Null)	2023-03-11 14:16:33	STD

USERNAME 指定用户名

DEFAULT_TABLESPACE 指定用户的 表空间

all_users

USERNAME	USER_ID	CREATED	COMMON	ORACLE_MAINTAINED	INHERITED	DEFAULT_COLLATION
SYS	0	2019-05-30 03:10:39	YES	Y	YES	USING_NLS_COMP
AUDSYS	8	2019-05-30 03:10:39	YES	Y	YES	USING_NLS_COMP
SYSTEM	9	2019-05-30 03:10:39	YES	Y	YES	USING_NLS_COMP

dba_roles

ROLE	ROLE_ID	PASSWORD_REQUIRED	AUTHENTICATION_TYPE	COMMON	ORACLE_MAINTAINED
CONNECT	2	NO	NONE	YES	Y
RESOURCE	3	NO	NONE	YES	Y
DBA	4	NO	NONE	YES	Y

select * from dba_sys_privs; 查看用户或角色系统权限

user_sys_privs

USERNAME	PRIVILEGE	ADMIN_OPTION	COMMON	INHERITED
SDK	UNLIMITED TABLESPACE	NO	NO	NO

dba_tab_privs

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE	HIERARCHY	COMM
SYSTEM	SYS	ORA$BASE	SYS	USE	YES	NO	NO
PUBLIC	SYS	DUAL	SYS	SELECT	YES	NO	NO

all_tab_privs

GRANTOR	GRANTEE	TABLE_SCHEMA	TABLE_NAME	PRIVILEGE	GRANTABLE	HIERARCHY
SYS	PUBLIC	SYS	DUAL	SELECT	YES	NO
SYS	PUBLIC	SYS	SYSTEM_PRIVILEGE_MAP	READ	NO	NO

user_tab_privs

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE	HIERARCHY	COMM
PUBLIC	SYS	SDK	SDK	INHERIT PRIVILEGES	NO	NO	NO

Statements

定长字符数组，数组大小为 10

```
TYPE v_code IS VARRAY(10) of VARCHAR2 (30);
code v_code:=vcode('100','101');
```

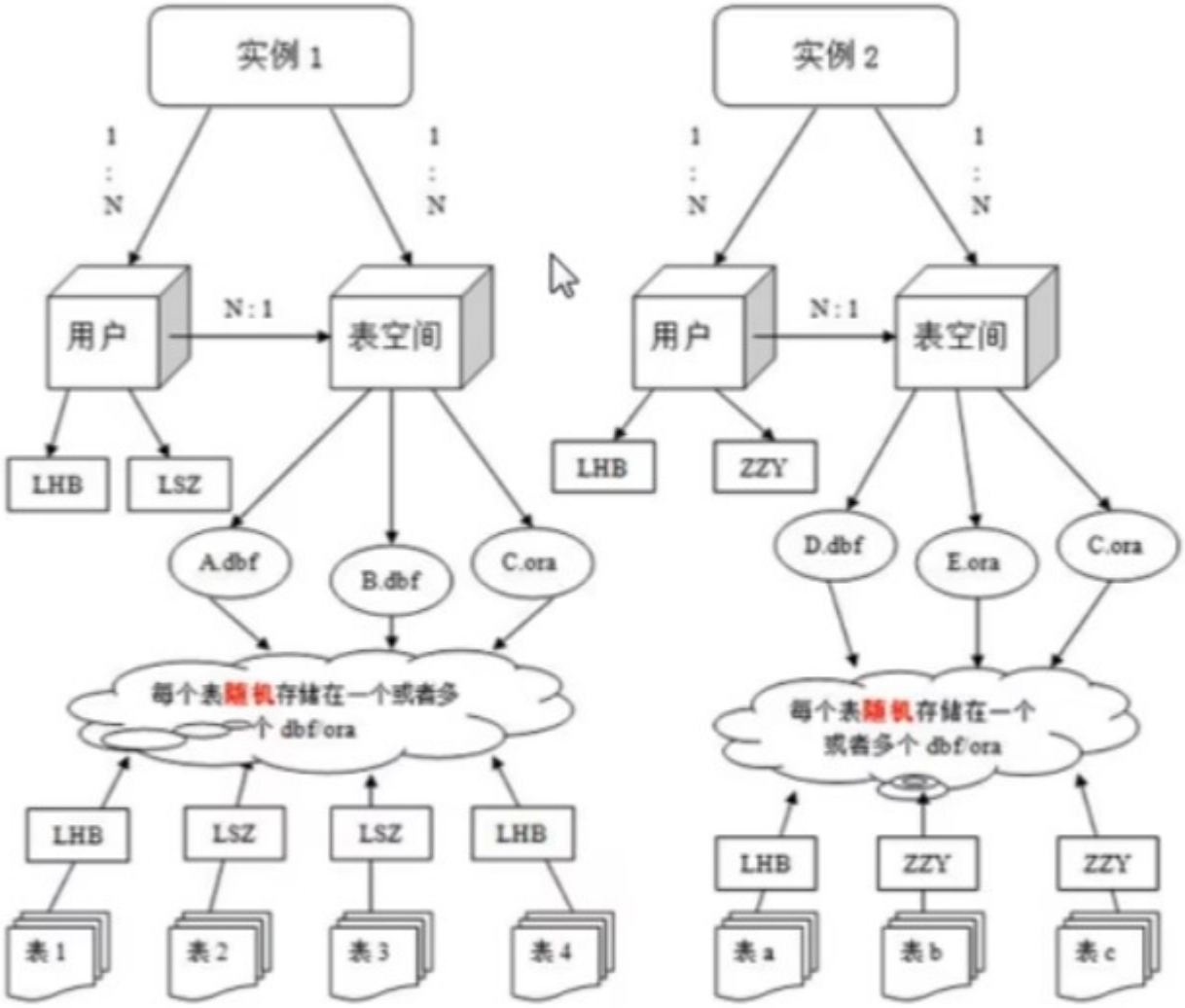
变长数组：

/*可变长字符数组，元素大小 30，索引标号 integer 类型自增长*/
TYPE v_code IS TABLE OF VARCHAR2 (30) INDEX BY BINARY_INTEGER;

Permissions

Users

用户是在表空间下建立的。用户登陆后只能看到和操作自己的表，ORACLE 的用户与 MYSQL 的数据库类似，每建立一个应用需要创建一个用户。



Default Users

There are three types of Oracle login identities, Each identity corresponds to different permissions.

- NORMAL
- SYSDBA (system administrator)
- SYSOPER (system operator).

SYS as SYSDBA The sys user is a super user with the highest authority, the SYSDBA role. And the database administrator authority to create a database.

SYSTEM The system user is a management operator with the SYSOPER role and database operator privileges, and cannot create a database

PDBADMIN

Roles

connect role(连接角色)

临时用户，特指不需要建表的用户，通常只赋予他们 connect role.

connect 是使用 oracle 简单权限，包括 select/insert/update 和 delete 等，这种权限只对其他用户的表有访问权限。

拥有 connect role 的用户还能够创建表、视图、序列（sequence）、簇（cluster）、同义词(synonym)、回话（session）和其他 数据的链（link）

resource role(资源角色)

更可靠和正式的数据库用户可以授予 resource role。

resource 提供给用户另外的权限以创建他们自己的表、序列、过程(procedure)、触发器(trigger)、索引(index)和簇(cluster)。

dba role(数据库管理员角色)

dba role 拥有所有的系统权限

包括无限制的空间限额和给其他用户授予各种权限的能力。system 由 dba 用户拥有

SID

SID

The unique name of the instance, such as the Oracle process running on the machine.

Service name

The alias to an instance or many instances.Ilt's the Database TNS Alias that is given when usersremotely connect to the database.

The service name defaults to the global database name, whichis made up of the database name and domain name.

DAM / MongoDB

Configuration

mongo.conf

mongo.conf >>

dbpath=F:\Program Files\MongoDB\Server\6.0\data 配置数据存储路径

logpath=F:\Program Files\MongoDB\Server\6.0\log\mongodb.log 配置日志存储路径

logappend=true 日志是否追加

journal=true

quiet=true

port=27017

auth=true 开启认证，添加用户再开启

User Management

db.createUser({user:"admin",pwd:"admin",roles:[{role:"userAdminAnyDatabase",db:"admin"}]}) create an admin user, It must be executed in the admin dtabase. (After creating a user, remember to create a new database.)

// db.createUser({user:"smp",pwd:"smp",roles:[{role:"readWrite",db:"smp"}]}) The “smp” database must be created before you create the new smp user.

db.createUser({user: "root",pwd: "root", roles: [{ role: "root", db: "admin" }]}) 创建 root 用户

db.grantRolesToUser ("smp", [{ role: "root", db: "admin" }]) 授予 smp 用户系统表操作权限（锁查询）

db.updateUser("chenpi",{roles:[{role:"readWrite",db:"chenpidb"},{role:"dbAdmin",db:"chenpidb"}]}) 更新用户

db.system.users.find() 查看所有用户

db.system.users.remove({user:"root"}) 删除用户

db.changeUserPassword("username","password") 修改用户密码

db.getUser("xxsmp")　　获取用户
db.auth("xxsmp","xxsmp")　验证用户

数据库用户角色

read　　允许用户读取指定数据库
readWrite　　允许用户读写指定数据库

数据库管理角色

dbAdmin　　允许用户在指定数据库中执行管理函数，如索引创建、删除，查看统计或访问 system.profile
userAdmin　　允许用户向 system.users 集合写入，可以找指定数据库里创建、删除和管理用户
dbOwner　　提供对数据库执行任何管理操作的功能。此角色组合了 readWrite，dbAdmin 和 userAdmin 角色授予的权限。

集群管理角色

clusterAdmin　　只在 admin 数据库中可用，赋予用户所有分片和复制集相关函数的管理权限。
clusterManager
clusterMonitor
hostManager

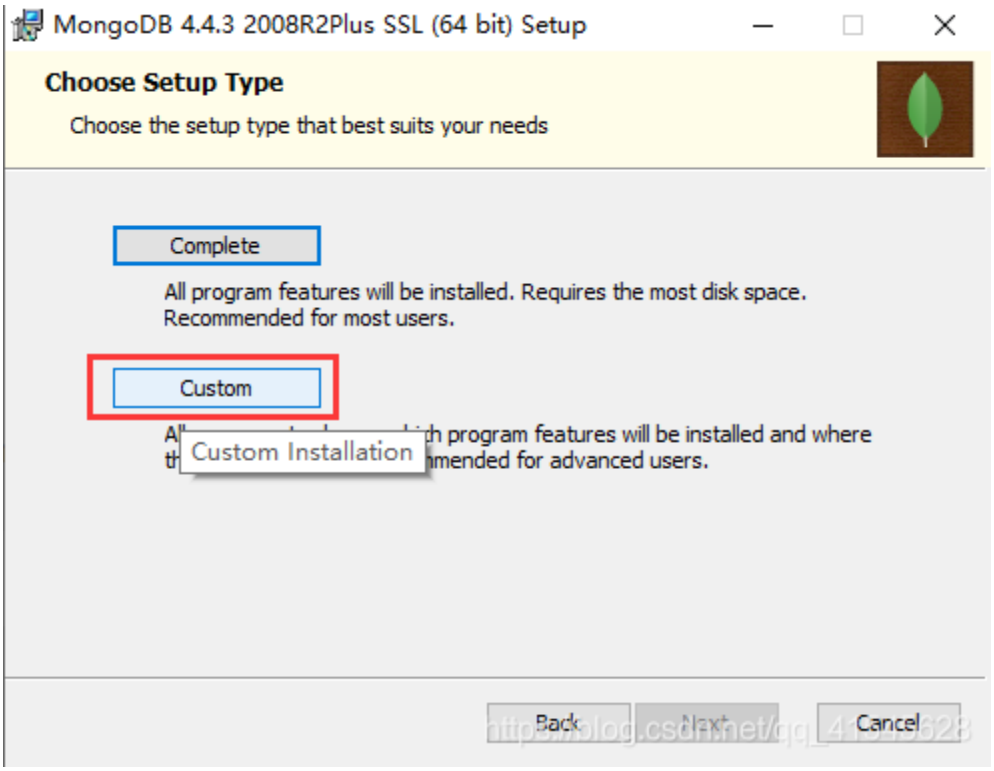
所有数据库角色

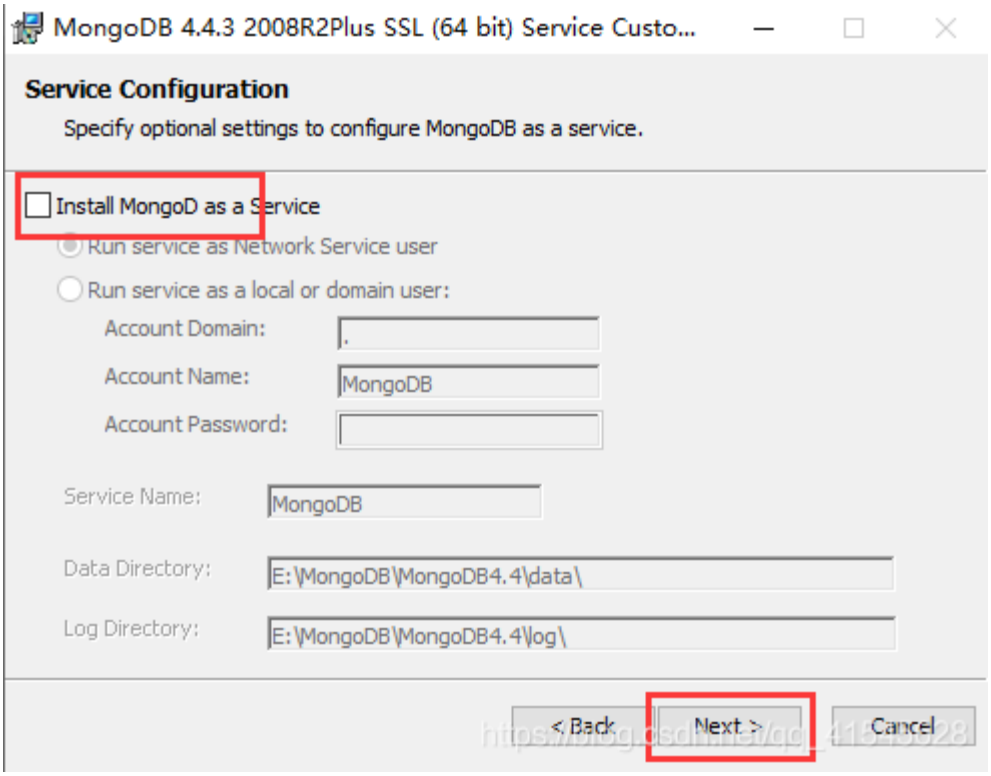
readAnyDatabase:　　只在 admin 数据库中可用，赋予用户所有数据库的读权限
readWriteAnyDatabase:　　只在 admin 数据库中可用，赋予用户所有数据库的读写权限
userAdminAnyDatabase:　只在 admin 数据库中可用，赋予用户所有数据库的 userAdmin 权限
dbAdminAnyDatabase:　　只在 admin 数据库中可用，赋予用户所有数据库的 dbAdmin 权限。

root　　只在 admin 数据库中可用。超级账号，超级权限（可以锁查询）

Windows Installation

MSI





Reinstall the service.

sc delete MongoDB Remove the MongoDb Service.
mongod --install --serviceName "MongoDB" --config " F:\Program Files\MongoDB\Server\6.0\mongo.conf"

Linux Installation

zip 安装
安装目录下创建 Data logs 目录
mongod --install --serviceName "MongoDB" --config " F:\Program Files\MongoDB\Server\6.0\mongo.conf"

进入 mongo 界面 bin/mongo
添加用户 db.createUser({user:"testAdmin",pwd:"123456",roles:[{role:"userAdminAnyDatabase",db:"admin"}]})
验证用户是否添加成功 mongod --dbpath d:\mongodb\data --auth
配置文件 mongo.conf dbpath=/xxx/data logpath=/xxx/logs/mongodb.log logappend=true journal=true
port = 27017
auth=true 权限检查

命令

mongod
mongod --verson 查看版本
mongo
ERROR 缺少 dll 安装 2015-2019 c++环境整合包

配置

mongod --dbpath “D:\Program Files\MongoDB\Server\3.6\bin\data\db” 安装为 mongodb 为 windows 服务

--logpath “D:\Program Files\MongoDB\Server\3.6\bin\data\log\MongoDB.log”
--serviceName MongoDB
--install

mongo cmd 连接本地的 mongodb 服务

exit 退出 cmd 连接

Structure

MongoDB follows a flexible and schema-less structure, which is different from traditional relational databases. Here’s an overview of the MongoDB structure:

Collections and Documents

Database

MongoDB stores data in databases, each of which can contain multiple collections. A database in MongoDB is similar to a database in SQL.

Collection

A collection is a group of MongoDB documents. It is equivalent to a table in a relational database. Collections do not enforce a schema, which means that documents within a collection can have different fields and structures.

Document

A document is a set of key-value pairs. It is the basic unit of data in MongoDB, similar to a row in a relational database. Documents are stored in BSON format (Binary JSON), which allows MongoDB to support various data types and nested structures.

MongoDB documents have a size limit of 16 MB. If your comments and their nested replies grow too large, you might encounter issues with document size.

Key Characteristics

Schema-less

MongoDB collections do not enforce a rigid schema. Documents within a collection can have varying structures and fields.

Dynamic Schema

Fields can be added to documents at any time without requiring a predefined schema alteration.

Indexing

MongoDB supports secondary indexes to improve query performance.

Aggregation Framework

MongoDB provides an expressive and powerful aggregation framework for data aggregation and analysis.

Scalability

MongoDB is designed for horizontal scalability through sharding, which allows distributing data across multiple servers.

High Availability

MongoDB supports replica sets, providing automated failover and data redundancy.

Core SQL

Database

db.version(); mongo 版本

show dbs 显示数据库（admin local 为系统数据库）

db 查看当前操作的数据库（默认为 test，插入数据才被创建）

use xxdb Switch or directly create a new database. (Remember to insert data, after that, we can see the new database in “show dbs”)

db.dropDatabse() 删除数据库

Collection

show tables 显示当前数据库的所有集合

show collections　　显示当前数据库的所有集合
db.createCollection(“xxtable”)　　创建集合
db.xxtab.drop()　　删除集合

Document

增
db.xxtab.save(　"_id":{"regionId":6,"name":"小燕子"},
{"articleid":"100000","content":"xxx","userid":"1001","nickname":"Rose","createdatetime":new Date(),"likenum":NumberInt(10),"state":null}　)　插入或更新一条数据
db.xxtab.insert({} , {writeConcern: <document>,　ordered: true　})　　插入一条数据
　　writeConcern　写入策略，一般可以不写，有个简单的了解即可。
　　ordered　　指定是否按顺序写入，默认 true，按顺序写入。为 true 时，插入多条数据时，有一条报错就中断后面的操作。为 false，报错的被跳过，后面的继续执行。
db.xxtab.insertOne({})　　插入一条数据，xxtab 集合如果不存在，则会隐式创建
　复制 json 插入时，删除原本的 $ 字段
db.xxtab.insertMany([{}, {}],　)　　插入多条数据
　　double　　mongo 中的数字，默认情况下是 double 类型
　　整型　　NumberInt(10)
　　日期　　new Date()

　　插入规则
　　可以自定义_id，除了可以用字符串、数字以外，还可以用一些比较复杂的数据格式做主键： "_id":{"regionId":6,"name":"小燕子"}
　　插入的数据没有指定 _id ，会自动生成主键值
　　如果某字段没值，可以赋值为 null，或不写该字段。
　　命名规则
　　键不能含有\0 (空字符)。这个字符用来表示键的结尾。
　　.和$有特别的意义，只有在特定环境下才能使用。
　　以下划线"_"开头的键是保留的(不是严格要求的)。

删
db.xxtab.deleteMany ({ status : "A" })　　删除多个
db.xxtab.deleteOne ({ status : "D" })　　删除一个　　“ $ id”:”123”

改
db.xxtab.update({ "_id": ObjectId(documentId), "count":{ $gt:20 } },　{ $set: {"name":"c3"} }　)　　Use id or other fields to filter and update data, only the first item is modified by default
db.xxtab.update({ "count":{$gt:20} }, { $set:{"name":"c4"} }, false, true)
　　upsert　　默认 false，找不到时，插入一条新数据 // {"name":"c4"}
　　multi　　默认 false，　更新所有参数

　　更新操作符
　　{ $inc : { age : 5 } }　　对一个数字字段的某个 field 增加 value　　// 把 age 增加 5
　　{ $set : { field : value } }　　把文档中某个字段 field 的值设为 value
　　{ $unset : { field : 1 } }　　删除某个字段 field
　　{ $push : { field : value } }　　把 value 追加到 field 里。注：field 只能是数组类型，如果 field 不存在，会自动插入一个数组类型
　　{ $pushAll : { "ailas" : ["A1","A2"] } }　　用法同$push 一样，只是$pushAll 可以一次追加多个值到一个数组字段内。
　　{ $addToSet : { field : value } }　　加一个值到数组内，而且只有当这个值在数组中不存在时才增加。
　　{ $pop : { field : -1 } }　　用于删除数组内的一个值，-1 表示第一个值，1 表示最后一个值

{ $pull : { "ailas":"A1"} }　　从 field 数组内删除一个等于 value 的值
{ $pullAll : { "ailas" : ["A1","A2"] } }　　用法同$pull 一样，可以一次性删除数组内的多个值。
{ $rename : {"name":"sname"} }　　字段重命名

查

db.xxtab.find({}, {})　　多条件查询集合中的所有文档
{ field: /xxreg/ }　　正则模糊查询

Find a field
"personList.1.age"　　根据序号选择数组内的第几个元素

group

lookup
{ $lookup: { from: '<reference-collection>}}

match
ObjectId("xxxxx")
// { "_id": ObjectId("xxxxx") }　id 键值对象查询

{ $size:3 }　　数组大小等于
{ $in:['1003','1004'] }　　包含查询
{ $nin:['1003','1004'] }　　不包含查询

{ $gt : 24 }　　大于
{ $lt : 24 }　　小于
{ $gte : 24 }　　大于等于
{ $lte : 24 }　　小于等于
{$ne : 24}　　不等于
{ $and : [{ content:/xxreg/ } , {userid:'1004'}] }　　并且，默认多字段时就是并且查询
{ $or: [{ content:/xxreg/ } , {userid:'1004'}] }　　或者

{$regex:/tom$/}　where name like "%tom"
{$regex:/^t/}　　where name like "t%"
{$regex:/o/}　　where nam like "%o%"
{$regex:/^t/i}　　不分区大小写
{$regex:/^t/, $options:"m"}　在一个文档值存在多字符串记录的情况下（中间以\n 或$分隔），以记录为单位，匹配符合的文档记录。如果文档中不包含\n 或$,则 m 选项不起作用
{$regex:/a/, $options:"x"}　　在字符串文档值中，忽略有空格的、hash 值的或带#的文档记录
{$regex:/^.*C/, $options:"si"}　　在多字符串记录情况下，使用 s 可以做到 pattern 带".*"情况下的多字符串匹配

db.xxtab.find().limit(2).skip(2)　　limit()方法读取指定数量的数据，　skip()方法跳过指定数量的数据
db.xxtab.find().sort({nickname:1})　　排序文档
db.xxtab.find().count()　　查询文档数量
db.xxtab.find().pretty()　　查询美观
db.xxtab.countDocuments({userid:'1003'})　统计文档数量

Index

Types of Indexes

Single Field Index: Indexes on a single field.

```
// Create a single-field index on the "username" field in the "users" collection
db.users.createIndex({ username: 1 }); // Ascending order
db.users.createIndex({ username: -1 }); // Descending order
```

Compound Index: Indexes on multiple fields.

```
// Create a compound index on the "username" and "email" fields in the "users"
collection
db.users.createIndex({ username: 1, email: 1 }); // Both in ascending order
db.users.createIndex({ username: 1, email: -1 }); // Mixed order
```

Multikey Index: Indexes on array fields, where each element of the array is indexed.

```
// Create a multikey index on the "tags" array field in the "posts" collection
db.posts.createIndex({ tags: 1 });
```

Text Index: Indexes for text search.

```
// Create a text index on the "description" field in the "products" collection
db.products.createIndex({ description: "text" });

// Create a text index on multiple fields
db.articles.createIndex({ title: "text", content: "text" });
```

Geospatial Index: Indexes for geospatial data.

```
// Create a 2d index for legacy coordinate pairs in the "places" collection
db.places.createIndex({ location: "2d" });

// Create a 2dsphere index for GeoJSON data in the "places" collection
db.places.createIndex({ location: "2dsphere" });
```

Hashed Index: Hash-based index for sharded collections.

```
// Create a hashed index on the "user_id" field in the "users" collection
db.users.createIndex({ user_id: "hashed" });
```

Key Uses of Indexes in MongoDB

1. Improve Query Performance:
   Indexes allow MongoDB to quickly locate documents that match query criteria, reducing the number of documents that need to be examined.
   This results in faster read operations, especially for large collections.
2. Sorting:
   Indexes can be used to sort the results of a query efficiently.
   If a query includes a sort operation, MongoDB can use the index to return sorted results without performing an in-memory sort.
3. Enforce Uniqueness:
   Unique indexes ensure that a particular field or combination of fields contains unique values across all documents in the collection.
   This is useful for enforcing constraints like unique usernames or email addresses.
4. Facilitate Join Operations:
   In MongoDB, join operations are often performed using the $lookup aggregation stage.
   Indexes can improve the performance of these operations by quickly matching documents from different collections.
5. Support Geospatial Queries:
   Geospatial indexes (2d and 2dsphere) enable efficient querying of spatial data, such as finding all points within a certain radius or finding the nearest location to a given point.
6. Enable Text Search:
   Text indexes allow for efficient text search within string fields.
   They enable complex search queries, such as searching for phrases, excluding certain words, and sorting by relevance.

Lock

MongoDB uses a locking mechanism to ensure data consistency and to handle concurrent operations.
However, MongoDB's locking strategy is designed to be less restrictive and more efficient than traditional database locking mechanisms.

Locking Mechanisms in MongoDB

1. Global Lock:
   In earlier versions of MongoDB (before 2.2), there was a single global lock that controlled access to the entire database instance.
   This could lead to contention and performance bottlenecks in multi-threaded applications.
2. Database Lock:
   MongoDB 2.2 introduced per-database locks, reducing contention by allowing different databases to be locked independently.
   This change improved concurrency by enabling operations on different databases to proceed in parallel.
3. Collection Lock:
   MongoDB 3.0 further improved concurrency by introducing collection-level locks.
   With this mechanism, operations on different collections within the same database can occur simultaneously, reducing lock contention even further.
4. Document Lock:
   MongoDB 4.0 introduced a more granular locking mechanism with document-level locks.
   This means that operations can lock individual documents rather than entire collections, significantly improving performance and concurrency in many scenarios.

Lock Modes

MongoDB supports various lock modes to handle different types of operations:

1. Shared Lock (S):
   Allows multiple readers to access the resource simultaneously but prevents writers from modifying it.
   ```
   // Multiple read operations acquire shared locks
   db.users.find({ username: "johndoe" });
   db.users.find({ age: { $gt: 25 } });
   ```
2. Exclusive Lock (X):
   Allows only one writer to access the resource and prevents readers from reading it until the lock is released.
   ```
   // An update operation acquires an exclusive lock
   db.users.updateOne({ username: "johndoe" }, { $set: { age: 31 } });
   ```
3. Intent Shared Lock (IS):
   Indicates that the operation will read a resource and is a precursor to acquiring a shared lock.
   ```
   // MongoDB internally manages intent shared locks when planning to read data
   db.products.find({ category: "electronics" });
   ```
4. Intent Exclusive Lock (IX):
   Indicates that the operation will write to a resource and is a precursor to acquiring an exclusive lock.
   ```
   // MongoDB internally manages intent exclusive locks when planning to write data
   db.orders.insertOne({ order_id: 1234, product: "laptop", quantity: 1 });
   db.orders.updateOne({ order_id: 1234 }, { $set: { quantity: 2 } });
   ```
5. Mixed Mode Locks:
   MongoDB also supports mixed mode locks like SIX (Shared Intent Exclusive), which combines intent and shared locks to optimize concurrency.
   ```
   // MongoDB internally manages shared intent exclusive locks
   // Example: Read and then update some documents in a collection
   var product = db.products.findOne({ product_id: 101 }); // Shared lock
   db.products.updateOne({ product_id: 101 }, { $set: { stock: product.stock - 1 } }); // Exclusive lock
   ```

Functional SQL

聚合查询

$eq　查询完全匹配

$ne　查询不匹配

```
"$not": {　"$size": 0　}
    "array": {
          "$gt": []
     }
    "array.0": {
          "$exists": true
     }
```

```
{
     '$addFields': {'objId':{'$toObjectId':'$studentId'}}          如果是 ObjectId 转化为 String，将'$toObjectId'改为'$toString'
}
```

```
{
     '$lookup':{                          关联查询
          'from': 'student',                    关联表
          'localField' : 'studentId',           当前表 字段
          'foreignField' : '_id',               关联表 字段
          'as' : 'student'                      当前表结果字段的名称，保存有关联表结果数组
     }
}
```

```
{
     '$project':{              提取，重命名 想要的字段
          'sex':'$student.sex'          为字段 student.sex 设置别名
          'name': 1,                    保留 name 字段
          'age':0                       去掉 age 字段
     }
}
```

```
{
     '$match':{              过滤数据，可以对$lookup 返回的 关联表数据 字段再进行筛选
          'sex;': {'$ne': null, $exists:true}          不为空
          ‘name’: /^.*xxxxx.*$/                       正则匹配
     },
}
```

```
{
      $unwind: "$xxx"          根据数组字段 xxx，拆分数据，其他字段保持一致，但是 xxx 字段拆分成多个元素
}
```

```
{
    $group: {                    在上一步的结果之上再执行统计，对返回字段 name 进行分组操作，也可以用于提取新字段（_id 表示识别的 类型，不同为一条新数据）
```

```
        "_id": {
            "country" : "$_id.country",
            "province": "$_id.province"
        },
        name: “$name”,
        totalQuantity: { $sum: “$quantity” }
        subjectId: { $addToSet: '$subjectId' }
        count : { $sum : 1 }   }
        details: {
            $push: {
                product_id: "$_id.product_id",
                name: '$product_name',
                price: '$price',
                amount: '$amount',
                packing: '$packing',
                quantity: '$quantity'
            }
        }
}

{
    $count: “resultCount”      定义一个字段显示返回数量
}

{
    $limit: 2                  限制返回的数据条数
}
```

DAM / Redis

Core

Reference:

https://www.geeksforgeeks.org/introduction-to-redis-server/

Redis (Remote DIctionary Server) is an open-source, in-memory, NoSQL key/value store that works by storing data in a computer's RAM, which makes it very fast to read and write.

Redis uses a single-threaded architecture, which means that it processes requests sequentially, one at a time.

This ensures data consistency and avoids the challenges of managing multiple threads accessing shared data simultaneously.

Redis also uses non-blocking I/O operations, which means that it doesn't block the entire server when it needs to perform I/O tasks, like reading or writing data from disk

Persistence

Redis offers two main mechanisms for persistence: **RDB snapshots (Redis Database snapshots)**and **AOF (Append-Only File)**.

These mechanisms ensure that Redis data is durable and can be recovered in case of server restarts or failures.

RDB Snapshots

RDB snapshots create point-in-time snapshots of the Redis dataset at specified intervals.

Locate the RDB File

The RDB file is usually named `dump.rdb` by default and is located in the directory specified in your Redis configuration

(`redis.conf`).

This file contains a binary snapshot of the Redis dataset.

When Redis starts, it looks for the `dump.rdb` file in the specified directory (`dir` configuration in `redis.conf`). If the file exists, Redis automatically loads the data from this snapshot file into memory.

How it Works

Redis periodically forks a child process that writes the dataset to a binary file (RDB file).

By default, Redis saves an RDB snapshot to disk:

Every 900 seconds (15 minutes) if at least 1 key changed (save 900 1).

Every 300 seconds (5 minutes) if at least 10 keys changed (save 300 10).

Every 60 seconds if at least 10,000 keys changed (save 60 10000).

These configurations can be adjusted in the redis.conf file or via runtime configuration commands (CONFIG SET).

Usage

- Enable RDB persistence by configuring save parameters in `redis.conf` or using `redis-cli config set`.
- Manually trigger a snapshot using the `redis-cli save` or `bgsave` (forks a child process for snapshot in the background) commands.
- To disable RDB persistence, set `save ""`.

Advantages:

High Efficiency:

RDB snapshots are compact and load quickly, making them ideal for large-scale data recovery.

Low Storage Space:

RDB files are binary and compressed, which makes them space-efficient.

Minimal Performance Impact:

Since RDB persistence is done in the background, it has a minimal effect on Redis server performance.

Disadvantages:

Potential Data Loss:

Since RDB snapshots are taken periodically, any changes made after the last snapshot and before a crash could be lost.

Not Suitable for Real-Time Data:

For scenarios requiring real-time data persistence, RDB may not be ideal due to its periodic nature.

AOF (Append-Only File)

AOF logs every write operation received by the Redis server, making it a log of commands that can be replayed to rebuild the dataset.

How it Works

Periodically, Redis forks a child process to perform the actual writing of commands from memory to the AOF file(`appendonly.aof`).

This is done asynchronously to minimize the impact on the main Redis process and to avoid blocking client requests during the disk I/O operations.

Redis writes each command to the AOF file as it executes the command.

AOF file is append-only, ensuring that the data is not overwritten and can be replayed to reconstruct the dataset.

AOF file can be configured to rewrite itself in the background to avoid growing too large (auto-aof-rewrite-percentage and auto-aof-rewrite-min-size settings).

Usage:

- Enable AOF persistence in redis.conf (appendonly yes) or using redis-cli config set.

  To disable AOF persistence, set appendonly no.
- Choose AOF rewrite options (always, everysec, or no) to control how often Redis rewrites the AOF file.

- Manually trigger an AOF rewrite using the bgrewriteof command to reduce file size and optimize performance.
    This command is executed in the background, meaning it does not block the Redis server's normal operations while it's running.

Advantages:

Better Data Protection:

AOF provides better data durability compared to RDB since it logs every write operation. This means less data is at risk of being lost.

Higher Data Real-Time Accuracy:

You can configure AOF to persist data more frequently, reducing the risk of data loss.

Disadvantages:

Larger File Size:

AOF files are generally larger than RDB files because they log every write operation as text.

Slower Recovery:

Restoring from an AOF file can be slower than loading an RDB snapshot, as Redis needs to replay all the logged commands.

Choosing Between RDB and AOF

Your choice between RDB and AOF depends on your application's needs:

If you prioritize fast recovery and minimal impact on performance, RDB is preferable.

If you need better data durability and can tolerate slower recovery, AOF is more suitable.

Many Redis users combine both mechanisms to balance between quick recovery times and minimal data loss, using RDB for regular backups and AOF for logging real-time changes.

Redis Expired Key Deletion Strategies

Redis manages expired keys using three main strategies: Periodic Deletion, Lazy Deletion, and Active Deletion.
These strategies work together to efficiently remove expired keys and free up memory.

Periodic Deletion

Redis periodically checks for expired keys in the database and removes them. The key aspects of this strategy include:

Scheduled Task:

By default, Redis runs a task every 100 milliseconds to randomly check a subset of keys that have expiration times set.

Sampling:

Redis doesn't check all keys at once. Instead, it samples a portion of the keys to keep the overhead low, ensuring that the server performance isn't significantly impacted.

Iteration:

If a significant number of keys are found to be expired in one run, Redis might repeat the check process a few more times.

Lazy Deletion

This strategy kicks in when a client attempts to access an expired key:

To manually check for expired keys and delete them, you would generally use the TTL command to check the time-to-live of a key.

If it returns a value of -2, the key has already expired.

On Access:

When a key is accessed, Redis checks if it has expired. If it has, Redis will delete the key immediately and return a result indicating the key doesn't exist.

On Demand:

This approach ensures that Redis doesn't waste resources checking all keys constantly, but only deletes keys when necessary.

Active Deletion

Redis uses this strategy when it is running low on memory:

Memory Pressure:

When Redis reaches its memory limit, it proactively scans the database for expired keys and removes them to free up space.

Eviction Policy:

If memory is still insufficient, Redis may apply its configured eviction policy to remove additional keys, which could include both expired and non-expired keys, depending on the policy.

Lua Script

Variables and Data Types

```
-- Variables
local a = 10
local b = "Hello, Lua"
local c = true

-- Table (Lua's only complex data structure, used for arrays, dictionaries, etc.)
local t = { key1 = "value1", key2 = "value2" }
```

Control Structures

```
-- If statement
if a > 5 then
    print("a is greater than 5")
else
    print("a is not greater than 5")
end

-- While loop
while a > 0 do
    print(a)
    a = a - 1
end

-- For loop
for i = 1, 5 do
    print(i)
end
```

Functions

```
-- Function definition
function add(x, y)
    return x + y
end

-- Function call
local sum = add(5, 3)
print(sum) -- Output: 8
```

Tables

```
-- Creating and using tables
local person = { name = "John", age = 30 }

-- Accessing table fields
print(person.name) -- Output: John
print(person["age"]) -- Output: 30

-- Iterating over a table
```

```
for key, value in pairs(person) do
    print(key, value)
end
```

Running Script in Redis

Lua is extensively used for scripting in Redis to perform atomic operations and complex transactions.
Redis ensures atomicity for Lua scripts, meaning that while a script is running, no other commands can be executed by other clients that might affect the data being accessed or modified by the script.

Operations executed within a Lua script using the EVAL command in Redis are atomic. This means that the script runs as a single, uninterrupted operation from Redis's perspective.

```
EVAL "local key = KEYS[1] local increment = tonumber(ARGV[1]) local current = tonumber(redis.call('GET', key) or '0') local new = current + increment redis.call('SET', key, new) return new" 1 mykey 10
```

Key Points on Atomicity and Concurrency with RedisTemplate

Atomic Operations

- Single Command: If you're performing single operations (e.g., SET, GET, INCR), these are atomic at the Redis level. For instance, calling redisTemplate.opsForValue().set(key, value) will be executed atomically by Redis, ensuring that no other command will interrupt this operation on that key.
- Lua Scripts: When using RedisTemplate to execute Lua scripts via the execute method, the script's operations are atomic and isolated, just as they would be with direct EVAL commands.

Multiple Commands

- Non-Atomic Multiple Operations: If you perform multiple operations in sequence (e.g., read-modify-write), there is no built-in atomicity across these commands unless explicitly managed. For instance, calling redisTemplate.opsForValue().get(key) followed by redisTemplate.opsForValue().set(key, newValue) is not atomic. Other clients could modify the key between these operations, leading to race conditions.

Transactions:

- Redis Transactions: You can use Redis transactions (via the multi/exec commands) to ensure that a sequence of commands is executed atomically. Redis transactions ensure that all commands in a transaction block are executed as a single operation, though they do not provide isolation in the same way as Lua scripts.
- Spring Data Redis Transactions: Spring Data Redis provides transaction support through the RedisTemplate by using the execute method with SessionCallback to execute a series of commands atomically.

Configuration
redis.conf

Advanced Configuration

hash-max-ziplist-entries <entries>

- Maximum number of entries in a hash to use a ziplist.

```
hash-max-ziplist-entries 512
```

hash-max-ziplist-value <bytes>

- Maximum size of an entry in a hash to use a ziplist.

```
hash-max-ziplist-value 64
```

list-max-ziplist-size <size>

- Maximum ziplist size for lists.

```
list-max-ziplist-size -2
```

list-compress-depth <depth>

- Depth for list compression.

```
list-compress-depth 0
```

set-max-intset-entries <entries>

Maximum entries for intsets in sets.

```
set-max-intset-entries 512
```

zset-max-ziplist-entries <entries>

Maximum entries in zsets to use a ziplist.

```
zset-max-ziplist-entries 128
```

zset-max-ziplist-value <bytes>

Maximum size of an entry in zsets to use a ziplist.

```
zset-max-ziplist-value 64
```

hll-sparse-max-bytes <bytes>

Maximum sparse representation size for HyperLogLog.

```
hll-sparse-max-bytes 3000
```

activerehashing <yes/no>

Enable active rehashing.

```
activerehashing yes
```

client-output-buffer-limit <class> <hard-limit> <soft-limit> <soft-seconds>

Client output buffer limits for different classes.

```
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
```

Basic Server Configuration

port <port-number>

The port Redis will listen on. Default is 6379.

```
port 6379
```

bind <ip>

The network interface(s) Redis will bind to.

```
bind 127.0.0.1
```

tcp-backlog <number>

TCP listen() backlog. Default is 511.

```
tcp-backlog 511
```

timeout <seconds>

Close the connection after a client is idle for N seconds (0 to disable).

```
timeout 0
```

tcp-keepalive <seconds>

TCP keepalive interval in seconds.

```
tcp-keepalive 300
```

Cluster

cluster-enabled <yes/no>

Enable cluster mode.

```
cluster-enabled no
```

cluster-config-file <filename>

Cluster configuration file.

```
cluster-config-file nodes.conf
```

cluster-node-timeout <milliseconds>

Cluster node timeout.

```
cluster-node-timeout 15000
```

cluster-replica-validity-factor <factor>

Cluster replica validity factor.

```
cluster-replica-validity-factor 10
```

cluster-migration-barrier <number>

Cluster migration barrier.

```
cluster-migration-barrier 1
```

cluster-require-full-coverage <yes/no>

Require full coverage for cluster.

```
cluster-require-full-coverage yes
```

Event Notification

notify-keyspace-events <events>

Event notifications.

```
notify-keyspace-events "AKE"
```

The notify-keyspace-events parameter in Redis is used to configure Redis keyspace notifications, which allow clients to receive events related to changes in the data stored in the Redis database.

Event Types

The string value of notify-keyspace-events consists of zero or more of the following characters, each representing a different type of event:

K: Keyspace events, published with the prefix __keyspace@<db>__.
E: Keyevent events, published with the prefix __keyevent@<db>__.
g: Generic commands like DEL, EXPIRE, RENAME, etc.
$: String commands like SET, APPEND, etc.
l: List commands like LPUSH, RPUSH, etc.
s: Set commands like SADD, SREM, etc.
h: Hash commands like HSET, HDEL, etc.
z: Sorted set commands like ZADD, ZREM, etc.
x: Expired events (events generated when keys expire).
e: Evicted events (events generated when keys are evicted).
A: Alias for g$lshzxe, representing all event types except for stream events.

Latency Monitor

latency-monitor-threshold <milliseconds>

Threshold for latency monitoring.

```
latency-monitor-threshold 100
```

Limit and Memory Management

hz <number>

The hz value is measured in Hertz (Hz), meaning it represents how many times per second these tasks are run.

The default value is 10, which means Redis performs these tasks ten times per second.

The hz parameter sets how many times per second Redis will perform its internal housekeeping tasks. These tasks include:

Checking for expired keys (periodic deletion).
Handling client connections and disconnections.
Performing AOF (Append-Only File) rewrites.
Resizing internal data structures, like hash tables.

maxclients <number>

Maximum number of connected clients.

```
maxclients 10000
```

maxmemory <bytes>

Maximum memory usage limit.

```
maxmemory 2gb
```

maxmemory-samples <number>

Number of samples to check for eviction.

```
maxmemory-samples 5
```

maxmemory-policy <policy>

Eviction policy when memory limit is reached.

```
maxmemory-policy noeviction
```

Choose the Eviction Policy:

- noeviction:
  - Default policy. Redis will return an error when the memory limit is reached and you try to insert more data.
- allkeys-lru:
  - Removes the least recently used (LRU) keys from the database to make room for new data.
- allkeys-lfu:
  - Removes the least frequently used (LFU) keys from the database.
- allkeys-random:
  - Randomly removes any keys from the database.

- volatile-lru:
  - Removes the least recently used (LRU) keys with an expiration set.
- volatile-lfu:
  - Removes the least frequently used (LFU) keys with an expiration set.
- volatile-random:
  - Randomly removes keys with an expiration set.

- LRU (Least Recently Used) Cache Eviction Strategy
  - Weaknesses:
    - Not Frequency-Aware:
      - LRU evicts keys that haven't been accessed recently, but it doesn't consider how frequently a key has been accessed.
      - A frequently accessed key that hasn't been used recently might be evicted, which could be inefficient for certain workloads.
    - Cache Pollution:
      - If a key is accessed once and then not used again, it could take up valuable space in the cache and prevent more useful keys from being cached.
  - Potential Solutions:
    - Hybrid Approach:
      - Although Redis doesn’t offer a built-in hybrid of LRU and LFU, you can simulate it by using different eviction policies in combination (e.g., volatile-lru for certain keys and allkeys-lfu for others).
    - Allkeys-LFU:
      - If you are concerned about cache pollution, consider using allkeys-lfu, which considers frequency, reducing the risk of keeping less useful data.
- LFU (Least Frequently Used) Cache Eviction Strategy
  - Weaknesses:
    - Stale Data:
      - LFU can hold onto data that was frequently accessed in the past but is no longer relevant. This can lead to cache staleness, where outdated or less useful keys remain in the cache longer than necessary.
    - Frequency Bias:
      - LFU may unfairly favor keys that had a high access frequency in the past but are no longer being accessed as frequently. This can prevent newer, more relevant keys from being cached.
    - Overhead:
      - Maintaining frequency counters for each key can add computational and memory overhead,

especially in scenarios with a large number of keys or high churn rates.

Potential Solutions:

Tuning LFU Decay Time:

Adjust the LFU decay time using the lfu-decay-time setting in Redis. This setting controls how quickly the frequency counter decays over time, reducing the likelihood of retaining stale data.

Using TTL with LFU:

Set a Time-To-Live (TTL) on keys alongside LFU to ensure that even frequently accessed keys are eventually evicted, preventing them from becoming stale.

Save and Close the File

For the changes to take effect, you need to restart the Redis server.

If you prefer not to modify the redis.conf file or need to change the configuration dynamically, you can use the CONFIG SET command in the Redis CLI.

This will change the configuration temporarily until Redis is restarted. To make it persistent, you would need to update the redis.conf file as described above.

Least Recently Used (LRU)

Redis maintains a doubly linked list of keys. Every time a key is accessed, it's moved to the head of the list. The tail of the list contains the least recently used keys.

Least Frequently Used (LFU)

Redis maintains a counter for each key, incrementing it on each access. To prevent excessive counter growth, Redis uses an approximated LFU algorithm.

Redis doesn't track the exact frequency of key accesses for every key.

Doing so would be computationally expensive and memory-intensive. Instead, it uses an approximated LFU algorithm to estimate which keys are accessed least frequently.

Morris Counter:

Redis employs a probabilistic counter called a Morris counter to estimate access frequency for each key. This counter requires very little memory.

Decay:

To prevent old access data from affecting eviction decisions, the counter value is decayed over time.

Sampling:

When selecting a key for eviction, Redis samples a small subset of keys and chooses the one with the lowest estimated frequency.

Logging

slowlog-log-slower-than <microseconds>

Log queries slower than the specified time in microseconds.

```
slowlog-log-slower-than 10000
```

slowlog-max-len <length>

Maximum length of the slow log.

```
slowlog-max-len 128
```

loglevel <level>

Server verbosity levels (debug, verbose, notice, warning).

```
loglevel notice
```

logfile <path>

Log file path. Empty string logs to stdout.

```
logfile ""
```

databases <number>

Number of databases. Default is 16.

```
databases 16
```

Lua Scripting

lua-time-limit <milliseconds>

Maximum execution time for Lua scripts.

```
lua-time-limit 5000
```

Replication

replicaof <masterip> <masterport>

The IP address and port of the master.

```
replicaof 192.168.1.1 6379
```

masterauth <password>

Master server password for authentication.

```
masterauth mypassword
```

repl-backlog-size <bytes>

Cnfigure the size of the replication backlog buffer. This buffer is used during the replication process between the master and slave servers.

```
repl-backlog-size 1mb  # The default size of the backlog buffer is 1 megabyte.
```

slave-priority <priority>

Defines the priority of a slave for promotion to master during failover.

Lower values indicate higher priority. A value of 0 means the slave will never be promoted.

```
slave-priority 100
```

replica-serve-stale-data <yes/no>

Whether to serve stale data or not when the replica is disconnected.

```
replica-serve-stale-data yes
```

replica-read-only <yes/no>

Make replica instances read-only.

```
replica-read-only yes
```

repl-diskless-sync <yes/no>

Use diskless replication.

```
repl-diskless-sync no
```

repl-diskless-sync-delay <seconds>

Diskless replication delay in seconds.

```
repl-diskless-sync-delay 5
```

repl-ping-slave-period <seconds>

Controls the frequency at which the master sends PING commands to its connected slaves to ensure that the connection is alive.

```
repl-ping-slave-period 5
```

repl-disable-tcp-nodelay <yes/no>

Disable TCP_NODELAY during synchronous replication.

```
repl-disable-tcp-nodelay no
```

replica-priority <number>

Replica priority for failover.

```
replica-priority 100
```

min-replicas-to-write <number>

Minimum number of replicas to perform writes.

```
min-replicas-to-write 3
```

min-replicas-max-lag <seconds>

Maximum lag for min-replicas-to-write.

```
min-replicas-max-lag 10
```

Security

requirepass <password>

Password for clients to connect.

```
requirepass mypassword
```

rename-command <original-command> <new-command>

Rename dangerous commands.

```
rename-command FLUSHALL ""
```

Sentinel

sentinel monitor <name> <ip> <port> <quorum>

Define a monitored master with a quorum.

```
sentinel monitor mymaster 127.0.0.1 6379 2
```

sentinel auth-pass <master-name> <password>

Authentication password for the master.

```
sentinel auth-pass mymaster mypassword
```

sentinel down-after-milliseconds <master-name> <milliseconds>

Time after which the master is considered down.

```
sentinel down-after-milliseconds mymaster 30000
```

sentinel parallel-syncs <master-name> <number>

Number of replicas to reconfigure in parallel.

```
sentinel parallel-syncs mymaster 1
```

sentinel failover-timeout <master-name> <milliseconds>

Failover timeout.

```
sentinel failover-timeout mymaster 180000
```

Snapshots

save <seconds> <changes>

Snapshotting configuration. Save the DB if both the given number of seconds and the given number of write operations against the DB occurred.

```
save 900 1       # To save an RDB snapshot every 900 seconds if at least 1 key changed:
save 300 10
save 60 10000
```

dbfilename <filename>

Filename for RDB snapshots.

x

```
dbfilename dump.rdb
```

dir <path>

Working directory.

by default, both the RDB (Redis Database) snapshot files and the AOF (Append-Only File) files are saved in the working directory specified by the dir configuration setting in the redis.conf file.

```
dir ./
```

stop-writes-on-bgsave-error <yes/no>

Stop writing if there are RDB saving errors.

```
stop-writes-on-bgsave-error yes
```

rdbcompression <yes/no>

Compress RDB files.

```
rdbcompression yes
```

rdbchecksum <yes/no>

Controls whether Redis performs a checksum on RDB (Redis Database Backup) files when reading or writing them. This checksum is used to detect potential data corruption.

```
rdbchecksum yes
```

Persistence

appendonly <yes/no>

Enable append-only file persistence.

```
appendonly yes
```

appendfilename <filename>

Name of the append-only file.

The default name of the AOF file is appendonly.aof.

```
appendfilename appendonly.aof
```

appendfsync <policy>

File synchronization policy for AOF (always, everysec, no).

fsync is a system call used in operating systems to ensure that all pending changes (data and metadata) in memory are synchronized and written to the persistent storage device (such as a hard disk or SSD).

```
appendfsync everysec
```

AOF Persistence:

In Redis, when using AOF persistence, fsync is triggered after appending each write command to the AOF file.

This ensures that every write operation is safely stored on disk, providing durability.

RDB Snapshots:

For RDB snapshots, fsync ensures that the snapshot file is fully written to disk before Redis considers the save operation complete.

This prevents data loss in case of a crash during the snapshotting process.

always

Perform fsync after every write operation to the AOF.

everysec

Perform fsync every second if there were changes to the AOF.

no

Let the operating system decide when to perform `fsync` (least safe option in terms of durability but highest performance).

no-appendfsync-on-rewrite <yes/no>

Disable fsync during AOF rewrite.

```
no-appendfsync-on-rewrite no
```

auto-aof-rewrite-percentage <percentage>

Determines when the AOF will be rewritten based on its size relative to its size after the last rewrite.

If set to 100, the AOF will be rewritten when its size doubles compared to its size after the last rewrite.

```
auto-aof-rewrite-percentage 100
```

auto-aof-rewrite-min-size <bytes>

Specifies the minimum size the AOF must reach before a rewrite is triggered, even if the percentage increase is met.

Smaller percentage:

More frequent rewrites, potentially improving performance but increasing CPU usage.

Larger percentage:

Less frequent rewrites, potentially reducing CPU usage but increasing the risk of data loss in case of a crash if the AOF becomes very large.

Larger minimum size:

Delays rewrites for smaller AOF files, reducing unnecessary CPU overhead.

```
auto-aof-rewrite-min-size 64mb
```

sentinel.conf

Basic Configuration

port <port-number>

The port Sentinel will listen on. Default is 26379.

```
port 26379
```

bind <ip>

The network interface(s) Sentinel will bind to.

```
bind 127.0.0.1
```

daemonize <yes/no>

Run Sentinel as a background daemon.

```
daemonize no
```

pidfile <path>

The path to the Sentinel process ID file.

```
pidfile /var/run/redis-sentinel.pid
```

logfile <path>

The path to the Sentinel log file.

```
logfile ""
```

loglevel <level>

The logging level (debug, verbose, notice, warning).

```
loglevel notice
```

Monitored Master Configuration

sentinel monitor <master-name> <ip> <port> <quorum>

Define a master to monitor with its IP, port, and quorum.

```
sentinel monitor mymaster 127.0.0.1 6379 2
```

Quorum

The number of Sentinel instances that must agree that the master is down before Sentinel initiates a failover process.

sentinel auth-pass <master-name> <password>

Password for authenticating with the monitored master.

```
sentinel auth-pass mymaster mypassword
```

sentinel down-after-milliseconds <master-name> <milliseconds>

Time in milliseconds after which the master is considered down.

This parameter determines how long it takes for a master instance to be considered down after it loses connection.

```
sentinel down-after-milliseconds mymaster 30000
```

sentinel parallel-syncs <master-name> <number>

Number of replicas to reconfigure in parallel during failover.

```
sentinel parallel-syncs mymaster 1
```

sentinel failover-timeout <master-name> <milliseconds>

Maximum time to wait for a failover to complete.

```
sentinel failover-timeout mymaster 180000
```

sentinel config-epoch <master-name> <epoch>

The configuration epoch for the master.

```
sentinel config-epoch mymaster 1
```

sentinel leader-epoch <master-name> <epoch>

The epoch of the current leader.

```
sentinel leader-epoch mymaster 1
```

Notification Script

sentinel notification-script <master-name> <script>

Path to a script that will be executed when a Sentinel event occurs.

```
sentinel notification-script mymaster /path/to/notification_script.sh
```

sentinel client-reconfig-script <master-name> <script>

Path to a script that will be executed to reconfigure clients.

```
sentinel client-reconfig-script mymaster /path/to/client_reconfig_script.sh
```

Customizable Timeouts

sentinel reconnect-period <milliseconds>

Period to attempt reconnection to a master or replica in milliseconds.

```
sentinel reconnect-period 10000
```

sentinel monitor-period <milliseconds>

Period for monitoring master and replica instances in milliseconds.

```
sentinel monitor-period 5000
```

Advanced Configuration

sentinel myid <id>

Unique identifier for this Sentinel instance.

```
sentinel myid 1234567890abcdef1234567890abcdef
```

sentinel announce-ip <ip>

The external IP address Sentinel will announce.

```
sentinel announce-ip 203.0.113.1
```

sentinel announce-port <port>

The external port Sentinel will announce.

```
sentinel announce-port 26379
```

sentinel current-epoch <epoch>

The current configuration epoch for the Sentinel.

```
sentinel current-epoch 1
```

sentinel known-replica <master-name> <ip> <port>

Additional known replica for the master.

```
sentinel known-replica mymaster 192.168.1.2 6379
```

sentinel known-sentinel <master-name> <ip> <port> <runid>

Additional known Sentinel for the master.

```
sentinel known-sentinel mymaster 192.168.1.3 26379 abcdef1234567890abcdef1234567890
```

sentinel deny-scripts-reconfig <yes/no>

Deny script reconfiguration.

```
sentinel deny-scripts-reconfig no
```

Cluster

Reference:

How Redis Cluster Maintains Availability while Re-Sharding

https://www.linkedin.com/pulse/how-redis-cluster-maintains-availability-while-tarun-annapareddy

Redis Cluster: Setup, Sharding and Failover Testing

https://opstree.com/blog/2019/10/29/redis-cluster-setup-sharding-and-failover-testing/

Purpose

Provides horizontal scaling and automatic data sharding.

Distributes data across multiple nodes using a hash slot mechanism.

Ensures data availability and partition tolerance.

Data sharding is useful when the system receives a large number of reads and writes, and a single master server cannot handle all the reads.

Data Sharding Process

In the sharding mechanism we discussed, we took the key and mapped it directly to the database by using consistent hashing. But Redis does it differently.

Partitioning

Distribute Slots

The cluster has 16,384 hash slots, distributed among the master nodes.

The entire Redis cluster has 16,384 slots, evenly divided into the shards. So if we have 10 shards, each shard will contain 1638 slots, and some will accommodate 4 extra slots.

Host	#Hash Slots	Total Slots
host1	0-5500	5501
host2	5501-11000	5500
host3	11001- 16383	5383
Total		16384

Distribute Data

Data is automatically partitioned across multiple master nodes using a hash slot mechanism.

Each key is assigned to a specific hash slot, and each master node is responsible for a subset of the hash slots.

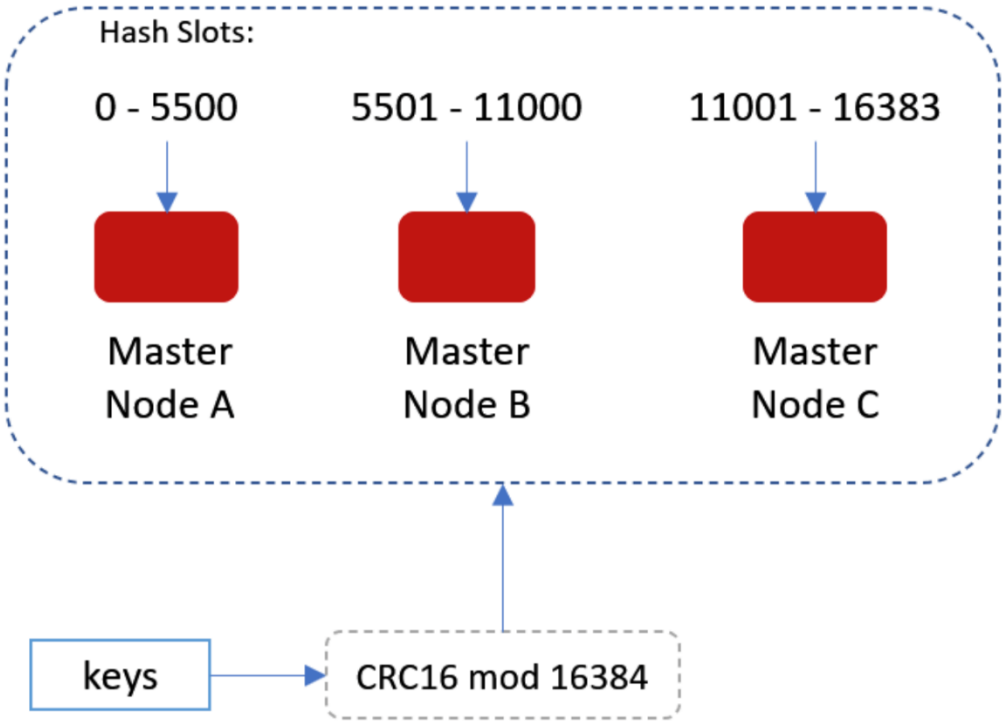
With sharding, Redis pick all the keys with the same hash and store them in a single database shard.

Now, every key we need to store to Redis is mapped to one of the Slots using the hash. I.e., it does key/16384. So, it indirectly maps to the Shard

CRC16

Redis uses the **CRC16** hash function for sharding when using its native partitioning mechanism

The CRC16 function, short for Cyclic Redundancy Check (CRC) with a 16-bit checksum, is a widely used error-detection algorithm that generates a short, fixed-size checksum for a given block of data,
such as a message or file. This checksum is used to detect accidental changes to raw data.



Rebalancing

Now, consider that we have created 10 shards for our order delivery application.

Our application became famous, and we started getting more orders.

After a point, we might have to scale out and increase the shards to 20.

Scale Redis Cluster

Consider that we must scale up the Redis cluster to add 2 more clusters.

Redis internally re-distributes the slots equally among the host nodes. So, now each shard contains 3277 slots.

Redis does not change the hash value of keys when adding a new node. Instead, it redistributes the slots among the

nodes, including the new one.

The cluster administrator or Redis automatically selects a set of slots that will be moved from the existing nodes to the new node.

Host	# Old Hash Slots	Old Total Slots	#New Hash Slots	New Total Slots
host1	0-5500	5501	0 – 3276	3,277
host2	5501-11000	5500	3277 – 6554	3,277
host3	11001- 16383	5383	6555 – 9831	3,277
host4			9832 – 13,108	3,277
host5			13,109 – 16,383	3,276
Total		16384		16384

Redistribute Data

Once the slots are reassigned, Redis begins migrating keys that belong to the moved slots from their original nodes to the new node.

Since the hash function (e.g., CRC16) remains unchanged, there is no re-hashing of individual keys.

The only change is which node manages which slot. Keys that belong to slots reassigned to the new node are migrated to that node, but their hash values and slot assignments do not change.

Advantage of Shards Over Consistent Hashing:

Slots eliminate the tight coupling between the Keys and Shards

- Even though we add extra shards, the slot that the key has to go into is fixed.
- So, until migration, every key goes to the old shard that holds it.
- After migration, the new shard updates its mapping to accommodate the new slots and starts accepting queries from them.
- Also, During the migration process, there will be no chance of duplication as Redis manages and migrates slot units. In case of discrepancy, the key is hashed again, and the shards redirect the query to new shards.
- So, there is no chance of data duplication here, and Redis handles the re-sharding of keys without downtime.

Example

There are 16384 slots. These slots are divided by the number of servers.

If there are 3 servers; 1, 2 and 3 then

- Server 1 contains hash slots from 0 to 5500.
- Server 2 contains hash slots from 5501 to 11000.
- Server 3 contains hash slots from 11001 to 16383.

Shell Command Test

```
redis-cli -c -h 172.19.33.7 -p 7000
172.19.33.7:7000> set a 1
-> Redirected to slot [15495] located at 172.19.45.201:7000
OK
172.19.45.201:7000> set b 2
-> Redirected to slot [3300] located at 172.19.33.7:7000
OK
172.19.33.7:7000> set c 3
-> Redirected to slot [7365] located at 172.19.42.44:7000
OK
172.19.42.44:7000> set d 4
-> Redirected to slot [11298] located at 172.19.45.201:7000
OK
172.19.45.201:7000> get b
```

```
-> Redirected to slot [3300] located at 172.19.33.7:7000
"2"
172.19.33.7:7000> get a
-> Redirected to slot [15495] located at 172.19.45.201:7000
"1"
172.19.45.201:7000> get c
-> Redirected to slot [7365] located at 172.19.42.44:7000
"3"
172.19.42.44:7000> get d
-> Redirected to slot [11298] located at 172.19.45.201:7000
"4"
```

Configure Cluster

Start Cluster Servers

Create configuration files for each node (node1.conf, node2.conf, ..., node6.conf):

```
port 7000
cluster-enabled yes
cluster-config-file nodes-7000.conf
cluster-node-timeout 5000
appendonly yes
```

Repeat for each node with appropriate port numbers (7001, 7002, ...).

Start each Redis server with their respective configuration:

```
redis-server /path/to/node1.conf
redis-server /path/to/node2.conf
...
redis-server /path/to/node6.conf
```

Create the cluster

```
redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004
127.0.0.1:7005 --cluster-replicas 1
```

Verification

Check cluster nodes:

```
redis-cli -p 7000 cluster nodes
```

Add data to the cluster:

```
redis-cli -p 7000
set key1 "value1"
get key1
```

Verify the key exists in one of the nodes in the cluster:

```
redis-cli -p 7001
get key1
```

Configure Redis-Cluster-Proxy

Set Up Your Redis Cluster

Install Redis on all nodes.

Configure Redis Nodes with the following in the redis.conf file

```
port 6379
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

Start Redis Instances on all nodes:

```
redis-server /path/to/redis.conf
```

Create the Cluster by connecting to one of the nodes and using the redis-cli tool:

```
redis-cli --cluster create <node1-ip>:6379 <node2-ip>:6379 <node3-ip>:6379 <node4-ip>:6379 <node5-
ip>:6379 <node6-ip>:6379 --cluster-replicas 1
```

Install Redis-Cluster-Proxy

Download and install Redis-Cluster-Proxy from the official GitHub repository. You can build it from source using the following commands:

```
git clone https://github.com/RedisLabs/redis-cluster-proxy.git
cd redis-cluster-proxy
make
```

After building, you should have an executable called redis-cluster-proxy.

Configure Redis-Cluster-Proxy

Create a configuration file for Redis-Cluster-Proxy, typically named proxy.conf. Below is an example configuration:

```
# Proxy port
port 7777

# Cluster nodes
cluster-announce-ip 127.0.0.1
cluster-announce-port 7777
cluster-announce-bus-port 17777

# List of initial nodes in the cluster
cluster-addr-mapping {
    node1 127.0.0.1:6379
    node2 127.0.0.2:6379
    node3 127.0.0.3:6379
    node4 127.0.0.4:6379
    node5 127.0.0.5:6379
    node6 127.0.0.6:6379
}

# Log file location
logfile "/var/log/redis-cluster-proxy.log"
```

Start Redis-Cluster-Proxy

Run the Redis-Cluster-Proxy with the configuration file:

```
./redis-cluster-proxy /path/to/proxy.conf
```

Connect to Redis-Cluster-Proxy

Once the proxy is running, you can connect to it as you would connect to a Redis server. For example, using redis-cli:

```
redis-cli -p 7777
```

You can also update your application configuration to point to the Redis-Cluster-Proxy's address and port (e.g., 127.0.0.1:7777),
instead of directly connecting to individual Redis cluster nodes.

High Availability and Failover

Redis-Cluster-Proxy handles failover automatically, ensuring that requests are redirected to the available nodes in the cluster if any node goes down.

It maintains the state of the cluster and provides consistent hashing, making it easier to manage client connections and maintain high availability.

Replication

Purpose

Redis supports master-slave replication, where data from a master instance is copied to one or more slave instances.

Replication ensures high availability and read scalability.

Read Requests

Slaves can serve read requests, offloading the read workload from the master. This enhances read scalability and performance.

Replication Lag

There might be a slight delay (lag) between the master and slaves since replication is asynchronous.

This means that the slaves might not always have the most up-to-date data immediately after a write operation.

Process

Initial Configuration and Connection

A Redis instance is configured as a slave using the replicaof (or slaveof in older versions) directive to specify the master's IP address and port.

When a slave connects to the master for the first time or reconnects after a disconnection, a synchronization process occurs.

Synchronization

Full Synchronization

The master use BGSAVE command to creates a snapshot of its current data and sends it to the slave. This snapshot is called an RDB (Redis Database) file.

The slave receives and loads this RDB file into memory, replacing its existing dataset.

Incremental Synchronization

After the initial synchronization, the master sends an ongoing stream of write commands to the slave to ensure it stays up-to-date with any changes.

The slave server executes these operations in sequence to maintain data consistency with the master server.

These commands are the same as those received from clients, ensuring that the slave mirrors the master's state.

Ping-Pong Mechanism and Acknowledgment Message

While the master is replicating data and receiving acknowledgments, it also periodically checks the health of the connection using the Ping-Pong mechanism.

This keeps the connection alive and confirms that the slave is still connected and operational.

Master to Slave

The master periodically sends PING commands to all connected slaves.

This is controlled by the repl-ping-slave-period parameter (default 10 seconds).

```
repl-ping-slave-period 5  # Example setting for a 5-second interval
```

Slave to Master

Upon receiving a PING, the slave responds with a PONG.

Acknowledgment

The slave sends regular acknowledgments back to the master. This includes information about the replication offset, indicating the last processed command.

Offset List

The master maintains a list of offsets for each slave, ensuring all slaves are synchronized and monitoring replication lag.

Reconnection

If the slave loses connection to the master (e.g., due to network issues), it will try to reconnect.

Upon reconnection, the slave may request a partial resynchronization if it has a replication offset that matches the master's backlog.

This allows the slave to catch up without requiring a full resynchronization.

```
repl-backlog-size 1mb  # Configurable size of the replication backlog buffer
```

Backlog

The replication backlog is a circular buffer that stores the most recent write commands executed by the master.

When a slave reconnects, it can request the missing commands from the master if the required commands are still in the backlog.

This avoids the overhead of a full dataset resynchronization.

The replication backlog allows slaves to catch up with the master without requiring a full resynchronization.

This is particularly useful when a slave temporarily loses connection to the master and later reconnects.

Failover and Promotion

In a scenario where the master fails, Redis Sentinel (if configured) can promote a slave to become the new master.

Sentinel constantly monitors the master and slaves, performing automatic failover if needed.

What Happens Without Replication

No Data Propagation

Without replication, data changes will only be applied to the master nodes.

There will be no slave nodes to receive and propagate these changes.

Single Point of Failure

Without slaves, each master node becomes a single point of failure.

If a master node fails, there is no slave to promote to master, which can lead to data unavailability or loss.

Redis Sentinel relies on replicas to perform automatic failover.

Without replicas, automatic failover isn't possible, and manual intervention is required to restore service.

Limited Read Scalability

Without slave nodes to handle read requests, all read and write operations must be performed on the master nodes, limiting read scalability and potentially impacting performance.

Manual Failover without Sentinel

Identify new Slave and Disconnect it.

Determine which slave will be promoted to master.

This decision can be based on factors like the most recent synchronization with the master or geographic proximity to the majority of your application's users.

Ensure that the chosen slave is disconnected from the master.

This prevents it from trying to replicate new changes from the master during the promotion process.

```
redis-cli -h 192.168.1.101 -p 6379 SLAVEOF NO ONE
```

Reconfiguration

Configure the chosen slave to become a master. This involves removing any slave configurations.

```
redis-cli -h 192.168.1.101 -p 6379 CONFIG SET slave-read-only no
```

Reconfigure other slaves, Point other slaves to the new master to ensure they start replicating from the newly promoted master.

```
redis-cli -h 192.168.1.102 -p 6379 SLAVEOF 192.168.1.101 6379
redis-cli -h 192.168.1.103 -p 6379 SLAVEOF 192.168.1.101 6379
```

Ensure that all clients and applications are updated to connect to the new master instance.

This might involve updating configuration files, environment variables, or connection strings in your application code.

Monitoring and Testing

Monitor the new master and the reconfigured slaves to ensure they are functioning correctly.

Perform some read and write operations to verify the setup.

Use Redis monitoring tools and commands like INFO to check the replication status.

Common problems during the replication process

Resource

Resource Exhaustion

- **Problem**

  Master or slaves run out of CPU, memory, or disk space.

- **Cause**

  High workload, insufficient hardware resources, or memory leaks.

- **Solution**

  Monitor resource usage, optimize the Redis configuration, and scale the hardware resources as needed.

Full Resynchronization

- **Problem**

  Slaves require full dataset synchronization from the master, which is resource-intensive.

- **Cause**

  Slaves disconnected for too long, backlog buffer size too small.

- **Solution**
  Increase the replication backlog size, ensure slaves reconnect promptly, and monitor disconnection times.

Disk I/O Bottlenecks

- **Problem**
  Slow disk performance affecting the persistence of data and replication logs.
- **Cause**
  High disk write load, slow disk hardware, or suboptimal disk configuration.
- **Solution**
  Use faster disks (SSD), optimize disk I/O settings, and balance write load.

Network

Replication Lag

- Problem
  Slaves are behind the master in terms of the latest data.
- Cause
  Network latency, high write traffic on the master, or resource constraints on the slaves.
- Solution
  Monitor the replication lag and optimize network conditions, reduce write load, or upgrade hardware resources.

Inconsistent Data

- **Problem**
  Data discrepancies between the master and slaves.
- **Cause**
  Replication delays, write load, or bugs.
- **Solution**
  Regularly verify data consistency and address any identified issues promptly.

Connection Timeouts

- **Problem**
  Master or slaves experiencing frequent connection timeouts.
- **Cause**
  Network instability, high latency, or overloaded Redis instances.
- **Solution**
  Improve network stability, reduce latency, and ensure Redis instances are not overloaded.

Network Partitions

- **Problem**
  Temporary or prolonged loss of network connectivity between the master and slaves.
- **Cause**
  Network outages, misconfigurations, or infrastructure issues.
- **Solution**
  Ensure reliable network connectivity, configure appropriate timeouts, and use robust network infrastructure.

Excessive Bandwidth Usage

- **Problem**
  High bandwidth consumption due to replication traffic.
- **Cause**
  High volume of write operations, full resynchronizations.
- **Solution**
  Optimize write operations, ensure partial resynchronization is effective, and monitor bandwidth usage.

Configuration

Failover Issues

- **Problem**
  Failover process not working correctly, leading to extended downtime.
- **Cause**
  Misconfigured Redis Sentinel, network issues, or bugs in the failover process.
- **Solution**
  Ensure proper Sentinel configuration, improve network reliability, and test failover mechanisms.

Configuration Errors

- **Problem**
  Incorrect or suboptimal replication configuration.
- **Cause**
  Misconfiguration of parameters like repl-backlog-size, repl-ping-slave-period, etc.
- **Solution**
  Carefully review and tune the replication configuration parameters according to best practices and the specific workload.

Replicationi Configuration

Start Master Server

1. Edit the redis.conf file for the master Redis server:
```
# redis.conf (Master)
port 6379
bind 127.0.0.1
```
2. Start the Redis server with the master configuration:
```
redis-server /path/to/master/redis.conf
```

Start Slave Servers

1. Edit the redis.conf file for the slave Redis server:
```
# redis.conf (Slave)
port 6380
bind 127.0.0.1
replicaof 127.0.0.1 6379
```
2. Start the Redis server with the slave configuration:
```
redis-server /path/to/slave/redis.conf
```

Verification

1. Connect to the master Redis server and set a key:
```
redis-cli -p 6379
set key1 "value1"
```
2. Connect to the slave Redis server and get the key:
```
redis-cli -p 6380
get key1
```

Sentinel

Reference:

https://medium.com/@larrie.loi/redis-sentinel-for-high-availability-f6b24f963bb7

Purpose

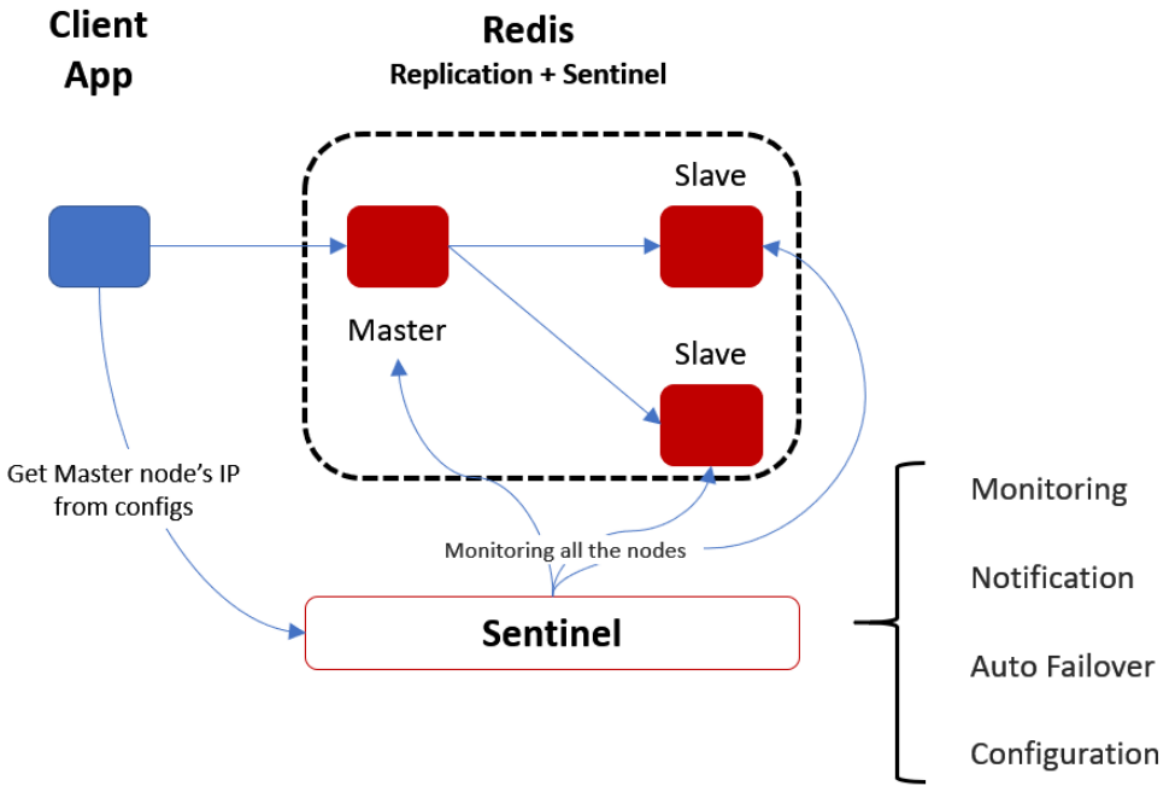
Provides high availability and monitoring.

Handles automatic failover of Redis instances.

Monitors master and slave instances, promotes a slave to master if the master fails.

When setting up a Redis Sentinel architecture, the Redis master and replica instances use `redis.conf` for their configuration,

while the Sentinel instances use `sentinel.conf`.



Election Process

Monitoring

Sentinel constantly checks if your master and replica instances are working as expected.

Sentinel instances monitor Redis instances by sending periodic PING commands.

If a Redis instance does not respond within a predefined timeout, it is considered as failing or unreachable.

Detection of Failures

Sentinels use a consensus mechanism to detect a master failure.

Sentinel can notify the system administrator, or other computer programs, via an API, that something is wrong with one of the monitored Redis instances.

If a master is not working as expected, Sentinel can start a failover process where a replica is promoted to master,

If a majority of Sentinels agree that the master is down (based on the failure detection and down-after-milliseconds settings), the failover process of promoting a replica to master begins.

Failover Process

Select a New Master:

Sentinels choose one of the available replicas to be promoted to master.

Replica Selection:

The other additional replicas are reconfigured to use the new master, and the applications using the Redis server are informed about the new address to use when connecting.

Sentinel selects the best replica based on criteria such as replication lag and priority.

```
replica-priority 50
```

Promotion:

The selected replica is promoted to master.

Update Configuration:

Sentinels update the list of available Redis instances, so clients can connect to the new master.

Sentinels also ensure that the new master has replicas by reconfiguring existing replicas to follow the new master.

Sentinel acts as a source of authority for clients service discovery:

Clients connect to Sentinels in order to ask for the address of the current Redis master responsible for a given

service. If a failover occurs, Sentinels will report the new address.
Sentinels update their configuration to reflect the new master and notify Redis clients.

Sentinel Configuration

Start Sentinel Server

1. Create a sentinel.conf file:
```
port 26379
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 10000
sentinel parallel-syncs mymaster 1
```
2. Start the Redis Sentinel:
```
redis-sentinel /path/to/sentinel.conf
```

Verification

1. Check the Sentinel status:
```
redis-cli -p 26379 sentinel masters
```
2. Simulate a master failure by stopping the master Redis server and observe Sentinel promoting a slave to master.

linux Installation

wget https://download.redis.io/releases/redis-6.2.6.tar.gz（官网查看版本 https://redis.io/download）
mv redis-5.0.0 /usr/local/ 直接移动
yum install gcc-c++
make 根目录
make install src 目录
mkdir etc bin 创建额外目录
cd src && mv mkreleasehdr.sh redis-benchmark redis-check-aof redis-cli redis-server ../bin 移动到 bin 目录
mv redis.conf ./etc
vi redis.conf 修改配置项
vi /etc/profile 修改环境变量

redis-server /usr/local/redis-6.0.0/redis.conf 服务器后台启动【--port 6379 指定端口】
redis-cli 客户端启动
redis-check-dump RDB 文件检查工具（快照持久化文件）
redis-check-aof AOF 文件修复工具
pkill redis 停止 redis

Command

Terminal

Global

redis-server

Starts the Redis server with the default configuration or with a specified configuration file.
```
redis-server /etc/redis/redis.conf
redis-cli shutdown   # Shuts down the Redis server gracefully.
```
Parameters:

/path/to/redis.conf: Path to the Redis configuration file (optional).

redis-cli <command>

Opens the Redis command-line interface for interactive mode.
```
redis-cli -h localhost -p 6379 -a mypassword
```
Parameters:

-h <hostname>: The server hostname (default: 127.0.0.1).

-p <port>: The server port (default: 6379).

-a <password>: The password for the Redis server.

-n <database>: The database number (default: 0).

redis-cli config get <parameter>

Gets the value of a configuration parameter.

```
redis-cli config get maxmemory
```

Parameters:

<parameter>: The configuration parameter to retrieve.

redis-cli config set <parameter> <value>

Sets the value of a configuration parameter.

```
redis-cli config set maxmemory 256mb
```

Parameters:

<parameter>: The configuration parameter to set.

<value>: The value to set for the parameter.

Key and Value

redis-cli keys <pattern>

Finds all keys matching the specified pattern.

redis-cli keys "*"

Parameters:

<pattern>: The pattern to match.

redis-cli del <key> [key ...]

Deletes one or more keys.

```
redis-cli del mykey
redis-cli del key1 key2 key3
```

Parameters:

<key>: The key(s) to delete.

redis-cli publish <channel> <message>

Posts a message to the given channel.

```
redis-cli publish mychannel "Hello, World!"
```

Parameters:

<channel>: The channel to publish to.

<message>: The message to publish.

redis-cli subscribe <channel> [channel ...]

Subscribes to the given channels.

redis-cli subscribe mychannel

Parameters:

<channel>: The channel(s) to subscribe to.

redis-cli eval <script> <numkeys> <key> [key ...] <arg> [arg ...]

Evaluates a Lua script.

```
redis-cli eval "return redis.call('set', KEYS[1], ARGV[1])" 1 mykey "Hello"
```

Parameters:

<script>: The Lua script to evaluate.

<numkeys>: Number of keys that the script will use.

<key>: The key(s) that the script will access.

<arg>: Additional arguments for the script.

Monitor

redis-cli info

Provides information and statistics about the Redis server.

```
redis-cli info
```

```
redis-cli INFO server
redis-cli INFO clients
redis-cli INFO memory
redis-cli INFO persistence
redis-cli INFO stats
redis-cli INFO replication
redis-cli INFO cpu
redis-cli INFO keyspace
```

Parameters:

[section]: Optional section to query (e.g., memory, cpu).

Metrics:

1) Server Metrics
   - redis_version: The version of the Redis server.
   - uptime_in_seconds: The total number of seconds the server has been running.
   - uptime_in_days: The total number of days the server has been running.
2) Clients Metrics
   - connected_clients: The number of client connections (excluding connections from replicas).
   - blocked_clients: The number of clients waiting for blocking commands (BLPOP, BRPOP, BRPOPLPUSH).
3) Memory Metrics
   - used_memory: The total number of bytes allocated by Redis using its allocator.
   - used_memory_rss: The number of bytes that Redis allocated as seen by the operating system (a.k.a. resident set size).
   - used_memory_peak: The peak memory consumed by Redis (in bytes).
   - used_memory_lua: The memory used by the Lua engine (in bytes).
   - mem_fragmentation_ratio: The ratio between used_memory_rss and used_memory.
4) Persistence Metrics
   - rdb_changes_since_last_save: The number of changes since the last RDB save.
   - rdb_bgsave_in_progress: Flag indicating an RDB save is in progress.
   - aof_enabled: Flag indicating AOF is enabled.
   - aof_rewrite_in_progress: Flag indicating an AOF rewrite operation is in progress.
   - aof_last_rewrite_time_sec: Duration of the last AOF rewrite operation in seconds.
   - aof_current_size: AOF current file size.
5) Stats Metrics
   - total_connections_received: The total number of connections accepted by the server.
   - total_commands_processed: The total number of commands processed by the server.
   - instantaneous_ops_per_sec: The number of commands processed per second.
   - total_net_input_bytes: The total number of bytes read from the network.
   - total_net_output_bytes: The total number of bytes written to the network.
   - rejected_connections: The number of connections rejected because of maxclients limit.
   - expired_keys: The total number of key expiration events.
   - evicted_keys: The number of evicted keys due to maxmemory limit.
   - keyspace_hits: The number of successful lookup of keys in the main dictionary.
   - keyspace_misses: The number of failed lookup of keys in the main dictionary.
6) Replication Metrics
   - role: The role of the Redis instance (master or slave).
   - connected_slaves: The number of connected replicas.
   - master_repl_offset: The offset of replication for the master.
   - repl_backlog_active: Flag indicating if the replication backlog is active.
7) CPU Metrics
   - used_cpu_sys: System CPU consumed by the Redis server.

- used_cpu_user: User CPU consumed by the Redis server.
- used_cpu_sys_children: System CPU consumed by the background processes.
- used_cpu_user_children: User CPU consumed by the background processes.

8) Keyspace Metrics
   - db0: The keyspace information for database 0, includes keys, expires, avg_ttl.
     Example: db0:keys=1,expires=0,avg_ttl=0

redis-cli monitor

Streams every command processed by the Redis server in real-time.

```
redis-cli monitor
```

redis-cli ping

Checks if the Redis server is running and responsive.

```
redis-cli ping
redis-cli ping "Hello, Redis!"
```

Parameters:

[message]: Optional message to send with the ping.

redis-cli dbsize

Returns the number of keys in the currently selected database.

```
redis-cli dbsize
```

redis-benchmark [options]

Measures the performance of a Redis server.

It runs a set of benchmarks to test various operations and reports the number of requests per second the server can handle.

```
redis-benchmark -h 127.0.0.1 -p 6379 -n 100000 -c 50
redis-benchmark -t set,get -n 100000 -c 50
```

Parameters:

-h <hostname>: Server hostname (default: 127.0.0.1).

-p <port>: Server port (default: 6379).

-s <socket>: Server socket (overrides hostname and port).

-a <password>: Password for Redis server.

-n <requests>: Total number of requests (default: 100000).

-c <clients>: Number of parallel connections (default: 50).

-d <size>: Data size of SET/GET value in bytes (default: 3).

-t <tests>: Only run the comma-separated list of tests (e.g., set,get).

-r <keyspacelen>: Use random keys for SET/GET/INCR, random values for SADD.

-P <pipelines>: Number of requests to pipeline (default: 1).

redis-cli slowlog [subcommand] [argument]

Logs and analyzes slow commands.

This command manages and queries the slow log, which records queries that exceed a specified execution time threshold.

By default, the slow log captures commands that take longer than 10 milliseconds to execute.

This command helps in identifying and optimizing long-running queries to improve performance.

redis-cli slowlog get [n]	Retrieve the last n slow log entries.
redis-cli slowlog len	Get the current length of the slow log.
redis-cli slowlog reset	Clear the slow log.

Persistence

redis-cli bgsave

Performs a background save of the dataset to disk.

```
redis-cli bgsave
```

redis-cli bgrewriteaof

Trigger an asynchronous rewrite of the Append-Only File (AOF) in the background.

```
redis-cli bgrewriteaof
```

redis-cli restore <key> <ttl> <serialized-value>

Restores a serialized value back into a key.

```
redis-cli restore mykey 0 "\x00\x00\x00\x00..."
```

Parameters:

<key>: The key to restore.

<ttl>: Time to live in milliseconds (0 if the key should not expire).

<serialized-value>: The serialized value to restore.

redis-cli save

Performs a synchronous save of the dataset to disk.

```
redis-cli save
```

redis-cli --rdb <filename>

Generates an RDB file by performing a synchronous save and writes it to the specified file.

```
redis-cli --rdb /tmp/dump.rdb
```

Parameters:

<filename>: The file to save the RDB snapshot.

redis-check-rdb <rdb-file>

Checks the integrity of a Redis RDB (Redis Database) file. It is useful for diagnosing and repairing corrupt RDB files.

```
redis-check-rdb /var/lib/redis/dump.rdb
```

Parameters:

<rdb-file>: The path to the RDB file that you want to check.

Sentinel

redis-cli sentinel monitor <master-name> <ip> <port> <quorum>

Adds a new master to be monitored by Sentinel.

```
redis-cli sentinel monitor mymaster 127.0.0.1 6379 2
```

Parameters:

<master-name>: The name of the master.

<ip>: The IP address of the master.

<port>: The port number of the master.

<quorum>: The quorum size for this master.

redis-cli sentinel failover <master-name>

Forces a failover of the specified master.

```
redis-cli sentinel failover mymaster
```

Parameters:

<master-name>: The name of the master.

Data Structures

Reference:

https://redis.io/glossary/redis-data-structures/

Redis Keys

Binary Safe

Redis keys can contain any binary data, meaning you can use strings with any characters, including null bytes (\x00).

Maximum Length

The maximum length of a Redis key is 512 MB.

Naming Conventions

Redis keys are typically short but descriptive, often following a specific naming pattern to avoid conflicts and make management easier.

Common naming conventions include using colons (:) as separators to create namespaces, e.g., user:1000:name.

Do not include special characters such as underscores, spaces, newlines, single and double quotes, and other escape characters;

Structure

Project Name or Abbreviation

Identifies the project or application.

```
ecommerce:order:orderid:1001
```

Table Name or Key Prefix

Represents the data entity or object type.

```
ecommerce:order:orderid:1001
```

Field for Key Differentiation

Distinguishes between different types of data or keys within the table.

```
ecommerce:order:orderid:1001
```

Key Value

Provides the actual unique identifier or data.

```
ecommerce:order:orderid:1001
```

Avoid Large Keys and Values

Key Size:

Keep keys relatively short to avoid excessive memory usage.

Redis keys are stored in a hash table, and very large keys can lead to inefficiencies.

Keys with large values (e.g., big lists, sets, or hashes) can become problematic if accessed frequently, as operations on large keys consume more resources (CPU, memory, I/O).

Even small keys can become hot if they are accessed with very high frequency.

Value Size:

Similarly, keep values manageable in size. For very large values, consider breaking them into smaller parts or using alternative storage mechanisms.

How to Identify Hot Keys:

Redis Monitoring:

Use Redis monitoring tools such as redis-cli commands (e.g., MONITOR, INFO, SLOWLOG) or monitoring solutions (e.g., RedisInsight, Prometheus, Grafana) to observe access patterns and identify keys with high request rates.

Application Logs:

Analyze application logs to see which keys are being accessed frequently.

Redis Keyspace Notifications:

Enable keyspace notifications to track operations on keys and identify patterns of frequent access.

Case Sensitivity

Redis keys are case-sensitive, so mykey and MyKey would be considered different keys.

Redis Values

Binary-safe strings

Redis strings can be of any size, but by default are limited to 512 MB.

The first byte of a Redis string is a dollar sign ($), followed by one or more decimal digits (0–9) that represent the string's length in bytes.

Redis can detect if a string value is a base-10 integer or floating-point value, and allows the user to manipulate it with INCR* and DECR* operations.

SET key value [EX seconds] [PX milliseconds] [NX|XX]

Sets the value of a key.

```
SET mykey "Hello, Redis!"
```

Parameters:

key: The name of the key.

value: The value to be set.

EX seconds: Set the specified expire time, in seconds.
PX milliseconds: Set the specified expire time, in milliseconds.
NX: Set the key only if it does not already exist.
XX: Set the key only if it already exists.

GET key

Gets the value of a key.

```
GET mykey
```

Parameters:

key: The name of the key.

INCR key

Increments the integer value of a key by one (if it contains an integer).

```
INCR counter
```

Parameters:

key: The name of the key.

TTL key

To manually check for expired keys and delete them, you would generally use the TTL command to check the time-to-live of a key.

If it returns a value of -2, the key has already expired.

DEL key

Delete a specific key manually

SETNX key value

SET if not exists

EXPIRE key seconds

Use the EXPIRE command to set a TTL (time-to-live) for a key.

```
EXPIRE mykey 60
```

Lists

Collections of string elements arranged in the order of insertion. They are essentially linked lists.

They are similar to linked lists in other programming languages and are optimized for adding and removing elements at the beginning or end of the list.

Redis lists are not limited to a fixed size and can expand or shrink as needed.

Adding or removing elements from the beginning or end of a list is efficient and has a constant time complexity of O(1).

However, accessing elements in the middle of a list can be slow.

Note: If there are no elements in the container, immediately delete the container to free up memory.

LPUSH key value [value ...]

Prepends one or multiple values to a list.

```
LPUSH mylist "world"
LPUSH mylist "hello"
```

Parameters:

key: The name of the list key.

value: The value(s) to be prepended to the list.

LRANGE key start stop

Gets a range of elements from a list.

```
LRANGE mylist 0 -1
```

Parameters:

key: The name of the list key.

start: The starting index of the range.

stop: The ending index of the range.

Hashes

Maps made up of fields that correspond to values. The field and the value are both strings.
The underlying storage uses ziplist (compressed list) and hashtable (dictionary). Each hash can store $2^{32}-1$ key-value pairs.
When the number of hash type elements is less than the hash-max-ziplist-entries configuration (default 512),
and all values are less than the hash-max-ziplist-value configuration (default 64 bytes), ziplist is used as the internal implementation of hashing , other times use hashtable.

ziplist uses a more compact structure to achieve continuous storage of multiple elements, so it is better than hashtable in terms of saving memory.
However, ziplist's reading and writing efficiency will decrease when there is a lot of data, while hashtable's reading and writing time complexity is O(1)

Set a hash field:
Get a hash field:

HSET key field value

Sets the value of a hash field.

```
HSET user:1000 name "John Doe"
HSET user:1000 age "30"
```

HGET key field

Gets the value of a hash field.

```
HGET user:1000 name
```

HGETALL key

Get all fields and values:

```
HGETALL user:1000
```

HINCRBY key field increment

Increment the integer value of a field in a hash by a specified increment.

Sets

Collections of distinct, unsorted string elements.
It is exactly the same as the hash storage structure. It only stores keys, not values (nil), and values are not allowed to be repeated.
If the added data already exists in the set, only one copy will be kept.
Although set has the same storage structure as hash, it cannot enable the space to store values in hash.

SADD key member [member ...]

Adds one or more members to a set.

```
SADD myset "hello"
SADD myset "world"
```

Parameters:

key: The name of the set key.

member: The member(s) to be added to the set.

SMEMBERS key

Get all members of a set:

Parameters:

key: The name of the set key.

Sorted Sets(ZSet)

Sorted sets are similar to sets, but each string element is associated with a floating number value known as a score.
Because the elements are always retrieved in order of their score, unlike Sets, it is possible to retrieve a range of elements (for

example you may ask: give me the top 10, or the bottom 10).

The primary purpose of the score is to sort the elements in the zset.
Members are ordered based on their scores in ascending order by default.
This allows for efficient retrieval of elements based on their rank or range.

The underlying storage uses ziplist (compressed list), skiplist (skip list), hashtable (dictionary)
When the number of deduplication set elements is less than the zset-max-ziplist-entries configuration (default 512),
and all values are less than the zset-max-ziplist-value configuration (default 64 bytes), ziplist is used as the internal hash Implementation, other times using skiplist and hashtable.

ZSet is suitable for message subscription and publishing modes because it can delete messages, which is difficult to achieve in other message middleware.
We can use internal timers to check data and send messages.

ZADD key [NX|XX] [CH] [INCR] score member [score member ...]

Adds one or more members to a sorted set, or updates its score if it already exists.

```
ZADD myzset 1 "one"
ZADD myzset 2 "two"
```

Parameters:

key: The name of the sorted set key.
NX: Only add new elements.
XX: Only update existing elements.
CH: Modify the return value from the number of new elements added, to the total number of elements changed.
INCR: Increment the score of the member.
score: The score of the member.
member: The member to be added to the sorted set.

ZRANGE key start stop [WITHSCORES]

Returns a range of members in a sorted set, by index.

```
ZRANGE myzset 0 -1 WITHSCORES
```

Parameters:

key: The name of the sorted set key.
start: The starting index of the range.
stop: The ending index of the range.
WITHSCORES: Return the scores of the elements.

Bitmaps

It is possible to treat String values like an array of bits by using special commands: you can set and clear individual bits, count all the bits set to 1, find the first set or unset bit, and so on.
Redis Bitmap is essentially a string data structure underneath.
Bitmaps in Redis are manipulated using string commands, but they are treated as a sequence of bits rather than characters.

Key Points about Redis Bitmaps

String Data Structure:

Internally, Redis stores bitmaps as binary-safe strings. Each bit within the string can be individually set or cleared.

Bitwise Operations:

Redis provides a set of bitwise operations to manipulate these bits, such as setting, getting, and performing bitwise AND, OR, XOR, and NOT operations.

Efficient Storage:

Since bitmaps are stored as strings, they can be very memory-efficient for representing binary states, especially when you need to track a large number of binary flags (on/off states).

Command Interface:

Redis uses commands like SETBIT, GETBIT, BITCOUNT, BITOP, and BITPOS to interact with bitmaps.

SETBIT key offset value

Sets or clears the bit at offset in the string value stored at key.

```
SETBIT mykey 7 1
```

GETBIT key offset

Returns the bit value at offset in the string value stored at key.

```
GETBIT mykey 7
```

BITCOUNT key [start end]

Counts the number of bits set to 1 in a bitmap.

```
BITCOUNT mykey  # Counts all bits set to 1 in 'mykey'
```

BITOP operation destkey key [key ...]

Performs bitwise operations between multiple bitmaps.

```
BITOP AND resultkey key1 key2  # Performs bitwise AND between 'key1' and 'key2', stores result in 'resultkey'
```

BITPOS key bit [start] [end]

Finds the first bit set to the specified value in a bitmap.

```
BITPOS mykey 1  # Finds the first bit set to 1 in 'mykey'
```

HyperLogLogs

HyperLogLogs are probabilistic data structures used for counting unique items with a low memory footprint.

Redis HyperLogLog is a probabilistic data structure used to estimate the cardinality (the number of unique elements) of a set.

It is highly memory efficient and can provide an approximate count with a standard error of only 0.81%.

Key Features:

Memory Efficiency:

HyperLogLog only uses around 12 KB of memory to count unique items with high precision, regardless of the number of items.

Probabilistic Counting:

It provides an approximate count of unique elements, which is useful for large datasets where an exact count is not necessary.

Cardinality Estimation

Ideal for use cases where you need to count unique users, IP addresses, or events over time.

PFADD key element [element ...]

Adds the specified elements to the specified HyperLogLog.

```
PFADD myhll "foo" "bar" "baz"      # Adds foo, bar, and baz to the HyperLogLog myhll.
```

PFCOUNT key [key ...]

Returns the approximated cardinality of the set(s) observed by the HyperLogLog at key(s).

```
PFCOUNT myhll
```

Geospatial Indexes

Geospatial Indexes allow you to store and query geographic locations.

Redis implements geospatial indexes using a combination of sorted sets (zsets) and the Geohash algorithm.

This allows Redis to efficiently store, query, and manage geospatial data, such as latitude and longitude coordinates, and perform operations like finding distances and searching for nearby locations.

Geohash Algorithm:

Geohash is a hierarchical spatial data structure that subdivides the world map into a grid of bounding boxes and encodes geographic coordinates (latitude and longitude) into a single string.

The Geohash algorithm allows for efficient indexing and querying of spatial data. Nearby locations have similar geohash prefixes.

Sorted Sets (zsets):

Redis sorted sets are used to store geohash values as scores and location names as members. The sorted set allows for efficient range queries based on scores, which is essential for geospatial queries.

Performance optimization:

Split geospatial index data by city to split internal skip lists, which can reduce update overhead in a single skip list.

GEOADD key longitude latitude member [longitude latitude member ...]

Adds the specified geospatial items (latitude, longitude, name) to the specified key.

```
GEOADD locations 13.361389 38.115556 "Palermo"
GEOADD locations 15.087269 37.502669 "Catania"
```

GEODIST key member1 member2 [unit]

Returns the distance between two members in the geospatial index represented by the key.

The unit of distance (m, km, mi, ft). Default is meters.

```
GEODIST locations "Palermo" "Catania"
```

GEORADIUS key longitude latitude radius unit [WITHDIST] [WITHCOORD] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]

Queries a geospatial index to fetch members within a specified radius around a given point.

GEORADIUS locations 15 37 200 km

Parameters:

key: The name of the geospatial key.
longitude: The longitude of the center point.
latitude: The latitude of the center point.
radius: The radius of the circle centered at the given point.
unit: The unit of measurement for the radius (m, km, mi, ft).
WITHDIST: Optionally returns the distance of the returned items from the specified center.
WITHCOORD: Optionally returns the longitude and latitude of the matching items.
WITHHASH: Optionally returns the raw geohash-encoded sorted set score of the matching items.
COUNT count: Limits the number of the returned items to count.
ASC|DESC: Sort the returned items in ascending or descending order of distance from the center.
STORE key: Stores the result in the specified key as a sorted set.
STOREDIST key: Stores the result with distances in the specified key as a sorted set.

GEORADIUSBYMEMBER key member radius unit [WITHDIST] [WITHCOORD] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]

Parameters:

key: The name of the geospatial key.
member: The name of the center member.
radius: The radius of the circle centered at the given member.
unit: The unit of measurement for the radius (m, km, mi, ft).
WITHDIST: Optionally returns the distance of the returned items from the specified center.
WITHCOORD: Optionally returns the longitude and latitude of the matching items.
WITHHASH: Optionally returns the raw geohash-encoded sorted set score of the matching items.

COUNT count: Limits the number of the returned items to count.
ASC|DESC: Sort the returned items in ascending or descending order of distance from the center.
STORE key: Stores the result in the specified key as a sorted set.
STOREDIST key: Stores the result with distances in the specified key as a sorted set.

Streams

Streams are a log data structure, similar to a message queue or log file.
Redis Streams is a powerful data structure in Redis designed for handling log or message data.
It allows for efficient and ordered storage and retrieval of messages or events, with features such as automatic ID generation, message trimming, and consumer groups for message processing.

XADD key [MAXLEN [~|=] count] [NOMKSTREAM] [MINID minid] [LIMIT limit] ID field value [field value ...]

Appends a new entry to a stream.

```
XADD mystream * name "Alice" age 30
```

Parameters:

key: The name of the stream key.
MAXLEN [~|=] count: Trim the stream to the specified length.
NOMKSTREAM: Do not create a stream if it doesn't exist.
MINID minid: Trim entries with IDs less than minid.
LIMIT limit: Limit the number of trimmed entries.
ID: The ID of the entry.
field value: Field-value pairs representing the entry.

XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]

Read the stream entries.

```
XREAD COUNT 2 STREAMS mystream 0
```

Parameters:

COUNT count: Specifies the maximum number of entries to return per stream. This limits the amount of data returned in a single read operation.
BLOCK milliseconds: Blocks the connection for the specified amount of milliseconds if no data is available. If no data is available after the timeout, the command returns an empty reply.
STREAMS: Specifies that the following arguments are the stream keys and the IDs to start reading from.
key [key ...]: The keys of the streams to read from.
ID [ID ...]: The IDs to start reading from for each stream. Use $ to start reading from the latest entry, or 0 to read all entries from the beginning.

XRANGE key start end [COUNT count]

Returns the stream entries matching a range of IDs.

```
XRANGE mystream - +
```

Parameters:

key: The name of the stream key.
start: The start ID.
end: The end ID.
COUNT count: Limit the number of returned entries.

XTRIM stream-name MAXLEN count [MINID minid] [KEEP history]

Trims the stream to a specified length. Messages older than the specified length are removed.

```
XTRIM mystream MAXLEN 1000
```

Parameters:

stream-name: The name of the stream.

MAXLEN count: Maximum length of the stream.

MINID minid: Optional. Minimum message ID to keep.

KEEP history: Optional. Keep a specified number of messages, preserving a history of the last n messages.

DAM / RabbitMQ

Core

RabbitMQ is an open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP). It is designed to facilitate the exchange of messages between different components of a system, ensuring that messages are delivered efficiently, securely, and reliably.

Components

Producers

Applications that send messages to RabbitMQ. They publish messages to exchanges.

Consumers

Applications that receive messages from queues and process them.

Broker

The RabbitMQ server itself, responsible for receiving, storing, and forwarding messages.

Exchanges

Components that receive messages from producers and route them to queues based on routing rules.

Types of exchanges include direct, fanout, topic, and headers exchanges.

Bindings

Rules that define the relationship between exchanges and queues, determining how messages are routed.

Queues

Buffers that store messages until they are processed by consumers. Queues can be durable, transient, or auto-deleted.

Virtual Hosts (Vhosts)

Logical partitions within an instance of RabbitMQ.

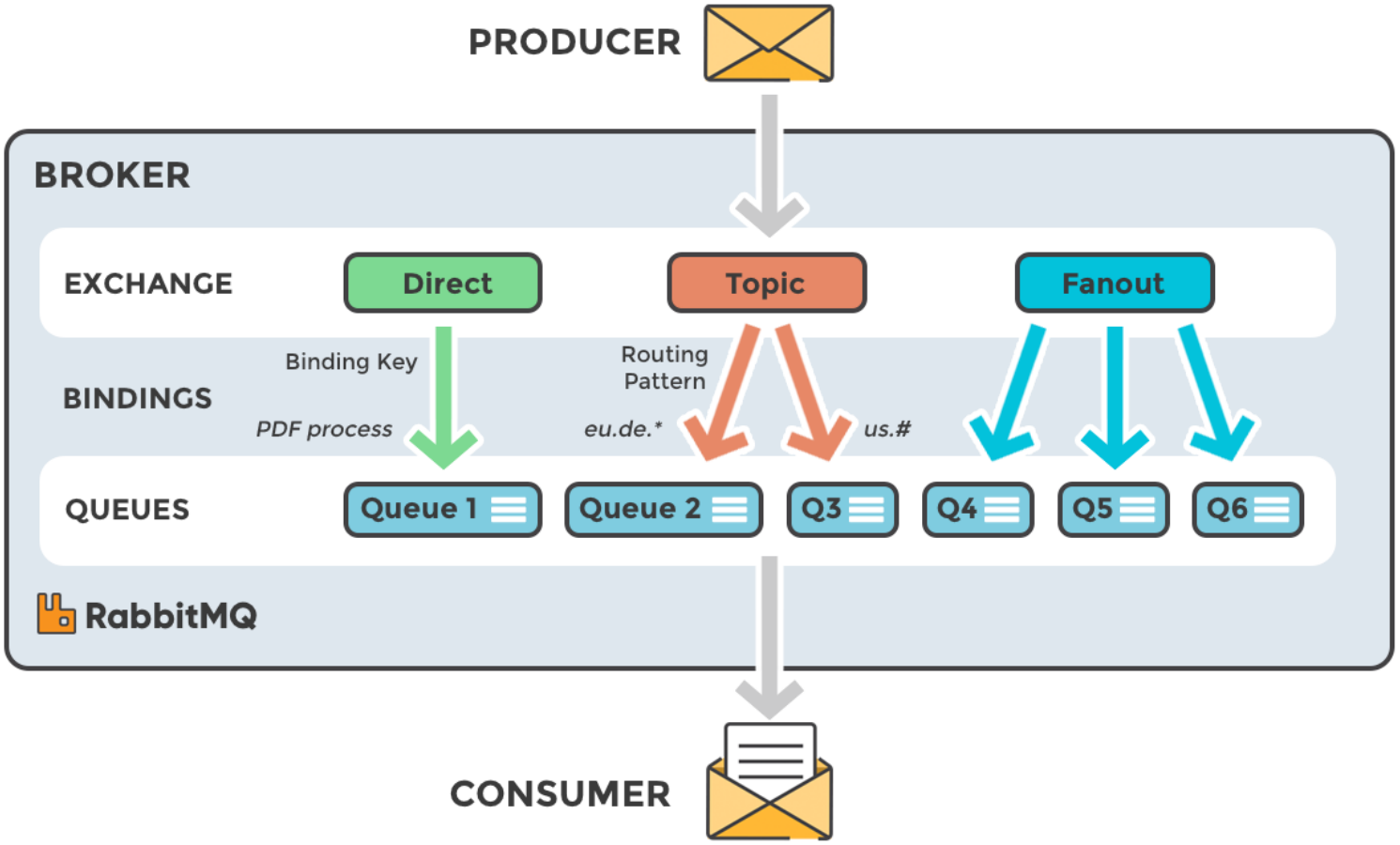
Each vhost can have its own set of exchanges, queues, bindings, users, and permissions.

Vhosts provide a way to segregate multiple applications or tenants within a single RabbitMQ instance.

Channels

Lightweight virtual connections inside a physical TCP connection between the client and the RabbitMQ broker.

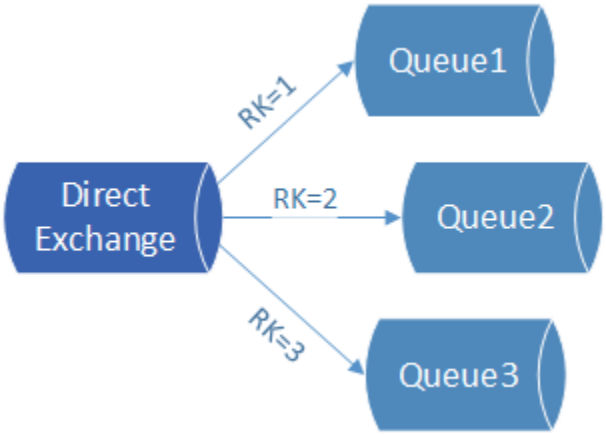
Multiple channels can be opened on a single TCP connection, allowing concurrent operations without the overhead of establishing multiple TCP connections. Channels enable concurrent publishing and consuming of messages, improving throughput and performance.



Exchange Types

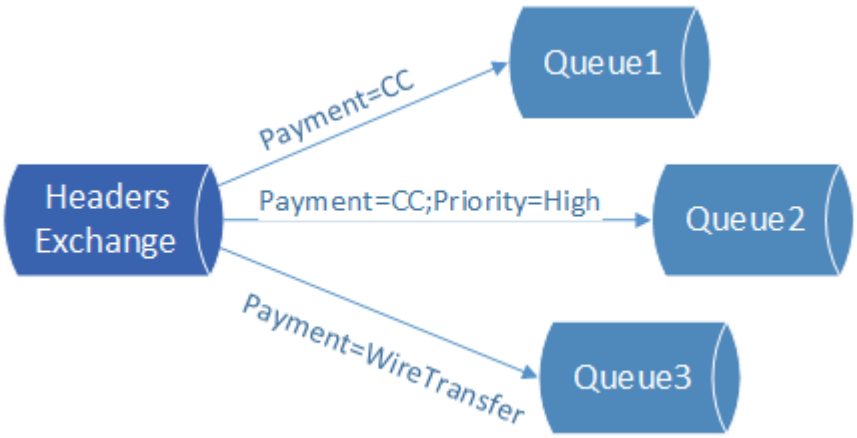
Direct Exchange

Routes messages to queues based on a direct match between the routing key specified by the producer and the routing key specified in the binding by the consumer.



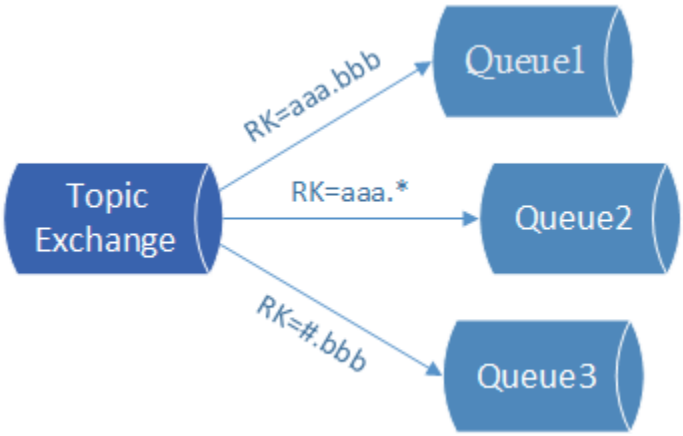
Headers Exchange

Routes messages based on message header attributes instead of routing keys. Allows for complex routing criteria based on headers.



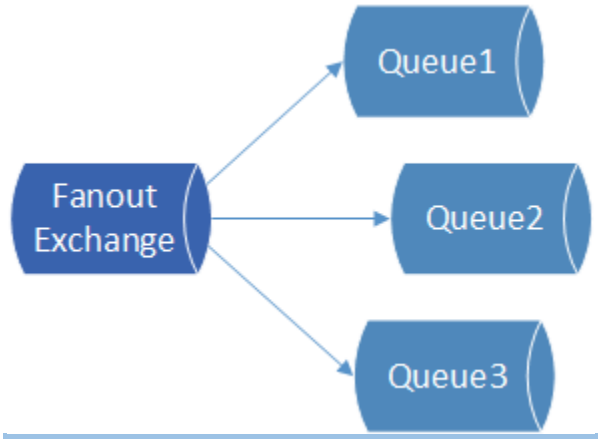
Topic Exchange

Routes messages to queues based on pattern matching of routing keys. Allows for more flexible routing than direct exchanges.



Fanout Exchange

Broadcasts messages to all queues that are bound to it, regardless of the routing key or pattern used by the producer.



Configuration
rabbitmq.conf

General Configuration

default_user <username>

Sets the default user for RabbitMQ.

```
default_user = guest
```

default_pass <password>

Sets the default password for RabbitMQ.

```
default_pass = guest
```

default_vhost <vhost>

Sets the default virtual host.

```
default_vhost = /
```

Networking

listeners.tcp.default <port-number>

Sets the default TCP port for RabbitMQ.

```
listeners.tcp.default = 5672
```

listeners.ssl.default <port-number>

Sets the default SSL port for RabbitMQ.

```
listeners.ssl.default = 5671
```

Node and Cluster Configuration

node.name <node-name>

Sets the name of the RabbitMQ node.

```
node.name = rabbit@hostname
```

node.port <port-number>

Sets the port used for inter-node communication.

```
node.port = 25672
```

cluster_formation.peer_discovery_backend <backend>

Specifies the backend to use for cluster formation.

```
cluster_formation.peer_discovery_backend = rabbit_peer_discovery_classic_config
```

cluster_formation.classic_config.nodes.<index> <node-name>

Lists nodes for classic config cluster formation.

```
cluster_formation.classic_config.nodes.1 = rabbit@node1
cluster_formation.classic_config.nodes.2 = rabbit@node2
```

Security

loopback_users.<username> <true/false>

Defines users restricted to loopback connections.

```
loopback_users.guest = true
```

auth_mechanisms.<index> <mechanism>

Lists authentication mechanisms to use.

```
auth_mechanisms.1 = PLAIN
auth_mechanisms.2 = AMQPLAIN
```

SSL Configuration

ssl_options.cacertfile <path>

Path to the CA certificate file.

```
ssl_options.cacertfile = /path/to/ca_certificate.pem
```

ssl_options.certfile <path>

Path to the server certificate file.

```
ssl_options.certfile = /path/to/server_certificate.pem
```

ssl_options.keyfile <path>

Path to the server key file.

```
ssl_options.keyfile = /path/to/server_key.pem
```

ssl_options.verify <verify_peer/verify_none>

Specifies whether to verify peer certificates.

```
ssl_options.verify = verify_peer
```

ssl_options.fail_if_no_peer_cert <true/false>

Specifies whether to fail if no peer certificate is provided.

```
ssl_options.fail_if_no_peer_cert = true
```

Performance Tuning

vm_memory_high_watermark.relative <fraction>

Sets the memory usage threshold as a fraction of total RAM.

```
vm_memory_high_watermark.relative = 0.4
```

disk_free_limit.absolute <size>

Sets the minimum amount of free disk space required.

```
disk_free_limit.absolute = 1GB
```

Logging

log.dir <directory>

Directory for log files.

```
log.dir = /var/log/rabbitmq
```

log.file <file>

Log file name.

```
log.file = rabbit.log
```

Management Plugin Configuration

management.listener.port <port-number>

Sets the port for the management plugin.

```
management.listener.port = 15672
```

management.listener.ssl <true/false>

Enables or disables SSL for the management plugin.

```
management.listener.ssl = false
```

Plugins

plugins.<plugin-name> <true/false>

Enables or disables specified plugins.

```
plugins.rabbitmq_management = true
plugins.rabbitmq_shovel = true
plugins.rabbitmq_federation = true
```

Queue Parameters

x-message-ttl=<seconds>

Defines the duration in seconds that a message remains in the queue before it is considered expired.

Once the TTL for a message expires, the message is removed from the queue. This TTL countdown starts from the moment the message is published to the queue.

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-message-ttl":3600}'
```

x-max-length=<count>

Defines the maximum number of messages that the queue can hold.

When the maximum number is reached, older messages are either dropped or moved to a dead-letter exchange (if configured).

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-max-length":1000}'
```

x-max-length-bytes=<bytes>

Specifies the maximum total size of messages that the queue can hold.

When the maximum size is reached, messages are either dropped or moved to a dead-letter exchange (if configured).

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-max-length-bytes":10485760}'
```

x-max-priority=<priority>

Sets the maximum priority for messages in the queue.

This parameter is used when messages have priorities assigned. The value specifies the highest priority level.

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-max-priority":10}'
```

x-dead-letter-exchange=<exchange_name>

Specifies an exchange to which messages should be forwarded if they are rejected, expired, or otherwise not processed successfully.

This helps manage message handling and allows for further processing or logging of problematic messages.

RabbitMq doesn't automatically create a dead-letter exchange. Instead, you must explicitly create the dead-letter exchange and define its behavior.

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-dead-letter-exchange":"dlx_exchange"}'
```

x-dead-letter-routing-key=<routing_key>

Defines the routing key to use when messages are sent to the dead-letter exchange. This allows for more specific routing of dead-letter messages.

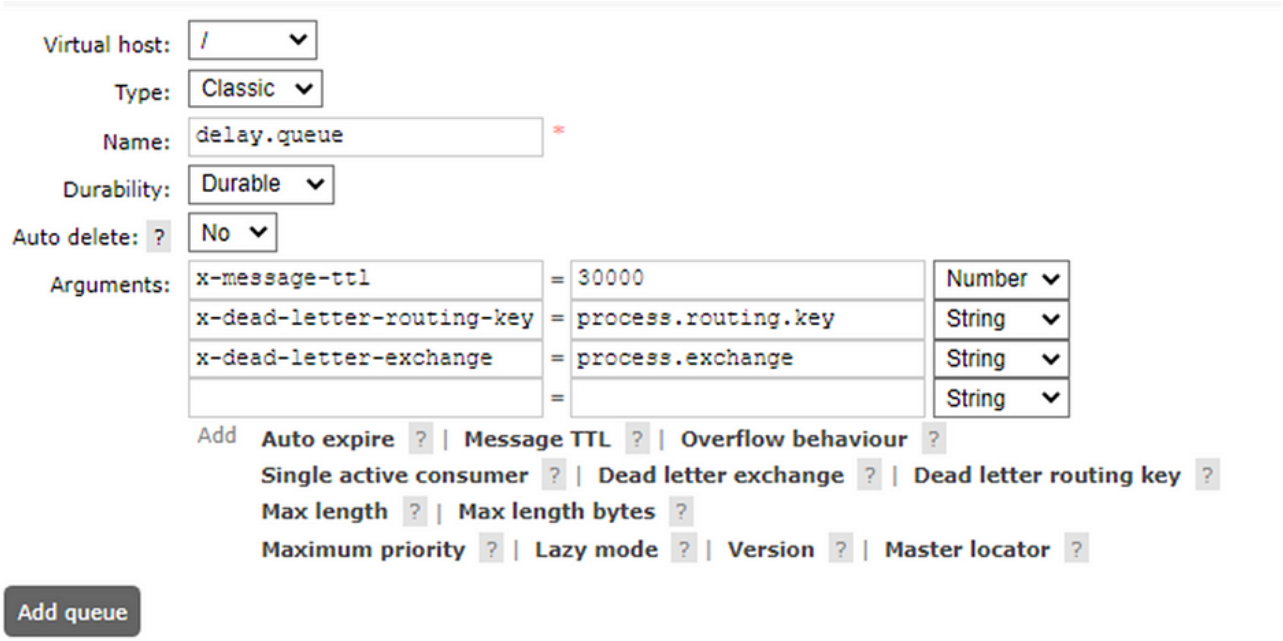
x-queue-mode=<mode>

Controls the mode of queue storage. The `default` mode uses traditional queue storage, while `lazy` mode reduces memory usage by storing messages on disk.

```
rabbitmqadmin declare queue name=my_queue arguments='{"x-queue-mode":"lazy"}'
```

x-expires=<seconds>

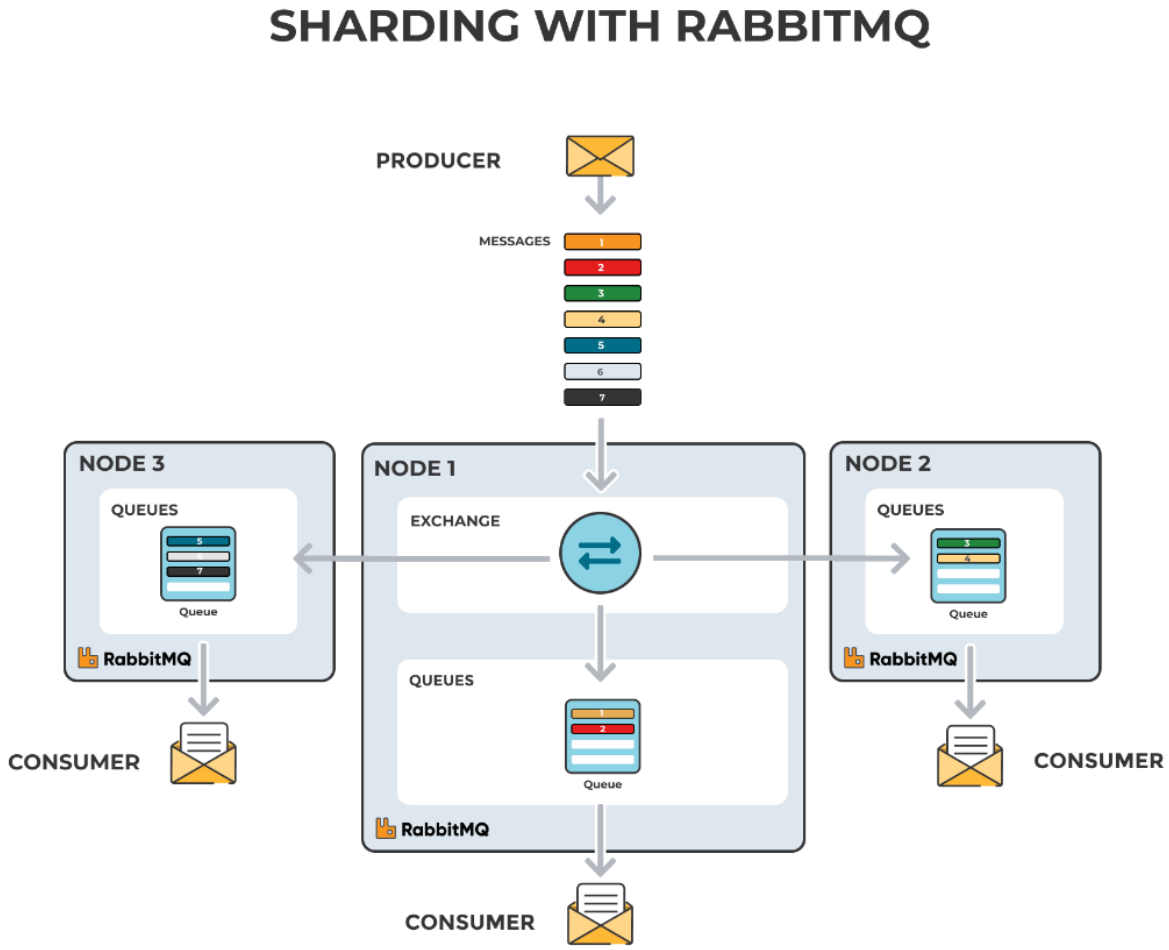
Specifies the Time-To-Live (TTL) for the queue itself. If the queue is not used for the specified duration, it will be automatically deleted. This helps in managing resources by cleaning up unused queues.



Plugins

RabbitMQ Sharding Plugin

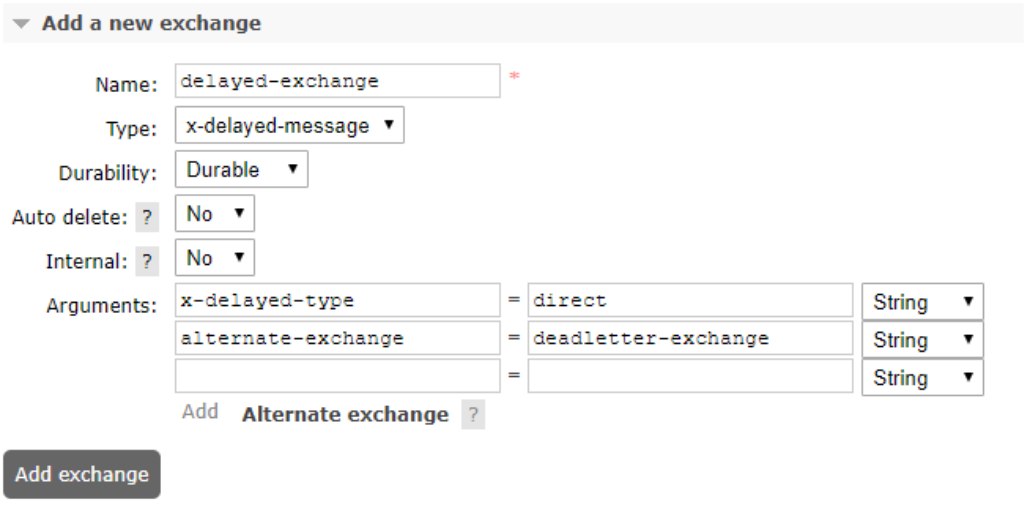
The RabbitMQ Sharding Plugin automatically distributes messages across multiple queues, which are referred to as shards. This distribution helps balance the load and can prevent any single queue from becoming a bottleneck.



Delay Excange Plugin

Enable plugin with command

```
rabbitmq-plugins enable rabbitmq_delayed_message_exchange
```



linux 安装

兼容:

RabbitMQ版本	Erlang最低版本要求	Erlang最高版本要求
3.7.15 - 3.7.16	20.3.x	22.0.x
3.7.7 - 3.7.14	20.3.x	21.3.x
3.7.0 - 3.7.6	19.3	20.3.x

yum install rabbitmq-server-3.7.7-1.el7.noarch.rpm 安装 rabbitmq
systectl start rabbitmq-server 启动 rabbitmq
/usr/sbin/rabbitmq-plugins enable rabbitmq_management 启用插件
vi /etc/rabbitmq/rabbitmq.conf 修改启动端口
listeners.tcp.default = ip:6573

rabbitmqctl add_user admin password 添加登录用户
rabbitmqctl set_user_tags admin administrator

yum install make gcc gcc-c++ openssl openssl-devel unixODBC unixODBC-devel kernel-devel m4 ncurses-devel 安装 Erlang 编译所依赖的环境（缺少 erlang 依赖时安装）
获取 erlang 依赖 rpm 仓库：https://github.com/rabbitmq/erlang-rpm
yum update -y
yum install -y erlang
erl 查看安装是否成功

集群安装

RABBITMQ_NODE_PORT=5673 RABBITMQ_NODENAME=rabbit1 rabbitmq-server start 启动节点 1
rabbitmqctl -n rabbit1 stop 结束节点
rabbitmqctl -n rabbit1 start_app rabbit1 操作作为主节点
rabbitmqctl -n rabbit1 stop_app
rabbitmqctl -n rabbit1 reset

rabbitmqctl -n rabbit2 join_cluster rabbit@'xxhostname' rabbit2 作为从节点
rabbitmqctl cluster_status -n rabbit1 查看集群状态

镜像队列（默认集群模式只有交换机，队列绑定同步，但是不会复制队列内的消息）
rabbitmqctl set_policy my_ha "^" '{"ha-mode":"all"}' 开启镜像队列

Command

rabbitmqctl

rabbitmqctl list_queues 查看队列【--vhost=saidake 指定虚拟主机】
rabbitmqctl environment 查看环境变量
rabbitmqctl list_exchanges 查看 exchanges
rabbitmqctl list_queues name messages_unacknowledged 查看未被确认的队列
rabbitmqctl list_users 查看用户
rabbitmqctl list_queues name memory 查看单个队列的内存使用
rabbitmqctl list_connections 查看连接
rabbitmqctl list_queues name messages_ready 查看准备就绪的队列
rabbitmqctl list consumers 查看消费者信息

rabbitmqctl trace_on 开启 Firehose 命令（打开 trace 会影响消息写入功能，适当打开后请关闭）
rabbitmqctl trace_off 关闭 Firehose 命令

firehose 的机制是将生产者投递给 rabbitmq 的消息，rabbitmq 投递给消费者的消息按照指定的格式发送到默认的 exchange 上。

这个默认的 exchange 的名称为 amq.rabbitmq.trace，它是一个 topic 类型的 exchange。

发送到这个 exchange 上的消息的 routing key 为 publish.exchangename 和 deliver.queuename。

其中 exchangename 和 queuename 为实际 exchange 和 queue 的名称，分别对应生产者投递到 exchange 的消息，和消费者从 queue 上获取的消息。

rabbitmq-plugins

rabbitmq-plugins enable <plugin-name> Enable plugin, you can enable the plugin by modifying the `rabbitmq.conf` file.

DAM / Kafka

Core

Apache Kafka is an open-source distributed event streaming platform used for building real-time data pipelines and streaming applications.

It was originally developed by LinkedIn and later became an open-source project under the Apache Software Foundation. Kafka is designed to handle high-throughput, fault-tolerant, and scalable data streaming across distributed systems.

Components

Producers

- Applications that send messages to Kafka topics. They publish messages to specific topics.

Consumers

- Applications that subscribe to Kafka topics and process the records they receive. Consumers can be part of consumer groups for parallel processing.

Groups

- A Kafka consumer group is a group of consumers that work together to consume messages from one or more Kafka topics.
- Each consumer in a group reads from a unique subset of the partitions of the topic(s) they are subscribed to.
- Load Balancing
  - Consumers with the same group ID will share the load of consuming messages from partitions.
  - Kafka ensures that each partition is consumed by exactly one consumer within a consumer group, which allows for load balancing and fault tolerance.
- Fault Tolerance
  - If a consumer in a group fails, Kafka will rebalance the load among the remaining consumers in the group. This ensures that the consumption can continue without interruption.
- Offset Management
  - Kafka tracks the offset of messages for each consumer group, allowing consumers to resume consumption from where they left off after a failure or restart.

Topics

- Categories or feed names to which records are sent by producers. Topics are the basic building block of Kafka.
- A topic is essentially a named stream of records or messages. It acts as a category to which data is published by producers and from which data is consumed by consumers.

Partitions

- Subdivisions of topics that allow for parallel processing. Each partition is an ordered, immutable sequence of records.
- Example: The "orders" topic might have three partitions, allowing for concurrent consumption by three consumers.
- Ordered, immutable sequences of records that allow topics to be parallelized.
- Each partition can be assigned to different brokers for scalability and fault tolerance.
- Message order
  - Key:

When producing messages, you can specify a key.

Kafka uses this key to determine the partition.

All messages with the same key will be routed to the same partition, allowing you to maintain the order for those messages.

Partition Load:

If you have a few partitions and a large volume of messages, some partitions might become overloaded, especially if certain keys are very frequent.

By increasing the number of partitions, you can reduce the risk of having any single partition becoming a hot spot, thus improving overall system performance and reliability.

Key Distribution:

While messages with the same key go to the same partition, the distribution of keys may vary.

Adding more partitions helps in balancing the distribution of different keys and managing the load better.

Single Partition:

Message order is only guaranteed within a partition, not across multiple partitions.

This way, a single consumer in the consumer group will consume all messages from that partition in the exact order they were produced.

Rebalancing

- The number of group members has changed.
- The number of subscribed topics has changed.
- The number of partitions of subscribed topics has changed.

Kafka will rebalance the partitions among the remaining consumers.

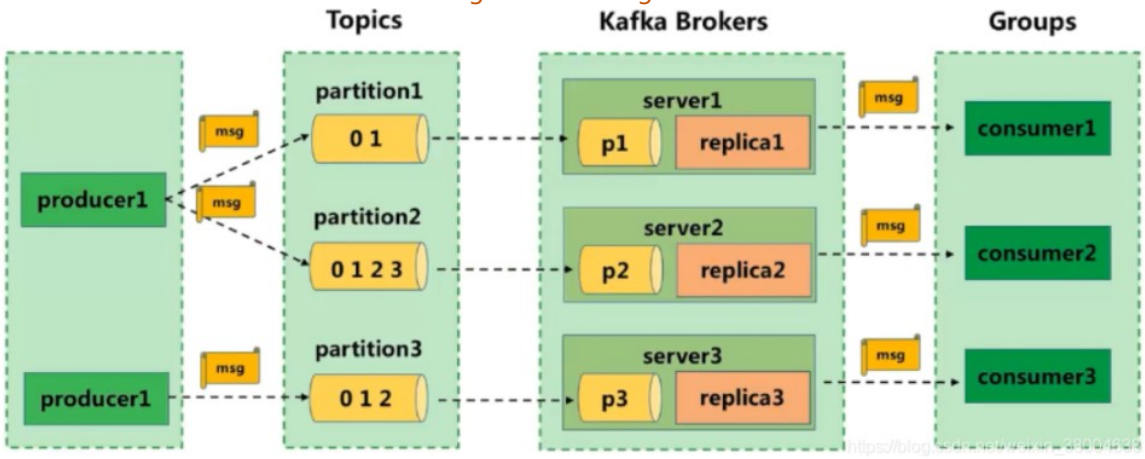
Messages in the uncommitted offsets will be reprocessed by the new consumer assigned to the partition.

Brokers

Servers that store data and serve client requests in a Kafka cluster. Each broker is responsible for a subset of topic partitions.

Zookeeper

A distributed coordination service used by Kafka to manage cluster metadata and ensure synchronization. Zookeeper handles tasks like leader election and configuration management.



File Storage Mechanism

Partition Structure

In Kafka, a partition is a fundamental unit of scalability and parallelism.

Partitions are represented as directories on the server where the data is stored.

Each partition holds a subset of the messages for a topic and is identified by a unique number.

Partition Representation:

Each partition is represented as a directory on the file system.

For example, if a topic named simon has two partitions, the corresponding directories would be named simon-0 and simon-1.

```
drwxrwxr-x 2 simon simon 4096 12月 18 10:41 simon-0
drwxrwxr-x 2 simon simon 4096 12月  8 22:18 simon-1
[simon@hadoop102 data]$
```

Segments:

Each partition is further divided into multiple segments to prevent any single log file from becoming too large, which would make data retrieval inefficient. Segments consist of:

.log files

These files contain the actual messages.

These files store the actual messages sent by producers.

Each log file in a segment is named based on the offset of the first message in that segment.

.index files

These files provide an index to the messages in the .log file, enabling quick lookups.

The .index file maps offsets to the physical position in the log file where the message can be found.

This allows Kafka to quickly locate messages without scanning the entire log file.

.timeindex files

These files are used to index messages based on their timestamp, facilitating time-based retrieval.

The .timeindex file maps timestamps to offsets, enabling efficient time-based searches.

Message Structure

When a producer sends a message to Kafka, it is written into a partition's log file. The structure of a message in Kafka includes:

Offset

The offset is a unique identifier for each message within a partition. It is an 8-byte integer that represents the position of the message in the partition.

Offsets are ordered and monotonically increase, allowing consumers to track their progress in reading messages.

Each message in a Kafka partition is assigned a sequential offset starting from 0.

Message Size

The message size is a 4-byte integer that indicates the size of the message body.

This allows Kafka to know how much data to read for a specific message.

Message Body

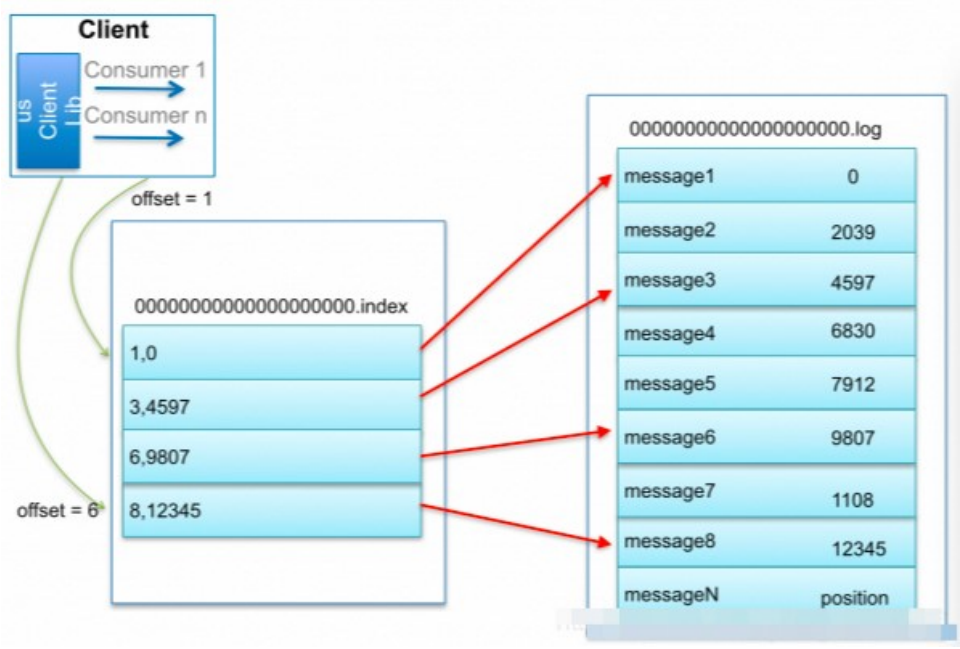
The actual data of the message, which can be compressed to save space.

The size of the message body varies depending on the content being stored.

Partition Allocation Algorithm: RangeAssignor

The RangeAssignor distributes partitions to consumers in a way that aims to minimize the number of partitions each consumer manages.

This strategy assigns contiguous ranges of partitions to each consumer.



Deletion Strategies

Kafka retains all messages, whether they have been consumed or not.

However, to manage storage efficiently and prevent the system from running out of space, Kafka implements the following deletion strategies:

Time-Based Retention:

By default, Kafka retains messages for 168 hours (7 days).

After this period, the messages are eligible for deletion.

Size-Based Retention:

Kafka also deletes messages based on the total size of the data in a partition.

The default maximum size is 1 GB (1073741824 bytes).

Once the partition reaches this size, older segments are deleted to make room for new messages.

Partition Data Redundancy [B]

Partition in Kafka provides data redundancy through a concept known as replication. Here's how partitions contribute to data redundancy:

Replication of Partitions

Multiple Copies:

Each partition in Kafka can be replicated across multiple brokers.

This means that for every partition, Kafka creates one or more copies (replicas) of the data on different brokers.

Load Distribution:

By replicating partitions across multiple brokers, Kafka not only ensures data redundancy but also distributes the load, leading to better performance and balanced resource utilization.

Durability

By having multiple replicas of the data, Kafka ensures that even if a broker goes down, the data remains safe and accessible from other brokers.

Replication Factor:

The number of replicas for a partition is determined by the replication factor, which is set when creating the topic.

For example, a replication factor of 3 means that each partition will have three copies on different brokers.

High Availability

Replicas

If one broker fails, Kafka can continue serving data from the other replicas of the partition.

This ensures that data is not lost and remains available even in the event of hardware failures.

Leader and Followers

In a replicated partition, one broker acts as the "leader," and the others are "followers."

The leader handles all read and write operations, while the followers replicate the data from the leader.

If the leader fails, Kafka automatically promotes one of the followers to be the new leader.

Unclean Leader Election

Kafka can be configured to avoid promoting an out-of-sync follower as the leader, which could lead to data loss.

This further strengthens data redundancy and ensures data integrity.

Broker Memory Data

The Kafka broker stores **HW**, **LEO**, and **LSO** in memory for efficient real-time operations.

These values are crucial for managing the state of log partitions and ensuring data integrity.

They are periodically checkpointed to disk to enable recovery in case of a broker restart or failure.

(Writes the current state of certain in-memory data to disk.)

HW (High Watermark):

The High Watermark (HW) in Kafka indicates the highest offset that has been successfully replicated to all in-sync replicas (ISR). Consumers can only read messages up to this offset.

HW ensures that only fully replicated data is visible to consumers, preventing them from reading data that may be lost in case of a broker failure.

LEO (Log End Offset):

The Log End Offset (LEO) is the offset of the next message that will be written to a partition.

It represents the end of the log.

LEO is used internally by Kafka to manage where new messages should be appended in the log.

It's also used to track how far each replica has caught up with the leader.

LSO (Log Start Offset):

The Log Start Offset (LSO) is the earliest offset in the log that has not been deleted due to log retention policies.

It represents the start of the log that is available to consumers.

LSO helps in managing log size by marking the beginning of the retained data, allowing Kafka to delete old data that is no longer needed according to the configured retention policies.

Fast Reading and Writing Speed

Kafka, as a disk-based log messaging queue system, achieves high read and write speeds due to several key characteristics and optimizations:

Parallel Processing

Kafka divides messages into topics, and each topic is further divided into partitions. These partitions are processed in parallel across different brokers.

This parallelism allows Kafka to handle large volumes of data efficiently, as different partitions can be written to and read from concurrently.

Parallel Consuming

Kafka is designed to support high concurrency, allowing multiple consumers to read messages from different partitions simultaneously.

This capability ensures that Kafka can handle a large number of concurrent requests without performance degradation.

Data Compression

Kafka supports data compression, which reduces the size of messages stored on disk and the amount of data transmitted over the network.

This not only saves storage space but also decreases the time required to read and write data, further improving overall performance.

Zero-Copy Technology

Kafka uses zero-copy technology to improve the efficiency of data transfer.

Zero-copy enables data to be transferred directly from the disk to the network socket without passing through the application layer multiple times.

This reduces CPU overhead and accelerates the transmission of data to consumers.

Sequential Disk Writes

Kafka employs sequential disk writes to append messages to the end of the log file, avoiding random I/O operations on the disk.

Sequential writes are significantly faster than random writes because they minimize the time spent on disk seek operations.

By appending data to the end of a log file, Kafka can take full advantage of the disk's sequential write speed.

Memory Optimization (Memory-Mapped Files)

Kafka uses memory-mapped files, which allow the operating system to map a portion of the disk file into memory.

This technique reduces the need for frequent disk I/O operations by enabling direct memory access to file contents, resulting in faster reads and writes.

Partition Write Strategies

Round-Robin Strategy

When a message does not specify a key, the producer uses a round-robin approach to distribute messages evenly across all available partitions.

Characteristics:

- Ensures balanced load across partitions.
- Does not guarantee message order, as messages are distributed randomly among partitions.

Key-Based Partitioning Strategy

When a message includes a key, Kafka computes the partition by hashing the key value.

The resulting hash determines the specific partition to which the message is assigned.

Characteristics:

- Ensures that all messages with the same key are sent to the same partition, preserving message order for that key.
- Useful for scenarios where message order is important, such as processing transactions for a specific user.

Custom Partitioning Strategy

Kafka allows producers to implement custom partitioning logic through a custom Partitioner class.

This allows messages to be assigned to partitions based on specific business logic or content within the messages.

Characteristics:

- Offers high flexibility, allowing you to meet complex business requirements.
- Developers are responsible for implementing and maintaining the custom partitioning logic.
- Useful for scenarios where standard partitioning strategies do not meet specific needs.

Partition Read Strategies

Polling

Consumers periodically poll the Kafka server to fetch new messages.

Characteristics**:**

- Simple to implement and understand.
- Requires configuring the polling interval to balance real-time performance and system load.
- Can introduce some delay between message production and consumption if the polling interval is too long.

Batch Fetch

Consumers read multiple messages from a partition in a single request, processing them in batches.

Characteristics:

- Reduces network overhead and improves throughput by minimizing the number of network requests.
- Batch size and fetch interval need to be balanced between latency (how quickly messages are processed) and throughput (how many messages are processed at once).
- Helps to maximize resource utilization and efficiency, especially for high-throughput applications.

Offset Commit

Automatic Offset Commit

Kafka automatically commits the offsets of messages that have been consumed, without the need for manual intervention from the consumer.

Characteristics:

- Simplifies the consumer code by managing offset commits automatically.
- May lead to message duplication or loss in case of failures, as offsets might be committed before the message is fully processed.
- Suitable for scenarios where exact message processing order is not critical.

Manual Offset Commit

Consumers explicitly manage and commit the offsets of messages after they have been processed.

Characteristics:

- Provides greater control over the consumption process, allowing for precise handling of offsets.
- Useful for scenarios requiring exactly-once or at-least-once processing guarantees.
- Requires additional handling logic in the consumer to ensure offsets are committed correctly, and to handle failures and retries.

Implementation

Topic Subscription Modes

Subscription:

Consumers subscribe to one or more topics. Kafka uses this subscription to manage partition assignment.

Topic Subscription:

Consumers specify the topics they are interested in, and Kafka ensures that each partition of those topics is assigned to a consumer in the group.

Partition Assignment Strategies

Round-Robin Assignment:

Partitions are distributed across consumers in a round-robin fashion, ensuring that each consumer gets a roughly equal number of partitions.

This method is simple but does not consider the partition load.

Range Assignment:

Partitions are assigned in ranges. For instance, the first consumer might get partitions 0-3, and the next consumer gets partitions 4-7.

This strategy can lead to imbalance if partitions have different loads.

Sticky Assignment:

This strategy tries to keep partitions assigned to the same consumer as much as possible, even when the number of consumers changes.

This helps to minimize the impact of reassignment on consumer state.

Thread Usage

Single-Threaded Consumption:

A consumer instance runs in a single thread and processes messages from one or more partitions sequentially. This approach is simpler and easier to manage but might not fully utilize the potential of modern multi-core CPUs.

Multi-Threaded Consumption:

A consumer instance runs multiple threads, where each thread processes messages from different partitions. This approach can increase throughput and make better use of multi-core processors but requires careful management of thread safety and partition allocation.

Workflow

Message Production Process

The message production process is the first step in Kafka's workflow. The producer (Producer) is responsible for publishing messages to Kafka topics. Below are the detailed steps:

Create Producer Instance

The producer first needs to create a KafkaProducer instance and configure necessary parameters like Kafka broker's address, serializer, etc.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

Construct Message

The producer constructs a ProducerRecord, which includes the target topic, message key, and message value.

```
ProducerRecord<String, String> record = new ProducerRecord<>("my-topic", "key", "value");
```

Send Message

The producer calls the send() method to send the message to the Kafka cluster. The producer can choose between synchronous and asynchronous message sending.

Partition Selection

Kafka selects the target partition based on the message key and partitioning strategy (e.g., round-robin or hash-based). If the message key is null, Kafka uses a round-robin strategy to evenly distribute the messages across partitions. Each message in a Kafka partition is assigned a sequential offset starting from 0. The offset is used to track the position of the message within that partition.

Message Serialization

The producer serializes the message key and value into byte arrays for network transmission and storage.

Message Sending

The producer sends the serialized message to the leader broker of the target partition.

Message Acknowledgement

After receiving the message, the leader broker writes it to the local log file and sends an acknowledgment (ACK) to the producer.

```
producer.send(record);
producer.close();
```

Message Storage Process

The message storage process is the second step in Kafka's workflow. The broker is responsible for persisting messages to disk and ensuring high availability and fault tolerance. Below are the detailed steps:

Receive Message

The leader broker receives the message from the producer and writes it to the local log file.

Replica Synchronization

The leader broker synchronizes the message to the follower brokers (replicas). Follower brokers write the message to their local log files and send acknowledgments back to the leader.

Message Commit

Once the leader broker receives acknowledgments from a sufficient number of replicas, it marks the message as committed. Committed messages are visible to consumers.

Log Management

Kafka periodically cleans up expired log segments to free up disk space. Kafka also supports log compaction, which retains only the latest message for each key.

Message Consumption Process

The message consumption process is the third step in Kafka's workflow. The consumer (Consumer) is responsible for subscribing to and consuming messages from Kafka topics. Below are the detailed steps:

Create Consumer Instance:

The consumer first needs to create a KafkaConsumer instance and configure necessary parameters like Kafka broker's address, deserializer, etc.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

Subscribe to Topics

The consumer calls the subscribe() method to subscribe to one or more topics. Consumers can consume messages individually or as part of a consumer group.

```
consumer.subscribe(Collections.singletonList("my-topic"));
```

Poll Messages

The consumer calls the poll() method to pull messages from the Kafka cluster. The consumer can set the polling interval and the number of messages to fetch.

```
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
```

Message Deserialization

The consumer deserializes the message key and value back to their original data types for processing.

Process Messages

The consumer processes the pulled messages, executing the necessary business logic.

Commit Offsets

Consumers use offsets to keep track of which messages have been read.

By remembering the last committed offset, consumers can resume processing from where they left off if they restart or encounter failures.

The consumer periodically commits the consumption progress (offset) to ensure accurate message processing and fault recovery.

Consumers can choose between automatic and manual offset commits.

```
consumer.commitSync();
```

Example:

Consider a topic partition with the following messages:

```
Offset | Message
-------|---------
  0    | "Message 1"
  1    | "Message 2"
  2    | "Message 3"
```

If a consumer reads "Message 1" and "Message 2" but hasn't yet processed "Message 3",

it might commit the offset as 1 (indicating that it has successfully processed up to "Message 2").

If the consumer fails and restarts, it will resume reading from offset 1, picking up "Message 2" and "Message 3".

Message Reliability

Producer

Acknowledgment (acks):

acks=all

Ensures that the producer waits for an acknowledgment from all in-sync replicas before considering a message successfully sent.

This setting provides high durability for the messages.

Idempotence and Retries

enable.idempotence=true

retries=Integer.MAX_VALUE

Enabling idempotence ensures that messages are not duplicated during retries.

This setting, combined with a high number of retries, ensures that the producer will attempt to send the message until it succeeds.

Consumer

Manual Offset Management:

enable.auto.commit=false: Disables auto-commit of offsets.

```
props.put("enable.auto.commit", "false");
```

commitSync: Manually commit offsets after processing each message or batch.

```
consumer.commitSync();
```

Idempotent Processing:

Ensure that your consumer processing logic is idempotent so that reprocessing messages does not cause issues.

Unique Message ID

Generating a Global Unique Message ID

When a message is produced, you assign a unique identifier to it. This can be achieved using various methods, such as:

UUIDs (Universally Unique Identifiers): Generate a UUID for each message to ensure its uniqueness.

Distributed ID Generators: Use distributed systems like Snowflake, Twitter's Snowflake, or custom implementations to generate unique IDs.

```
import java.util.UUID;

String messageId = UUID.randomUUID().toString();
ProducerRecord<String, String> record = new ProducerRecord<>("my-topic", messageId,
"message-value");
producer.send(record);
```

Storing the Message ID in a Database

After processing a message, you store its unique ID in a database. This step ensures that you can keep track of which messages have already been processed.

```
CREATE TABLE processed_messages (
    message_id VARCHAR(255) PRIMARY KEY,
    processed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Checking for Duplicate Consumption

Before processing a new message, check if its unique ID is already in the database. If the ID is present, it indicates that the message has already been processed, so you skip processing it.

```
public boolean isMessageProcessed(String messageId) {
    // Code to check if messageId exists in the database
    return true; // or false based on the actual check
}
```

Optimistic Locking

To use optimistic locking, your database schema should include a version column that tracks changes to each record.

```
CREATE TABLE messages (
    message_id VARCHAR(255) PRIMARY KEY,
    status VARCHAR(50),
    version INT DEFAULT 0,
    processed_at TIMESTAMP
);
```

When a message is consumed, follow these steps to implement optimistic locking:

Fetch the Record:

Retrieve the record using the message ID. This will include the current version number.

Process the Message:

Perform any necessary processing.

Update with Optimistic Locking:

Attempt to update the record with a new status and increment the version number.

If the version number has changed since you fetched the record, it means another process has updated it, and you should handle this conflict.

Handling Consumer Errors:

Properly handle exceptions and ensure offsets are not committed if processing fails.

Kafka Broker Configuration

Replication Factor:

Replication Factor: Set a replication factor greater than 1 to ensure data is replicated across multiple brokers.

```
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 3 --topic my_topic
```

Min In-Sync Replicas:

min.insync.replicas: Set this to a value that ensures enough replicas acknowledge the write.

```
bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-name my_topic --alter --add-config min.insync.replicas=2
```

Log Retention and Segment Size:

log.retention.hours: Set the retention period for logs.

log.segment.bytes: Adjust segment size to ensure frequent flushing.

```
log.retention.hours=168
log.segment.bytes=1073741824
```

Unclean Leader Election:

unclean.leader.election.enable=false: Prevents an out-of-sync replica from becoming the leader, which might lead to data loss.

```
unclean.leader.election.enable=false
```

Configuration
server.properties

General Configuration

broker.id=<id>

Sets the unique ID for each Kafka broker in the cluster.

log.dirs=<path>

Sets the directories where Kafka stores its log files.

```
log.dirs=/var/lib/kafka/logs
```

zookeeper.connect=<url>

Specifies the Zookeeper connection string.

```
zookeeper.connect=localhost:2181
```

Networking

listeners=<address>

Sets the listener addresses for Kafka.

```
listeners=PLAINTEXT://:9092
```

advertised.listeners=<address>

Sets the advertised listener addresses for Kafka.

```
advertised.listeners=PLAINTEXT://your.host.name:9092
```

Log Configuration

log.retention.hours=<number>

Sets the number of hours to retain log files.

```
log.retention.hours=168
```

log.segment.bytes=<number>

Sets the maximum size of a log segment file.

```
log.segment.bytes=1073741824
```

log.flush.interval.messages=<number>

Sets the number of messages to write to a log segment before forcing a flush to disk.

```
log.flush.interval.messages=10000
```

log.flush.interval.ms=<number>

Sets the maximum time in milliseconds between flushes of log segments to disk.

```
log.flush.interval.ms=10000
```

log.flush.scheduler.interval.ms=<number>

Sets the interval in milliseconds between scheduled flush checks, which determine when logs are flushed.

```
log.flush.scheduler.interval.ms=300000
```

Replication Configuration

default.replication.factor=<number>

Sets the default replication factor for automatically created topics.

```
default.replication.factor=3
```

min.insync.replicas=<number>

Sets the minimum number of in-sync replicas required for a message to be considered committed.

```
min.insync.replicas=2
```

num.partitions=<number>

Default Number of Partitions for New Topics

```
num.partitions=3
```

Leader Election

unclean.leader.election.enable=<true/false>

Allow an out-of-sync replica from becoming the leader

```
unclean.leader.election.enable=true
```

By default, this setting is false, however, some managed Kafka services, such as Amazon MSK, set this option to true by default to enhance availability.

Assigned Replicas (AR)

AR refers to all the replicas assigned to a particular partition. This includes both the leader and the follower replicas.

AR is used to distribute data across multiple Kafka brokers, ensuring high availability and fault tolerance.

In-Sync Replicas (ISR)

These are replicas that are fully caught up with the leader and have acknowledged all the recent writes.

They are considered reliable because they have the latest data.

Out-of-Sync Replicas (OSR)

These replicas are lagging behind the leader and do not have the latest data.

They may have fallen behind due to various reasons such as network issues, high load, or temporary unavailability.

OSR nodes may lack the latest messages, risking data loss if elected as the leader.

Producer Configuration

acks=<number/all>

Sets the number of acknowledgments the producer requires the leader to have received before considering a request complete.

```
acks=all
```

retries=<number>

Sets the number of retries for a failed send attempt.

```
retries=Integer.MAX_VALUE
```

enable.idempotence=<true/false>

Enables idempotence for exactly-once delivery semantics.

```
enable.idempotence=true
```

delivery.timeout.ms=<number>

Controls the maximum time to retry sending a message.

```
delivery.timeout.ms=120000
```

max.in.flight.requests.per.connection=<number>

Ensures in-flight requests do not exceed a threshold to maintain order and avoid duplication.

```
max.in.flight.requests.per.connection=5
```

Consumer Configuration

group.id=<group-id>

Sets the consumer group ID.

auto.offset.reset=<type>

Specifies what to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server.

```
auto.offset.reset=earliest
```

enable.auto.commit<true/false>

Enables or disables automatic offset commits.

```
enable.auto.commit=false
```

zoo.cfg

General Configuration

dataDir=<path>

Sets the directory where Zookeeper stores its data.

```
dataDir=/var/lib/zookeeper
```

clientPort=<port >

Sets the port where Zookeeper listens for client connections.

```
clientPort=2181
```

Monitoring and Logging

log4j.rootLogger=<level, appenders>

Sets the root logger level and appenders.

```
log4j.rootLogger=INFO, stdout
```

log4j.appender.stdout

Configures the stdout appender.

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n
```

Cluster Configuration

inter.broker.protocol.version=<version>

Sets the inter-broker protocol version.

broker.rack=<rack-id>

Specifies the rack ID for the broker.

Installation

官网安装包 https://kafka.apache.org/

DAM / ElasticSearch

Core

Elasticsearch is a search server based on Lucene (free open source information retrieval software library).

It's 'elastic' in the sense that it's easy to scale horizontally-simply add more nodes to distribute the load.

Today, many companies, including Wikipedia, eBay, GitHub, and Datadog, use it to store, search, and analyze large amounts of data on the fly.

Elasticsearch represents data in the form of structured JSON documents, and makes full-text search accessible via RESTful API and web clients for languages like PHP, Python, and Ruby.

In Elasticsearch, related data is often stored in the same index, which can be thought of as the equivalent of a logical wrapper of configuration.

Each index contains a set of related documents in JSON format. Elasticsearch's secret sauce for full-text search is Lucene's inverted index.

ELK Stack

Logs

Server logs that need to be analyzed are identified

Beats

Part of the Elastic Stack.

Beats are lightweight data shippers that send data from various sources to Elasticsearch or Logstash.

Logstash

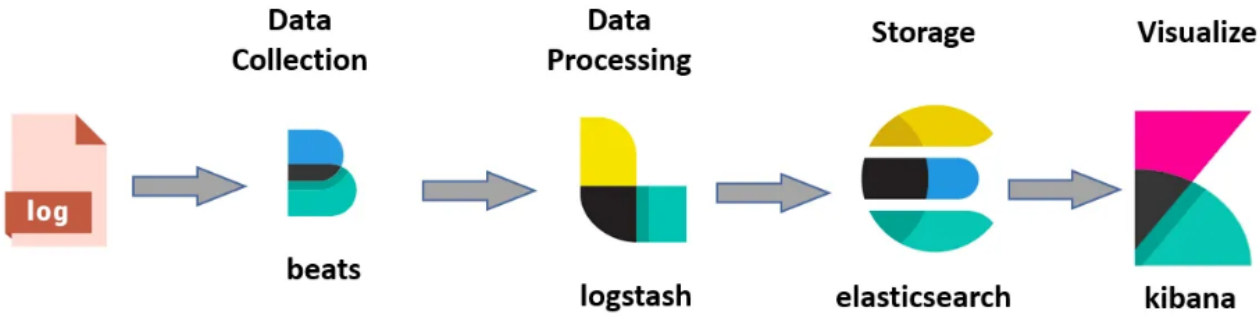
A data processing pipeline that collects, parses, and transforms logs or other event data before sending it to Elasticsearch.

ElasticSearch

A search and analytics engine where the transformed and indexed data from Logstash is stored and made searchable.

Kibana

A visualization tool that interacts with Elasticsearch to explore, visualize, and share data.



Components

Index

An index is a collection of documents that share similar characteristics. It is similar to a database in a relational database.

An index is identified by a unique name, which is used to refer to the index when performing operations like indexing, searching, updating, and deleting documents.

Example: An index named "products" could be used to store information about various products.

Document

A document is a basic unit of information that can be indexed. It is similar to a row in a relational database.

Documents are stored in JSON format and contain fields, which are the data entries in the document.

Example: A document in the "products" index might contain fields like name, price, and description.

Shard

A shard is a subset of an index and is the unit of distribution in Elasticsearch. An index can be divided into multiple shards, which can be stored on different nodes in a cluster.

Each shard is a fully functional and independent "index" that can be hosted on any node in the cluster.

Example: An index with 5 primary shards can be distributed across multiple nodes for load balancing and redundancy.

Cluster

A cluster is a collection of one or more nodes that together hold all of the data and provide federated indexing and search capabilities.

A cluster is identified by a unique name and can scale horizontally by adding more nodes.

Example: A cluster named "elasticsearch-cluster" might contain multiple nodes that share the indexing and search workload.

Node

A node is a single server that is part of the cluster. It stores data and participates in the cluster's indexing and search capabilities.

Each node is identified by a unique name and can be configured to serve different roles (e.g., master node, data node, ingest node).

Example: A node might be configured to handle only search requests or to store only specific indices.

Data Types

String Data Types

Keyword

Used for structured data that can be filtered or aggregated, such as tags, keywords, and IDs.

```
{ "status": "active" }
```

Text

Used for full-text search fields, such as product descriptions or log messages. Text fields are analyzed, meaning they are broken down into individual terms for searching.

```
{ "description": "A beautiful day in San Francisco" }
```

Numeric Data Types

Long

64-bit signed integer.

```
{ "views": 1234567890 }
```

Integer

32-bit signed integer.

```
{ "age": 30 }
```

Short

16-bit signed integer.

```
{ "temperature": 25 }
```

Byte

8-bit signed integer.

```
{ "rating": 5 }
```

Double

64-bit IEEE 754 floating point.

```
{ "price": 19.99 }
```

Float

32-bit IEEE 754 floating point.

```
{ "average_rating": 4.5 }
```

Date Data Type

Date

Used for storing dates. Elasticsearch supports multiple date formats, including epoch_millis (milliseconds since the epoch) and ISO 8601 format.

```
{ "published_date": "2023-12-31T23:59:59Z" }
```

Boolean Data Type

Boolean

Used for fields that can hold true or false values.

```
{ "is_active": true }
```

Binary Data Type

Binary

Used for storing binary data encoded as Base64.

```
{ "image": "base64encodedstring" }
```

Range Data Types

Integer Range

Used for storing a range of integers.

```
{ "age_range": { "gte": 10, "lte": 20 } }
```

Float Range

Used for storing a range of float values.

```
{ "temperature_range": { "gte": -20.5, "lte": 50.0 } }
```

Long Range

Used for storing a range of long values.

```
{ "timestamp_range": { "gte": 1609459200000, "lte": 1609545600000 } }
```

Double Range

Used for storing a range of double values.

```
{ "price_range": { "gte": 10.99, "lte": 99.99 } }
```

Date Range

Used for storing a range of dates.

```
{ "date_range": { "gte": "2023-01-01", "lte": "2023-12-31" } }
```

Array Data Type

Array

Used for storing multiple values in a single field. Any field can contain an array of values.

```
{ "tags": ["elasticsearch", "search", "analytics"] }
```

Object Data Type

Object

Used for storing nested JSON objects. Each key-value pair within the object can be of any data type.

```
{ "address": { "street": "123 Main St", "city": "Springfield", "zip": "12345" } }
```

Nested Data Type

Nested

Used for storing arrays of objects as nested fields, allowing for more complex queries on arrays of objects.

```
{
  "user": "john_doe",
  "comments": [
    { "date": "2023-01-01", "comment": "First comment" },
    { "date": "2023-02-01", "comment": "Second comment" }
  ]
}
```

Inverted Index (Features)

When a document is indexed, Elasticsearch automatically creates an inverted index for each field;
the inverted index maps terms to the documents that contain those terms as shown below:

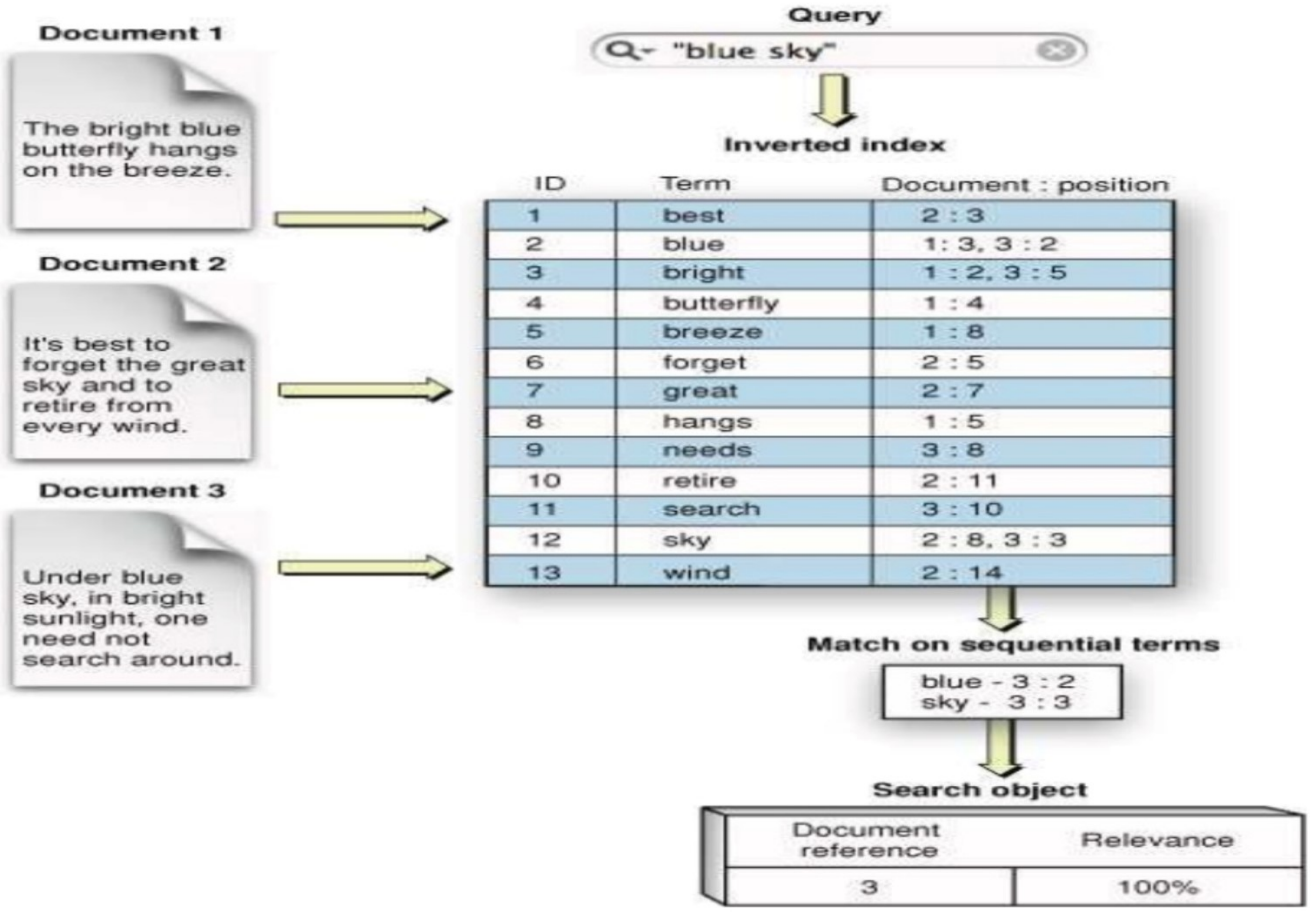
Terms Dictionary

A dictionary of unique terms found across all documents.

Posting List

For each term, a list (or posting list) of documents where the term appears.

This list often includes additional information such as term frequency and positions within the documents.



Credit: https://developer.apple.com/library/mac/documentation/userexperience/conceptual/SearchKitConcepts/searchKit_basics/searchKit_basics.html

Word Segmentation

Analyzer

Analyzers are components in Elasticsearch that handle text analysis during indexing and searching.

They consist of three parts:

- Tokenizer
  - Breaks text into terms based on certain rules (e.g., whitespace, punctuation).
- Token Filters
  - Modify or remove tokens (e.g., lowercasing, removing stop words).
- Character Filters
  - Modify the text before tokenization (e.g., removing HTML tags).

Segmenter (for non-space-separated languages)

Specialized tokenizers designed for languages where words are not explicitly separated by spaces.

Update Latency

Refresh Interval:

- Elasticsearch uses a mechanism called **refresh** to make recently indexed or updated documents visible to search.
- The default refresh interval is 1 second (index.refresh_interval), meaning updates are visible to search after this interval.
- Implication
  - There is a natural delay because Elasticsearch batches changes and makes them visible at fixed intervals.

Translog (Transaction Log)

- Elasticsearch writes updates to a transaction log (translog) before applying them to the index.

This ensures durability but adds a slight delay.

Implication

The process of writing to the translog and ensuring its durability can introduce latency.

Merge Operations

Elasticsearch periodically merges smaller segments of the index into larger ones.

During this process, updated documents are marked as deleted in the old segments and added to new segments.

Implication

Until the merge is completed and the old segments are purged, the updated document may not be searchable.

Concurrency and Load

High concurrency and heavy load can increase update latency.

When multiple update operations are performed simultaneously, the system needs to manage and apply these changes efficiently.

Implication

A heavily loaded cluster may experience increased latency due to resource contention and the need to manage multiple update requests.

Cluster State and Replication:

Elasticsearch needs to ensure that updates are propagated to all replicas of a shard. This includes updating the primary shard and then synchronizing changes with replica shards.

Implication

Network latency and the time required to update replicas can contribute to overall update latency.

Indexing Buffer Size:

The size of the indexing buffer affects how quickly updates can be processed.

A smaller buffer might cause frequent flushing to disk, while a larger buffer can handle more in-memory operations.

Implication

The size and management of the indexing buffer can influence how quickly updates are applied.

Configuration
elasticsearch.yml

Cluster Settings

cluster.name: <name>

The name of the cluster. Used to identify the cluster among others.

```
cluster.name: my-cluster
```

node.master <true/false>

node.master: true indicates that the node is eligible to become a master node, whereas node.master: false means it is not.

```
node.master: false
```

Node Settings

node.name: <name>

The name of the node. Used to identify the node within the cluster.

```
node.name: my-node
```

Network Settings

network.host: <host>

The host or IP address for the network. Can be used to bind the node to a specific network interface.

```
network.host: 0.0.0.0
```

http.port: <port>

The port on which the HTTP server listens for incoming connections.

```
http.port: 9200
```

Discovery Settings

discovery.seed_hosts: <hosts>

List of host addresses to use for initial discovery.

```
discovery.seed_hosts: ["host1", "host2"]
```

cluster.initial_master_nodes: <nodes>

List of node names or IP addresses that are eligible to become master nodes.

```
cluster.initial_master_nodes: ["node-1", "node-2"]
```

Path Settings

path.data: <path>

Path to the directory where data is stored.

```
path.data: /var/lib/elasticsearch
```

path.logs: <path>

Path to the directory where logs are stored.

```
path.logs: /var/log/elasticsearch
```

Memory Settings

bootstrap.memory_lock: <true|false>

Lock the memory to prevent swapping.

```
bootstrap.memory_lock: true
```

Heap Settings

jvm.options

Path to the JVM options file. Configure JVM options like heap size.

```
jvm.options: /etc/elasticsearch/jvm.options
```

Security Settings

xpack.security.enabled: <true|false>

Enable or disable security features.

```
xpack.security.enabled: true
```

xpack.security.transport.ssl.enabled: <true|false>

Enable or disable SSL for transport layer.

```
xpack.security.transport.ssl.enabled: true
```

Index Settings

index.number_of_shards: <number>

Number of primary shards for indices.

```
index.number_of_shards: 3
```

index.number_of_replicas: <number>

Number of replica shards for indices.

```
index.number_of_replicas: 2
```

Snapshot Settings

path.repo: <path>

Path to the repository where snapshots are stored.

```
path.repo: /mnt/snapshots
```

Logging

logger.level: <level>

Log level for Elasticsearch logging. Available levels are TRACE, DEBUG, INFO, WARN, ERROR.

```
logger.level: INFO
```

logger.rotate.size: <size>

Rotate logs when they reach this size. Example sizes: "10mb", "1gb".

```
logger.rotate.size: 10mb
```

Discovery Settings

discovery.type: <type>

Discovery type to use. Options include single-node, zen-disco, etc.

```
discovery.type: single-node
```

Cluster

Structure

A cluster is made up of multiple nodes, which can be master-eligible nodes, data nodes, or client nodes.

Client nodes

Nodes that route client requests to data nodes, such as search queries or indexing requests, balancing the load and coordinating complex operations without storing data.

Elasticsearch API can only be used in client nodes.

Specifically, you should disable both the `node.master` and `node.data` roles.

```
node.master: false
node.data: false
```

Master-eligible nodes

Master-eligible nodes in an Elasticsearch cluster are nodes that can be elected to become the master node, which manages the cluster's metadata and coordinates activities like shard allocation and node management.

```
node.master: true
```

Data nodes

Data nodes in an Elasticsearch cluster are responsible for storing and managing data, as well as performing data-related operations such as indexing, searching, and aggregating data.

node.data: true

Sharding:

Data is divided into shards. Each shard can be replicated to ensure high availability.

Indexes:

Data is stored in indexes, which are divided into shards.

Shard Size

Recommended Shard Size

General Guidelines:

Size Range:

A commonly recommended range is between 20 GB and 50 GB per shard. This is a general guideline and may need adjustment based on specific use cases and hardware capabilities.

For Small Clusters:

If you have a small cluster or limited resources, consider keeping shards on the smaller side, around 10 GB to 20 GB, to ensure that the cluster remains responsive and manageable.

For Large Clusters:

Larger clusters with ample resources can handle larger shards, but it's generally advisable to avoid shards exceeding 50 GB to ensure efficient recovery and management.

Impact of Shard Size

Small Shards:

Pros: Easier to manage and recover; lower risk of a single shard becoming a performance bottleneck.

Cons: Can lead to a large number of shards, increasing cluster overhead and management complexity.

Large Shards:

Pros: Fewer shards reduce management overhead; improved search performance as fewer shards are queried.

Cons: Longer recovery times; potential for increased resource contention and decreased performance if a shard becomes too large.

Configure Shards

Set the Number of Primary Shards and Replicas:

You configure the number of primary shards and replicas when creating an index. These settings can be specified in the index creation request.

Dynamic Configuration (Optional):

You can also dynamically update the number of replicas for an existing index, but the number of primary shards cannot be changed after the index is created.

You can configure the number of primary shards and replicas using the Elasticsearch API when creating an index.

Create an Index with Specific Shard Configuration

```
PUT /my-index
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 2
  }
}
```

number_of_shards: Sets the number of primary shards to 3.

number_of_replicas: Sets the number of replicas to 2, meaning each primary shard will have two replica shards.

Primary Shard Allocation: Each primary shard is allocated to a different node.

Replica Shard Allocation: Replica shards are distributed to nodes that do not contain the corresponding primary shard, ensuring that no node holds both a primary shard and its replicas.

Typically, the number of replicas should not exceed the number of data nodes significantly. A more reasonable approach would be to have fewer replicas or more data nodes.

Updating the Number of Replicas Dynamically

You can change the number of replicas for an existing index dynamically.

```
PUT /my-index/_settings
{
  "index": {
    "number_of_replicas": 1
  }
}
```

Example

Scenario:

Number of Primary Shards: 3

Number of Data Nodes: 2

Data Nodes: 2

Client Nodes: 1

Elasticsearch will distribute the three primary shards across the two data nodes.

- Data Node 1: Shard 0, Shard 1
- Data Node 2: Shard 2

Scenario:

- Number of Primary Shards: 3
- Number of Replicas: 2
- Total Shards (including replicas): 3 primary shards + 2 replicas per primary shard = 9 shards
- Data Nodes: 3
- Client Nodes: 1

Elasticsearch will attempt to distribute the shards and their replicas as evenly as possible across the available data nodes. Each primary shard will have two replicas, meaning each primary shard and its replicas must be spread across three different nodes.

Example Distribution:

- Data Node 1:
  - Primary Shard 0
  - Replica of Shard 1
  - Replica of Shard 2
- Data Node 2:
  - Primary Shard 1
  - Replica of Shard 0
  - Replica of Shard 2
- Data Node 3:
  - Primary Shard 2
  - Replica of Shard 0
  - Replica of Shard 1

Cluster Configuration

elasticsearch.yml

```
# Cluster and Node Configuration
cluster.name: my-cluster
node.name: node-1

# Network Settings
network.host: 0.0.0.0
http.port: 9200

# Discovery and Cluster Formation
discovery.seed_hosts: ["node-1:9300", "node-2:9300", "node-3:9300"]
cluster.initial_master_nodes: ["node-1", "node-2", "node-3"]

# Node Roles
node.master: true
node.data: true
node.ingest: true

# JVM Settings (in jvm.options file)
-Xms1g
-Xmx1g
```

Start Elasticsearch

Start the Elasticsearch service on each node.

```
sudo systemctl start elasticsearch
```

Verify Cluster Health

Use the _cluster/health API to check the cluster status.

```
curl -X GET "localhost:9200/_cluster/health?pretty"
```

Replication and Sharding

```
PUT /my-index/_settings
{
  "index": {
    "number_of_replicas": 1
  }
}
```

Election

Master-Eligible Nodes

Master-Eligible Node Configuration:

Nodes that can participate in the master election process have node.master: true set in their configuration.

Quorum:

A majority of master-eligible nodes must agree on the elected master for the cluster to be operational. This majority is often referred to as quorum.

Election Process

Discovery Phase:

When an Elasticsearch cluster starts or a node joins the cluster, nodes enter the discovery phase.

During this phase, master-eligible nodes discover each other and communicate to form a cluster.

Ping Requests:

Master-eligible nodes send ping requests to each other to detect existing masters and to share their node information.

If a node discovers an existing master, it will join the cluster by sending a join request.

Master Election:

If no master is found, an election process begins.

Nodes use the Zen Discovery module to select a master. This module implements the Bulldog Algorithm to choose a master node.

Each node in the cluster votes for a master-eligible node based on the highest node ID, which is a combination of the node's internal unique identifier and the node's address.

Election Winner:

The node with the highest ID and majority votes becomes the master node.

The elected master node then becomes responsible for managing cluster state and shard allocation.

Publishing Cluster State:

The new master node publishes the current cluster state to all nodes in the cluster.

All nodes acknowledge the master and synchronize their state with the master's state.

Failover and Re-Election

Master Node Failure

If the master node fails, the remaining master-eligible nodes detect the failure through missed heartbeats or pings.

Re-Election Process

A new master election process is initiated to select a new master from the remaining master-eligible nodes.

Minimal Disruption

Elasticsearch aims to minimize disruption during master node failure by quickly electing a new master and maintaining cluster stability.

Split-Brain Scenario

Minimum Master Nodes Setting

To prevent a split-brain scenario (where two nodes both think they are the master), Elasticsearch has a discovery.zen.minimum_master_nodes setting.

Quorum Requirement

This setting ensures that a majority of master-eligible nodes must be available to elect a new master, preventing the cluster from splitting into multiple parts with different masters.

Other

es 配置项

elasticsearch.yml >>

```
xpack.security.enabled: false          #关闭安全选项
http.cors.allow-origin: "*"            #跨域配置
http.cors.enabled: true
network.host: 0.0.0.0                  #ip 地址
http.port: 9201                        #端口
path.data: /opt/data                   #数据存储路径
path.logs: /opt/logs                   #日志存储路径

xpack.license.self_generated.type: basic     #xpack 插件的授权类型，basic 是免费的，还有其他收费版本
xpack.security.enabled: true                 #是否开启安全验证
xpack.security.transport.ssl.enabled: true   #是否开启远程访问安全验证
#————————————————————————————————————集群
cluster.name: itcast-es      #集群名称
node.name: itcast-1          #节点名称

node.master : true           #是不是有资格成为主节点
node.data: true              #是否存储数据
node.max_local_storage_nodes : 3      #最大集群节点数
transport.tcp.port : 9700             #内部节点之间沟通端口
discovery.seed_hosts: ["loca7host:9700" , "localhost:9800", "localhost:9900"]      #es7.x>>质新增的配置，节点发现
cluster.initial_master_nodes: ["itcast-1", "itcast-2", "itcast-3"]                 #es7.x 之后新增的配置，初始化一个新的集群时需要此配置来选举 master
discovery.zen.ping.timeout: 60s                增加新节点加入集群的等待时长，默认 3s。如果一次加入不上，默认重试 20 次
```

jvm.options >>

```
-Xms2g
-Xmx2g
```

es 命令（es 服务器）

GET	/_all	获取所有信息
GET	/	获取 es 信息
PUT	/person	添加索引
DELETE	/person	删除索引
GET	/person	查询索引
POST	/person/_close	关闭索引
POST	/person/_open	打开索引

kibana 配置项

kibana.yml >>

```
server. port: 5601
server.host: "0.0.0.0"
```

server.name : "kibana-itcast"
elasticsearch.hosts : ["http://localhost:9201","http://localhost:9202", "http://localhost:9203"]
elasticsearch.requestTimeout: 9999

elasticsearch.username: "kibana"
elasticsearch.password: "之前设置的密码"

kibana 命令

索引操作

```
PUT person                创建索引（默认主分片 1，副本分片 1，添加映射可省）
{
  "mappings":{
    "properties":{
      "name":{
        "type":"keyword"
      },
      "age":{
        "type":"integer"
      },
      "address":{
        "type":"text",
        "analyzer":"ik_max_word"
      }
    }
  },
  "settings":{
    "number_of_shards":3,          主分片数量
    "number_of_replicas":1         副本分片数量
  }
}
DELETE person          删除索引

POST   person_v1/_alias/person          给索引起别名，便于 javaapi 访问
POST _reindex          重建索引（字段相同的数据会被完全拷贝过去）
{
  "source":{
    "index":"person_v1"
  },
  "dest":{
    "index":"person_v2"
  }
}
GET person                              查询索引
```

映射操作

```
PUT person/_mapping       创建映射（映射创建后无法修改，只能删除后新建）
{
  "properties":{
    "name":{
```

```
            "type":"keyword"
        },
        "age":{
            "type":"integer"
        },
        "address":{
            "type":"text",
            "analyzer":"ik_max_word"        选择映射分词器
        }
      }
    }
    GET person/_mapping        查询映射
```

文档操作

```
    POST person/_doc              创建文档，不指定 id（无序字符串）
    {
      "name":"张三",
      "age":20
    }
    DELETE person/_doc/4TOJoH8BKOnWMWnCJoIq      删除文档

    PUT person/_doc/1             修改/创建文档
    {
      "name":"张三",
      "age":
    }

    GET person/_doc/4TOJoH8BKOnWMWnCJoIq      查询文档
```

查询所有文档（条件可省）

```
    GET person/_search
    {
      "query":{
        "match_all":{}                        match_all 查询所有文档，from，size 控制分页
      },
      "from":0,
      "size":100
    }
    GET person/_search
    {
      "query":{
        "term":{                              term 词条查询（查询字符串 完全匹配 词条）   match 全文查询（查询字符串分词后匹配词条，取并集） wildcard 全文分词查询（同 match，但可以使用通配符 ？任意单个字符  *  零个或多个字符）
    regexp  正则查询（ "address" : "\\w+(.)*" ）   prefix   前缀查询（ "address" : "三" ）
            "address":"北京"
        }
      }
    }
    GET person/_search
    {
```

```
  "query":{
    "range":{                                range 范围查询
      "price":{
        "gte":2000,
        "lte":3000,
        "order":"desc"
      }
    }
  }
}
GET person/_search
{
  "query":{
    "query_string":{                         query_string 多字段查询，查询字符串分词后匹配词条，取并集）
      "fields":["name","address"],
      "query":"华为 OR 手机"                  支持 AND，OR 条件指定 取并集还是交集
    }
  }
}
GET person/_search
{
  "query":{
    "query_string":{                         bool 布尔查询
      "must":[        {"term":{"address":"上海"}},  {"match":{"name":"张三"}},],          条件必须成立（and）
      "filter":[      {"term":{"address":"上海"}},  {"match":{"name":"张三"}},],          条件必须成立，性能比 must
高，不会计算得分（and）
      "must_not":[    {"term":{"address":"上海"}},  {"match":{"name":"张三"}},],          条件必须不成立（not）
      "should":[      {"term":{"address":"上海"}},  {"match":{"name":"张三"}},],          条件可以成立（or）
    }
  }
}
GET person/_search
{
  "query":{
    "term":{
      "address":"北京"
    }
  },
  "aggs":{
    "max_age":{                      自定义显示的结果字段
      "max":{                        指标聚合查询，相当于 MySQL 的聚合函数 max、min、avg、sum 等
        "field":"age"                查询到结果后，按照此字段 计算最终函数值（text 字段在聚合和排序中默认禁用，应使
用关键字字段）
      }
    }
  }
}
GET person/_search
```

```
{
  "query":{
    "term":{
      "address":"北京"
    }
  },
  "aggs":{
    "person_ages":{                    自定义显示的结果字段
      "terms":{                        桶聚合查询，分组
        "field":"age",                 查询到结果后，计算 age 各种值出现的次数（text 字段在聚合和排序中默认禁用，应使用关键字字段）
        "size":100                     显示的分组结果条数
      }
    }
  }
}
GET person/_search
{
  "query":{
    "match":{
      "address":"儿子"
    }
  },
  "highlight":{                        高亮查询，将匹配的 词条 加上前后缀
    "fields":{
      "address":{
        "pre_tags":"<font color='red'>",
        "post_tags":"</font>"
      }
    }
  }
}
```

分词器操作

```
GET _analyze                   对指定语句分词
{
  "analyzer": "standard",
  "text": "张三丰"
}
```

批量操作：

```
POST /_bulk
{"delete":{"_index":"person","_id":"5"}}          删除 5 号记录
{"create":{"_index":"person","_id":"5"}}          创建 5 号记录
{"name":"六号","age":20,"address":"北京"}
{"update":{"_index":"person","_id":"2"}}          更新 5 号记录
{"doc":{"name":"二号"}}
```

windows 安装

elasticsearch（es 官网下载）：　　elasticsearch-7.15.1-windows-x86_64.zip　　（/bin/elasticsearch.bat 启动，需要几分钟）

/bin/elasticsearch-setup-passwords.bat interactive 设置密码 elastic，apm_system，kibana，logstash_system，beats_system，remote_monitoring_user（需要先启动 es）

elasticsearch-head（git 下载）： npm install，npm start 启动

kibana（es 官网下载） kibana-7.15.1-windows-x86_64 （/bin/kibana.bat 启动）

ik 分词器（git 下载）： https://github.com/medcl/elasticsearch-analysis-ik（解压后放到/plugins，重启 es）

访问 es 接口： http://localhost:9200

访问 es 集群状态： http://localhost:9200/_cat/health?v

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending_tasks max_task_wait_time active_shards_percent

1647760192 07:09:52 elasticsearch yellow 1 1 18 18 0 0 8 0 - 69.2%

cluster 集群名称

staus 状态（green：健康，yellow 代表分配了所有主分片，但至少缺少一个副本，此时集群数据仍然完整）

node.total 代表在线的节点总数量

node.data 代表在线的数据节点的数量

shards 存活的分片数量

pri 存活的主分片数量，正常情况下 shards 的数量是 pri 的两倍。

relo 迁移中的分片数量，正常情况为 0

init 初始化中的分片数量正常情况为 0

unassign 未分配的分片正常情况为 0

pending_tasks 准备中的任务，任务指迁移分片等正常情况为 0

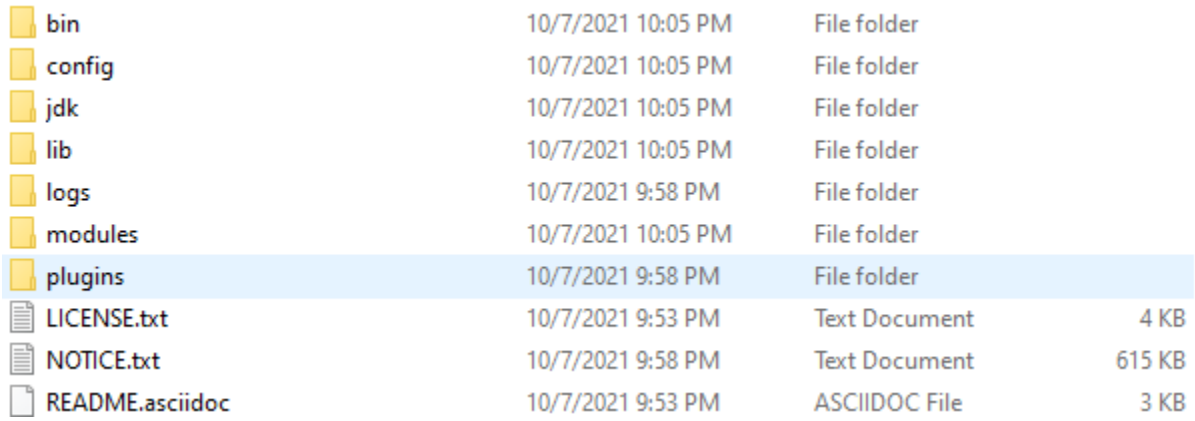
max_task_wait_time 任务最长等待时间

active_shards_percent 正常分片百分比正常情况为 100%

访问 es-head 首页： http://localhost:9100

访问 kibana 首页： http://localhost:5601

ES 目录 >>



bin 启动文件（elasticsearch.bat 启动）

config 配置文件

- log4j2 日志配置文件
- jvm.options jvm 虚拟机配置
- elasticsearch.yml es 配置文件
  - http.port: 9200 默认端口
  - http.cors.enabled: true 跨域配置
  - http.cors.allow-origin: "*"
- lib 相关 jar 包

modules	功能模块
plugins	插件
logs	日志

Command

Thread pool

For Elasticsearch versions before 7.x, you can configure thread pools directly in the elasticsearch.yml file.
As of Elasticsearch 7.x and later, direct configuration of thread pools in elasticsearch.yml is deprecated, and you should use dynamic settings instead.

```
PUT /_cluster/settings
{
  "persistent": {
    "threadpool.search.size": 10,
    "threadpool.write.size": 15,
    "threadpool.search.queue_size": 1000,
    "threadpool.write.queue_size": 2000
  }
}
```

threadpool.search.size:	Number of threads for handling search requests.
threadpool.write.size:	Number of threads for handling write requests.
threadpool.search.queue_size:	Size of the queue for search requests.
threadpool.write.queue_size:	Size of the queue for write requests.

Reduce Update Latency

Adjust Refresh Interval

Reduce the index.refresh_interval to a lower value for faster visibility of updates. However, this can increase the load on the cluster.

Example: Set the refresh interval to 500ms

```
PUT /my-index/_settings
{
  "index": {
    "refresh_interval": "500ms"
  }
}
```

Tune Translog Settings

Adjust translog settings to balance between durability and performance.

Example: Set the translog durability to async for faster updates, but with potential risk of data loss in case of a crash.

```
PUT /my-index/_settings
{
  "index.translog.durability": "async"
}
```

Manage Segment Merges

Adjust merge policies to optimize the merging process and reduce the impact on update latency.

Example: Increase the index.merge.scheduler.max_thread_count to speed up merges.

```
PUT /my-index/_settings
{
  "index.merge.scheduler.max_thread_count": 2
}
```

Optimize Indexing Performance

Ensure that the cluster has adequate resources (CPU, memory, I/O) to handle the indexing load.

Use bulk indexing to reduce the overhead associated with individual update requests.

Use Optimistic Concurrency Control

Implement optimistic concurrency control to manage conflicts and reduce the need for multiple updates.

Example: Use the version or seq_no and primary_term parameters to handle updates.

Monitor Cluster Health

Regularly monitor and optimize the cluster to ensure it is not overloaded.

Use monitoring tools to identify bottlenecks and address them proactively.

Analyzers & Segmenters

IK Analyzer (for Chinese)

IK Analyzer is a widely used tokenizer for Chinese text.

Modes:

Smart Mode: Provides a more general segmentation.

Full Mode: Offers a finer-grained segmentation by recognizing more words and phrases.

```
PUT /my-index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "ik_max_word": {
          "type": "ik_max_word"
        },
        "ik_smart": {
          "type": "ik_smart"
        }
      }
    }
  }
}
```

Jieba Analyzer (for Chinese)

Jieba Analyzer is another popular tool for Chinese word segmentation.

Known for its flexibility and ease of use.

```
PUT /my-index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "jieba": {
          "type": "jieba_analyzer"
        }
      }
    }
  }
}
```

Kuromoji Analyzer (for Japanese)

Kuromoji is used for Japanese text analysis and includes segmentation and part-of-speech tagging.

```
PUT /my-index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "kuromoji": {
          "type": "kuromoji"
        }
      }
    }
  }
}
```

Nori Analyzer (for Korean)

Nori is designed for Korean text analysis, handling segmentation and normalization.

```
PUT /my-index
```

```
{
  "settings": {
    "analysis": {
      "analyzer": {
        "nori": {
          "type": "nori"
        }
      }
    }
  }
}
```

DAM / Zookeeper

Core

Apache ZooKeeper is an open-source distributed coordination service for distributed applications.
It provides a simple and robust mechanism to coordinate and manage the configuration, synchronization, and naming of distributed systems.
ZooKeeper is widely used in large-scale distributed systems for tasks such as leader election, configuration management, and distributed locking.

Features

Hierarchical Namespace:

- ZooKeeper maintains a hierarchical namespace, similar to a file system, where each node is called a znode.
- Each znode can store data and have child nodes, enabling a tree-like structure.

Ephemeral Nodes:

- ZooKeeper supports ephemeral znodes that exist only as long as the session that created them is active.
- This is useful for detecting the presence or absence of nodes in a distributed system.

Watches:

- Clients can set watches on znodes to get notifications of changes, enabling reactive programming models.

Watches

Setting a Watch

Registering a Watch:

- When a client performs a read operation (such as getData, getChildren, or exists), it can also set a watch on the znode.
- The watch is a one-time trigger; it will notify the client the next time the znode changes and then needs to be re-registered if further notifications are desired.

Types of Watches:

- Data Watches:
  - Set using operations like getData or exists. These watches notify the client when the data of the specified znode changes.
- Child Watches:
  - Set using getChildren. These watches notify the client when the children of the specified znode change.

Triggering a Watch

Znode Changes:

- When a znode changes (e.g., data is modified, a child is added or removed), the server handling the change identifies which clients have set watches on the znode.

Notification:

- The server sends a notification to the client(s) that have set a watch on the changed znode.
- Notifications are sent asynchronously, and the client will receive an event indicating the type of change.

Handling Watch Events

Client-Side Event Handling:

- The client library includes a watcher interface (e.g., Watcher in Java) that clients implement to handle watch events.

When a watch triggers, the ZooKeeper client library invokes the appropriate watcher method, passing an event object with details about the change.

Event Types:

NodeCreated:	The watched znode was created.
NodeDeleted:	The watched znode was deleted.
NodeDataChanged:	The data of the watched znode was changed.
NodeChildrenChanged:	The children of the watched znode were changed.

Configuration

zoo.cfg

```
dataDir=/usr/local/zookeeper-cluster/zookeeper-2/data
extendedTypesEnabled=true
```

Cluster

Components

A ZooKeeper cluster typically consists of an odd number of servers (e.g., 3, 5, 7) to ensure quorum-based decision-making.

Leader:

- The primary server that handles all write requests.
- Coordinates updates and ensures data consistency.

Followers:

- Servers that replicate the leader's data.
- Handle read requests and participate in leader elections.

Observers (optional):

- Do not participate in leader elections.
- Replicate data and handle read requests to increase read scalability without affecting quorum size.

Leader Election

Initial State:

- When a ZooKeeper server starts, it initially votes for itself as the leader.

Broadcasting Votes:

- Each server broadcasts its vote to all other servers in the ensemble.

Receiving Votes:

- Each server receives votes from other servers. Initially, each server may have voted for itself.

Processing Votes:

- Each server processes the votes it receives based on the epoch and the server ID.
- The epoch is a term or logical clock used to identify the round of leader election. A higher epoch indicates a more recent vote.
- If two servers have the same epoch, the server ID is used to break ties. The server with the higher ID is preferred.

Updating Votes:

- Servers update their votes if they receive a vote with a higher epoch or, if the epoch is the same, a higher server ID.
- This ensures that votes gradually converge towards a single candidate.

Convergence:

- The process continues until a quorum of servers (a majority) agree on the same leader.
- Once a server sees that a quorum has been reached for a particular candidate, it recognizes that candidate as the leader.

Replication

Features

Ensemble:

- A ZooKeeper ensemble typically consists of an odd number of servers to prevent split-brain scenarios and to ensure a majority (quorum) can make decisions.

Common ensemble sizes are 3, 5, or 7 servers.

Quorum:

A majority of servers (quorum) must agree on any changes to the data.

This ensures that updates are consistently applied and the system can tolerate failures.

Replication Process

Proposal:

When a client sends a write request, the leader creates a proposal for the update.

The proposal includes the update and a unique transaction ID.

Broadcast:

The leader broadcasts the proposal to all followers.

Followers receive the proposal and write it to their local logs.

Acknowledgment:

Followers send an acknowledgment (ack) back to the leader once they have written the proposal to their logs.

The leader waits for acknowledgments from a quorum of followers.

Commit:

Once the leader receives acks from a quorum, it marks the proposal as committed.

The leader then broadcasts a commit message to all followers.

Followers apply the committed update to their local state.

Client Notification:

After the update is committed, the leader sends a response to the client, indicating the success of the operation.

Other

内部命令

./ zkcli.sh -server 127.0.0.1:2181　　连接 ZooKeeper 服务端

help　　查看命令帮助

quit　　断开连接

ls /xxpath　　显示指定目录下节点

create /xxnpath value　　创建节点，值不能省略，可设置为""【-e 临时节点　-s 顺序节点　-es 临时顺序节点　-t 30000　毫秒过期时间，如果在此时间内没有得到更新并且没有孩子节点，会被删除（配置项 extendedTypesEnabled）】

delete /xxnpath　　删除单个节点

deleteall　/xxnpath　　删除带有子节点的节点

set /xxnpath value　　设置节点值

get /xxnpath　　获取节点值

- czxid　　节点被创建的事务 ID
- ctime　　创建时间
- mzxid　　最后一次被更新的事务 ID
- mtime　　修改时间
- pzxid　　子节点列表最后一次被更新的事务 ID
- cversion　　子节点的版本号
- dataversion　　数据版本号
- aclversion　　权限版本号
- ephemeralOwner　　用于临时节点，代表临时节点的事务 ID，如果为持久节点则为 0
- dataLength　　节点存储的数据的长度

numChildren　　当前节点的子节点个数

bin 命令

./zkServer.sh start　启动

./zkServer.sh stop　停止

./zkServer.sh status 查看状态
./zkServer.sh restart 重启

linux 安装

下载地址（直接执行）：https://zookeeper.apache.org/releases.html
mv zoo_sample.cfg zoo.cfg 修改配置文件名

ZAB 协议

概念

ZAB 协议：概念：Zookeeper Atomic Broadcast，是为 ZooKeeper 专门设计的一种支持崩溃恢复的分布式一致性协议。基于该协议，ZooKeeper 实现了一种主从模式的系统架构来保持集群中各个副本之间的数据一致性。

原理：Zab 协议要求每个 Leader 都要经历三个阶段：发现，同步，广播。

发现：要求 zookeeper 集群必须选举出一个 Leader 进程，同时 Leader 会维护一个 Follower 可用客户端列表。将来客户端可以和这些 Follower 节点进行通信。

同步：Leader 要负责将本身的数据与 Follower 完成同步，做到多副本存储。这样也是提现了 CAP 中的高可用和分区容错。Follower 将队列中未处理完的请求消费完成后，写入本地事务日志中。

广播：Leader 可以接受客户端新的事务 Proposal 请求，将新的 Proposal 请求广播给所有的 Follower。

Zab 协议的核心

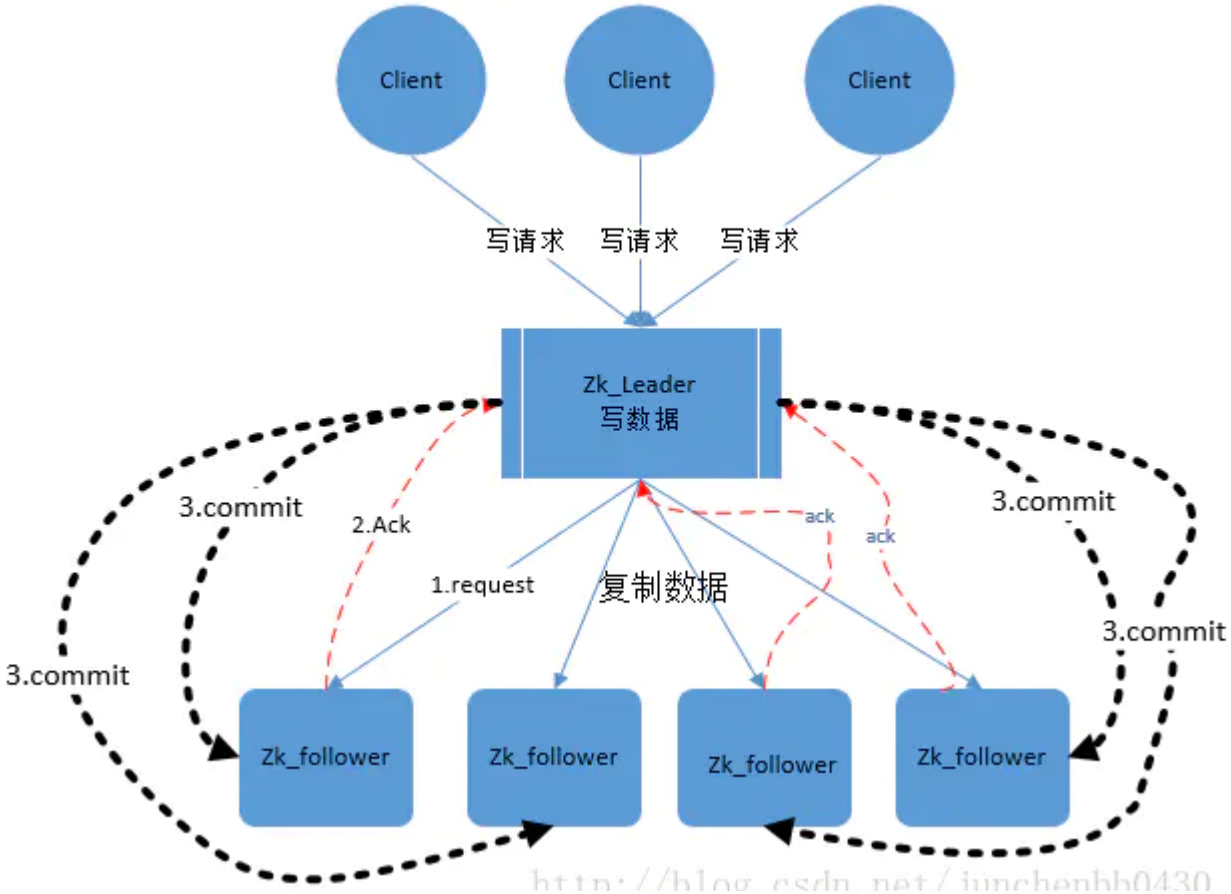
Zab 协议的核心：定义了事务请求的处理方式（事务请求为写请求，读请求直接从服务器读取数据）

1）所有的事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被叫做 Leader 服务器。其他剩余的服务器则是 Follower 服务器。

2）Leader 服务器 负责将一个客户端事务请求，转换成一个 事务 Proposal，并将该 Proposal 分发给集群中所有的 Follower 服务器，也就是向所有 Follower 节点发送数据广播请求（或数据复制）

3）分发之后 Leader 服务器需要等待所有 Follower 服务器的反馈（Ack 请求）

在 Zab 协议中，只要 Leader 服务器 收到半数以上的 Follower 的 Ack 请求，那么 Leader 就会再次向所有的 Follower 服务器发送 Commit 消息，要求其将上一个 事务 proposal 进行提交。



崩溃恢复

一旦 Leader 服务器出现崩溃或者由于网络原因导致 Leader 服务器失去了与过半 Follower 的联系，那么就会进入崩溃恢复模式（Leader 选举 和 数据恢复）

崩溃恢复要求满足如下 2 个要求：

1）确保那些已经在 Leader 服务器上 commit 提交的事务最终被所有的服务器提交

2）确保丢弃那些只在 Leader 服务器 上被提出而没有被提交的事务（新选举出来的 Leader 不能包含未提交的 Proposal。同时新选举的 Leader 节点中含有最大的 zxid）

leader 服务器发生崩溃时分为如下场景：

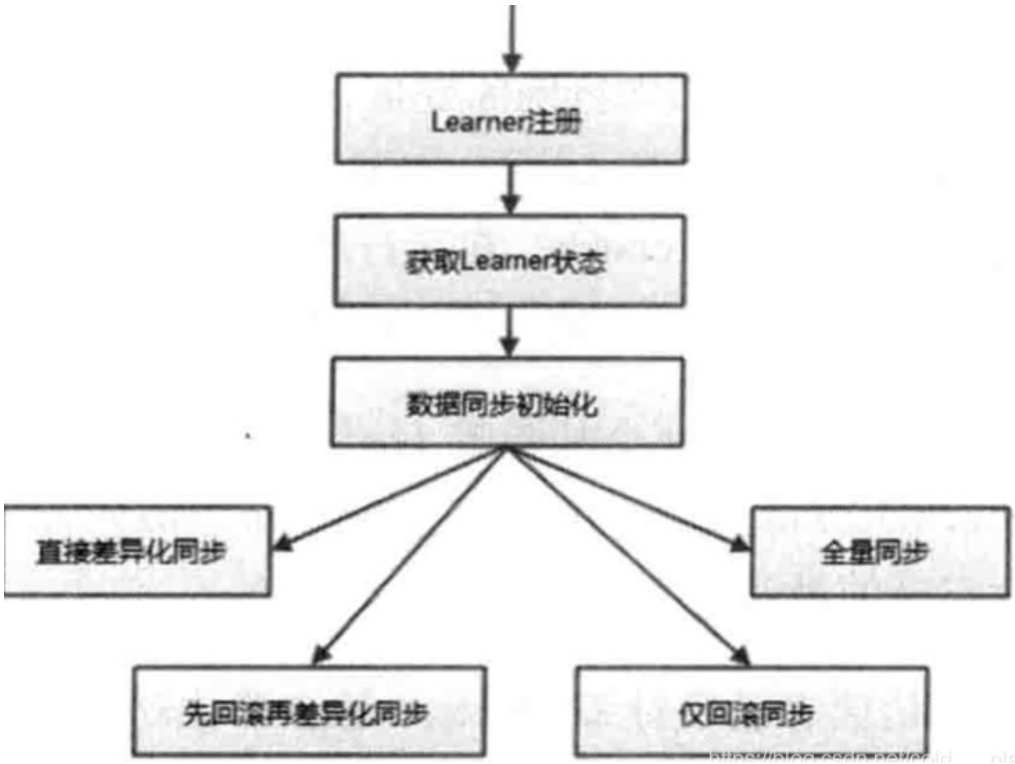
1）leader 在提出 proposal 时未提交之前崩溃，则经过崩溃恢复之后，新选举的 leader 一定不能是刚才的 leader。因为这个 leader 存在未提交的 proposal。

2）leader 在发送 commit 消息之后，崩溃。

即消息已经发送到队列中。经过崩溃恢复之后，参与选举的 follower 服务器（刚才崩溃的 leader 有可能已经恢复运行，也属于 follower 节点范畴）中有的节点已经消费了队列中所有的 commit 消息，那么最大 zid 的 follower 节点将会被选举为最新的 leader。

剩下动作就是数据同步过程。

数据同步：数据同步过程就是 Leader 服务器将那些没有在 Learner 服务器上 commit 提交过的事务请求同步给 Learner 服务器（Learner=Observer+Follwer）



获取 Learner 状态：

在向 Leader 服务器注册 Learner 的最后阶段，Learner 服务器会发送给 Leader 服务器一个 ACKEPOCH 数据包，Leader 服务器会从这个数据包中解析出该 Learner 的当前 Epoch 和最新 Zxid。

数据同步初始化：在开始数据同步之前，Leader 服务器会进行数据同步初始化，首先会从 ZooKeeper 的内存数据库中提取出事务请求对应的提议缓存队列 proposals（用"提议缓存队列"来指代该队列），同时完成对以下三个 Zxid 值的初始化。

peerLastZxid　　该 Learner 服务器最后处理的 Zxid。

minCommittedLog　Leader 服务器提议缓存队列 committedLog 中的最小 Zxid。

maxCommittedLog　Leader 服务器提议缓存队列 committedLog 中的最大 Zxid。

数据同步分类：在初始化阶段，Leader 服务器会优先初始化以全量同步方式来同步数据。当然，这并非最终的数据同步方式，在以下步骤中，会根据 Leader 和 Learner 服务器之间的数据差异情况来决定最终的数据同步方式。

直接差异化同步(DIFF 同步)

minCommittedLog < peerlastZxid <maxCommittedLog

1) DIFF 指令：Leader 服务器会首先向这个 Learner 发送一个 DIFF 指令，用于通知 Learner 进人差异化数据同步阶段 Leader 服务器即将把一些 Proposal 同步给自己。

2) 数据包：在实际 Proposal 同步过程中，针对每个 Proposal，Leader 服务器都会通过发送两个数据包来完成，分别是 Proposal 内容数据包和 commit 指令数据包（这 和 ZooKeeper 运行时 Leader 和 Follower 之间的事务请求的提交过程是一致的）

例子：假如某个时刻 Leader 服务器的提议缓存队列对应的 ZXID 依次是：0x500000001 0x500000002 0x500000003 0x500000004 0x500000005

而 Learner 服务器最后处理的 ZXID 为 0x500000003，于是 Leader 服务器就会依次将 0x500000004 和 0x500000005 两个提议同步给 Learner 服务器，同步过程中的数据包发送顺序如下：

发送顺序	数据包类
1	PROPOSAL
2	COMMIT
3	PROPOSAL
4	COMMIT

通过以上四个数据包的发送，Learner 服务器就可以接收到自己和 Leader 服务器的所有差异数据。

Leader 服务器在发送完差异数据之后，就会将该 Learner 加入到 forwardingFollowers 或 observingLearners 队列中，这两个队列在 ZooKeeper 运行期间的事务请求处理过程中都会使用到。

2) NEWLEADER 指令：

随后 Leader 还会立即发送一个 NEWLEADER 指令，用于通知 Learner，已经将提议缓存队列中的 Proposal 都同步给自己了。

Learner 接收到来自 Leader 的 NEWLEADER 指令，此时 Learner 就会反馈给 Leader 一个 ACK 消息，表明自己也确实完成了对提议缓存队列中 Proposal 的同步。

Leader 在接收到来自 Learner 的这个 ACK 消息以后，就认为当前 Learner 已经完成了数据同步，同时进入“过半策略”等待阶段

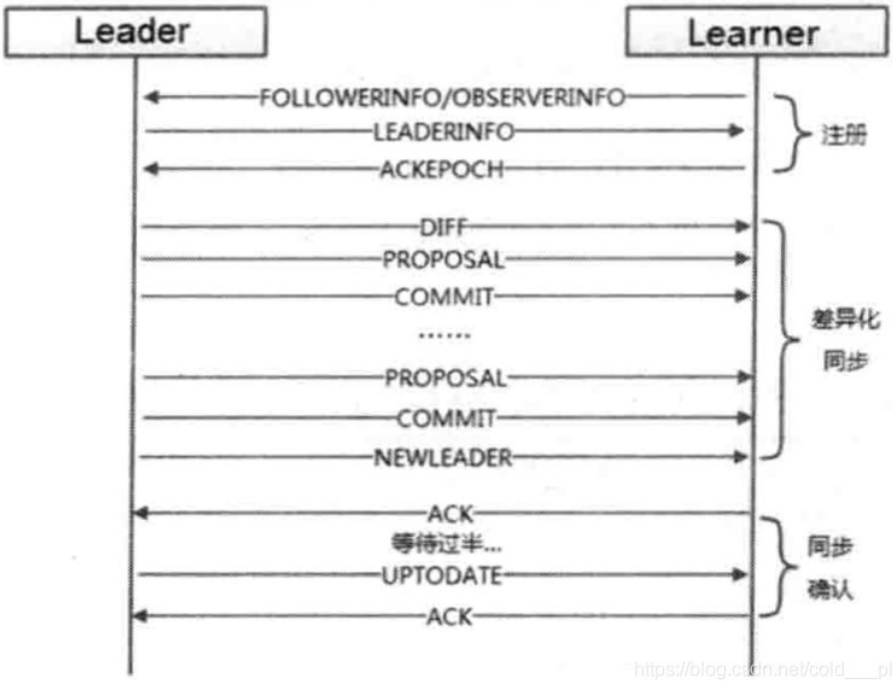
Leader 会和其他 Learner 服务器进行上述同样的数据同步流程，直到集群中有过半的 Learner 机器响应了 Leader 这个 ACK 消息。

3) UPTODATE 指令：

一旦满足“过半策略”后，Leader 服务器就会向所有已经完成数据同步的 Learner 发送一个 UPTODATE 指令，用来通知 Learner 已经完成了数据同步，

同时集群中已经有过半机器完成了数据同步，集群已经具备了对外服务的能力了。

Learner 在接收到这个来自 Leader 的 UPTODATE 指令后，会终止数据同步流程，然后向 Leader 再次反馈一个 ACK 消息。



先回滚再差异化同步(TRUNC+DIFF 同步)

minCommittedLog < peerlastZxid <maxCommittedLog

1) DIFF 指令：Leader 服务器会首先向这个 Learner 发送一个 DIFF 指令，用于通知 Learner 进入差异化数据同步阶段 Leader 服务器即将把一些 Proposal 同步给自己。

2) 数据包：（直接差异化同步场景中，会有一个罕见但是确实存在的特殊场景，发送数据包这一步后会有所不同）

例子：设有 A、B、C 三台机器，假如某一时刻 B 是 Leader 服务器，此时的 LeaderEpoch 为 5，同时 Leader 当前已经被集群中绝大部分机器都提交的 ZXID 包括：0x500000001 和 0x500000002。

此时，Leader 正要处理 ZXID：0x500000003，并且已经将该事务写人到了 Leader 本地的事务日志中去。就在 Leader 恰好要将该 Proposal 发送给其他 Follower 机器进行投票的时候，Leader 服务器挂了 Proposal 没有被同步出去。

此时 ZooKeeper 集群会进行新一轮的 Leader 选举，假设此次选举产生的新的 Leader 是 A，同时 LeaderEpoch 变更为 6，B 成为了 Follower

之后 A 和 C 两台服务器继续对外进行服务，又提交了 0x600000001 和 0x600000002 两个事务。【peerLastZxid=0x500000003 minCommittedLog=0x500000001 maxCommittedLog=0x600000002】

此时，服务器 B 再次启动，并开始数据同步。

对于这个特殊场景,就使用先回滚再差异化同步(TRUNC+DIFF 同步)的方式。当 Leader 服务器发现某个 Learner 包含了一条自己没有的事务记录，那么就需要让该 Learner 进行事务回滚

回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID。

在上面这个例子中，Leader 会需要 Learner 回滚到 ZXID 为 0x500000002 的事务记录。

先回滚再差异化同步的数据同步方式在具体实现上和差异化同步是一样的，都是会将差，异化的 Proposal 发送给 Learner。同步过程中的数据包发送顺序如下表所示。

发送顺序	数据包
1	TRUNC
2	PROPOSAL
3	COMMIT
4	PROPOSAL
5	COMMIT

仅回滚同步(TRUNC 同步)

peerlastZxid > maxCommittedLog

这种场景其实就是上述先回滚再差异化同步的简化模式，Leader 会要求 Learner 回滚到 ZXID 值为 maxCommitedLog 对应的事务操作。

全量同步(SNAP 同步)

场景 1: peerLastZxid 小于 minCommittedLog。

场景 2: Leader 服务器上没有提议缓存队列，peerLastZxid 不等于 lastProcessedZxid（Leader 服务器数据恢复后得到的最大 ZXID）

上述这两个场景非常类似，在这两种场景下，Leader 服务器都无法直接使用提议缓存队列和 Learner 进行数据同步，因此只能进行全量同步(SNAP 同步)。

所谓全量同步就是 Leader 服务器将本机上的全量内存数据都同步给 Learner。Leader 服务器首先向 Learner 发送一个 SNAP 指令,通知 Learner 即将进行全量数据同步。

后,Leader 会从内存数据库中获取到全量的数据节点和会话超时时间记录器,将它们序列化后传输给 Learner。

Learner 服务器接收到该全量数据后，会对其反序列化后载入到内存数据库中。

Zab 数据同步过程中，如何处理需要丢弃的 Proposal：

每当选举产生一个新的 Leader ，就会从这个 Leader 服务器上取出本地事务日志中最大编号 Proposal 的 zxid

并从 zxid 中解析得到对应的 epoch 编号，然后再对其加 1，之后该编号就作为新的 epoch 值并将低 32 位数字归零，由 0 开始重新生成 zxid。

Zab 协议通过 epoch 编号来区分 Leader 变化周期，能够有效避免不同的 Leader 错误的使用了相同的 zxid 编号提取了不一样的 Proposal 的异常情况。

消息广播

消息广播：在 zookeeper 集群中，数据副本的传递策略就是采用消息广播模式。只要半数以上的 Follower 成功反馈即可

广播流程：

1）客户端发起一个写操作请求。

2) Leader 服务器将客户端的请求转化为事务 Proposal 提案, 同时为每个 Proposal 分配一个全局的 ID，即 zxid。

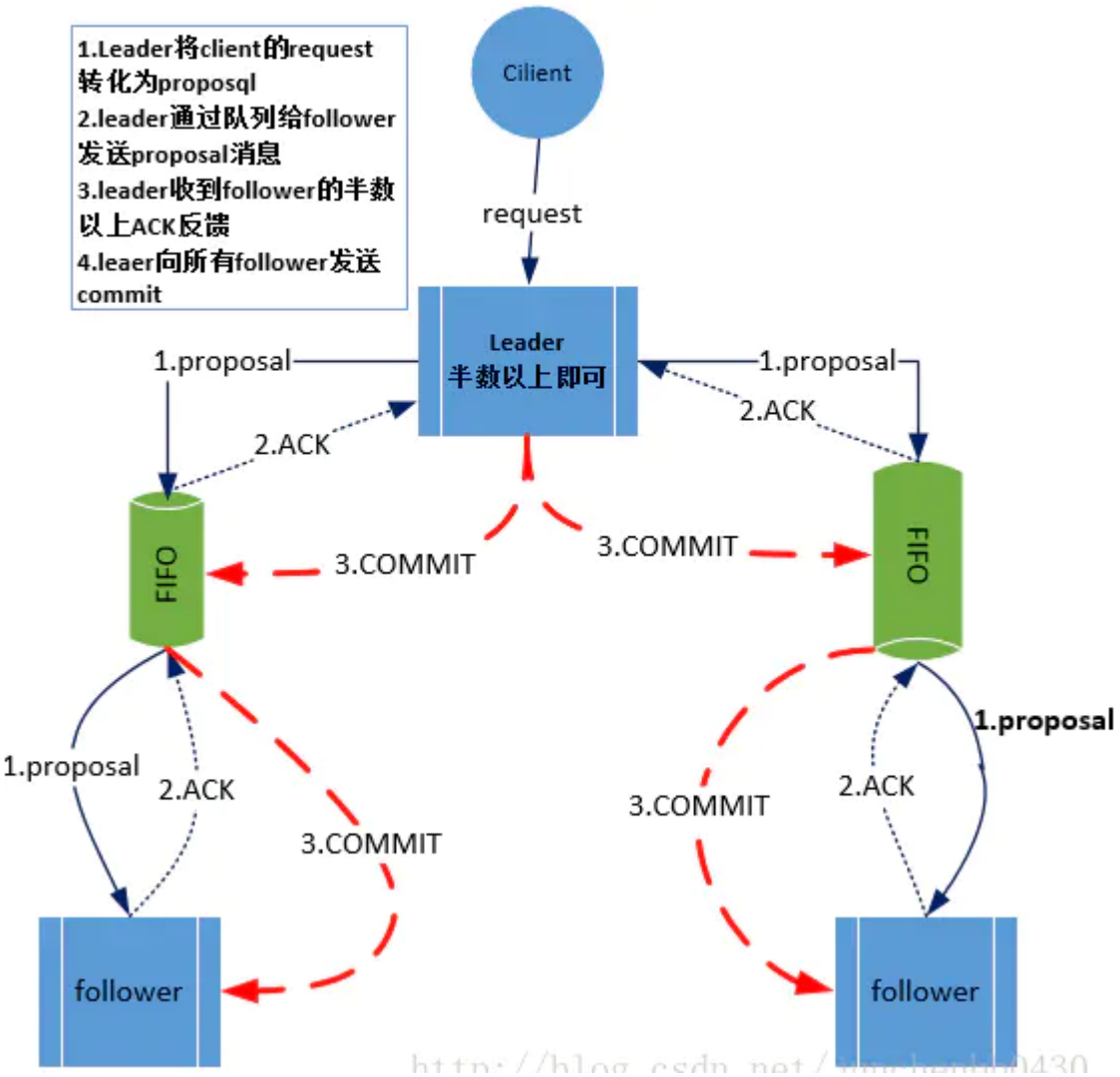
Leader 服务器为每个 Follower 服务器分配一个单独的队列, 然后将需要广播的 Proposal 依次放到队列中取，并且根据 FIFO 策略进行消息发送。

3）Follower 接收到 Proposal 后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向 Leader 反馈一个 Ack 响应消息。

4）Leader 接收到超过半数以上 Follower 的 Ack 响应消息后，即认为消息发送成功，可以发送 commit 消息。

Leader 向所有 Follower 广播 commit 消息，同时自身也会完成事务提交。

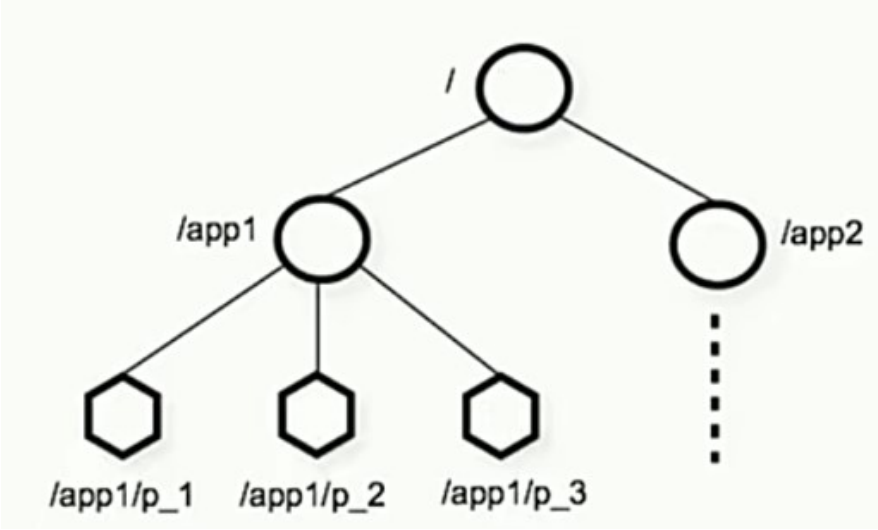
5）Follower 接收到 commit 消息后，会将上一条事务提交。

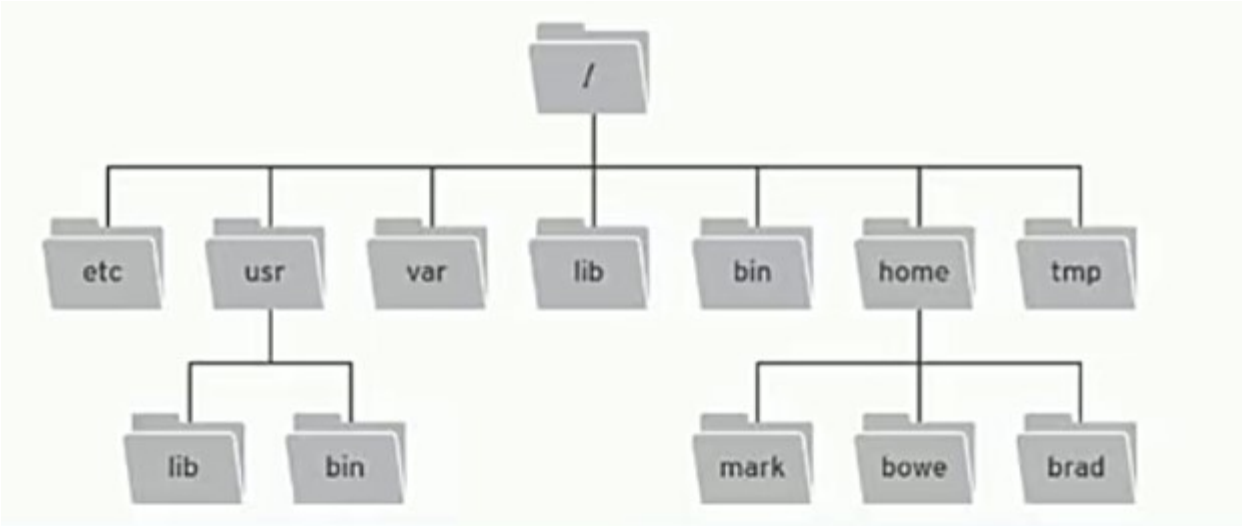


介绍

概念：Zookeeper 属于 Apache Hadoop 项目下的一个子项目，是一个树形目录服务。

节点：每个 ZNode 节点上都会保存自己的数据和节点信息，节点可以拥有子节点，同时也允许少量(1MB）数据存储在该节点之下。



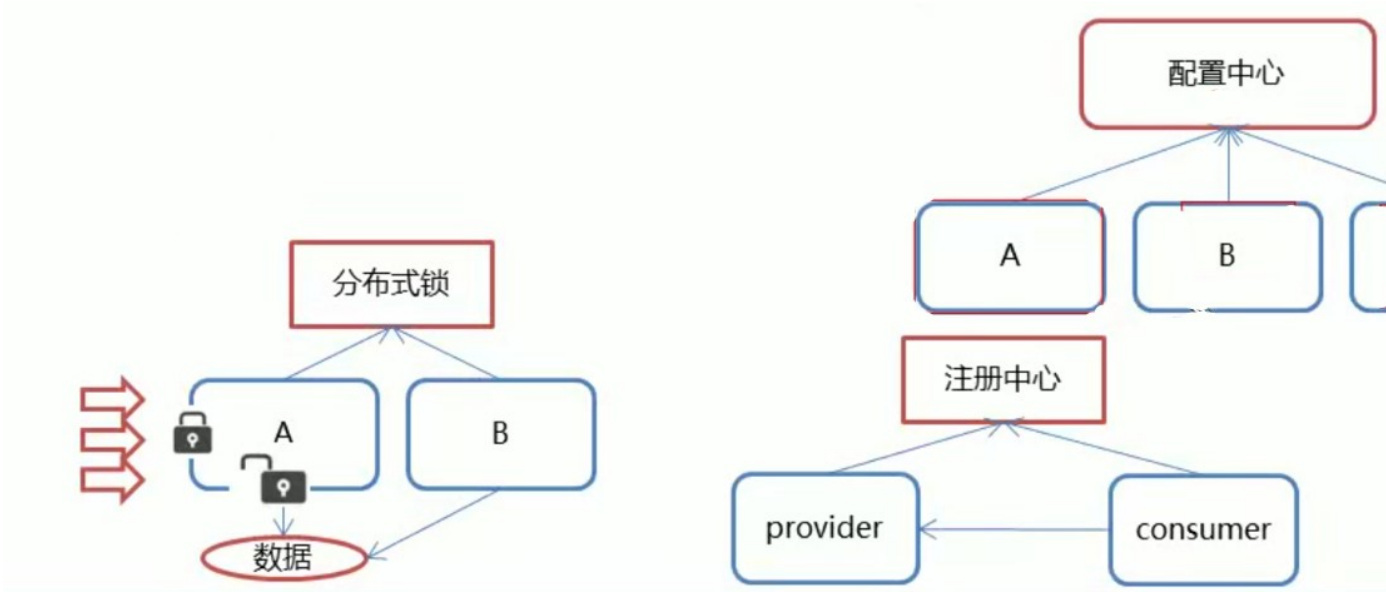


主要功能:

配置中心

注册中心

分布式锁



注册中心原理

Zookeeper 可以充当一个服务注册表(Service Registry)，让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址(IP+端口)去访问具体的服务提供者

Watch 事件监听

概念：ZooKeeper 允许用户在指定节点上注册一些 Watcher 来实现发布/订阅功能，能够让多个订阅者同时监听某一个对象，当一个对象自身状态变化时，会通知所有订阅者。(该机制是 ZooKeeper 实现分布式协调服务的重要特性)

发送消息：一次监听事件触发时，一个通知会被发送到 client，但是 client 只会收到一次这样的信息。

注册监听器： getData、exists、 getChildren

消息问题：通知事件从服务器发送给客户端是异步的，当不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素可能导致客户端在不通的时刻监听到事件

在客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。

Zookeeper 只能保证最终的一致性，而无法保证强一致性。

断开连接：当客户端与一个服务器失去连接的时候，是无法接收到事件通知的。

而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。

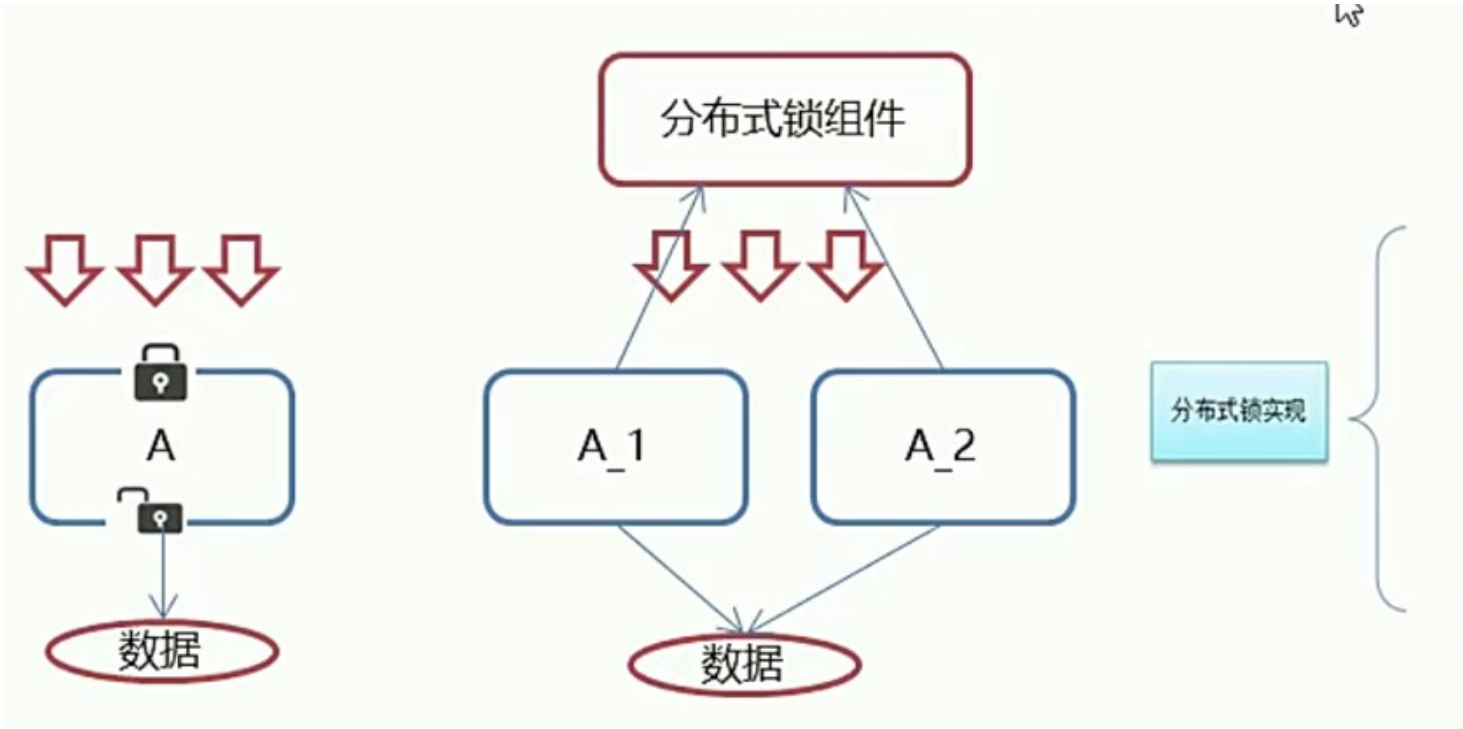
只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

ZooKeeper 原生支持通过注册 Watcher 来进行事件监听，需要反复注册 Watcher 比较繁琐，Curator 引入了 Cache 来实现对 ZooKeeper 服务端事件的监听。

NodeCache	监听一个节点
PathChildrenCache	监控一个节点的所有子节点.
TreeCache	可以监控整个树上的所有节点，类似于 PathChildrenCache 和 NodeCache 的组合

分布式锁

在分布式集群多 JVM 的工作环境下，跨 JVM 之间无法通过多线程的锁解决同步问题，分布式锁就是用来解决这种跨机器的进程之间的数据同步问题。



核心思想：当客户端要获取锁，则创建节点，使用完锁，则删除该节点。

临时节点方式：客户端获取锁时，创建一个无序临时节点 Lock，其他客户端只需要监听 lock 就行了

但是当临时节点被删除的时候，其余客户端会竞争创建节点，如果存在一千个的客户端，这时只有一个客户端能创建成功，却要让一千个客户端来竞争，同时浪费了一千个客户端的资源。这就叫做惊群现象。

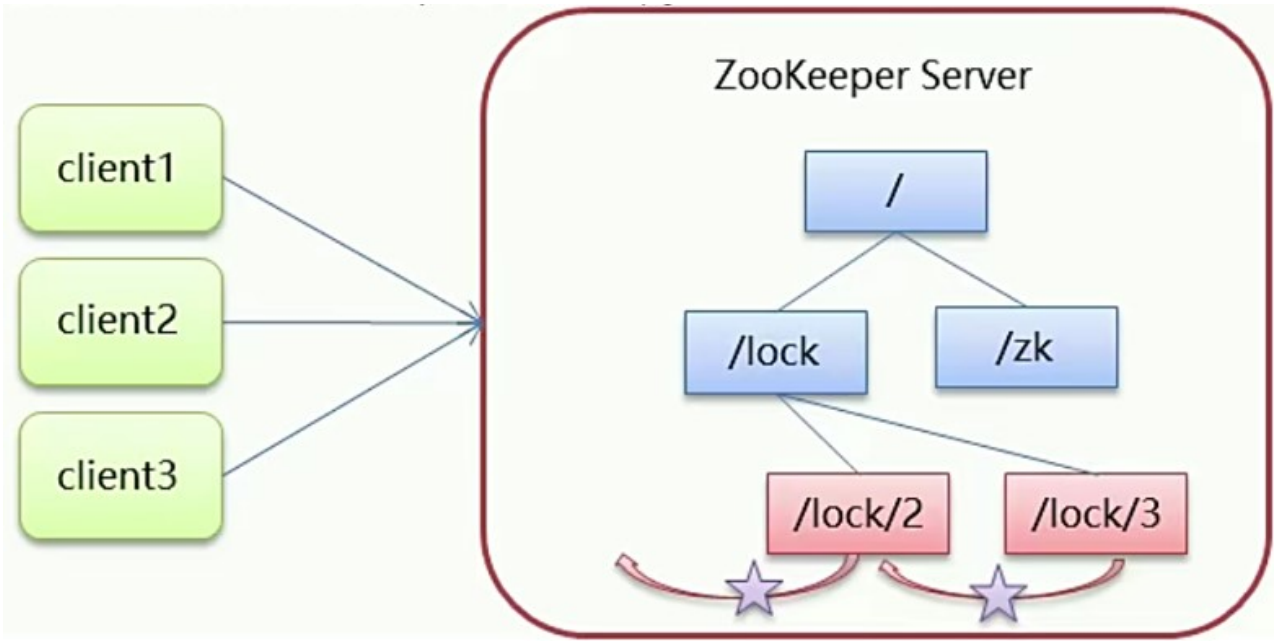
顺序节点方式：

客户端获取锁时，在 lock 节点下创建临时顺序节点。

其他客户端尝试获取锁时，也会在 lock 节点下创建一个临时顺序节点，再会获取 lock 下面的所有子节点，如果发现自己创建的子节点序号最小，那么就认为该客户端获取到了锁。使用完锁后，将该节点删除。

如果发现自己创建的节点并非 lock 所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，同时对其注册事件监听器，监听删除事件。

如果发现比自己小的那个节点被删除，则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 lock 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。（只会惊动一个节点）



curator 锁方案

InterProcessSemaphoreMutex	分布式非可重入排它锁
InterProcessMutex	分布式可重入排它锁
InterProcessSemaphoreV2	共享信号量（令牌桶算法）
InterProcessReadWriteLock	分布式读写锁
InterProcessMultiLock	联锁。对多个锁进行一组操作。当 acquire 的时候就获得多个锁资源，否则失败。

DAM / Mycat

介绍

MyCat 是一款数据库集群软件，不仅支持 MySQL，常用关系型数据库也都支持。

其实就是一个数据库中间件产品，支持 MySQL 集群，提供高可用性数据分片集群。

分库分表

分库分表：将庞大的数据量拆分为不同的数据库和数据表进行存储。

水平拆分

将同一表中的所有数据拆分到多台数据库服务器上，也叫做横向拆分。

例如：一张 1000 万的大表,按照一模一样的结构，拆分成 4 个 250 万的小表，分别保存到 4 个数据库中。

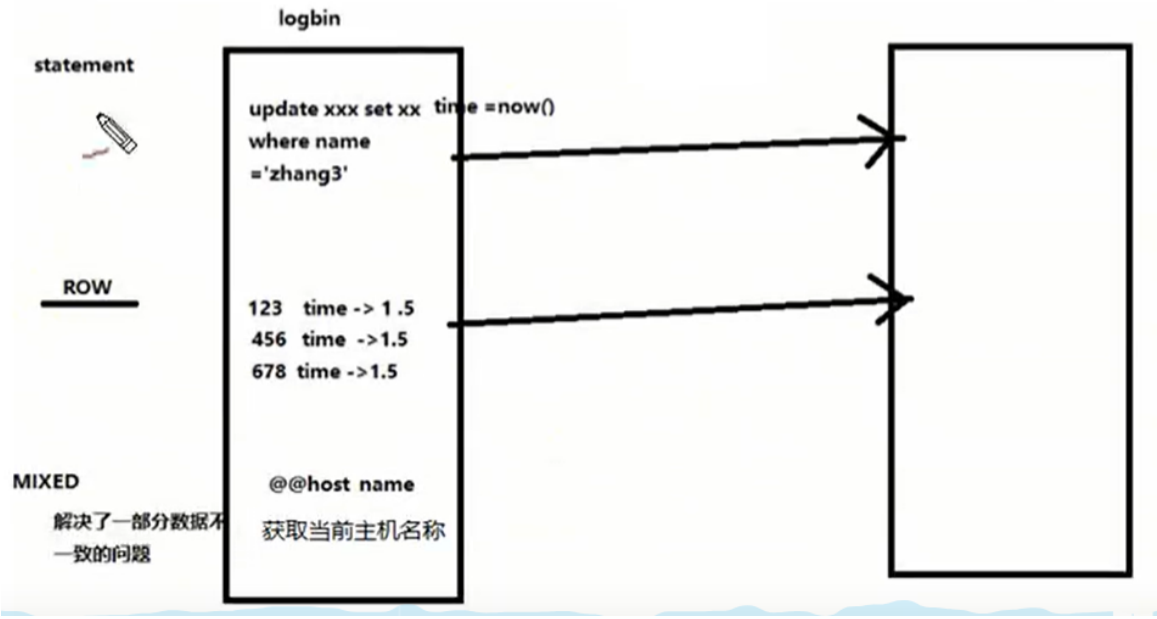
垂直拆分

将不同业务类型的表拆分到不同的数据库上，也叫做纵向拆分。

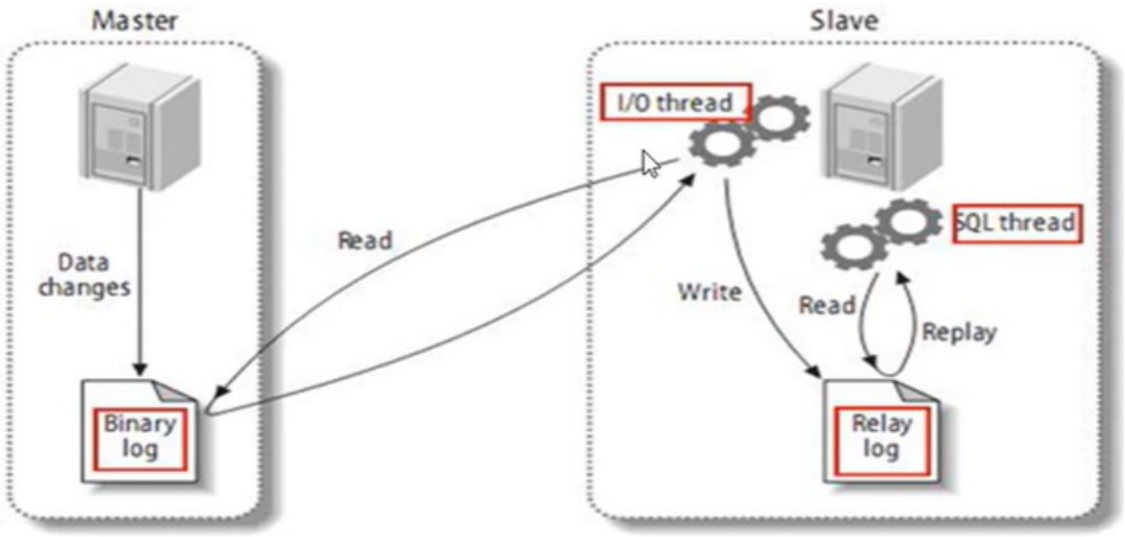
例如：所有的动物表都保存到动物库中，所有的水果表都保存到水果库中，同类型的表保存在同一个库中。

分片规则：？？？？？

binlog 格式：



主从复制原理：



linux 安装

下载安装包　　wget http://dl.mycat.io/1.6-RELEASE/Mycat-server-1.6-RELEASE-20161028204710-linux.tar.gz

（ https://github.com/MyCATApache/Mycat-download ）

mycat console　　控制台启动（默认端口号：8066）

mycat start　　启动命令

mycat stop　　停止命令

mycat restart　　重启命令

change master to master_host='192.168.59.143', master_port=3306, master_user='root', master_password='xxx', master_log_file='mysql-bin.000001', master_log_pos=154;

start slave;　开启从节点

show slave status\G;　　查询结果

拆分配置

水平拆分配置

mycat/conf/server.xml >>　根据 id 的余数拆分存入数据库

<property name="sequenceHandlerType">0</property>

mycat/conf/sequence_conf.properties >>

```
GLOBAL.HISIDS=xxx                    #自定义关键字
GLOBAL.MINID=10000          #最小值
GLOBAL.MAXID=20000          #最大值
```

mycat/conf/schema.xml >>

```
    <schema name="SDKDB" checkSQLschema="false" sqlMaxLimit="100" >
          <table name="product" primaryKey="id" dataNode="dn1,dn2,dn3" rul3="mod-long">
    </schema>

    <dataNode name="dn1" dataHost="host1" database="db1" />
    <dataNode name="dn2" dataHost="host1" database="db2" />
    <dataNode name="dn3" dataHost="host1" database="db3" />

    <dataHost name="host1" maxCon="1000" minCon="10" balance="1"   writeType="0" dbType="mysql"
switchType="1" slaveThreshold="100" dbDriver="native">         数据主机
          <heartbeat>select user()</heartbeat>                    心跳检测
          <writeHost host="hostM1" url="localhost:3306" user="root"    password="123456">          主服务器负责写的操作
               <readHost host="hostS1" url="192.168.203.149:3306" user="root" password="123456" />      从服务器负责读
的操作
          </writeHost>
    </dataHost>
```

mycat/conf/rule.xml >>

```
<function name="mod-long" class="io.mycat.route.functionPartitionByMod">
    <property name="count">3<property>
</function>
```

分库分表后，insert 操作时必须指定列名，主键也不能省略为 null 了

水平拆分配置

mycat/conf/schema.xml >>

```
    <schema name="SDKDB" checkSQLschema="false" sqlMaxLimit="100" >
          <table name="dog" primaryKey="id" autoIncrement="true" dataNode="dn4">
          <table name="cat" primaryKey="id" autoIncrement="true" dataNode="dn4">

          <table name="apple" primaryKey="id" autoIncrement="true" dataNode="dn5">
          <table name="banana" primaryKey="id" autoIncrement="true" dataNode="dn5">

    </schema>

    <dataNode name="dn4" dataHost="host1" database="db4" />
    <dataNode name="dn5" dataHost="host1" database="db5" />

    <dataHost name="host1" maxCon="1000" minCon="10" balance="1"   writeType="0" dbType="mysql"
switchType="1" slaveThreshold="100" dbDriver="native">         数据主机
          <heartbeat>select user()</heartbeat>                    心跳检测
          <writeHost host="hostM1" url="localhost:3306" user="root"    password="123456">          主服务器负责写的操作
               <readHost host="hostS1" url="192.168.203.149:3306" user="root" password="123456" />      从服务器负责读
的操作
          </writeHost>
```

```
    </dataHost>
```

读写分离配置

mycat/conf/server.xml >>　配置连接 mycat 的用户，系统相关变量，端口

```
    <user name="test">
        <property name="password">test</property>          root 用户
        <property name="schemas">SDKDB</property>         逻辑库名为 TESTDB
    </user>

    <user name="user">
        <property name="password">user</property>          读取的用户
        <property name="schemas">SDKDB</property>         逻辑库名为 TESTDB
        <property name="readOnly">true</property>
    </user>
```

mycat/conf/schema.xml >>

```
    <schema name="SDKDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1">    逻辑库 添加数据节点
    </schema>
    <dataNode name="dn1" dataHost="host1" database="db1" />
    <dataHost name="host1" maxCon="1000" minCon="10" balance="1"   writeType="0" dbType="mysql"
switchType="1" slaveThreshold="100" dbDriver="native">        数据主机
        <heartbeat>select user()</heartbeat>                  心跳检测
        <writeHost host="hostM1" url="localhost:3306" user="root"    password="123456">        主服务器负责写的操作
            <readHost host="hostS1" url="192.168.203.149:3306" user="root" password="123456" />      从服务器负责读
的操作
        </writeHost>
    </dataHost>
```

连接 mycat 逻辑库：　　mysql -uroot -pqew3dqr4wmymycat　-h localhost -P 8066

mysql 配置

my.ini >>　修改主机

```
server-id =1                        #主服务器唯一 id
log-bin=mysql-bin
innodb_flush_log_at_trx_commit=1
sync_binlog=1

binlog-do-db=testdb                 #需要复制的主数据库名
binlog_format=STATEMENT             #binlog 日志格式
binlog-ignore-db=mysql              #设置不要复制的数据库
binlog-ignore-db=information-schema
```

DAM / Activemq

介绍

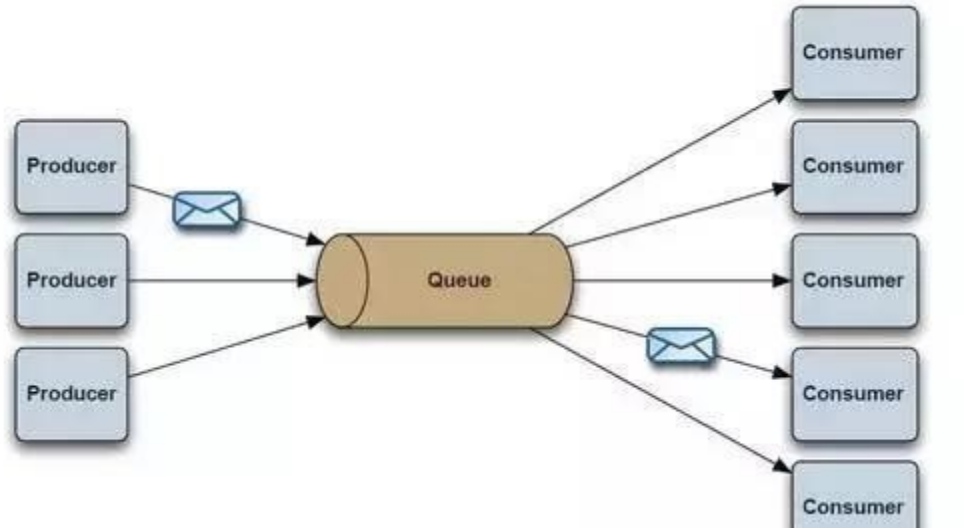
JMS 概念：

- Provider/MessageProvider：　生产者
- Consumer/MessageConsumer：　消费者
- PTP：　Point To Point，点对点通信消息模型
- Pub/Sub：　Publish/Subscribe，发布订阅消息模型
- Queue：　队列，目标类型之一，和 PTP 结合
- Topic：　主题，目标类型之一，和 Pub/Sub 结合
- ConnectionFactory：　连接工厂，JMS 用它创建连接

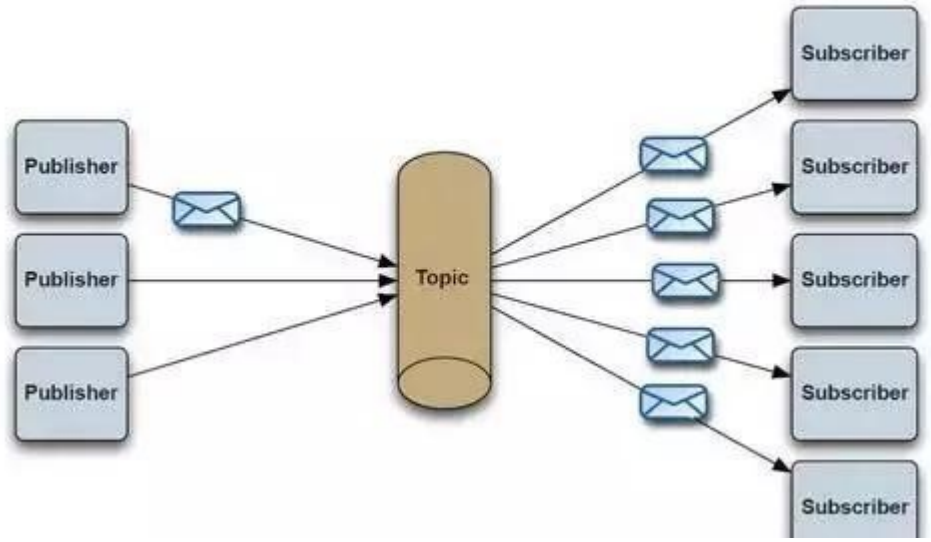
Connnection：　　JMS Client 到 JMS Provider 的连接
Destination：　　消息目的地，由 Session 创建
Session：　　会话，由 Connection 创建，实质上就是发送、接受消息的一个线程，因此生产者、消费者都是 Session 创建的

工作模式

PTP　点对点，基于队列



Pub/Sub　发布订阅，基于主题



bin 命令

./activemq start　　启动
./activemq stop　　停止
./activemq dstat　　显示默认 broker 的所有主题和队列统计信息
./activemq dstat topics　　显示的主题统计信息
./activemq dstat queue　　显示队列的统计信息
dtsat　　就是显示图形化中 query 的图。
Queue Size　　表示 queue 的大小。 Producer 表示生产者
Consumer 消费者

Enqueue 一共进入了多少队列
Dequeue 一共消费的多少队列

linux 安装

官网下载安装包：http://activemq.apache.org/components/classic/download/

tar -zxvf apache-activemq-5.16.0-bin.tar.gz

conf/jetty.xml >>

```
<bean id="jettyPort" class="org.apache.activemq.web.WebConsolePort" init-method="start">
    <property name="host" value="0.0.0.0"/>            <!-- 修改管理后台远程登录地址 -->
    <property name="port" value="48123"/>              <!-- 修改管理后台访问端口 -->
</bean>
 <bean id="securityConstraint" class="org.eclipse.jetty.util.security.Constraint">
     <property name="name" value="BASIC" />
     <property name="roles" value="user,admin" />     <!-- 定义用户组 -->
     <!-- set authenticate=false to disable login -->
     <property name="authenticate" value="true" />
 </bean>
```

conf/jetty-realm.properties >>

admin: admin, admin　　#用户名 ：密码 ,角色名

conf/activemq.xml >>

```
<transportConnectors>
    <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100MB -->
    <transportConnector name="openwire" uri="tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>  <!-- 修改 tcp 地址 -->
    <transportConnector name="amqp" uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>  <!-- 修改 amqp 地址，防止与 rabbitmq 冲突 -->
    <transportConnector name="stomp" uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>
    <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>
    <transportConnector name="ws" uri="ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>
</transportConnectors>
<plugins>                                    <!-- 添加密码 -->
    <simpleAuthenticationPlugin>
        <users>
            <authenticationUser username="admin" password="admin1231" groups="users,admins"/>
        </users>
    </simpleAuthenticationPlugin>
</plugins>
```

DAM / Hana

Core

Configuration

Download and Install the SAP HANA Client

https://help.sap.com/docs/hana-cloud/sap-hana-cloud-getting-started-guide/download-and-install-sap-hana-client

SAP HANA 2.0 Express Edition, HANA Studio Full Installation with 1 Year Free Registration

https://www.youtube.com/watch?v=bgKRBDUpFDM&t=126s

SAP Hana Express

http://www.datadisk.co.uk/html_docs/hana/hana_2.html

https://www.sap.com/products/technology-platform/hana/express-trial.html

HDB info

This should show the status of the HANA instance (e.g., whether it is running or not).

HDB start

ip a

Commands

Hana Terminal

HDB info

HDB stop

Stop HANA services

HDB start

Start HANA services