

In-memory Database

- **What is In-memory database?**

- In-memory database is a database management system that stores data in main memory (RAM) in order to achieve faster response time.

- **Advantages over traditional databases:**

- Faster
- Require fewer CPU instructions
- Eliminates seek time
- Different parts of the database can be managed through direct pointers

- **Drawbacks:**

- With just an In-memory database all the data will be lost if the power goes off / system crashes as main memory is volatile. So durability is a concern.

- **Where to use In-memory database?**

- Real time data analysis
- Embedded system
- Mobile advertising
- Games, live leaderboards etc.
- Caching
- Any application where response time is critical
- Testing a prototype with temporary data
- Chat messaging

- **In-memory database with persistence:**

- An In-memory database achieves durability with the help of **transaction logs**. A transaction log is stored in the hard disk. But it is not a bottleneck for the system because transaction logs are stored in a sequential / append only manner. Hard disk is much faster in sequential access than random access and an In-memory database only uses it when a write operation is made. This transaction log is used to recover data. Transaction logs are compacted using **snapshotting**. Snapshotting is dumping the whole database from main memory to disk once in a while. Once a snapshot has been taken all the unnecessary transaction logs can be erased. This helps to make the recovery process faster.


- **Some popular In-memory database:**

- **Redis:** "Remote Dictionary Server" - Redis is a durable key-value in-memory database. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams and spatial indices. It has a sub millisecond latency. It is durable, supporting snapshots and backups(RDB snapshot/AOF log). It can be used as cash as well as a primary datastore.
- **Memcached:** Memcached is a general purpose key-value memory caching system. It is multi-threaded. It does not have any data types. It is volatile. It can be easily scaled

vertically by providing it with more cores and memory. When memory is full it searches for a suitable expired entry and replaces this with a new entry.



- **Aerospike**
- **Apache ignite**
- **Hazelcast**
- **HSQL**
- **H2**
- **SQLite** (To force SQLite to store data in main memory, we need to use “::memory::”

- **Resources:**

- [In-memory database](#)
- [What is an In-Memory Database? Definition and FAQs](#)
- [What Is an In-Memory Database?](#)
- [What an in-memory database is and how it persists data efficiently](#)
- [List of in-memory databases](#)
- [In-Memory Databases](#)
- [Redis](#)
-  [Redis in 100 Seconds](#)
- [Redis: in-memory data store. How it works and why you should use it](#)
- [Redis vs Memcached: which one to pick?](#)
- [Memcached | Distributed Key-Value Store | AWS](#)

Redis

Resources:

- The Little Redis Book by Karl Seguin ([Untitled](#))
-  Redis Crash Course
-  Redis Course - In-Memory Database Tutorial
- [Redis in Action - Foreword](#)
- [Redis: in-memory data store. How it works and why you should use it](#)
- [An introduction to Redis data types and abstractions – Redis](#)

What is Redis?

- Redis is an open source in-memory, persistent, key-value database management system.

Why use Redis?

- *In-memory database*: it is much much faster than traditional databases.
- *Data persistence*: Though store data in memory, it persists data through snapshotting and transaction log.
- *Key-value structure*: It is a NoSQL database. Store data in key-value manner.
- *Data structure*: It exposes a set of data structures (e.g. string, hash, set).
- *Scalability*

Where to use Redis?

We can use redis as a primary database but it is mainly used for:

- Caching
- Real time data analytics
- Session store
- Chat messaging

In summary, wherever we need fast data access with persistence.

Redis-cli:

Redis-cli is the redis command line interface.

Few Basic Commands:

- **PING** To check whether redis server is running or not
- **DEL** To delete a key
- **EXPIRE** To set the expiry of a key after a specified time in seconds.
- **PERSIST** Removes expiry from a key
- **EXISTS** To check whether a key exists or not
- **RENAME** To rename a key
- **TYPE** To check the type of a key
- **FLUSHALL** To clear the database
- **TTL** Remaining time to live for a key

"STRING"

- **SET** key value Set the value of the 'key' to 'value'
- **GET** key Get the value of 'key'
- **STRLEN** key Get length of the 'value' of the 'key'
- **GETRANGE** key start end Get the value in range start to end
- **APPEND** key value Appends 'value' to the existing value of the 'key'
- **INCR** key Increment key value. Similarly, **INCRBY**, **DECR**, **DECRBY**.
- **MSET** key1 value1 key2 value2... Set multiple key-value
- **MGET** key1 key2 key3 Get values of multiple keys

"HASH"

- **HSET** key field value Set the field 'field' to value 'value' of the key 'key'
- **HGET** key field Get the value of the field 'field' of the key 'key'
- **HMSET** key field value field value... Set multiple field value of a key
- **HMGET** key field field.... Get multiple field value of a key
- **HGETALL** key Get all field - value of a key
- **HKEYS** key Get all the field names of a key
- **HDEL** key field Delete specified field

"LIST"

- **LPUSH** key value Push the 'value' to the left of the list 'key'
- **RPUSH** key value Push the 'value' to the right of the list 'key'
- **LPOP** key Pop the leftmost element from the list 'key'
- **RPOP** key Pop the rightmost element from the list 'key'
- **LRANGE** key start end Returns the sublist from start to end (index can be negative)
- **LTRIM** key start end Removes all the elements not in start to end
- **LLEN** key Returns the length of the list 'key'

"SET"

- **SADD** key value1 value2... Adds elements to the set 'key'
- **SMEMBERS** key Returns all elements of a set in any order
- **SISMEMBER** key value Check if 'value' is a member of set 'key'
- **SRANDMEMBER** key Returns a random element from the set 'key'
- **SCARD** key Returns total number of element in the set 'key'
- **SPOP** key Removes and returns a random element from the set 'key'
- **SINTER** key1 key2... Returns intersections of multiple sets
- **SUNION** key1 key2... Returns union of multiple sets

"SORTED SET"

(Values are sorted using score, if score is same then lexicographically)

- **ZADD** key score value Add an item 'value' with score 'score' in a sorted set 'key'
- **ZRANGE** key start end Returns a list with item in 'key' from start to end
- **ZREVRANGE** key start end Same as ZRANGE but in reverse order (Descending order)
- **ZRANK** key value Position or rank of 'value' in sorted order.
- **ZREVRANK** key value Position or rank of 'value' in reverse sorted order.
- **ZRANGEBYSCORE** key val1 val2 Returns list of sorted element having score in val1 and val2

Other supported data structures: Bitmaps, HyperLogLogs.

Redis Stream

What is Redis Stream?

- A redis stream is a new data structure introduced in redis 5.0 for managing data channels between producers and consumers.
- It models log data structure in a more abstract way. Conceptually stream is an append only data structure, where each entry has an unique id and value.

Why Redis Stream?

- Each entry in the stream has a unique id and it is guaranteed to be in increasing order.
- Velocity in which data is being produced or consumed does not affect the performance.
- Supports multiple consumers
- Supports multiple producers
- Consumers can wait until new data comes in.
- Data is safe when a consumer fails to consume data.
- Persist data when consumers are disconnected.

Adding Data to a Stream:

XADD stream_name * key1 value1 key2 value2....

Here '*' is used to let the redis generate an ID for us. We could specify our own ID (not recommended). This command adds {key1: value1, key2: value2...} entry to stream stream_name and returns the ID generated for it.

Some Notes on Entry ID:

Format: <millisecondsTime>-<sequenceNumber>

If two entries appended to the stream in the same time or current millisecondsTime is less than previous then redis simply uses previous millisecondsTime and increases the <sequenceNumber>. This is how stream ensures monotonically increasing ID for new entries.

Querying Stream by Range:

- **XRANGE:**
 - **XRANGE** stream_name - +
=> '-' is used to mean lowest ID possible, '+' is used to mean highest ID possible
=> Returns all entry available in stream_name
 - **XRANGE** stream_name start_id end_id
=> Returns entries starting from start_id to end_id
 - **XRANGE** stream_name start_id end_id COUNT 5
=> Returns first five entries starting from start_id to end_id
- **XREVRANGE:**
 - Same as XRANGE. start_id and end_id needs to be specified in reverse order.

Reading item from streams:

XREAD:

- `XREAD STREAMS stream_name ID`
Here, XREAD reads from streams specified after STREAMS, all the data having an ID greater than the specified ID in the command.
- `XREAD STREAMS mystream 0`
Here 0 at the end means ID 0-0. So it will read every data added to the stream mystream.
- `XREAD COUNT 5 STREAMS mystream 0`
Here with the extra count option we are reading the first five data from the stream mystream.
- `XREAD COUNT 2 STREAMS mystream $`
Here a special symbol '\$' is assigned in place of ID. It means new data that came after this command was executed.
- `XREAD BLOCK 2000 COUNT 2 STREAMS mystream $`
Here we are using blocking, what it does is it waits for 2000 milliseconds for new data. After two second it will return null if no new data comes.
- `XREAD BLOCK 0 COUNT 2 STREAMS mystream $`
Using 0 after block, this command will wait indefinitely for new data.

Working with single consumer:

While working with a single consumer, say, the last data that it consumed has ID **last_ID**. Now if it somehow crashes or stops working, it needs to start consuming data that came after the last_ID. If we do not store this last_ID, all the data that was produced while it was not working, will be lost.

- We can use a '**file**' to store this last_ID
- We can **cash** this last_ID in Redis for future usage.

Working with multiple consumers:

Consumer Group:

- Basic concept is we have multiple workers/consumers working on the same streams. We are calling them a consumer group. So first we need to create this group.

Creating A Consumer Group:

- `XGROUP CREATE stream_name group_name ID`
This command will create a consumer group with the name 'group_name' which will start consuming from ID.
- `XGROUP CREATE mystream consumers $`
This command will create a consumer group named consumers, which will start consuming all the new data from stream mystream.

Reading data using Consumer Group:

- `XREADGROUP GROUP mygroup worker1 STREAMS mystream >`

Here worker1 from mygroup is consuming data from mystream. Special sign '>' means that it will only consume data that was not provided to any other consumer from mygroup before. We can also use 'BLOCK' in this command like before.

- `XREADGROUP GROUP mygroup worker1 STREAMS mystream 0`


If we use anything other than '>', this command will only show us the data that were provided to worker1 but not yet been acknowledged. So a worker needs to acknowledge the data when it finishes processing it to the consumer group.

Acknowledging the data:

- `XACK mystream mygroup data_ID`

Whenever a consumer finishes processing data, it needs to run this command. Here data_ID is the ID of that data. If a worker fails to acknowledge a data or it crashes, that data will be considered as pending data and redis stream will try to assign that data to another consumer of that group. All the pending data and their information can be viewed using the 'XPENDING' command.

Resources:

-  [Redis Streams Featuring Salvatore Sanfilippo - Redis Labs](#)
- [Introduction to Redis Streams – Redis](#)
- [How to use Redis Streams](#)
- [Working with Redis Streams](#)

Practice code - 1: <https://github.com/hijibijee/Producer-Consumer-Redis-streams-Jedis-Java-practice> (Java)

Description:

- Used Jedis library to connect java to redis server. To use Jedis, I added maven dependency.
Details: [redis/jedis: A blazingly small and sane redis java client](#)
- To connect java to redis server, I ran redis server in terminal and created a Jedis object. That's it. By default it connects to the redis server at 127.0.0.1:6379.
- Producer.java adds curr_msg (key) "hello #{an incrementing number}" (value) to the stream called "messages" every second.
- Consumer_1.java starts consuming the first 10 messages produced by the producer after it starts consuming. Initial id is set to system time in millisec-0. 'lastID' variable contains the id of the last consumed operation and used in the next operation.
- Executing redis commands in java using Jedis was a bit challenging for me. I used below resources:
 - [Jedis \(Jedis 3.1.0-m3 API\)](#)
 - [JedisCommands \(Jedis 3.1.0-m3 API\)](#)
 - [jedis/StreamsCommandsTest.java at master · redis/jedis · GitHub](#)
 -  [Using the Jedis Force to Manage Streams](#)
 - <https://github.com/marcosnils/redistories>

Practice code - 2: <https://github.com/hijibijee/Redis-stream-Node-Js> (Node js)

Description: Detailed description will be found in the readme.md section of the github repository.