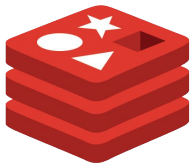


redislabs
HOME OF REDIS

EdgeX F2F
Redis Technical Session

November 2019 | André Srinivasan

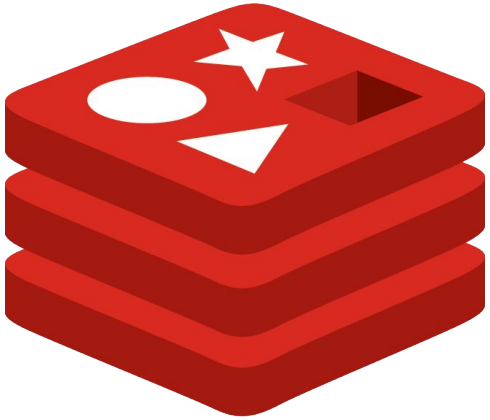
Agenda

- Redis Labs and Redis
- API
- Transactions
- Persistence
- TL;DR

Redis Labs and Redis

Redis Labs is the Home of Redis

Our Roots Are in Open Source



redis

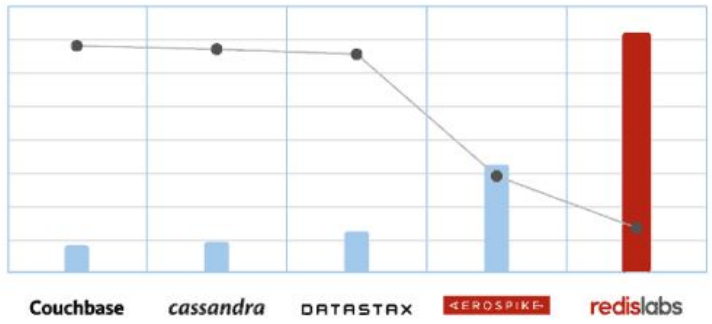
An **In-memory open source database**, supporting a variety high performance operational, analytics or hybrid use case

How Redis Labs Thinks About Redis

1

Performance

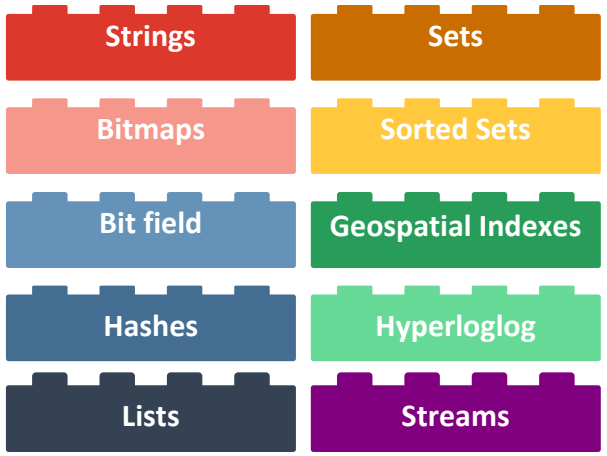
Database for the Instant Experience
(Low latency at High Throughput)



2

Simplicity

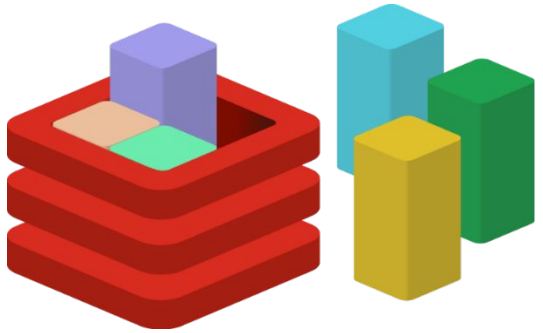
Redis Data Structures



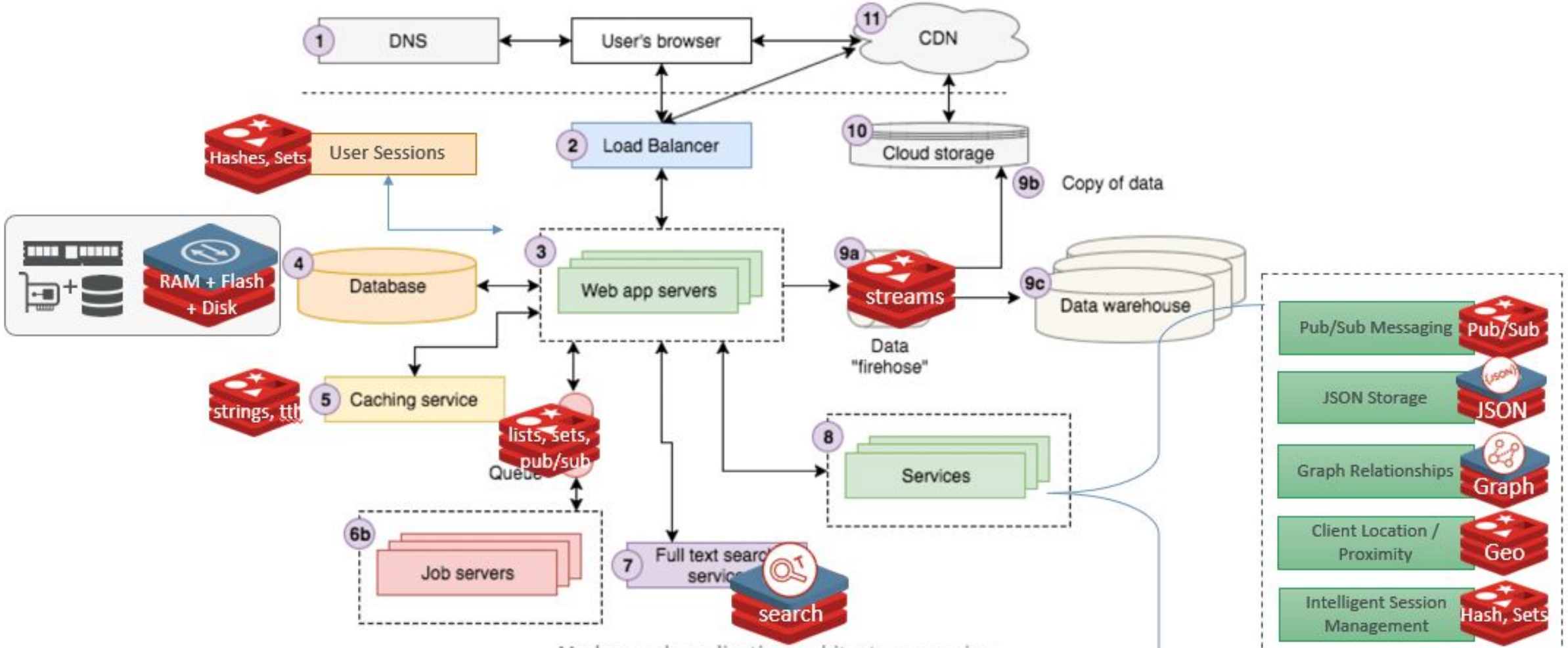
3

Extensibility

Redis Modules



Redis in Context



Modern web application architecture overview

Redis Database

- Redis is a **key**-value store; sometimes referred to as data-structure store
- Schema-less
- Keys must be unique (like primary keys in relational databases)
- The Redis language is basically CRUD

- Design Patterns
  - Common naming convention uses delimiters

    ```
    db.DeviceService + ":name"
    ```

  - Cryptic keys are hard to read
  - Wordy keys take up memory

Properties of Keys and Values

All Keys

- Up to 512MB in size, cAse SeNsitivE, and binary safe strings
- Can register to listen for changes
- Deleting the key deletes the value
  - DEL - blocking
  - UNLINK - non blocking

All Data Structures

- Can have expiry (TTL – Time-To-Live)
- Can be up to 512MB in size

API

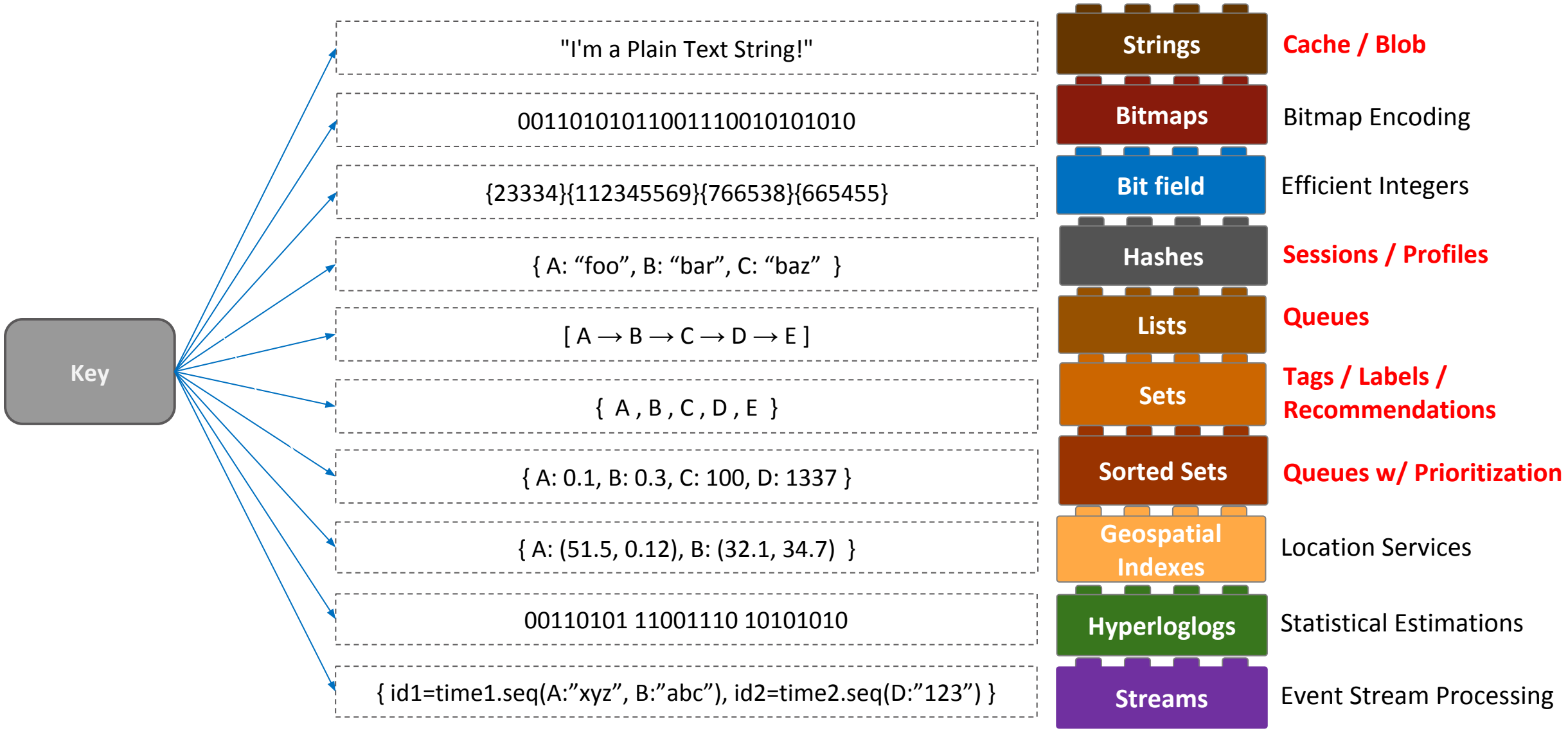
Redis CLI

- `redis-cli` is the Redis command line interface
  - sends commands to Redis
  - reads the replies sent by the server directly from the terminal

- It has two main modes
  - Interactive mode where there is a REPL (Read Eval Print Loop) where the user types commands and get replies
  - Command line mode – The command is sent as arguments of redis-cli, executed, and printed on the standard output

```
127.0.0.1:6379> PING
PONG
```

Data Structure Use Patterns



String Data Structure

- Foundational data type
- Binary safe
- Can store:
  - Strings, Byte Arrays
  - Numeric - Integers, Floats
  - Serialized Objects - JSON, CBOR, Images, HTML, Files, ...
- Simple GET and SET operations

String Data Structure

Basics

- GET / SET
- GETSET
- MGET/MSET
- SETNX

```
127.0.0.1:6379> SET hello world
OK
127.0.0.1:6379> GET hello
"world"
127.0.0.1:6379> GETSET hello class
"world"
127.0.0.1:6379> GET hello
"class"
127.0.0.1:6379> MSET first:name John last:name Smith
OK
127.0.0.1:6379> GET first:name
"John"
127.0.0.1:6379> GET last:name
"Smith"
127.0.0.1:6379> SETNX last:name Doe
(integer) 0
127.0.0.1:6379> GET last:name
"Smith"
127.0.0.1:6379> MGET first:name last:name
1) "John"
2) "Smith"
```

String Data Structure

SET w/ Expire

- **SETEX (sec)**
- **PSETEX (ms)**

```
127.0.0.1:6379> SETEX exp:test 1 "value"
OK
127.0.0.1:6379> GET exp:test
(nil)
127.0.0.1:6379> SETEX exp:test 10 "value"
OK
127.0.0.1:6379> GET exp:test
"value"
127.0.0.1:6379> PSETEX exp:test:2 1000 "value2"
OK
127.0.0.1:6379> GET exp:test:2
(nil)
127.0.0.1:6379> PSETEX exp:test:2 10000 "value2"
OK
127.0.0.1:6379> GET exp:test:2
"value2"
```

String Data Structure

Utility Commands

- APPEND
- STRLEN
- SETRANGE
- GETRANGE

```
127.0.0.1:6379> SET key "redis"
OK
127.0.0.1:6379> APPEND key " is fun"
(integer) 12
127.0.0.1:6379> GET key
"redis is fun"
127.0.0.1:6379> SETRANGE key 9 easy
(integer) 13
127.0.0.1:6379> GET key
"redis is easy"
127.0.0.1:6379> STRLEN key
(integer) 13
127.0.0.1:6379> GETRANGE key 0 4
"redis"
```

String Data Structure

Numeric Commands

- INCR
- INCRBY
- DECR
- DECRBY
- INCRBYFLOAT

```
127.0.0.1:6379> SET counter 1
OK
127.0.0.1:6379> GET counter
"1"
127.0.0.1:6379> INCR counter
(integer) 2
127.0.0.1:6379> INCRBY counter 5
(integer) 7
127.0.0.1:6379> DECR counter
(integer) 6
127.0.0.1:6379> SET float 2.0
OK
127.0.0.1:6379> GET float
"2.0"
127.0.0.1:6379> INCRBYFLOAT float .5
"2.5"
127.0.0.1:6379> INCR float
(error) ERR value is not an integer or out of range
```

Why We Care

- Support ById query where returned value is EdgeX data structure
  - Key is UUID
  - Serialized EdgeX data structures are stored as Redis Strings

Create

```
m, err := marshalEvent(e)
_ = conn.Send("SET", e.ID, m)
```

Read

```
obj, err := redis.Bytes(conn.Do("GET", id))
event, err = unmarshalEvent(obj)
```

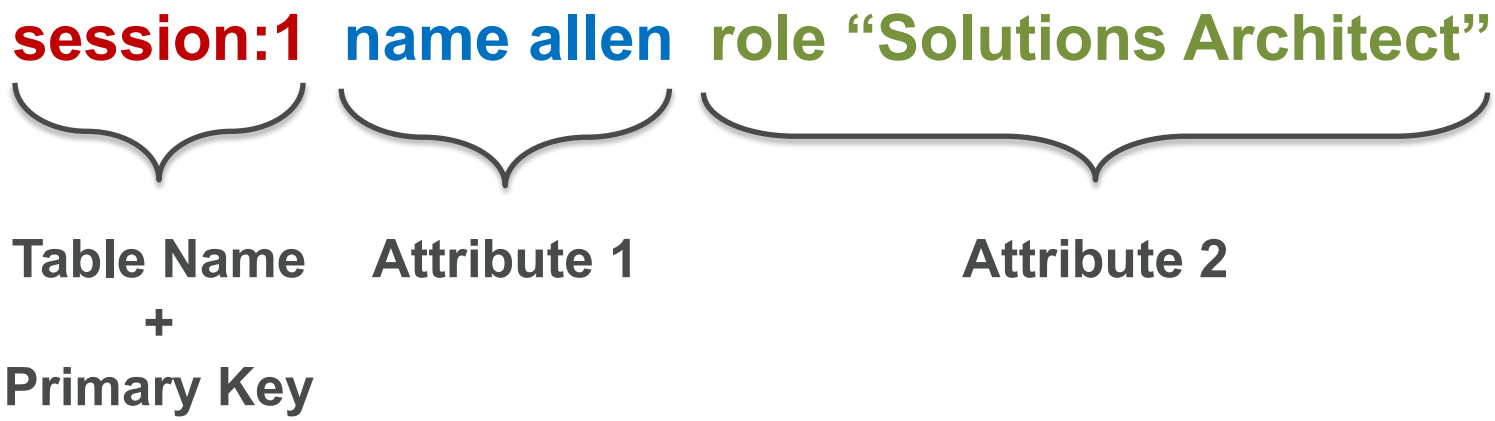
Hash Data Structure

- Dictionary or Map of key-value pairs (attributes)
- Flat data model – No explicit hierarchy and/or nesting
- Each attribute's value is a String data structure
- Advantages of Hashes instead of Strings
  - Well suited to represent Objects (Classes)
  - Uses very little space when ~100 fields or fewer are used
  - High throughput for partial reads/exists/scan/writes

Hash Data Structure

Table - Session

key	Name	Role
1	Allen	Solutions Architect
2	John	Account manager
3	Dan	Solutions Architect



Hash Data Structure

Basics

- HSET/HMSET
- HGET
- HGETALL
- HVALS
- HMGET
- HMSET

```
127.0.0.1:6379> HSET session:1 name allen
(integer) 1
127.0.0.1:6379> HSET session:1 role sa
(integer) 1
127.0.0.1:6379> HGET session:1 name
"allen"
127.0.0.1:6379> HMSET session:2 name john role sales quota 100
OK
127.0.0.1:6379> HGETALL session:2
1) "name"
2) "john"
3) "role"
4) "sales"
5) "quota"
6) "100"
127.0.0.1:6379> HMGET session:2 name role
1) "john"
2) "sales"
127.0.0.1:6379> HVALS session:1
1) "allen"
2) "sa"
```

Hash Data Structure

Utility Commands

- **HEXISTS**
- **HKEYS**
- **HSCAN**
- **HLEN**
- **HDEL**

```
127.0.0.1:6379> HGETALL session:2
1) "name"
2) "john"
3) "role"
4) "sales"
5) "quota"
6) "100"
127.0.0.1:6379> HEXISTS session:2 name
(integer) 1
127.0.0.1:6379> HKEYS session:2
1) "name"
2) "role"
3) "quota"
127.0.0.1:6379> HSCAN session:2 0 MATCH nam*
1) "0"
2) 1) "name"
   2) "john"
127.0.0.1:6379> HLEN session:2
(integer) 3
127.0.0.1:6379> HDEL session:2 quota
(integer) 1
127.0.0.1:6379> HLEN session:2
(integer) 2
```

Why We Care

- Support ByName or other field specific queries (think dictionary lookup)
  - Result is UUID so can get to serialized data structure

Create

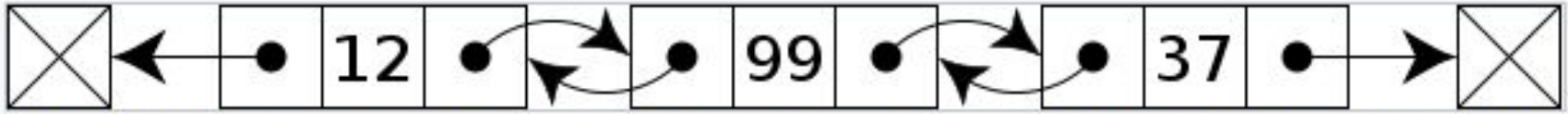
```
_ = conn.Send("HSET", db.DeviceService+":name", ds.Name, id)
```

Read

```
id, err := redis.String(conn.Do("HGET", hash, field))
object, err := redis.Bytes(conn.Do("GET", id))
return unmarshal(object, out)
```

List Data Structure

- Implemented as doubly linked list
- High Read / Write Performance for Head / Tail modifications
- Consumers can perform blocking operations



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

List Data Structure

Basics

LPUSH 2

RPUSH 1

LPUSH 3

RPOP / BRPOP

LRANGE

```
127.0.0.1:6379> LPUSH list 2
(integer) 1
127.0.0.1:6379> RPUSH list 1
(integer) 2
127.0.0.1:6379> LPUSH list 3
(integer) 3
127.0.0.1:6379> LRANGE list 0 -1
1) "3"
2) "2"
3) "1"
127.0.0.1:6379> RPOP list
"1"
127.0.0.1:6379> LRANGE list 0 -1
1) "3"
2) "2"
```

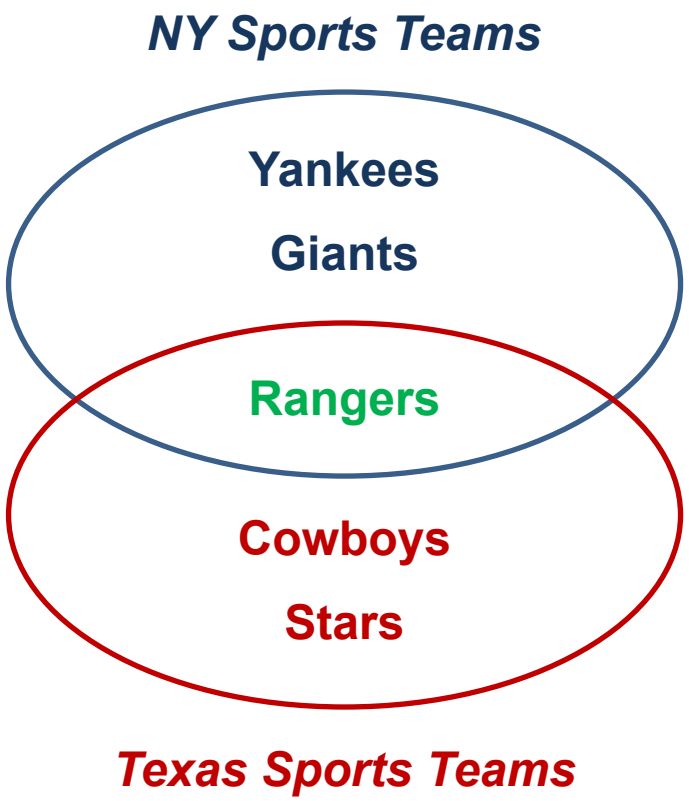
Why We Care

- We don't at the moment
  - Just wanted to show the data structure exists

Set Data Structure

- Follows the principles of basic discrete mathematics
- Distinct values only
- Unordered collection (order of elements is not guaranteed)
- High performance operations
- Perfect for operations on one or more collections
  - Intersection, Union, Difference

Set Data Structure




```
127.0.0.1:6379> SADD ny_teams yankees giants rangers
(integer) 3
127.0.0.1:6379> SADD tx_teams rangers cowboys stars
(integer) 3
127.0.0.1:6379> SDIFF ny_teams tx_teams
1) "giants"
2) "yankees"
127.0.0.1:6379> SUNION ny_teams tx_teams
1) "rangers"
2) "giants"
3) "yankees"
4) "stars"
5) "cowboys"
127.0.0.1:6379> SINTER ny_teams tx_teams
1) "rangers"
127.0.0.1:6379> SMOVE ny_teams tx_teams giants
(integer) 1
127.0.0.1:6379> SMEMBERS tx_teams
1) "cowboys"
2) "rangers"
3) "giants"
4) "stars"
```

Set Data Structure

Utility Commands

- SISMEMBER
- SCARD
- SSCAN
- SRANDMEMBER
- SREM
- SPOP

```
127.0.0.1:6379> SADD genre mystery romance comedy
(integer) 3
127.0.0.1:6379> SISMEMBER genre comedy
(integer) 1
127.0.0.1:6379> SCARD genre
(integer) 3
127.0.0.1:6379> SSCAN genre 0 MATCH m*
1) "0"
2) 1) "mystery"
127.0.0.1:6379> SRANDMEMBER genre
"comedy"
127.0.0.1:6379> SREM genre romance
(integer) 1
127.0.0.1:6379> SPOP genre 1
1) "comedy"
127.0.0.1:6379> SMEMBERS genre
1) "mystery"
```

Why We Care

- Support ByNameAndDeviceId. Think multiple tags/labels.

Create

```
_ = conn.Send("SADD", db.Command+":name:"+cmd.Name, cid)
_ = conn.Send("SADD", db.Command+":device:"+id, cid)
```

Read

```
ids, err := redis.Values(conn.Do("SINTER", args...))
objects, err = redis.ByteSlices(conn.Do("MGET", ids...))
```

Why We Care

- Support ByCategoriesLabels.

Create

```
_ = conn.Send("SADD", db.Subscription+":label:"+label, id)
```

Read

```
ids, err := redis.Values(conn.Do("SUNION", args...))
objects, err = redis.ByteSlices(conn.Do("MGET", ids...))
```

Sorted Set Data Structure

- Stores members like Sets however guarantees order
- Sorting is based on an inserted score/weight/time-interval
- Does not include DIFF / UNION commands
- Used for
  - Ordered list such such as time series
  - Leaderboards
  - Priority/Weighted Queues
  - Publish-Subscribe
  - Activity Tracking

Sorted Set Data Structure

Basics

- ZADD
- ZRANGE
- ZRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE

```
127.0.0.1:6379> ZADD game 100 Jill
(integer) 1
127.0.0.1:6379> ZADD game 25 Allen
(integer) 1
127.0.0.1:6379> ZADD game 50 Dave
(integer) 1
127.0.0.1:6379> ZRANGE game 0 -1
1) "Allen"
2) "Dave"
3) "Jill"
127.0.0.1:6379> ZRANGEBYSCORE game 0 50
1) "Allen"
2) "Dave"
127.0.0.1:6379> ZREVRANGE game 0 -1
1) "Jill"
2) "Dave"
3) "Allen"
127.0.0.1:6379> ZREVRANGEBYSCORE game 100 30 WITHSCORES
1) "Jill"
2) "100"
3) "Dave"
4) "50"
```

Sorted Set Data Structure

Utility Commands

- ZCARD
- ZCOUNT
- ZSCORE
- ZRANK
- ZSCAN

```
127.0.0.1:6379> ZADD game 100 Jill 50 Dave 25 Allen
(integer) 3
127.0.0.1:6379> ZCARD game
(integer) 3
127.0.0.1:6379> ZCOUNT game 0 50
(integer) 2
127.0.0.1:6379> ZSCORE game Dave
"50"
127.0.0.1:6379> ZRANK game Dave
(integer) 1
127.0.0.1:6379> ZSCAN game 0 MATCH *e*
1) "0"
2) 1) "Allen"
   2) "25"
   3) "Dave"
   4) "50"
```

Sorted Set Data Structure

Basics++

- ZINCRBY
- ZREM
- ZPOPMAX
- ZPOPMIN

```
127.0.0.1:6379> ZINCRBY game 50 Allen
"75"
127.0.0.1:6379> ZRANGE game 0 -1
1) "Dave"
2) "Allen"
3) "Jill"
127.0.0.1:6379> ZREM game Allen
(integer) 1
127.0.0.1:6379> ZPOPMAX game 1
1) "Jill"
2) "100"
127.0.0.1:6379> ZPOPMIN game 1
1) "Dave"
2) "50"
```

Why You Care

- Support ByCreationTime or ByDevice range queries

Create

```
    _ = conn.Send("SET", r.Id, m)
    _ = conn.Send("ZADD", db.ReadingsCollection, 0, r.Id)
    _ = conn.Send("ZADD", db.ReadingsCollection+":created", r.Created,
r.Id)
    _ = conn.Send("ZADD", db.ReadingsCollection+":device:"+r.Device,
r.Created, r.Id)
    _ = conn.Send("ZADD", db.ReadingsCollection+":name:"+r.Name,
r.Created, r.Id)
```

Why You Care

- Support ByCreationTime or ByDevice range queries

Read

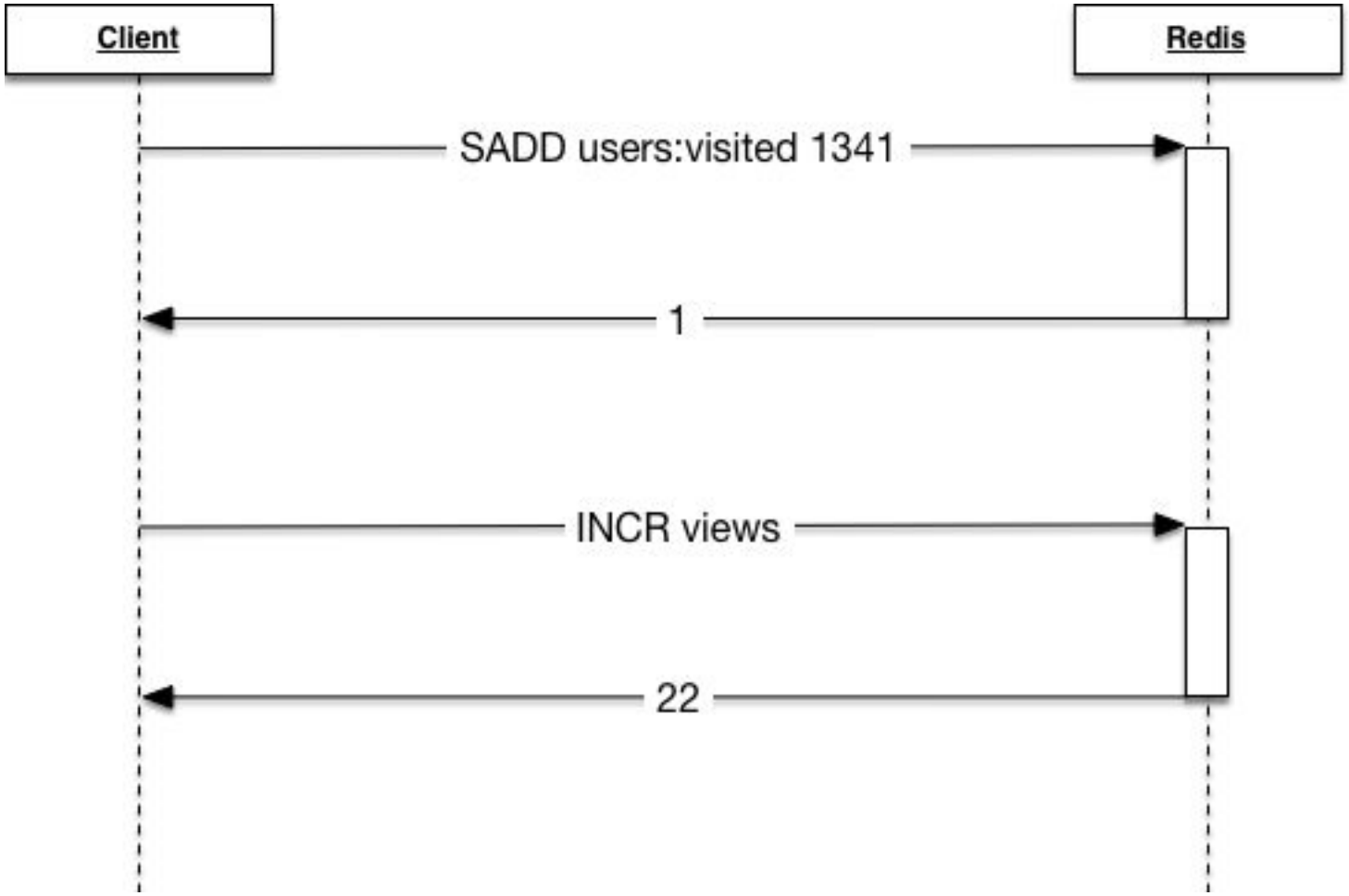
```
ids, err := redis.Values(conn.Do("ZRANGE", key, 0, -1))

ids, err := redis.Values(conn.Do("ZREVRANGE", key, 0, -1))
```

Transactions

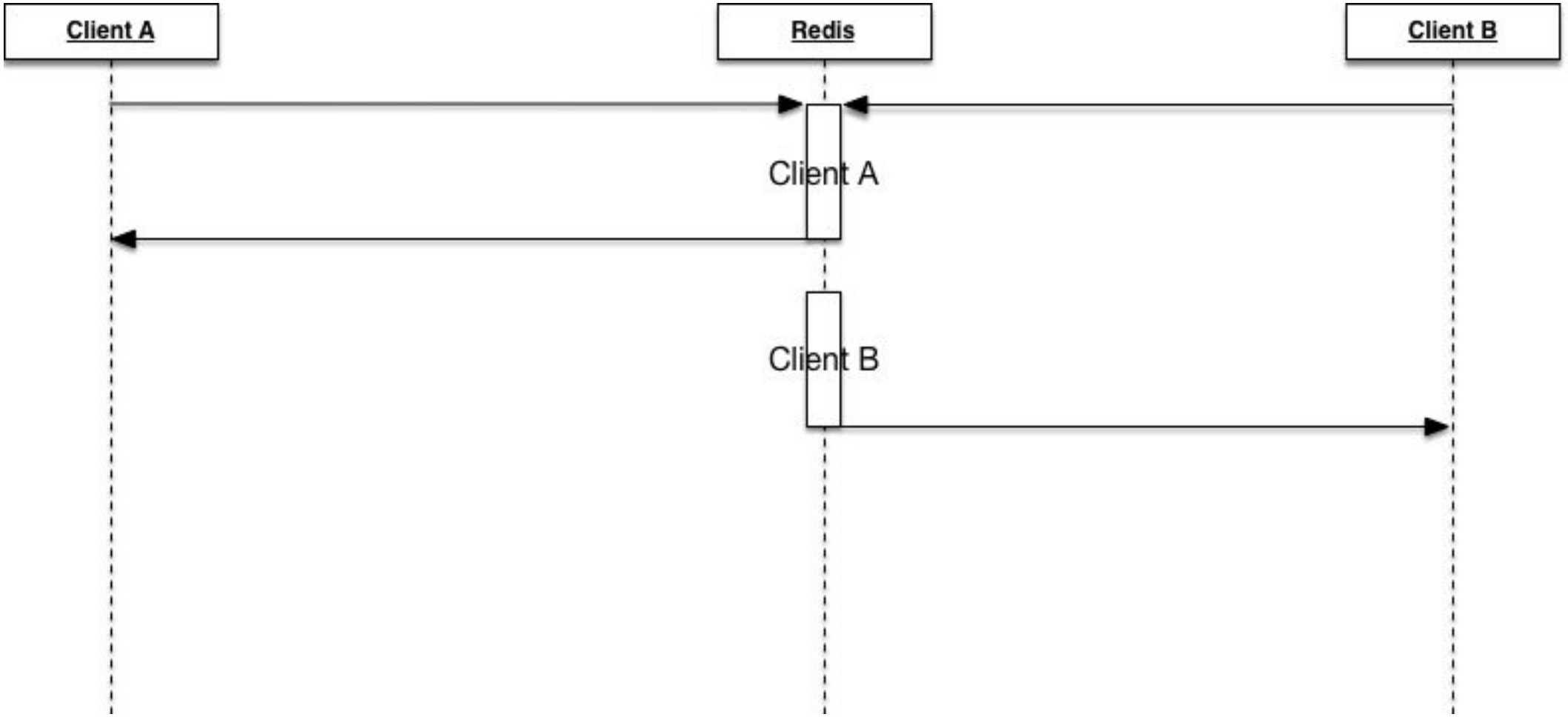
ACID Transactions

A - Atomicity
  -- Transaction executes as an indivisible unit
C - Consistency
  -- Transaction takes database from one valid state to another
I – Isolation
  -- Transactions result in a state as if they were executed sequentially
D – Durability
  -- Transaction changes are available event in the event of failure

Single Client – Execution Flow



Two Client – Execution Flow



Multiple Command Transactions

- MULTI to start transaction block
- EXEC to close transaction block

- Commands are queued until exec
- All commands or no commands are applied
- **Transactions can have errors**

MULTI Example

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> sadd site:visitors 124
QUEUED
127.0.0.1:6379> incr site:raw-count
QUEUED
127.0.0.1:6379> hset sessions:124 userid salvatore ip 127.0.0.1
QUEUED
127.0.0.1:6379> EXEC
1) (integer) 1
2) (integer) 1
3) (integer) 2
```

DISCARD Example

```
127.0.0.1:6379> sadd site:visitors 124
QUEUED
127.0.0.1:6379> incr site:raw-count
QUEUED
127.0.0.1:6379> DISCARD
OK
```

Transactions with Errors – Syntactic Error

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set site:visitors 10
QUEUED
127.0.0.1:6379> ste site:raw-count 20
(error) ERR unknown command 'ste'
127.0.0.1:6379> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
```

Transactions with Errors – Semantic Error

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set messages:hello "Hello World!"
QUEUED
127.0.0.1:6379> incr messages:hello
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) (error) ERR value is not an integer or out of range
```

Persistence

Disk Based Persistence - Options

- Redis continues to serve commands from main memory
- Multiple Persistence modes
  - **Snapshot (RDB):** store a compact point-in-time copy every minute, hourly, or daily – tunable
  - **Append-only-file (AOF):** write to disk (fsync) every second or every write - tunable
- Provides durability of data across power loss
  - Look into replication to prevent data loss in case of node loss

RDB Persistence

- Persistence
  - Fork Redis process
  - Child process writes new RDB file
  - Atomic replace old RDB file with new
- Trigger manually
  - SAVE command (sync)
  - BGSAVE (background)

AOF Persistence

- Configuration
  - APPENDONLY directive (Redis.conf): APPENDONLY YES
  - Runtime : CONFIG SET APPENDONLY YES
- AOF File fsync options
  - Trade off speed for data security
  - Options: None, every second, always
- BGREWRITEAOF
  - AOF file grows indefinitely
  - BGREWRITEAOF – trigger compaction of AOF file

Transactions and Persistence Summary

- No Rollback – transaction commands are queued then sent to server
- Transactions
  - Atomic – through MULTI/EXEC
  - Isolation, Consistency – single threaded nature
  - Durability – persistence modes: snapshots and append-only-file
- Transactions work differently than other databases, achieves the same goals
- Single threaded event-loop for serving commands

TL;DR

Current EdgeX/Redis Status

- All microservices that use persistence have a Redis implementation
  - redigo client library
  - Lua for to optimize server centric operations
- Based on how data queried, data structures used do date
  - String
  - Hash
  - Set
  - Sorted Set
- No plans to support Logger
- Need integrate with security config for username/password auth

Thinking About

- Redis Streams implementation of message bus
- Storage optimization
- Expose Redis configuration to EdgeX configuration

Tools

- monitor command
- redis-cli
- RedisInsight - https://redislabs.com/redisinsight/

More Info

- redis.io
- Stack Overflow
- Redis In Action - https://redislabs.com/ebook/foreword/
- Redis University - https://university.redislabs.com/
- andre@redislabs.com

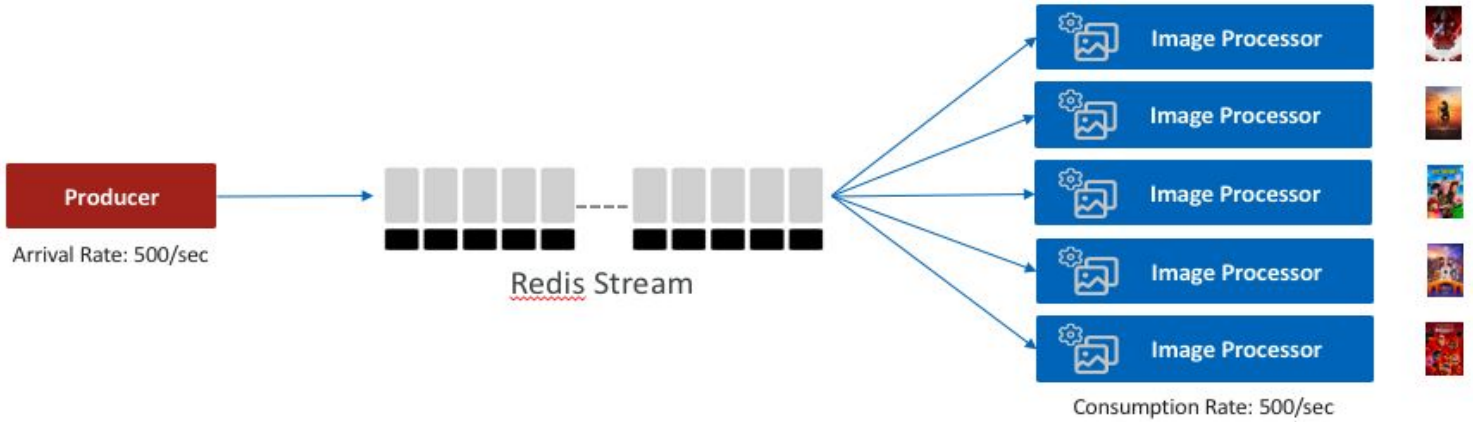
Thanks

André Srinivasan
andre@redislabs.com

Backup

Streams Data Structure

- Redis Steams is **purpose-built to implement Message Streaming** and **Event-Processing patterns**
- Streams is an append-only log-like data structure which **naturally guarantees ordering by time**
- Broadcasts in parallel to consumers for **max throughput at sub-millisecond latency**
- Naturally provides **at-least once** delivery or can implement **exactly-once** delivery guarantees
- Naturally allows for **replay** and **querying of historical messages**
- Scales consumer-side processing by allowing **consumer groups** to partition individual stream(s)

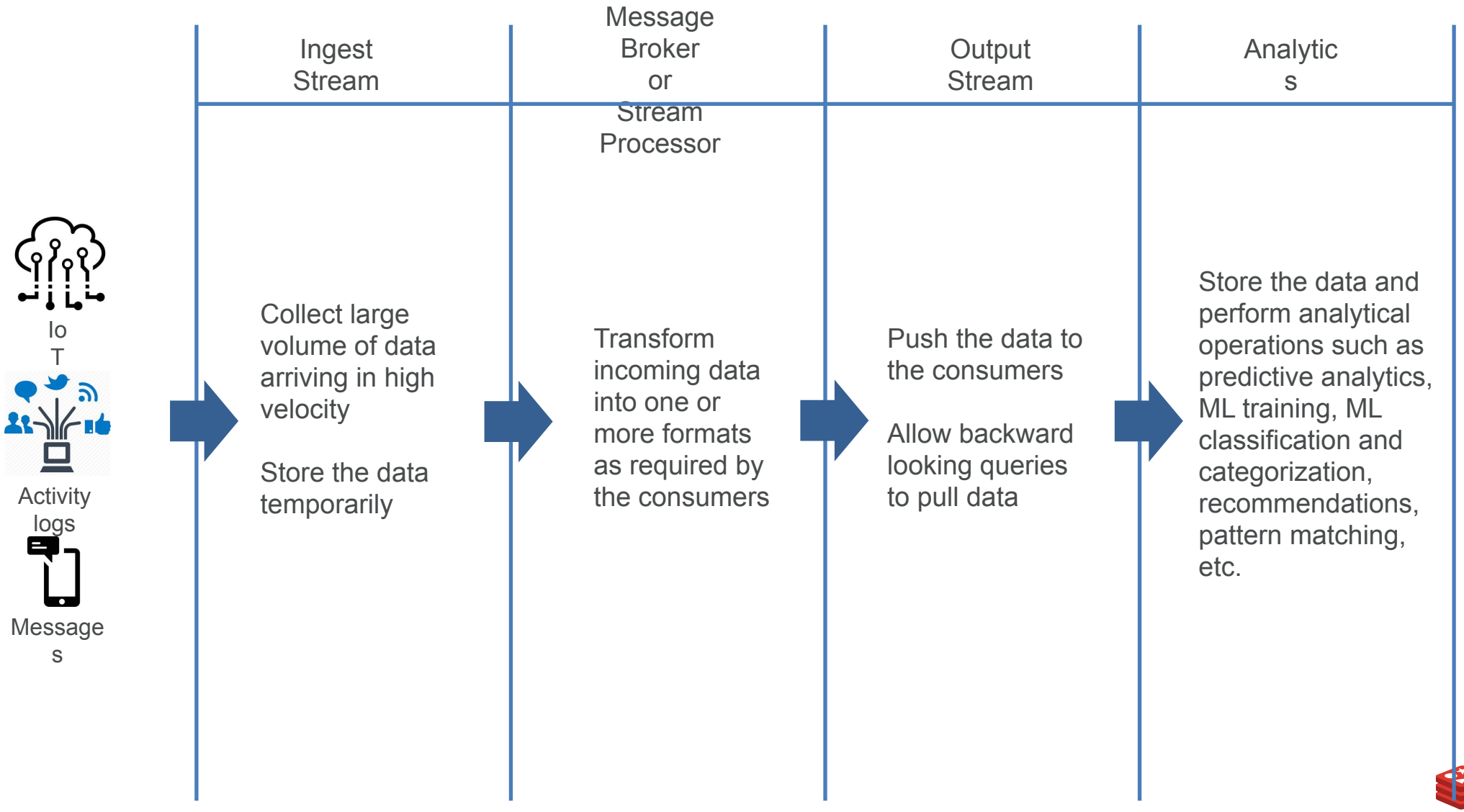


Learn more at: https://redis.io/topics/streams-intro

Streams Data Structure

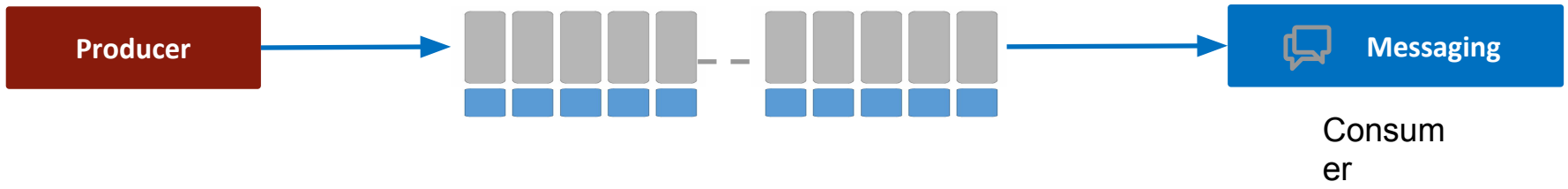
- **Option to consumers to read streaming data and data at rest**
- **Consumer groups to help the consumers to coordinate among themselves while reading the data from the same stream**
- **Super-fast lookup queries powered by radix trees**
- **Automatic eviction of data based on the upper limit**

Streams Data Structure



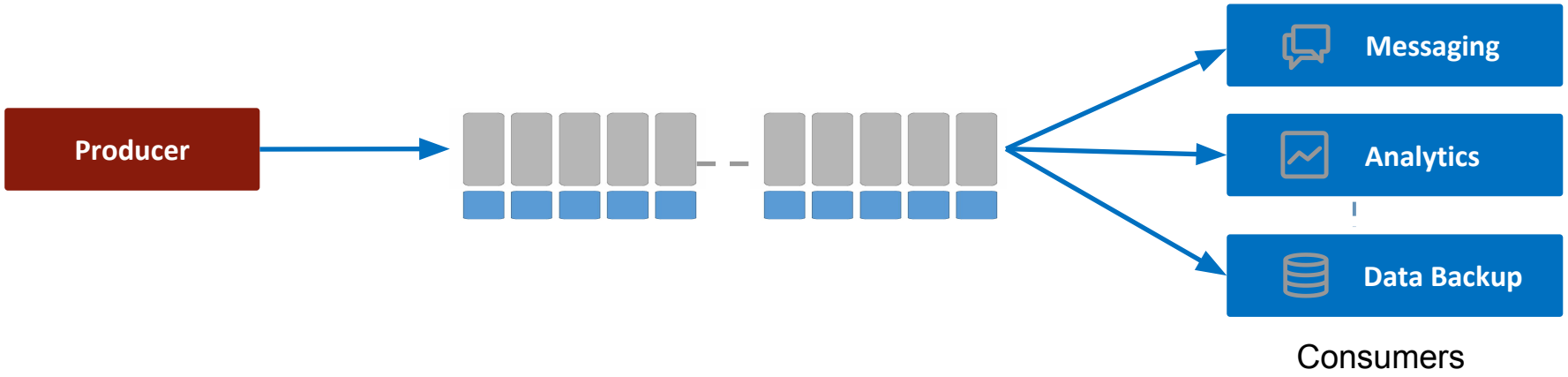
Streams Data Structure

1. **It enables asynchronous data exchange between producers and consumers**



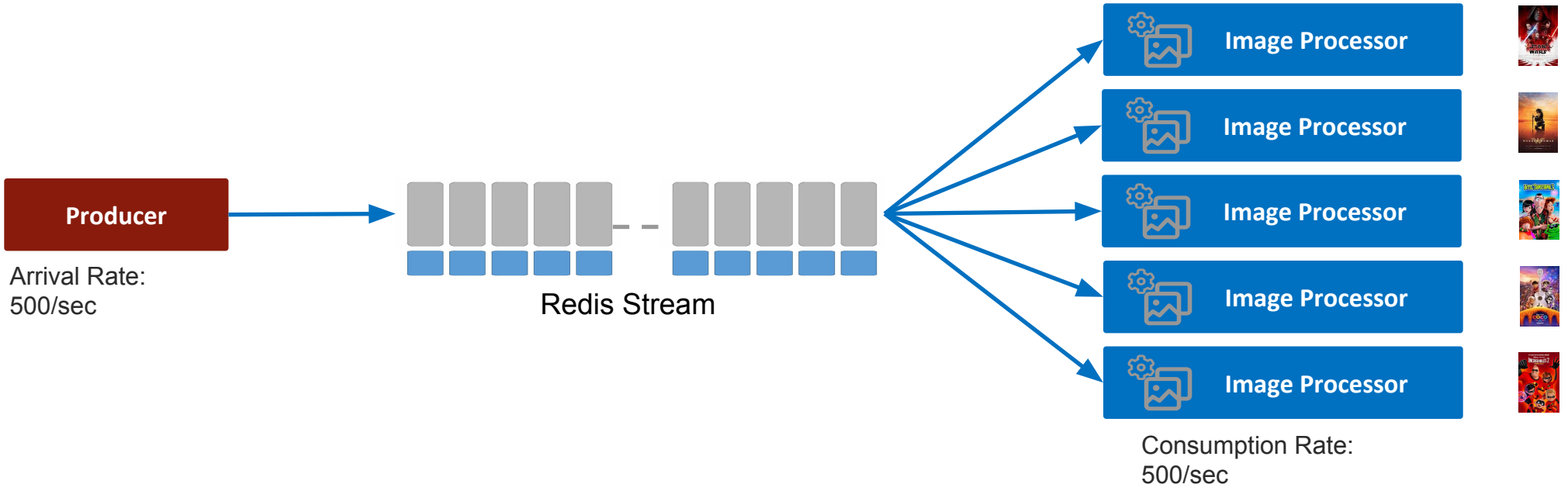
Streams Data Structure

2. **You can consume data in real-time as it arrives or lookup historical data**



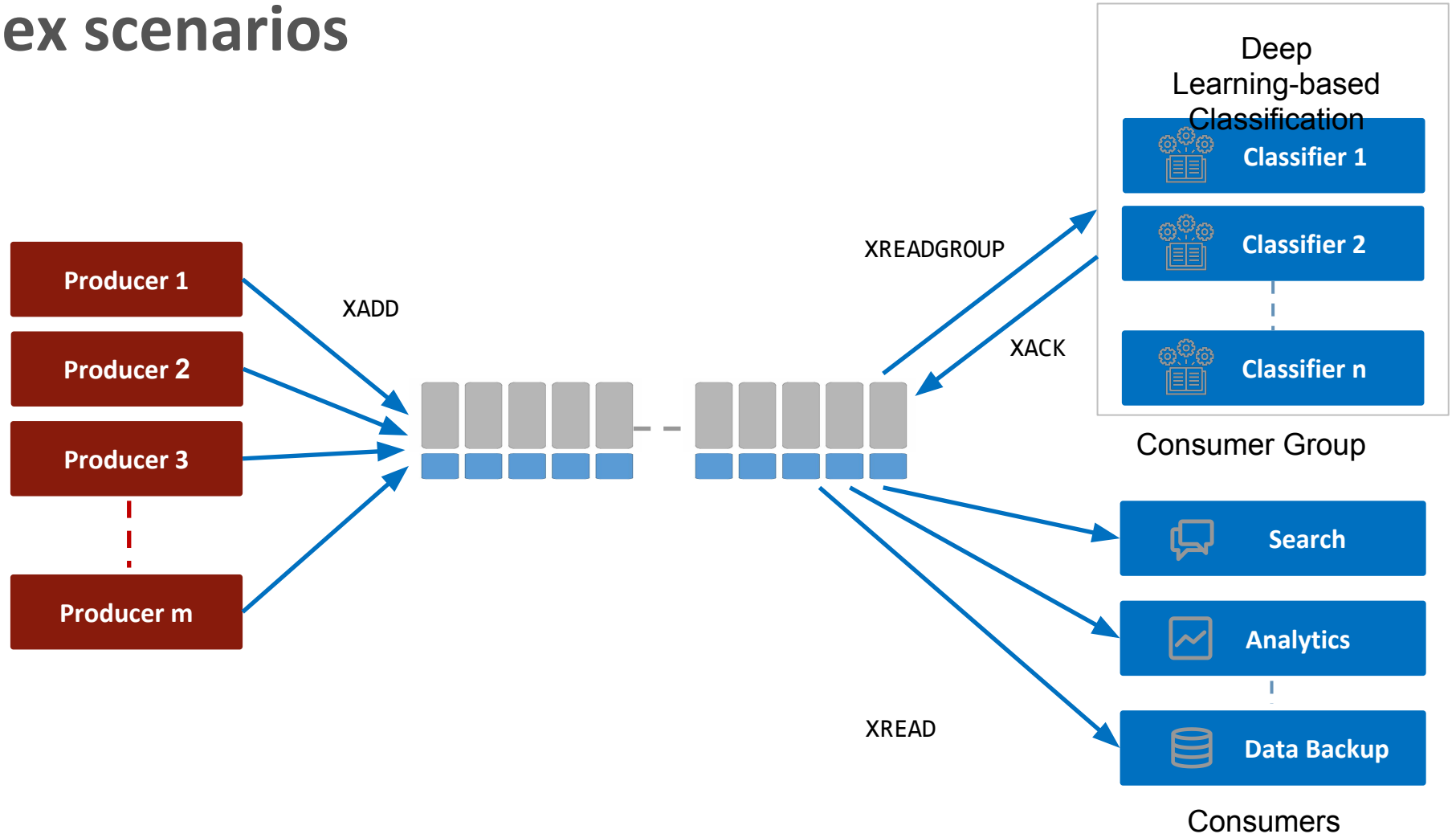
Streams Data Structure

3. With consumer groups, you can scale out and avoid backlogs



Streams Data Structure

Simplify data collection, processing and distribution to support complex scenarios



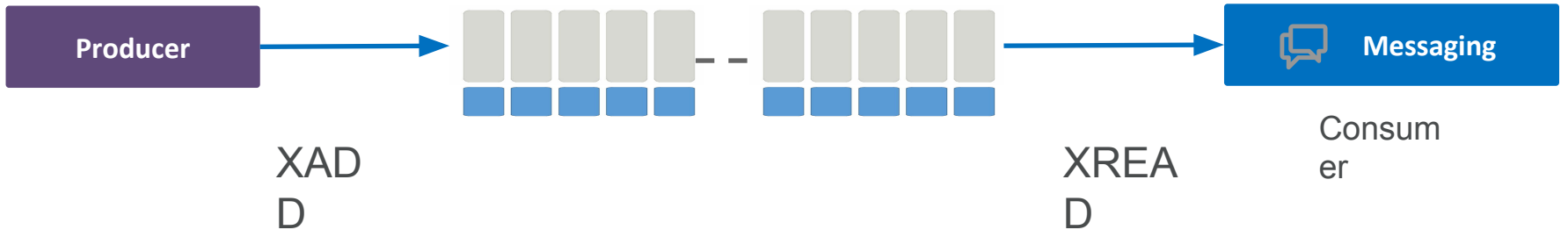
Streams Data Structure

Benefits

- High Velocity Collection (the only bottleneck is your network I/O)
- Many to Many Mapping – Producer to Consumer
- Effectively manage your consumption of data even when producers and consumers don’t operate at the same rate
- Persist data when your consumers are offline or disconnected
- Communicate between producers and consumers asynchronously (Rate limitation problems)
- Scale your number of consumers based on data consumption
- Implement transaction-like data safety when consumers fail consuming data (XACK)
- Self Contained Architecture

Streams Data Structure

Asynchronous producer-consumer message transfer

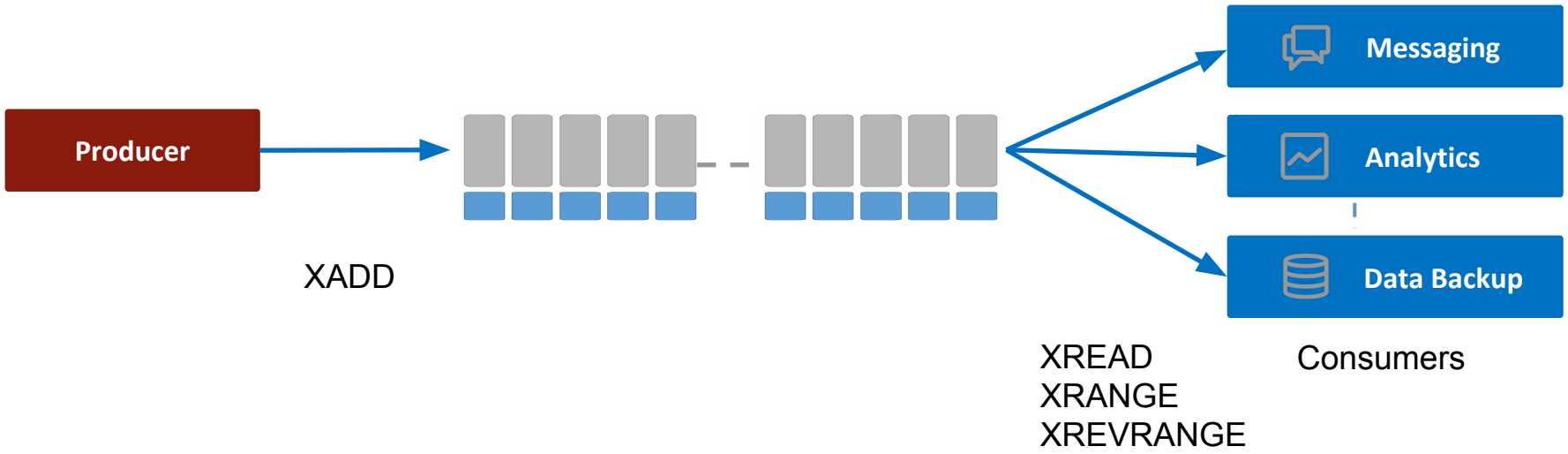



```
XADD mystream * name Anna
XADD mystream * name Bert
XADD mystream * name Cathy
```

```
XREAD COUNT 100 STREAMS mystream 0

XREAD BLOCK 10000 STREAMS mystream $
```

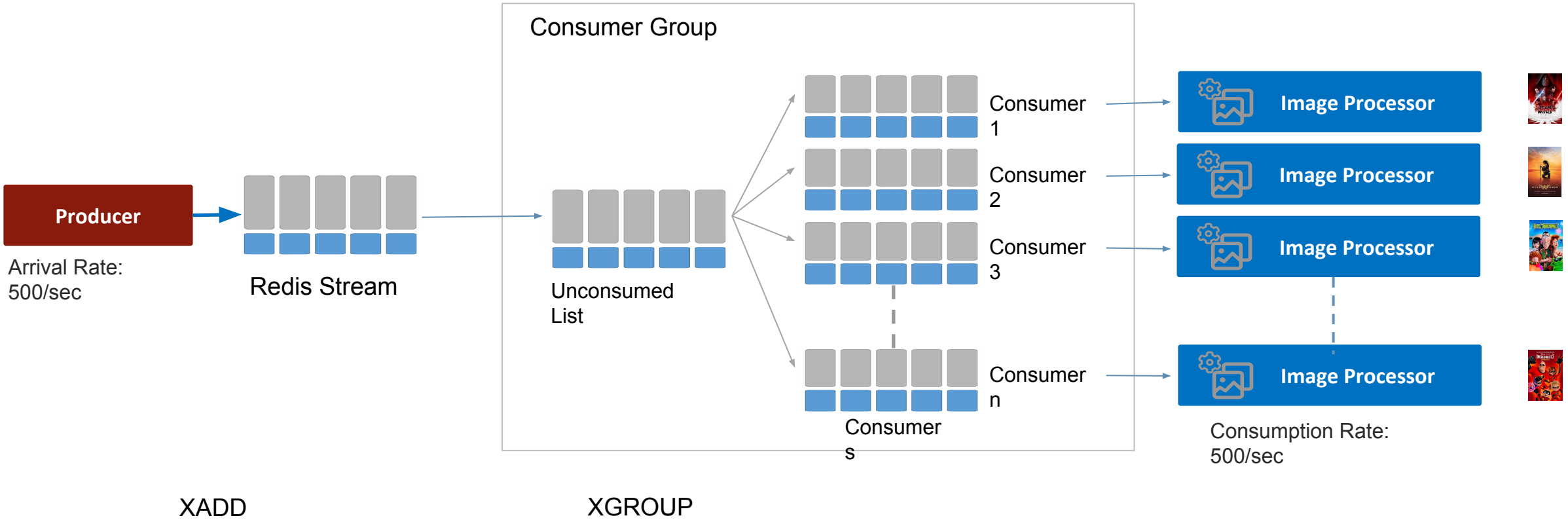
Streams Data Structure

Lookup historical data



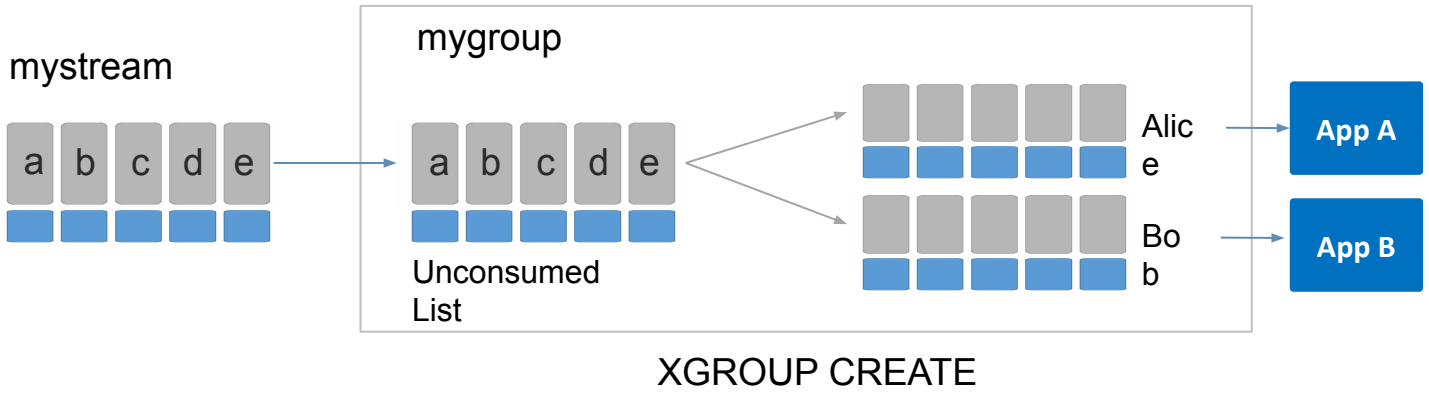

```
XRANGE mystream 1518951123450-0 1518951123460-0 COUNT 10

XRANGE mystream - + COUNT 10
```

Streams Data Structure



Streams Data Structure

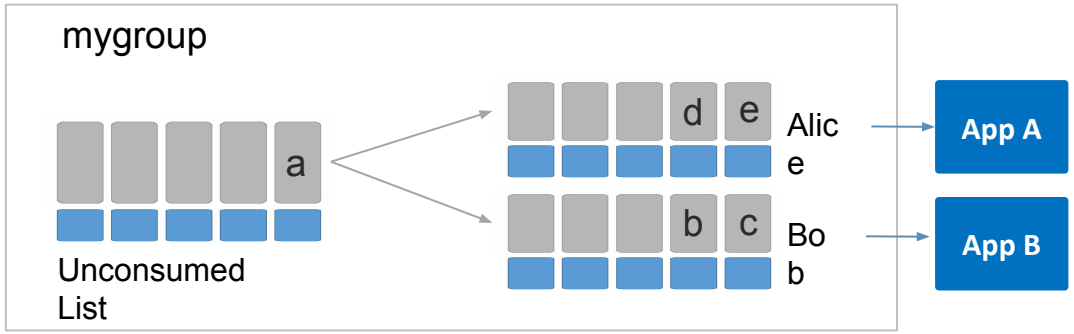
Create a consumer group




```
XGROUP CREATE mystream mygroup $
```

Streams Data Structure

Read the data

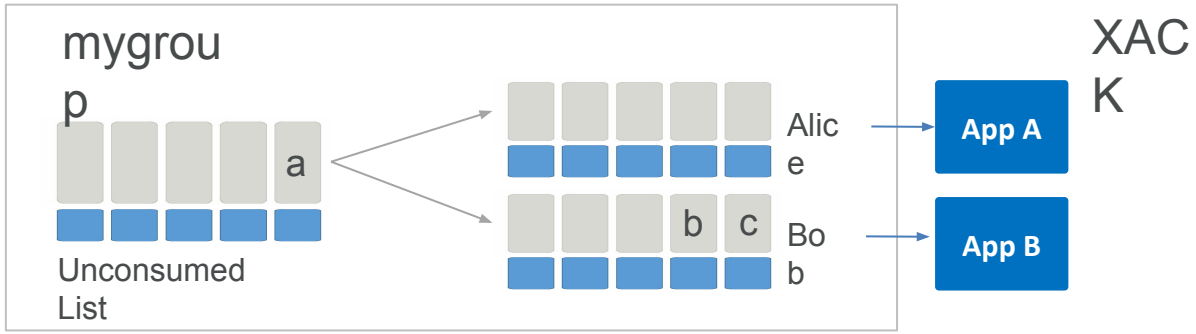


XREADGROUP

```
XREADGROUP GROUP mygroup COUNT 2 Alice STREAMS mystream >

XREADGROUP GROUP mygroup COUNT 2 Bob STREAMS mystream >
```

Streams Data Structure

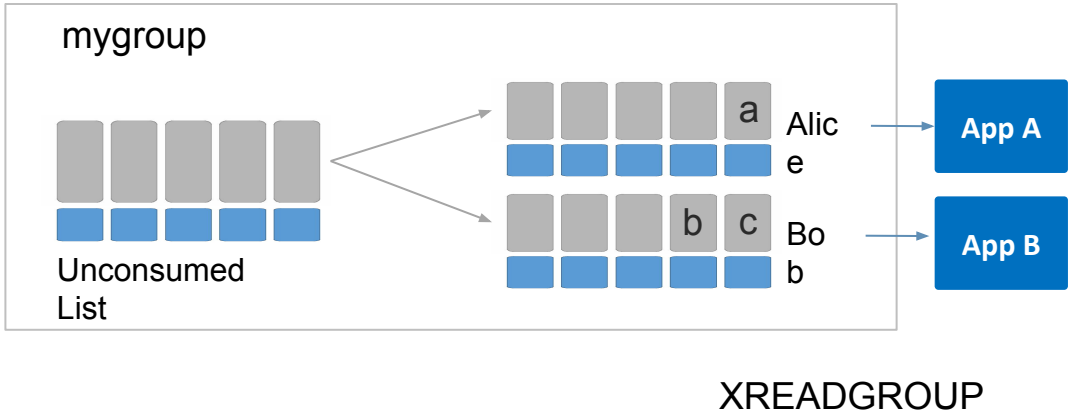
Consumers acknowledge that they consumed the data




```
XACK mystream mygroup 1526569411111-0 1526569411112-0
```

Streams Data Structure

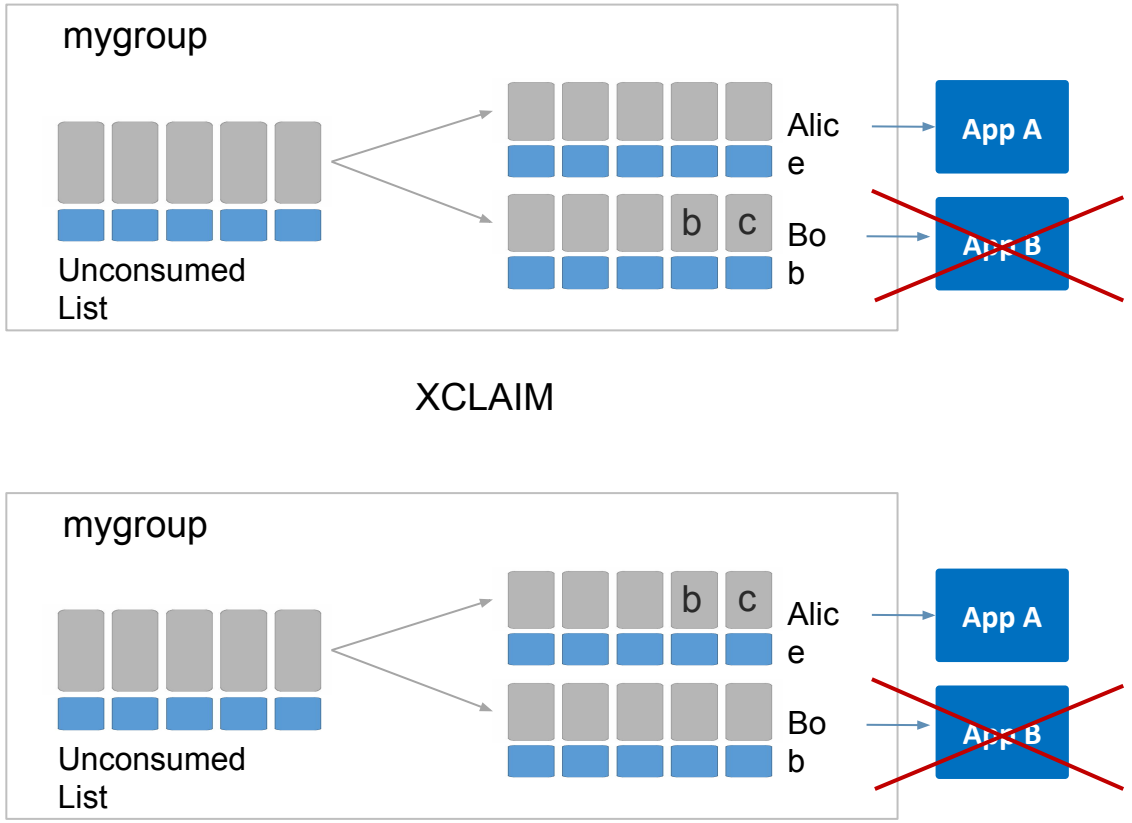
Repeat the cycle




```
XREADGROUP GROUP mygroup COUNT 2 Alice STREAMS mystream >
```

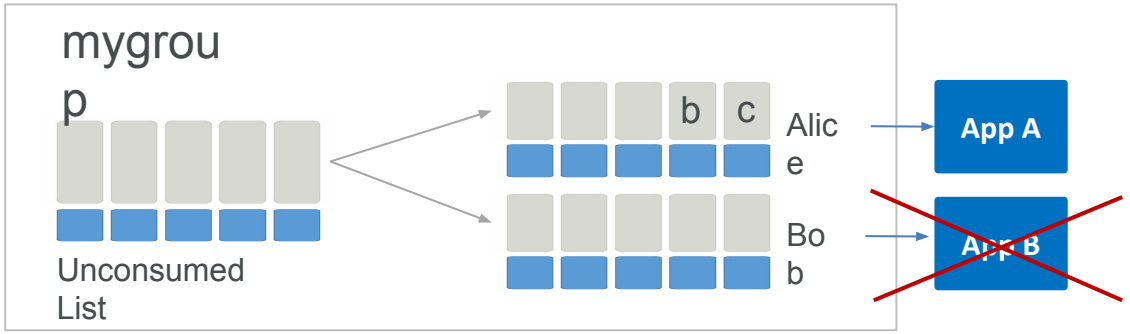
Streams Data Structure

How to claim the data from a consumer that failed while processing the data?



Streams Data Structure

Claim pending data from other consumers

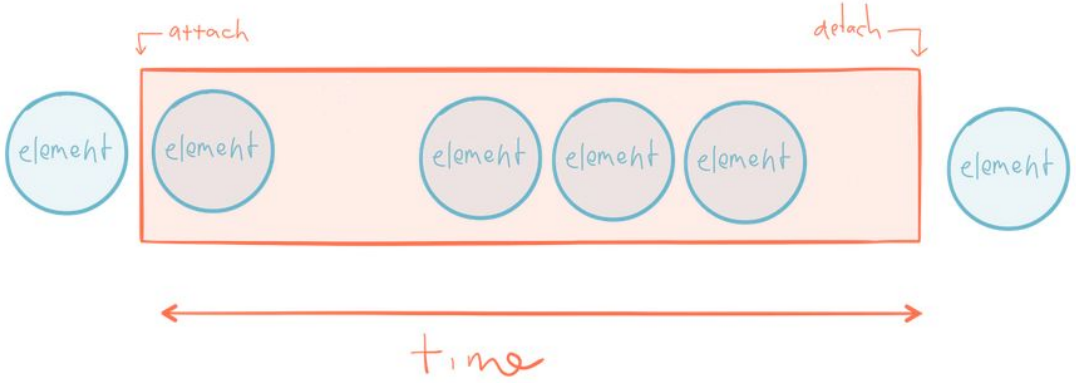



```
XPENDING mystream mygroup - + 10 Bob

XCLAIM mystream mygroup Alice 0 1526569411113-0 1526569411114-0
```

(Hands-on Exercise) *Streams* Data Structure

Producer

```
127.0.0.1:6379> XADD stream * message value
"1569986851098-0"
127.0.0.1:6379> XADD stream * message value
"1569986853350-0"
127.0.0.1:6379> XADD stream * message value
"1569986854353-0"
127.0.0.1:6379> XADD stream * message value
"1569986873230-0"
```



Consumer

```
127.0.0.1:6379> XRANGE stream - +
1) 1) "1569986851098-0"
   2) 1) "message"
      2) "value"
2) 1) "1569986853350-0"
   2) 1) "message"
      2) "value"
3) 1) "1569986854353-0"
   2) 1) "message"
      2) "value"
```

Consumer

```
127.0.0.1:6379> XREAD BLOCK 0 STREAMS stream $
1) 1) "stream"
   2) 1) 1) "1569986873230-0"
         2) 1) "message"
            2) "value"
(4.97s)
```