Domain narrative & design goals

The chess-club database is **read-heavy**: new games arrive slowly, while players query histories, leaderboards and analytics many times a day.
Design therefore follows two main principles:

1. **Query-pattern-first**: each required question must be answerable by *one Redis operation* or, at worst, a short script that touches a single key family.
2. **Write-once, read-many**: we pre-aggregate during **Write Events** and store redundant counts rather than scan large lists later.

Note: the assignment restrict our usage of Redis types to only Lists, Sets, Keys and BloomFilters.

Write Events

ID	Event
E1	Initial load (via `load_transform.py`)
E2	When a new player is added
E3	When a new game is scheduled
E4	When a game record is inserted
E5	When 72 hours passes on a scheduled game

Key Generation Method

Namespace convention

```
<root>:<entity-id>[:subid][:attribute]
```

- `player` and `game` are the only entity roots.
- A trailing plural (`games` , `opponents`) always holds a **collection** (List / Set).
- `global:` and `analytics:` roots hold cross-player aggregates.

Keys

Player

#	Key pattern	Redis type	Value contents	Written / updated	Read by
P1	`player:{pid}:email`	string	e-mail address	E1, E2	
P2	`player:{pid}:wins`	string (int)	win counter	E1, E2, E4	`filtered_fof`
P3	`player:{pid}:losses`	string (int)	loss counter	E1, E2, E4	
P4	`player:{pid}:games`	list of strings	`game_id` s	E1, E2, E4	`player_match_history(pid)`
P5	`player:{pid}:games-set`	set of strings	`game_id` s	E1, E2, E4	`games_against_opponent(pid, oid)`
P6	`player:{pid}:scheduled`	list of strings	future `game_id` s	E1, E2, E3, E5	`future_games(pid)`
P7	`player:{pid}:opponents`	Set of strings	opponent PIDs	E1, E2, E4	`friends_of_friends(pid)`
P8	`player:{pid}:seq`	bloom filter	3-move string by PID	E1, E2, E4	`player_seq()`
P9	`player:{pid}:openings:{eco}`	string (int)	player ECO counter	E1, E2, E4	`player_most_freq_opening(pid)`
P10	`player:{pid}:most_freq_opening`	string	most used opening	E1, E2, E4	`player_most_freq_opening(pid)`

Game

#	Key pattern	Redis type	Value contents	Written / updated	Read by
G1	`game:{gid}:winner`	string	winner color	E1, E4	
G2	`game:{gid}:victory_status`	string	how winner won	E1, E4	
G3	`game:{gid}:number_of_turns`	string (int)	turns counter	E1, E4	
G4	`game:{gid}:white_player_id`	string	white player PID	E1, E4	
G5	`game:{gid}:black_player_id`	string	black player PID	E1, E4	
G6	`game:{gid}:opening_eco`	string	opening move	E1, E4	
G7	`game:{gid}:moves`	list of strings	game moves	E1, E4	three-move extraction (loader)

Analytics

#	Key pattern	Redis type	Value contents	Written / updated	Read by
A1	`analytics:shortest_game`	string	current shortest game	E1, E4	`shortest_game()`
A2	`analytics:check:{gid}`	string (int)	"+" count	E1, E4	`check_counts()`
A3	`analytics:top_wins`	list of strings	sorted list of joined `pid:count` s	E1, E4	`top_10()`
A4	`analytics:bottom_losses`	list of strings	sorted list of joined `pid:count` s	E1, E4	`bottom_10()`

#	Key pattern	Redis type	Value contents	Written / updated	Read by
A5	`analytics:most_freq_opening`	string	current leader ECO	E1, E4	`most_freq_opening()`
A6	`analytics:most_freq_opening_count`	string (int)	its count (avoids extra GET)	E1, E4	`most_freq_opening()`
A7	`analytics:most_common_seq`	string	most used 3-move string	E1, E4	`most_common_seq()`
A8	`analytics:least_common_seq`	string	least used 3-move string	E1, E4	`least_common_seq()`
A9	`analytics:least_common_seq_count`	string (int)	current minimum count	E1, E4	`least_common_seq()`

Global

#	Key pattern	Redis type	Value contents	Written / updated	Read by
GA1	`global:seq`	bloom filter	every 3-move string seen	E1, E4	`global_seq(seq)`
GA2	`global:seq:{seq}:count`	string (int)	every 3-move string seen	E1, E4	analytics queries
GA3	`global:opening:{eco}:count`	string (int)	total games that used this ECO	E1, E4	`most_freq_opening()`
GA4	`global:players:emails`	set of strings	`player_email`	E1, E2	`in_league(email)`

Write-up

Player Queries

`player_match_history(player_id)`

- **Key**: `player:{pid}:games`
- **Value**: List of every game-id the player has participated in (most-recent first).
- **Example**: `["g17", "g31", "g54"]`
- **Update**:
  - **E1**: initial load pushes all historical game-ids (oldest → newest) so the final list is in recency order.
  - **E2**: create an empty list for a new player.
  - **E4**: for each finished game, `LPUSH` the new `gid` onto each participant's list.
- **Read**: Use `player:{pid}:games` to return the full history in one operation.
- **Cost**:
  - Two `LPUSH` calls per game record (one for each player), and one `LRANGE` per query.
  - Each operation is O(1) for push and O(N) to scan the list of length N, but it's a single Redis call.

`future_games(player_id)`

- **Key**: `player:{pid}:scheduled`
- **Value**: List of upcoming `game_id`s in the order they were scheduled.
- **Example**: `["g88", "g90"]`
- **Update**:
  - **E1**: initial load builds any pre-scheduled games.
  - **E2**: create an empty list.
  - **E3**: `RPUSH` the new `gid` when a match is scheduled.
  - **E5**: external timer/job invokes removal after 72 h removing older games for each player.
- **Read**: Use `player:{pid}:scheduled` to return the list in one go.
- **Cost**:
  - One `RPUSH` per player when scheduling.
  - One `LREM` per player when expiring.
  - One `LRANGE` per query.

`in_league(player_email)`

- **Key**: `global:players:emails`
- **Value**: Set of every registered e-mail address.
- **Example**: `{"bob@club.org", "alice@club.org"}`
- **Update**:
  - **E1**: during initial load, `SADD` each player's email.
  - **E2**: when adding a new player, `SADD` their email.
- **Read**: Use `SISMEMBER` with `global:players:emails` to return a boolean in one operation.
- **Cost**:
  - One `SADD` per player write.
  - one `SISMEMBER` per lookup.
  - Both O(1).

`games_against_opponent(pid, oid)`

- **Keys**:
  - `player:{pid}:games-set`
  - `player:{oid}:games-set`
- **Value**: Each is the full set of games per player.

- **Example**: Intersecting `{"g31","g54"}` with `{"g31","g88"}` → `{"g31"}`
- **Update**:
  - **E1**: initial load does `SADD` for each historical game.
  - **E2**: create an empty set.
  - **E4**: on game insert, two `SADD` calls (one per participant).
- **Read**: Use `SINTER` with the two keys to get the answer in one operation.
- **Cost**:
  - Two `SADD` per game record
  - one `SINTER` per query.
  - O(N).

`player_most_freq_opening(pid)`

- **Keys**:
  - `player:{uid}:openings:{eco}`
  - `player:{uid}:most_freq_opening`
  - **Value**: A string representing the most used opening by this player.
  - **Example**:
  - `player:{pid}:openings:{eco}` → 13
  - `player:{pid}:most_freq_opening` → `f3`
- **Update**:
  - **E2**: Init to zero.
  - **E1/E4**:

```
SET OR INCR player:{pid}:openings:{eco}           → n
GET    player:{pid}:most_freq_opening             → fav
GET    player:{pid}:openings:{fav} (skip if None) → fcnt
IF fav is None OR n > fcnt:
  SET player:{pid}:most_freq_opening {eco}
```

- **Read**: Ues `GET` on `player:{pid}:most_freq_opening` to get it in one operation.
- **Cost**
  - Reading is one operation.
  - Writing is ~4 operations.

Analytics Queries

`most_freq_opening()`

- **Key**: `analytics:most_freq_opening`
- **Value**: A string representing the most used ECO code ever.
- **Example**: `analytics:most_freq_opening` → C65
- **Update**:
  - **E2**: initialize to empty or `""`.
  - **E1/E4**:

```
SET / INCR  global:opening:{eco}:count           → n
GET         analytics:most_freq_opening_count    → best       # None on first game
IF best is None OR n > best THEN
        SET analytics:most_freq_opening        {eco}
        SET analytics:most_freq_opening_count  n
```

- **Read**: A simple `GET` on the key to read it in one query.
- **Cost**:
  - On write, ?.
  - On read, one GET per query O(1).

`most_common_seq()`

- **Key**: `analytics:most_common_seq`
- **Value**: The 3-move sequence with the **highest** global count.
- **Example**: `analytics:most_common_seq` → `"e4 e5 Nf3"`
- **Update**:
  - **E1**:
    - after writing every `global:seq:{seq}:count`, run a one-off Python loop that scans those keys
    - finds the max count
    - `SET` the `analytics:most_common_seq` key.
  - **E2**: initialize to the empty string `""`.
  - **E4**: for every 3-move span extracted from a new game:
    1. `INCR global:seq:{seq}:count` → **`newCount`**
    2. `GET analytics:most_common_seq` → **`bestSeq`**
    3. `GET global:seq:{bestSeq}:count` → **`bestCount`**
    4. If **`newCount`** **>** **`bestCount`**, `SET analytics:most_common_seq {seq}`.

- **Read**: One `GET` operation on `analytics:most_common_seq`.
- **Cost**
  - **Writes**:
    - One `INCR` per span O(1)
    - Plus `GET` and `SET` when a new maximum appears.
  - **Reads**: one `GET` per query O(1)

`least_common_seq()` (Unstable and I don't like it)

- **Key**: `analytics:least_common_seq`
- **Value**: The 3-move sequence with the lowest positive global count.
- **Example**: `analytics:least_common_seq` → `"a2 a3 h6"`
- **Update**:
  - **E1**:
    - after writing every `global:seq:{seq}:count`, run a one-off Python loop that scans those keys
    - finds the minimum non-zero count
    - `SET` the `analytics:least_common_seq` key.
  - **E2**: initialize to the empty string `""`.
  - **E4**: for every 3-move span extracted from a new game:

```
new_count = INCR global:seq:{seq}:count

least_seq   = GET analytics:least_common_seq
least_count = GET analytics:least_common_seq_count

# first ever value or a strictly smaller count
if least_count is None or new_count < int(least_count):
    MSET analytics:least_common_seq {seq} \
         analytics:least_common_seq_count {new_count}

# brand-new sequence (new_count == 1) is always the new minimum
elif new_count == 1:
    MSET analytics:least_common_seq {seq} \ analytics:least_common_seq_count 1
```

- **Read**: One `GET` operation on `analytics:least_common_seq`.
- **Cost**
  - **Writes**:
    - One `INCR` per span O(1)

        - Plus `GET` and `SET` when a new minimum appears.
    - **Reads**: one `GET` per query O(1)

Leaderboard Queries

`top_10()`

- **Key family**: `analytics:top_wins`
- **Value**: List of at most 10 `player_id` s, sorted descending by win count.
- **Example**: `["42","17",…,"5"]`
    - **E1**: load-time script sorts all players by wins, then `RPUSH` the first 10 into `analytics:top_wins` .
        - **E2**: `RPUSH` new PID at the tail with a 0 count.
        - **E4**: on each win:

```
wins = INCR player:{pid}:wins                               # W1
oldTuple = f"{pid}:{wins-1}"
LREM  analytics:top_wins 0 oldTuple                        # W2   remove stale copy
```

ids = LRANGE analytics:top_wins 0 -1 # R1 single read

find insert position (at most 10 comparisons)

pos = first i where wins > int(ids[i].split(':')[1])
newTuple = f"{pid}:{wins}"

if pos is None: # wins is now the smallest
RPUSH analytics:top_wins newTuple # W3
else:
LINSERT analytics:top_wins BEFORE ids[pos] newTuple # W3

if LLEN analytics:top_wins > 10:
RPOP analytics:top_wins # W4 trim list

- **Read**: Use `LRANGE` on `analytics:top_wins` and get it in one query.

- **Cost**:
  - Reading is 1 op (`LRANGE`)
  - Writing ≤ 4 ops (`INCR` , `LREM` , insert, optional `RPOP`).

`bottom_10()`

- **Key**: `analytics:bottom_losses`
- **Value**: List of the **10 highest-loss players**,.
- **Example**: `["99","8",…,"23"]`
- **Update**
  - **E1**: build initial list.
  - **E2**: `RPUSH` new PID with 0.
  - **E4**:

```
losses = INCR player:{pid}:losses                          # W1

oldTuple = f"{pid}:{losses-1}"
LREM  analytics:bottom_losses 0 oldTuple                   # W2

ids = LRANGE analytics:bottom_losses 0 -1                  # R1

pos = first i where losses > int(ids[i].split(':')[1])
newTuple = f"{pid}:{losses}"

if pos is None:
  RPUSH  analytics:bottom_losses newTuple                # W3
else:
  LINSERT analytics:bottom_losses BEFORE ids[pos] newTuple   # W3

if LLEN analytics:bottom_losses > 10:
  RPOP analytics:bottom_losses                           # W4
```

- **Read**: Use `LRANGE` on `analytics:bottom_losses` and get it in one query.
- **Cost**:
  - Reading is 1 op (`LRANGE`)
  - Writing ≤ 4 ops (`INCR` , `LREM` , insert, optional `RPOP`).

Game Queries

`player_seq(user, seq)` (Incorrect)

- **Key**: `player:{pid}:seq`
- **Value**: ...
- **Example**: Filter `"e4 e5 Nf3"`.
- **Update**: on E4, for each game's move list, take every consecutive span of three half-moves and `BF.ADD` to the filter.
- **Read**: Use `BF.EXISTS` with `player:{pid}:seq` in one call.
- **Cost**:
  - One `BF.EXISTS` per read query O(1).
  - Writes proportional to the number of moves per game.

`global_seq(seq)` (Incorrect)

- **Key**: `global:seq`
- **Value**: ...
- **Example**: `BF.EXISTS global:seq "e4 e5 Nf3"` → `1` (present) or `0` (absent)
- **Update**
  - **E1**: during initial load, for each three-move span do `BF.ADD global:seq seq`.
  - **E4**: on each new game, for each three-move span do `BF.ADD global:seq seq`.
- **Read**: One `BF.EXISTS` on `global:seq "{seq}"`
- **Cost**
  - One `BF.EXISTS` per query O(1).
  - Writes: one `BF.ADD` per three-move span.

Analytics Queries

`shortest_game()`

- **Key**: `analytics:shortest_game`
- **Value**: Game-id of the record holder for fewest turns.
- **Example**: `"g17"`
- **Update**:
  - On **E1**, set to the shortest from the load.
  - On **E4**, compare the new game's turn count and `SET` if lower.
- **Read**: Simple `GET` on `analytics:shortest_game`.
- **Cost**:

    - One comparison + possible write per game.
    - One O(1) read per query.

`check_counts(game_id)`

- **Key**: `analytics:check:{gid}`
- **Value**: Integer count of “+” symbols in that game’s moves.
- **Example**: `5`
- **Update**: On **E1/E4**, count the occurrences and `SET` the `analytics:check:{gid}` key.
- **Read**: Use `GET` with `analytics:check:{gid}`.
- **Cost**:
    - One write per game O(1)
    - One O(1) read per query.

Graph Queries

`friends_of_friends(user)`

- **Key**: `player:{pid}:opponents`
- **Value**: Set of every opponent the user has faced.
- **Example**: `{"17","99","103"}`
- **Update**:
    - On **E1**, populate from history
    - On **E2**, create empty set
    - On **E4**, two `SADD` calls (one per player).
- **Read**: To be written by the person working on Graph queries.
- **Cost**: To be written by the person working on Graph queries.

`filtered_friends_of_friends(user)`

- **Keys**: Same FoF set plus `player:{pid}:wins` counters.
- **Value**: FoF PIDs who have more wins than the user.
- **Example**: `{"17","103"}` after filtering.
- **Update**: No extra writes beyond wins increments on **E4**.
- **Read**: To be written by the person working on Graph queries.
- **Cost**: To be written by the person working on Graph queries.

`largest_connected_component()`

- **Key fabric**: All `player:{pid}:opponents` sets form the graph.
- **Value**: Computed on demand.
- **Update**: Only the usual `SADD` on **E4**.
- **Read**: To be written by the person working on Graph queries.
- **Cost**: To be written by the person working on Graph queries.