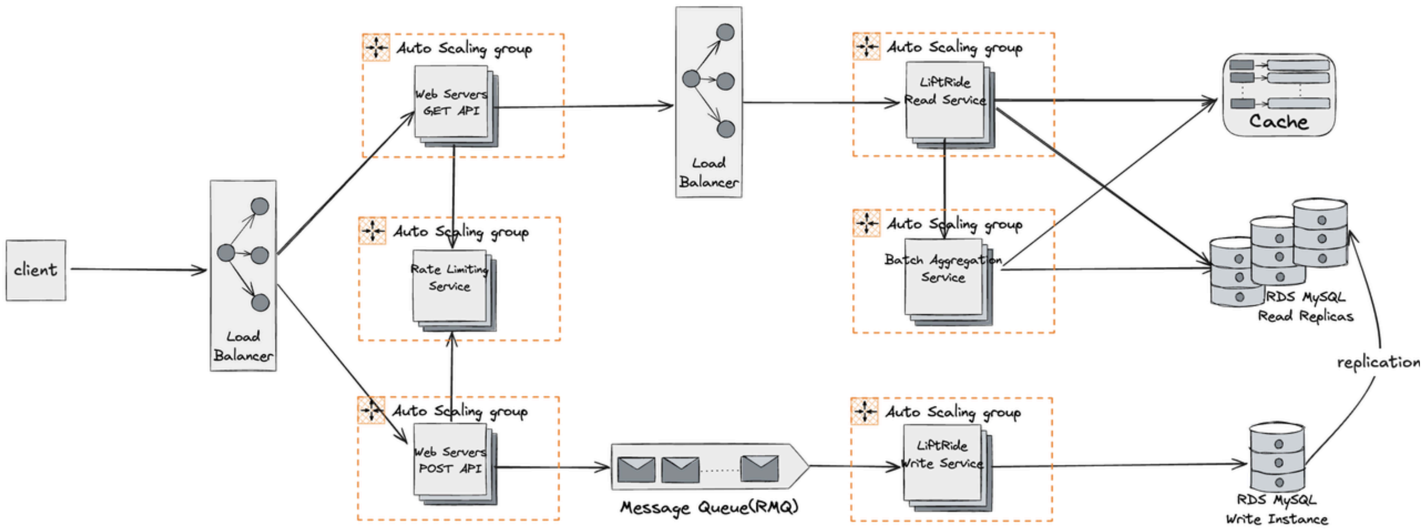
CS6650 Assignment4 Group Notion

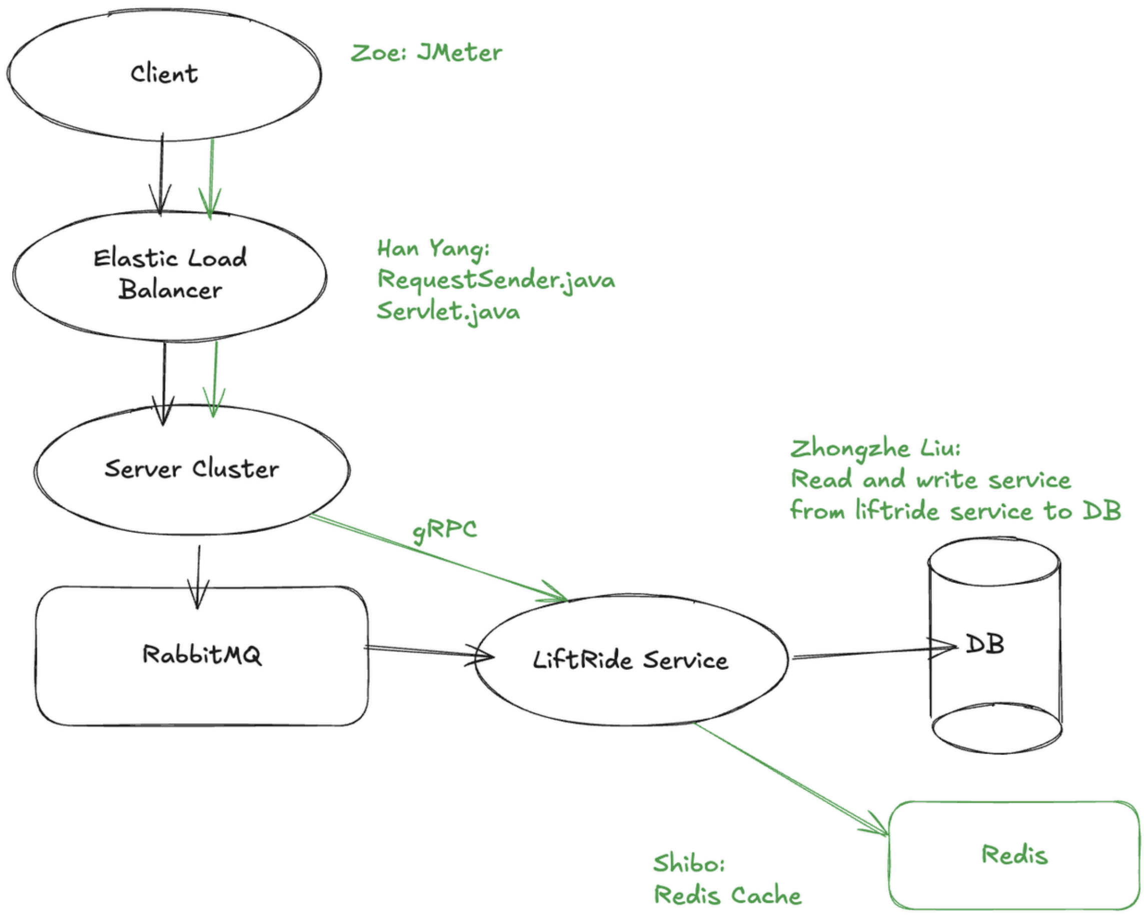
HY Han Yang S shibozheng ZL Zhongzhe Liu Modified Today

GitHub Repo: https://github.com/704500626/cs6650assignment4

System Architecture Final Version



System Architecture Early Version



Project Goal

- Requires high availability and responsiveness
- Can tolerate short-term cache staleness
- Read-heavy: reduces DB pressure
- Eventual consistency is acceptable as final state is resolved

Core Components

1. **Client**
    - Local multithreaded Java client sending 200K POST requests.
2. **Jmeter**
    - Used for load testing and validation of system throughput.
3. **Web Server + POST API (Write Servlet)**
    - Java servlet running on Tomcat, deployed on AWS EC2.
    - Performs request validation, rate limiting, queue depth monitoring.
    - Publishes to RabbitMQ using a round-robin load balancing strategy.
4. **RabbitMQ (Message Queue)**
    - Hosted on EC2, durable with multiple queues.
    - Enables decoupled, fault-tolerant ingestion.
    - Circuit breaker integrated to manage overload.
5. **Write Service (Consumer)**
    - Runs on EC2, consumes messages from RabbitMQ.
    - Uses batched DB insertions with retries and selective ACK/NACK logic.
    - Ensures at-least-once delivery while minimizing duplicates.
6. **MySQL (RDS)**
    - RDS Write instance + read replicas.
    - Holds normalized schema for lift rides, resorts, and seasons.
    - Optimized with indexes on `(resort_id, season_id, day_id)` and `(skier_id, resort_id, season_id, day_id)`.
7. **LiftRideReadService (gRPC)**
    - Serves GET queries via gRPC.
    - Combines Bloom filter + Redis cache + local LRU cache.
    - Prewarms and periodically syncs cache with Redis.
8. **Web Server + GET API (Read Servlet)**
    - Receives external HTTP requests.
    - Parses and forwards gRPC requests to `LiftRideReadService`.
    - Rate limiting and error handling integrated.
9. **Batch Aggregation Service (gRPC)**
    - Periodically aggregates DB rows into Redis and Bloom filters.
    - Supports 3 strategies: `FULL`, `BLOOM_ONLY`, `REFRESH_EXISTING_CACHE`.
    - Serves serialized Bloom filters to read services for fast membership checking.
10. **Rate Limiter Service (gRPC)**
    - Token bucket implementation (local/remote/Redis modes).
    - Shared by write and read servlets for fairness and protection.

Component-Level Design

SkierWriteServlet

- Validates URL and JSON.

- Uses `RateLimiter` and `waitForAcceptableQueueDepth()` to throttle load.
- Circuit breaker triggers if RabbitMQ queues exceed thresholds.
- Fault isolation ensures bad requests or downstream issues don t cascade.

RabbitMQPublisher

- Uses a pool of channels
- Round-robin distribution across multiple queues
- Periodically monitors queue depth with thread pool.
- Circuit breaker backed by live queue metrics.

LiftRideWriteService

- For each queue, spawns `M` consumer channels → `N x M` total DB workers.
- Batched insertions with retry/backoff + fallback to individual retries.
- Uses `basicAck` / `basicNack` on delivery tags to avoid message loss.
- Concurrent flushing via `ScheduledExecutor`.

LiftRideReadService

- Queries are served with three-layered caching:
  a. **Bloom Filter**: fast negative filtering.
  b. **Local LRU Cache**: fast in-memory lookups, refreshed from Redis.
  c. **Redis**: centralized in-memory store, refreshed by batch aggregation.
- Writes to Redis are offloaded to a worker thread with batching.
- Tracks metrics (cache hit rate, LRU hits, failures, etc.) using atomic counters.

BatchAggregationService

- Periodic strategies:
  - `BLOOM_ONLY`: only update Bloom filters from unique key queries.
  - `FULL`: update both Redis cache and Bloom filters (conditional on row count).
  - `REFRESH_EXISTING_CACHE`: refresh Redis for existing hot keys.
- Produces serialized + compressed Bloom filters served to read services.

Read Servlet (HTTP → gRPC Adapter)

- Parses RESTful URLs into structured requests.
- Handles retryable gRPC errors (e.g., RESOURCE_EXHAUSTED).
- Fully decoupled from read logic for easy testing and deployment.

RateLimiterService

- gRPC-based token bucket per logical group.
- Stateless client library with retries and exponential backoff.
- Pluggable modes: `LOCAL`, `REMOTE`, `REDIS`.

Aggregation Service (Batch Processing)

- gRPC server that periodically:
  - Updates Redis hot keys
  - Recomputes Bloom Filters
  - (Optional) Performs full DB aggregation if below threshold
- Publishes serialized bloom filters to consumers
- Compresses Bloom filters via GZIP before sending

Optimizations:

Summary

Component	Key Role	Core Strength
Write Servlet	Validates + publishes lift events	Circuit breaker + channel pooling + rate limiting
Write Service	Asynchronous, batched DB ingestion	Reliable batching, resilient error handling
Read Service	Handles queries via gRPC	Layered caching + bloom filters + metrics

Read Service

Feature	Description
Bloom Filters	Reduce unnecessary DB reads (0% false negative, <1% false positive)
Local LRU Cache	Fast hot key lookup, refreshed every few seconds
Cache Queue	Decouples reads from Redis writes (batch writes to Redis)
Prewarm LRU Cache	Preloads keys from Redis during boot
Metrics Collector	Tracks cache hit/miss/bloom filter stats
Thread-safe Design	Fully concurrent reads and cache writes
gRPC with Thread Pooling	Customizable thread pool for gRPC server
Read Strategies Encapsulated	Separate classes like TotalVerticalQuery, ResortDaySkiersQuery, etc.

Servlet Write:

Feature	Description
Channel Pooling	Uses RMQChannelPool with pre-created AMQP channels
Queue Monitoring	Background threads keep track of queue length, enabling adaptive control
Circuit Breaker	Dynamically rejects requests when queue depth exceeds a threshold
Backpressure Aware	Waits if queues are nearing overflow instead of dropping or blindly publishing
Rate Limiter Abstraction	Supports LOCAL, REMOTE (gRPC), or future Redis mode

Write Service:

Feature	Description
Batch Insert	Uses executeBatch + commit() for efficiency
Flush Timer	Scheduled executor flushes batch every X ms if idle
Reliable ACK/NACK	Manual RabbitMQ acknowledgment per event or batch
Retry with Backoff	All SQL failures retry up to 5 times
Thread/Channel Control	Highly tunable consumer threads via configuration
Backpressure-tolerant	Even in high failure or latency scenarios, events are retried or safely NACK'd

- Multi-tier cache strategy and read through cache
  - **Redis first, fallback to MySQL**
  - **Bloom filters** reduce unnecessary DB load
  - **Local LRU** handles most frequent queries (preloaded & refreshed)
  - **Asynchronous cache write queue** for decoupled persistence

a. LRU Cache (local in-memory)

Code block

```
localLRU.get(itemKey);
```

- Implemented as `LinkedHashMap` with `removeEldestEntry`
- Refreshed periodically via Redis `SCAN`
- Only enabled if `LIFTRIDE_READ_SERVICE_LRU_SWITCH` is true

b. Redis Cache

```
cache.getSync().get(itemKey);
```

- Keys follow structured naming conventions for predictable lookups
- Cache writes are **asynchronous**, queued in a `BlockingQueue<CacheWrite>`

c. Bloom Filters

Code block

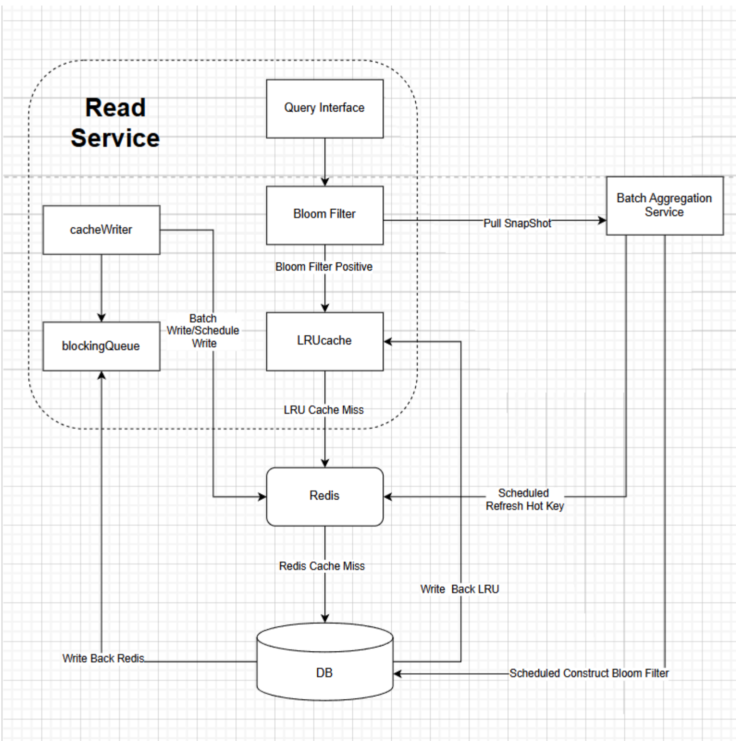
```
if (!bloomFilter.getUniqueSkiersFilter().mightContain(key)) → short-circuit DB
```

- False-negative-safe; skips DB query when possible
- Refreshed via gRPC call to the `BatchAggregationService`
- Stored in-memory, deserialized from compressed byte snapshots

- **Aggregation Service (Batch Processing)**
  - gRPC server periodically:
    - Updates Redis hot keys
    - Recomputes Bloom Filters
    - Performs full DB aggregation if below threshold
  - Publishes serialized bloom filters to consumers
  - Compresses Bloom filters via GZIP before sending

Read Request Workflow

1. Request comes in via servlet, gRPC call forwarded to `LiftRideReadServiceImpl`
2. `queryResortDaySkiers(...)` runs:
   - Check Bloom filter (fail-fast if negative)
   - Check Local LRU (superfast)
   - Check Redis
   - Fallback to MySQL DAO
3. Result is asynchronously queued for Redis caching
4. Respond to client via gRPC with a `SkierCountResponse`



Scalability, and Fault Tolerance

- **Write Path**:
  - Asynchronous RabbitMQ + multi-threaded consumers.
  - Batched DB inserts and pipelined flushing.
  - Adaptive retries with exponential backoff.
- **Read Path**:
  - Aggressive caching via Redis and LRU.
  - Bloom filters eliminate invalid read requests.
  - All caches are warmed and refreshed proactively.

Metrics and Observability

- Each request is logged with cache hit level.
- Periodic output of request count, cache hits, write failures.
- Error logs are informative for retry/fallback strategies.

Fault Isolation

- **Circuit breaker** isolates downstream RabbitMQ pressure.
- **Retry logic** in every component: DB, Redis, gRPC, and RabbitMQ.
- **Graceful shutdown hooks** for every long-lived component.

Configurability

- All concurrency and capacity values are tunable via `Configuration` class.

Database Fallback (MySQL via LiftRideReader)

On cache miss, queries are performed via LiftRideReader:

- getSkierDayVertical()
- getResortUniqueSkiers()
- getSkierResortTotals()

Example:

Code block

```
int vertical = dbReader.getSkierDayVertical(...);
```

Async Redis Cache Write

Cached values are **not** written synchronously. Instead, they are enqueued:

Code block

```
cacheQueue.offer(new CacheWrite(itemKey, value));
```

Processed in batch by CacheWriterWorker.

Background Workers

CacheWriterWorker

Dequeues from BlockingQueue<CacheWrite> and writes to Redis in batches:

- Queue capacity is configurable
- Batching interval is time or size based

LRU Refresher

Bloom Filter Refresher

Code block

```
ScheduledExecutorService bloomRefresher
```

- gRPC call to BatchAggregationService.getBloomFilterSnapshot
- Deserializes compressed Bloom filters
- Updates all four filters

Metrics Collector

Code block

```
ScheduledExecutorService metricsCollector
```

Prints statistics:

- Requests per hit level
- Cache write drops
- Total requests

Query Modules: Example – SkierDayRidesQuery

Code block

```
public CacheHitLevel querySkierDayRides(...) {
    if (bloomFilter != null && !bloomFilter.mightContain(key)) return BLOOM_NEGA
    if (localLRU.get(key) != null) return LRU_HIT;
    if (redis.get(key) != null) return REDIS_HIT;
    int val = dbReader.getSkierDayVertical(...);
    enqueue CacheWrite
    return DB_HIT;
}
```

This structure is **identical** in the other modules (ResortDaySkiersQuery, TotalVerticalQuery)

Cache Hit Tracking (CacheHitLevel)

Hit levels include:

- BLOOM_NEGATIVE
- LRU_HIT
- REDIS_HIT
- DB_HIT

They re logged and counted:

Code block

```
cacheStats.get(cacheHitLevel).incrementAndGet();
```

Deployment on AWS

1. Security Group:

Inbound rules (7)

Manage tags | Edit inbound rules

Search

Name	Security group rule ID	IP version	Type	Protocol	Port range	Source
–	sgr-04247a7de59a06bb5	IPv4	Custom TCP	TCP	8080	24.16.119.100/32
–	sgr-0f2e13f5dd47ea4ef	–	Custom TCP	TCP	6379	sg-07f8b902a8e7beb8c...
–	sgr-05254dc8447fe68fd	–	MYSQL/Aurora	TCP	3306	sg-07f8b902a8e7beb8c...
–	sgr-01f3f9e6d5bfc17c4	IPv4	SSH	TCP	22	24.16.119.100/32
–	sgr-0d38e9f81075e7aac	–	Custom TCP	TCP	8081	sg-07f8b902a8e7beb8c...
–	sgr-07304f50527acbdea	–	Custom TCP	TCP	5672	sg-07f8b902a8e7beb8c...
–	sgr-0815f0714347fa643	IPv4	Custom TCP	TCP	15672	24.16.119.100/32

2. Launched Instances:
    a. t2.large for servlet write
    b. t2.large for servlet read
    c. t2.large for read service
    d. t2.large for write service
    e. t2.medium for rabbit MQ
    f. t2.large medium for Redis
    g. t2.large for DB instance (MYSQL)
    h. t2.large for batch aggregation Service
    i. t2.medium for ratel imiter
    j. Subnet choose us-west-2b

Microservice Component

1. Client & Jmeter
2. Write Servlet
3. Read Servlet
4. Ratelimiter
5. ServerWriteService
6. DB read service
7. Cache read service
8. Batch aggregation Service

3. DB instance screenshot on EC2:

```
mysql> SELECT COUNT(*) FROM LiftRides;
+----------+
| COUNT(*) |
+----------+
|   600000 |
+----------+
1 row in set (0.08 sec)
```

Tuning

1. Under config.peroperties file, we distinguish between KNOB Major and Minor
2. This is how we tune our parameters to optimize performance

```java
public class Configuration {  Vinson Liu +2
    // RabbitMQ Configuration
    public String RABBITMQ_HOST = "localhost"; // RabbitMQ broker IP  4 usages
    public String RABBITMQ_USERNAME = "admin"; // username  3 usages
    public String RABBITMQ_PASSWORD = "admin"; // password  3 usages
    public String RABBITMQ_EXCHANGE_NAME = "skiers_exchange"; // Exchange name for the messages  6 usages
    public String RABBITMQ_ROUTING_KEY = "skiers.route"; // Routing key for the messages  1 usage
    public String RABBITMQ_QUEUE_NAME_PREFIX = "skiers_queue"; // Queue name for the messages  1 usage
    public int RABBITMQ_MAX_QUEUED_MSG = 5000; // KNOB: the maximal desired number of queued messages in the RMQ(all queues), if there are
     more messages than this threshold, the request will be asked to wait until the number queued messages fall under the
     threshold  4 usages
    public int RABBITMQ_CIRCUIT_BREAKER_THRESHOLD = 10000; // KNOB: the maximal allowed number of queued messages in the RMQ(all queues),
     if there are more messages than this threshold, circuit breaker will be turn on and new requests will be rejected  2 usages
    public int RABBITMQ_CIRCUIT_BREAKER_TIMEOUT_MS = 2000; // KNOB(minor): The number of milliseconds the circuit breaker will be
     on  2 usages
    public int RABBITMQ_QUEUE_MONITOR_THREAD_COUNT = 5; // KNOB(minor): the number or threads that are used to monitor the total number of
     queued messages  2 usages
    public int RABBITMQ_QUEUE_MONITOR_INTERVAL_MS = 100; // KNOB(minor): the time interval of checking the total number of queued
     messages  2 usages
    public int RABBITMQ_NUM_QUEUES = 10; // KNOB(major): the number of queues used on RMQ to store and process the messages. Since each
     queue in RMQ is single-threaded, more queues can improve concurrency. On the production side, messages will be evenly distributed to
     all N queues. On the consumption side, we assign each queue an equal number of consumers(channels).  5 usages
    public int RABBITMQ_PRODUCER_CHANNEL_POOL_SIZE = 100; // KNOB: the number of channels in a pool shared among write servlet threads,
     which is the producers of the messages to RMQ.  2 usages
    public int RABBITMQ_CONSUMER_NUM_CONNECTIONS = 1; // KNOB(major): the number of connections from the consumer to rabbitmq, for each
     connection we create the same number of channels for each queue. In practice, 1 seems ok.  2 usages
```

Database Design

Apis specified in docs and in assignment requirements:

1. GET/resorts/{resortID}/seasons/{seasonID}/day/{dayID}/skiers
    a. Get Number of unique skiers at resort/season/day
2. GET/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}
    a. Get total vertical for the skier for the specified sku day
3. GET/skiers/{skierID}/vertical
    a. Get the total vertical for skier of specified resort, if not specified, return all season vertical

Request body:

1. LiftRide: { lift_id, time }
2. resortID
3. seasonID
4. dayID
5. skierID

Each resort can have multiple seaons

Each resort has mutiple lifts

Each skier can go to multiple resorts in any days in any seasons and take mutiple lifts

Resort : Season → 1 to N

Resort : Lift → 1 to N

Skier: LiftRide → 1 to N

LiftRide : Lift → N to 1

Database Schema: Table Descriptions & Constraints

1. `resorts`

Code block

```sql
CREATE TABLE IF NOT EXISTS Resorts (
    resort_id INT PRIMARY KEY,
    resort_name VARCHAR(255) NOT NULL
);
```

---

2. `lifts`

Code block

```sql
CREATE TABLE IF NOT EXISTS Lifts (
    lift_id SMALLINT,
    resort_id INT NOT NULL,
    PRIMARY KEY (resort_id, lift_id),
    FOREIGN KEY (resort_id) REFERENCES Resorts(resort_id)
);
```

- Composite primary key ensures each lift is unique within a resort.
- Enforces that all `resort_id` values must exist in the `resorts` table.

---

3. `skiers`

Code block

```sql
CREATE TABLE IF NOT EXISTS Skiers (
    skier_id INT PRIMARY KEY
);
```

- Unique skier IDs.

---

4. `seasons`

Code block

```sql
CREATE TABLE IF NOT EXISTS Seasons (
     season_id CHAR(4),
     resort_id INT NOT NULL,
     PRIMARY KEY (resort_id, season_id),
     FOREIGN KEY (resort_id) REFERENCES Resorts(resort_id)
);
```

- Composite key ensures one entry per resort per season.

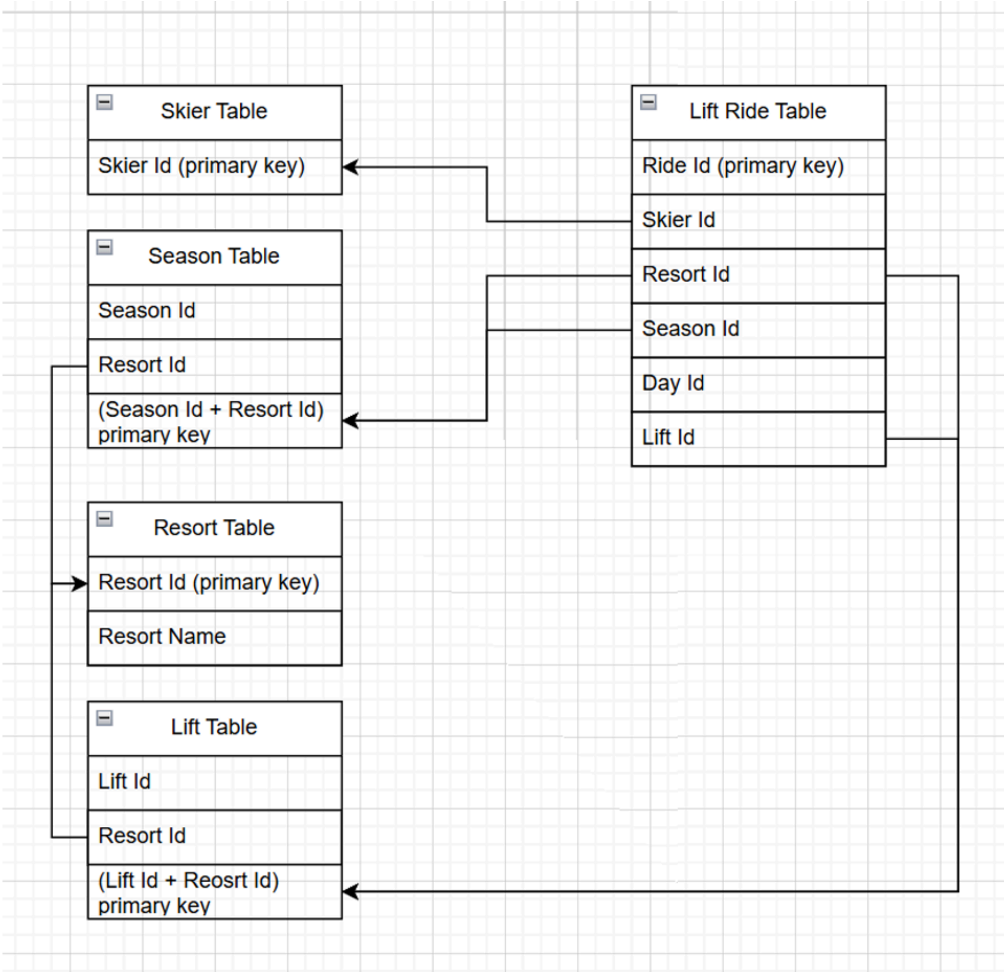
---

5. `lift_ride`

Code block

```sql
CREATE TABLE IF NOT EXISTS LiftRides (
    ride_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    skier_id INT NOT NULL,
    resort_id INT NOT NULL,
    season_id CHAR(4) NOT NULL,
    day_id SMALLINT NOT NULL,
    lift_id SMALLINT NOT NULL,
    ride_time SMALLINT NOT NULL, -- representing minutes or seconds from day sta
    FOREIGN KEY (skier_id) REFERENCES Skiers(skier_id),
    FOREIGN KEY (resort_id, lift_id) REFERENCES Lifts(resort_id, lift_id),
    FOREIGN KEY (resort_id, season_id) REFERENCES Seasons(resort_id, season_id)
);
```

- **`lift_ride`** is the central fact table capturing each skier s activity.
- Includes detailed fields like `day_id`, `lift_time`, etc.
- Uses `SERIAL` primary key for unique ride records.



Trade off between SQL and NO SQL

Strong Data Integrity with Relationships

- The data model includes multiple interrelated entities: `resorts`, `lifts`, `skiers`, `seasons`, and `lift_ride`.
- Referential integrity is enforced through **foreign key constraints.**
- NoSQL solutions **don t support joins or relational constraints**, leading to data duplication or inconsistent references.

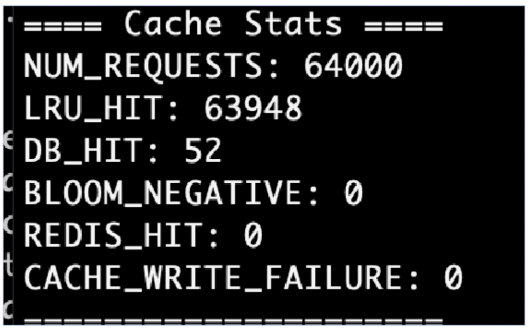
Complex Querying Requirements

- These require **grouping, filtering, and joining** across multiple tables — which are more efficient and expressive in SQL.
- NOSQL database can do complex queries by assigning partition key + sort key
- Still not effective, not scalable, and redundant data

Result:

getUniqueSkier api, most of the read requests were cached by LRU cache

10 resorts * 1 season * 3 days, total 30 keys, LRU works as expected!



getSingleVertical api, most of the read requests were filtered by bloom filters

```
==== Cache Stats ====
NUM_REQUESTS: 64000
BLOOM_NEGATIVE: 52297
REDIS_HIT: 16
LRU_HIT: 137
DB_HIT: 11550
CACHE_WRITE_FAILURE: 0
=====================
```

getTotalVertical api, most of the read requests were filtered by bloom filters

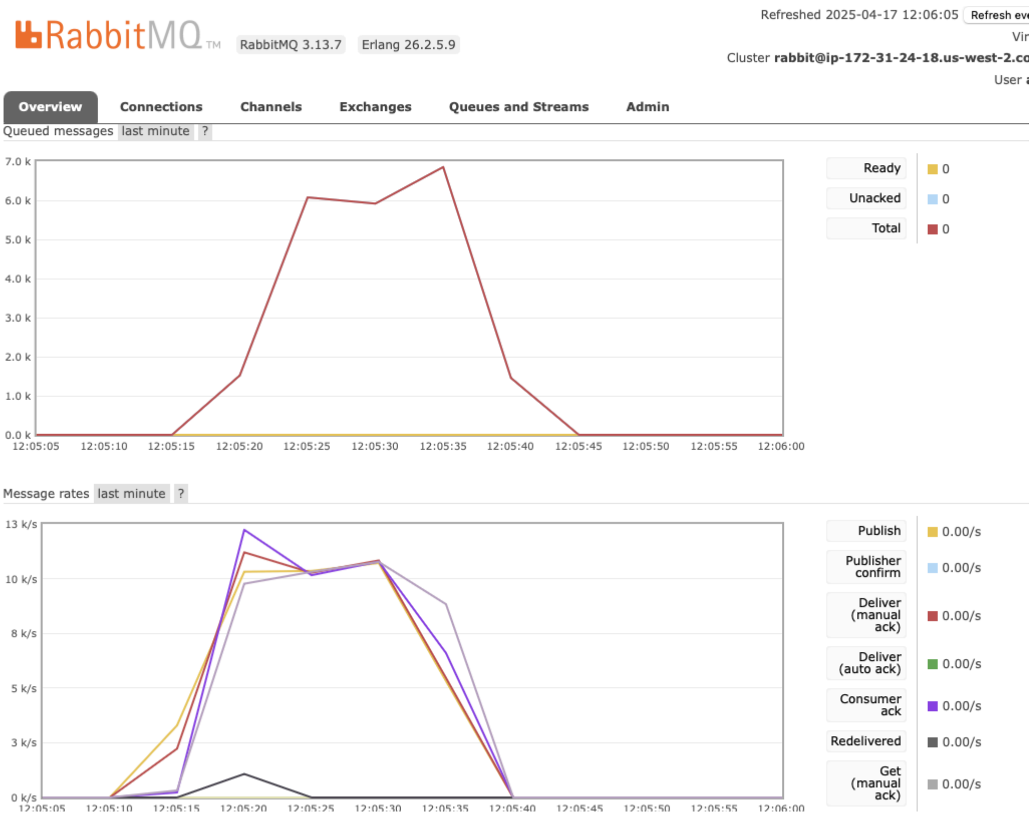
Less random skier_ids than the last api, but most requests are still filtered by bloom filters, as expected

```
==== Cache Stats ====
NUM_REQUESTS: 64000
LRU_HIT: 290
DB_HIT: 28331
BLOOM_NEGATIVE: 35196
REDIS_HIT: 183
CACHE_WRITE_FAILURE: 0
=====================
```

Client:

```
Server URL: http://52.89.73.132:8080/cs6650Server_war
Total successful request for Phase 1: 26955
Total unsuccessful request for Phase 1: 0
Phase 1 completed with 3284 ms
Phase 1 Avg Response time: 115.38133926915229 ms
Phase 1 Throughput: 8207.978075517662 req/s
Total successful request: 200000
Total unsuccessful request: 0
Total time to send 200000 requests: 18894 ms
Total Avg Response time: 92.447995 ms
Phase 2 Avg Response time: 88.87569707301569 ms
Phase 2 Throughput 11086.232301877122 req/s
Total Throughput 10585.371017254156 req/s
Number of threads in phase 1 responsible for sending requests: 32
Number of threads in phase 2 responsible for sending requests: 1
Number of threads in thread pool responsible for receiving requests: 1000
Mean response time: 92.447995 ms
Median response time: 83.0 ms
Min response time: 39.0 ms
Max response time: 837.0 ms
P99 response time: 321.0 ms
```

RabbitMQ



Jmeter Result:

getAllVertical:

Test and Report information

Source file	"results.jtl"
Start Time	"4/17/25, 11:01 PM"
End Time	"4/17/25, 11:02 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	HTTP Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**64000**	**0**	**0.00%**	**18.94**	**0**	**133**	**11.00**	**31.00**	**38.00**	**58.00**	**3795.52**	**754.14**	**704.19**
HTTP Request	64000	0	0.00%	18.94	0	133	11.00	31.00	38.00	58.00	3795.52	754.14	704.19

Errors

Type of error	Number of errors	% in errors	% in all samples

getSingleVertical

Test and Report information

Source file	"results.jtl"
Start Time	"4/17/25, 11:04 PM"
End Time	"4/17/25, 11:04 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	HTTP Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**64000**	**0**	**0.00%**	**8.61**	**0**	**121**	**4.00**	**19.00**	**25.00**	**40.00**	**4716.63**	**881.27**	**861.29**
HTTP Request	64000	0	0.00%	8.61	0	121	4.00	19.00	25.00	40.00	4716.63	881.27	861.29

Errors

Type of error	Number of errors	% in errors	% in all samples

getUniqueSkier

Test and Report information

Source file	"results.jtl"
Start Time	"4/17/25, 10:55 PM"
End Time	"4/17/25, 10:55 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	HTTP Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**64000**	**0**	**0.00%**	**11.83**	**0**	**1093**	**3.00**	**16.00**	**22.00**	**46.00**	**4607.96**	**863.99**	**814.94**
HTTP Request	64000	0	0.00%	11.83	0	1093	3.00	16.00	22.00	46.00	4607.96	863.99	814.94

Errors

Type of error	Number of errors	% in errors	% in all samples

Future Improvement:

- **Introduce Distributed Redis Cache Service**
- **Dynamic Hot Key Detection in Redis**
- **Cache Invalidation Strategy**
- **Hybrid Database Architecture**
- **Write-Behind Strategy for Cache Updates**
  Implement a write-behind approach where updates go to the database first, followed by asynchronous cache refresh to ensure durability.
- **Infrastructure Enhancements**
  Build observability stack (logging, metrics, tracing), and integrate long-term data storage via data warehouse or lake with ETL pipelines.