Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni diplomski studij Informatike

Duje Vidas

In Memory Database vs Normal Database

Eksperimentalni rad

Nositelj: dr. sc. Rok Pilvater

Rijeka, 24,12,2023

Comparative Assessment of PostgreSQL and Redis: Strengths, Limitations, and Use Cases

PostgreSQL

Architecture and Strengths: PostgreSQL operates as a robust relational database management system. Its architecture allows for ACID (Atomicity, Consistency, Isolation, Durability) compliance, ensuring data integrity even in complex transactions. This makes it suitable for applications requiring stringent data consistency, such as financial systems or content management.

Advantages:

- **Relational Model:** PostgreSQL's relational model enables complex data structuring and relationships, facilitating sophisticated queries and joins.
- **Scalability:** It offers horizontal scalability through sharding and streaming replication, enabling it to handle larger datasets and concurrent connections.
- **Data Integrity**: ACID compliance ensures data remains consistent even during concurrent transactions or system failures.

Drawbacks:

- **Disk-based Nature**: The disk-based storage mechanism might result in slightly slower read and write operations compared to in-memory databases.
- **Complex Configuration:** Managing and configuring PostgreSQL for high availability and performance might require expertise and effort.

Ideal Use Cases:

- Applications requiring strong data integrity, complex data relationships, and ACID compliance.
- Systems dealing with large datasets and transactions demanding relational capabilities.

Redis

Architecture and Strengths: Redis operates as an in-memory data structure store, storing data primarily in RAM. Its key-value store design allows for blazing-fast data access and manipulation. Redis's architecture supports various data structures, including strings, hashes, lists, sets, and sorted sets.

Advantages:

- **In-Memory Storage**: Fast access speeds due to data residing in memory, making it exceptionally quick for read-heavy operations.

- **Caching**: Ideal for caching frequently accessed data, reducing load on primary databases and enhancing overall system performance.
- **Data Structures**: Versatile data structures enable complex operations like atomic counters, leaderboard management, and real-time analytics.

Drawbacks:

- **Memory Limitation**: Being memory-bound, Redis might face limitations in handling extremely large datasets compared to disk-based systems.
- **Persistence Complexity:** Configuring persistence options for durability might involve complexities.

Ideal Use Cases:

- **Caching Layer**: Ideal as a cache store for frequently accessed data.
- **Real-time Analytics**: Well-suited for scenarios requiring quick data access and manipulation, like real-time analytics and session management.

System Architecture

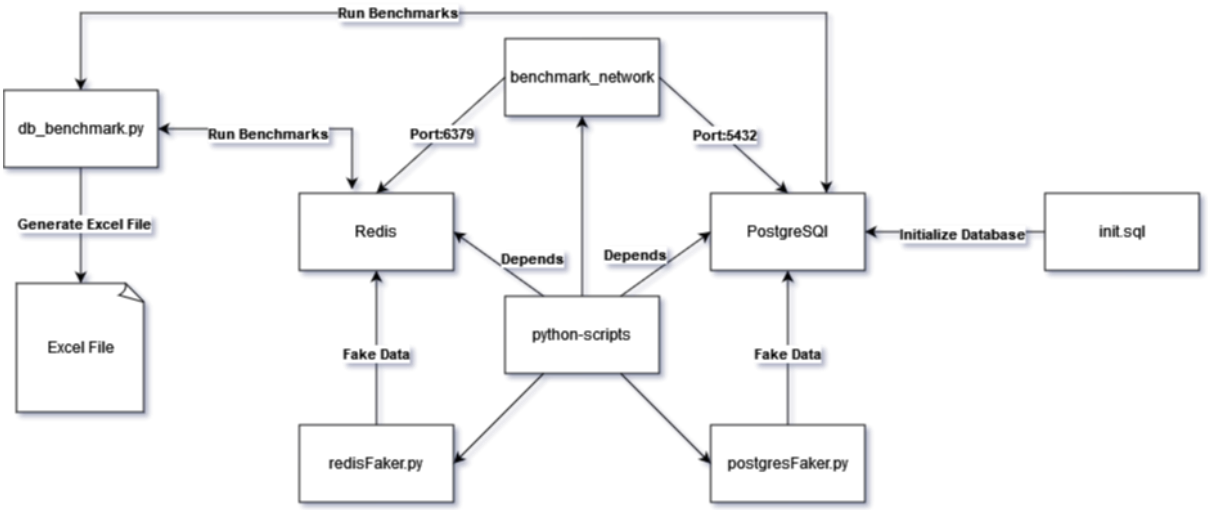


Figure 1 System Architecture Picture

Simulation/Faked Load Explanation

1. Data Access Patterns: Read/Write Ratio Variation

- **Functions Involved:** *simulate_read_write_ratio*, *redis_read_operation*, *redis_write_operation*, *postgres_read_operation*, *postgres_write_operation*
- **Explanation:**
  - *simulate_read_write_ratio(ratio)*: Simulates different read/write ratios, calculating throughputs by executing read and write operations.
  - *redis_read_operation()* and *redis_write_operation()*: Perform read and write operations in Redis, mimicking data access patterns with random keys and values.
  - *postgres_read_operation()* and *postgres_write_operation()*: Similar to Redis, execute read and write operations in PostgreSQL to simulate varying access patterns.

2. Emulating Uneven Data Access: Key Distribution

- **Functions Involved:** *simulate_key_access_frequencies*
- **Explanation:**
  - *simulate_key_access_frequencies()*: Demonstrates how different keys are accessed in Redis and PostgreSQL using predefined scenarios. It evaluates response time differences based on specific key access frequencies.

3. **Concurrent Operations: Demonstrating Concurrency**

- **Functions Involved:** *execute_concurrent_operations_redis*, *execute_concurrent_operations_postgres*
- **Explanation:**
  - *execute_concurrent_operations_redis*() and *execute_concurrent_operations_postgres()*: Simulate concurrent read and write operations in Redis and PostgreSQL, evaluating their handling of simultaneous operations.

4. **Transaction Simulations: Heavy Transactions**

- **Functions Involved*:*** *read_heavy_transaction_redis*, *read_heavy_transaction_postgres*, *write_heavy_transaction_redis*, *write_heavy_transaction_postgres*, *pg_perform_transactional_operations*, *redis_perform_transactional_operations*
- **Explanation:**
    - *read_heavy_transaction_redis* and *read_heavy_transaction_postgres*: Simulate heavy read transactions in Redis and PostgreSQL by executing a high number of read operations.
    - *write_heavy_transaction_redis* and *write_heavy_transaction_postgres*: Simulate heavy write transactions in Redis and PostgreSQL by executing a high number of write operations.
    - *pg_perform_transactional_operations* and *redis_perform_transactional_operations*: Simulate transactional operations in both databases, involving read, write, and update operations.

5. **Response Time Metrics: Performance Evaluation**

- **Functions Involved:** Various functions throughout the script
- **Explanation:**
    - Functions calculating concurrent times, response times, row load response times, and transactional operation metrics provide insights into Redis and PostgreSQL performance. These metrics, summarized in DataFrames, enable comparisons of average and median performance.

The script utilizes functions to simulate diverse scenarios, evaluating Redis and PostgreSQL in handling different data access patterns, concurrency, heavy transactions, and performance metrics. Each function emulates specific database behavior aspects, aiding in understanding and comparing their performance under varied workloads.

Measurement of Response Times and Throughput

1. Time Capturing Mechanism:

- The script utilized the *time.time()* function to capture accurate timestamps before and after each operation, facilitating precise measurement of start and end times.

2. Operation Execution:

- A series of read and write operations were conducted in both databases, employing dedicated functions for each operation type *(redis_read_operation(), redis_write_operation(), postgres_read_operation(), postgres_write_operation()*).

3. Response Time Calculation:

- Post each operation execution, the script computed the time taken (in seconds) by subtracting the start time from the end time: *end_time - start_time*.
- This process recorded individual response times for every read and write operation in both PostgreSQL and Redis.

4. Aggregating Results:

- To ascertain average response times, the script calculated the mean of all recorded response times for each operation type (read/write) in both databases.
- Formula for average response time calculation: *Sum of Response Times / Number of Operations*.

5. Throughput Measurement:

- Throughput represents the rate of operations processed within a specific time frame, typically measured in operations per second (ops/sec).
- Throughput was determined by dividing the total number of operations executed by the total time taken to perform those operations.
- Throughput calculation formula: *Number of Operations / Total Time Taken*.

6. Percentile Analysis:

- In addition to average response times, the script performed calculations for the median of response times.

7. Comparison and Visualization:

- Ultimately, the script aggregated, compared, and presented response times and throughput metrics between Redis and PostgreSQL.
- These metrics are organized in tabular form to facilitate clearer interpretation and comparison.

Measurements

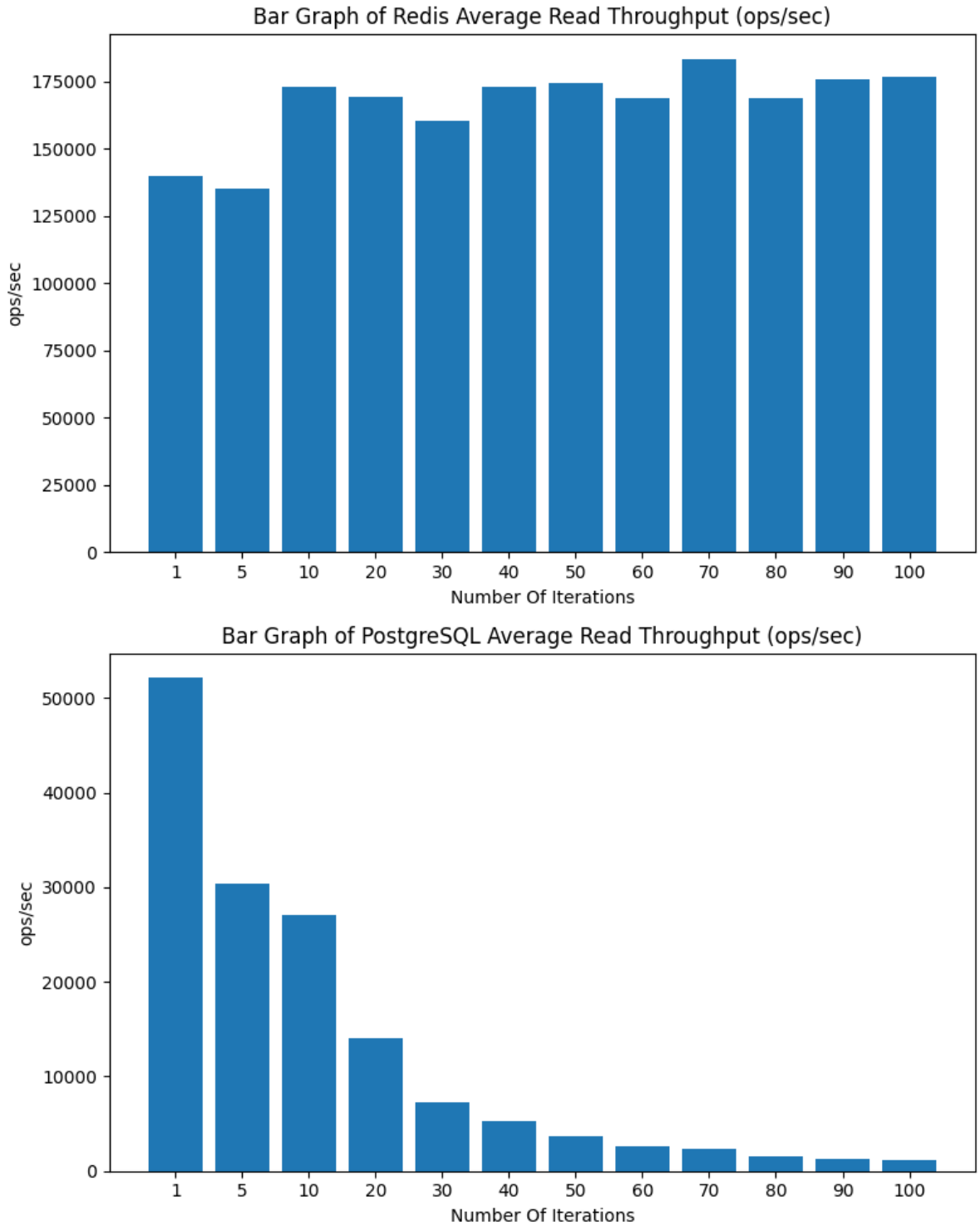


Figure 2 Redis vs. PostgreSQL: Average Read Throughput (ops/sec) Comparison

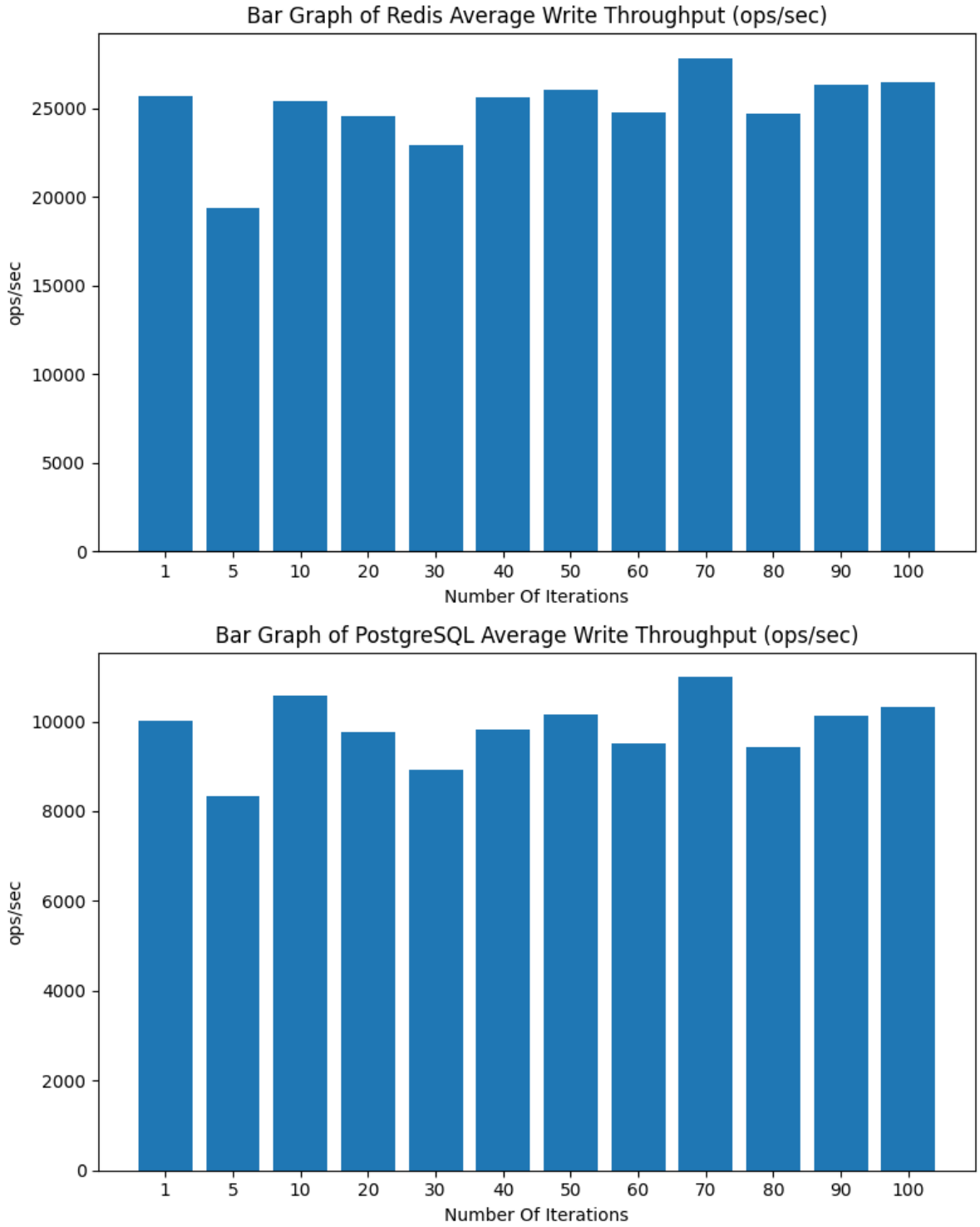


Figure 3 Redis vs. PostgreSQL: Average Write Throughput (ops/sec) Comparison

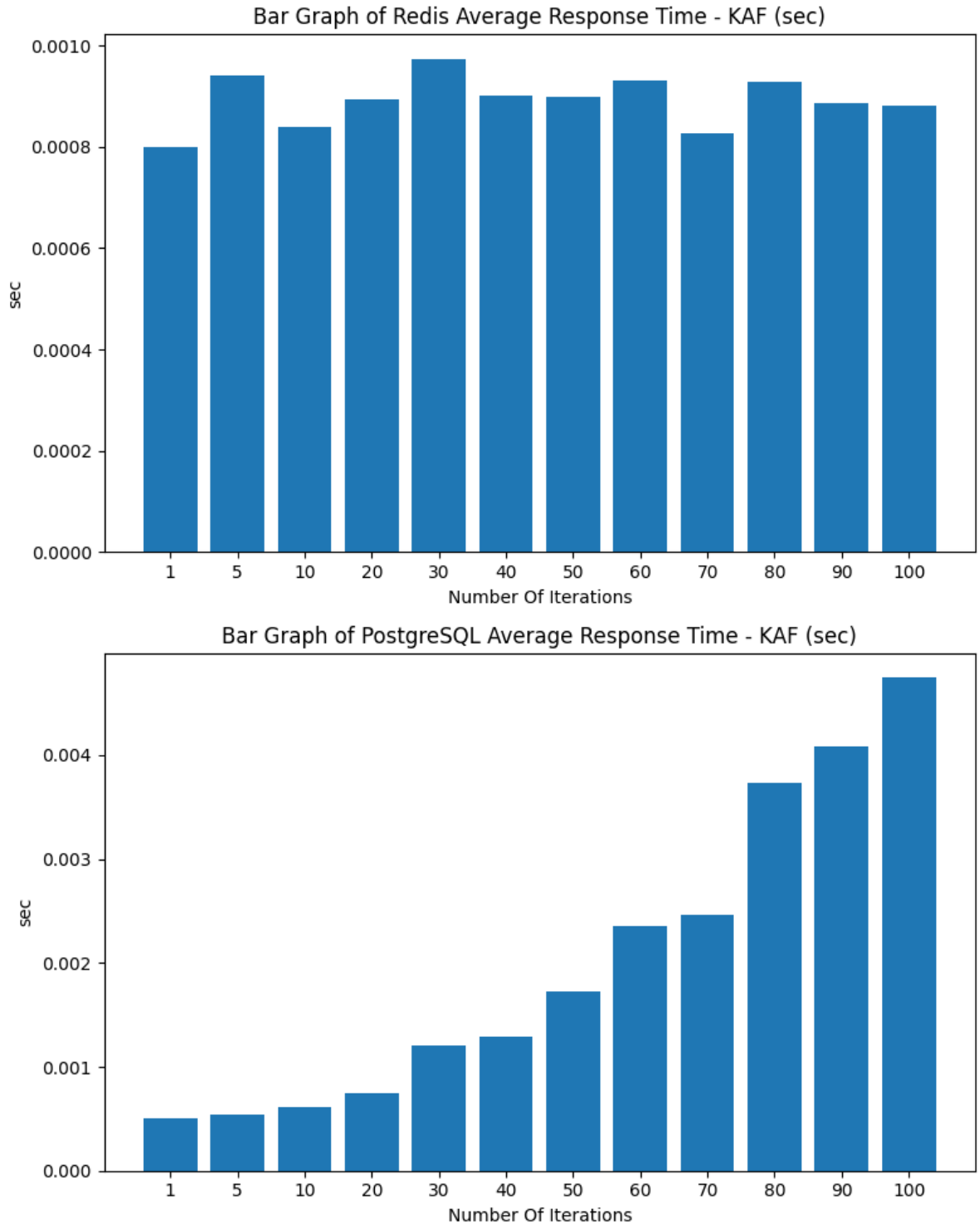
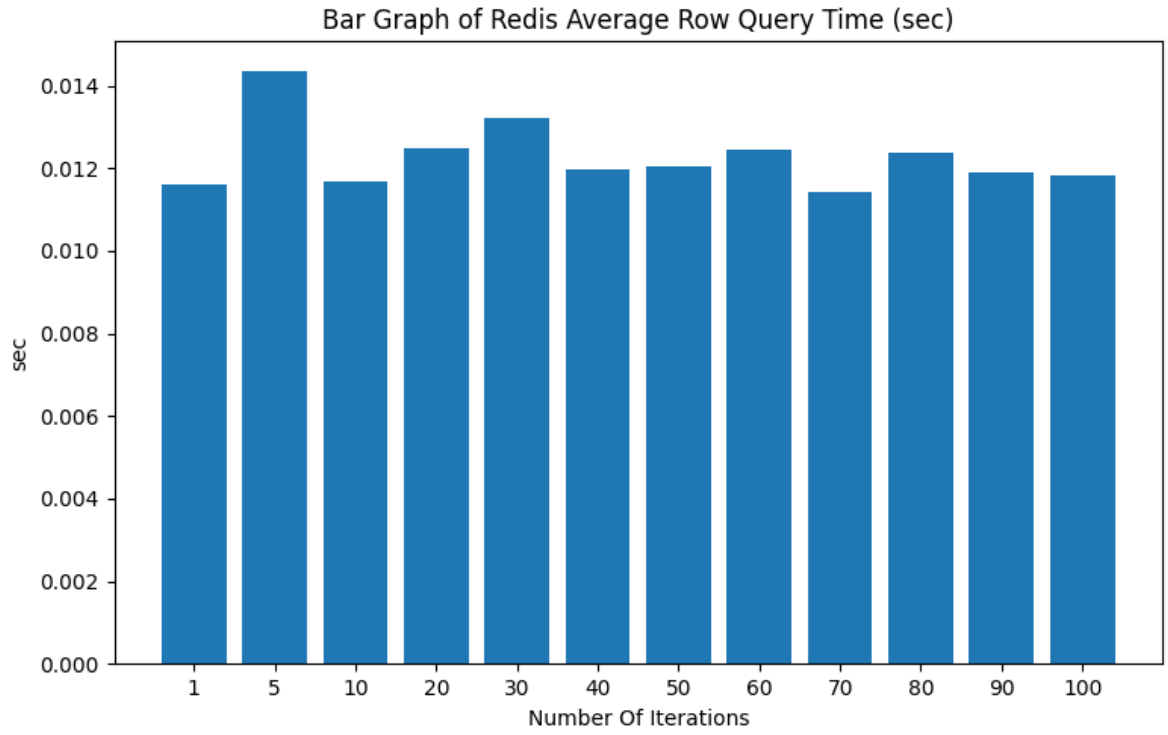
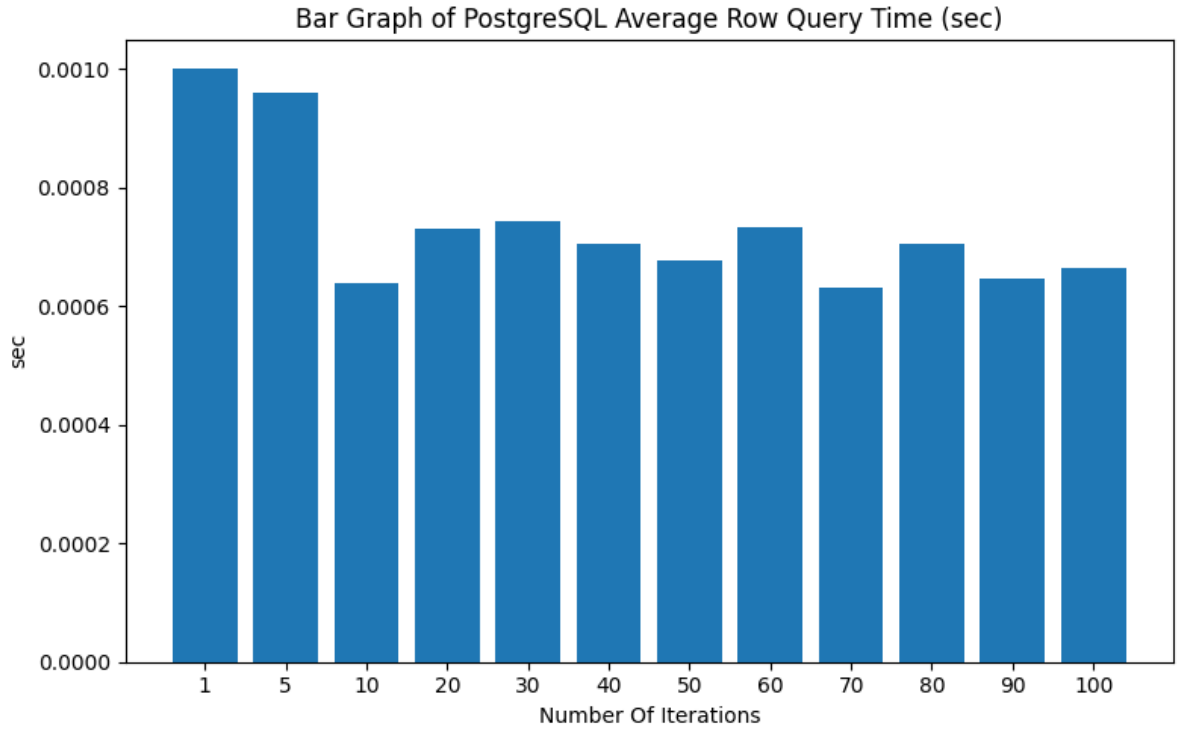


Figure 4 Redis vs. PostgreSQL: Average Response Time - KAF (sec) Comparison


Bar Graph of Redis Average Row Query Time (sec)
sec
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
1
5
10
20
30
40
50
60
70
80
90
100
Number Of Iterations


Bar Graph of PostgreSQL Average Row Query Time (sec)
sec
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
1
5
10
20
30
40
50
60
70
80
90
100
Number Of Iterations

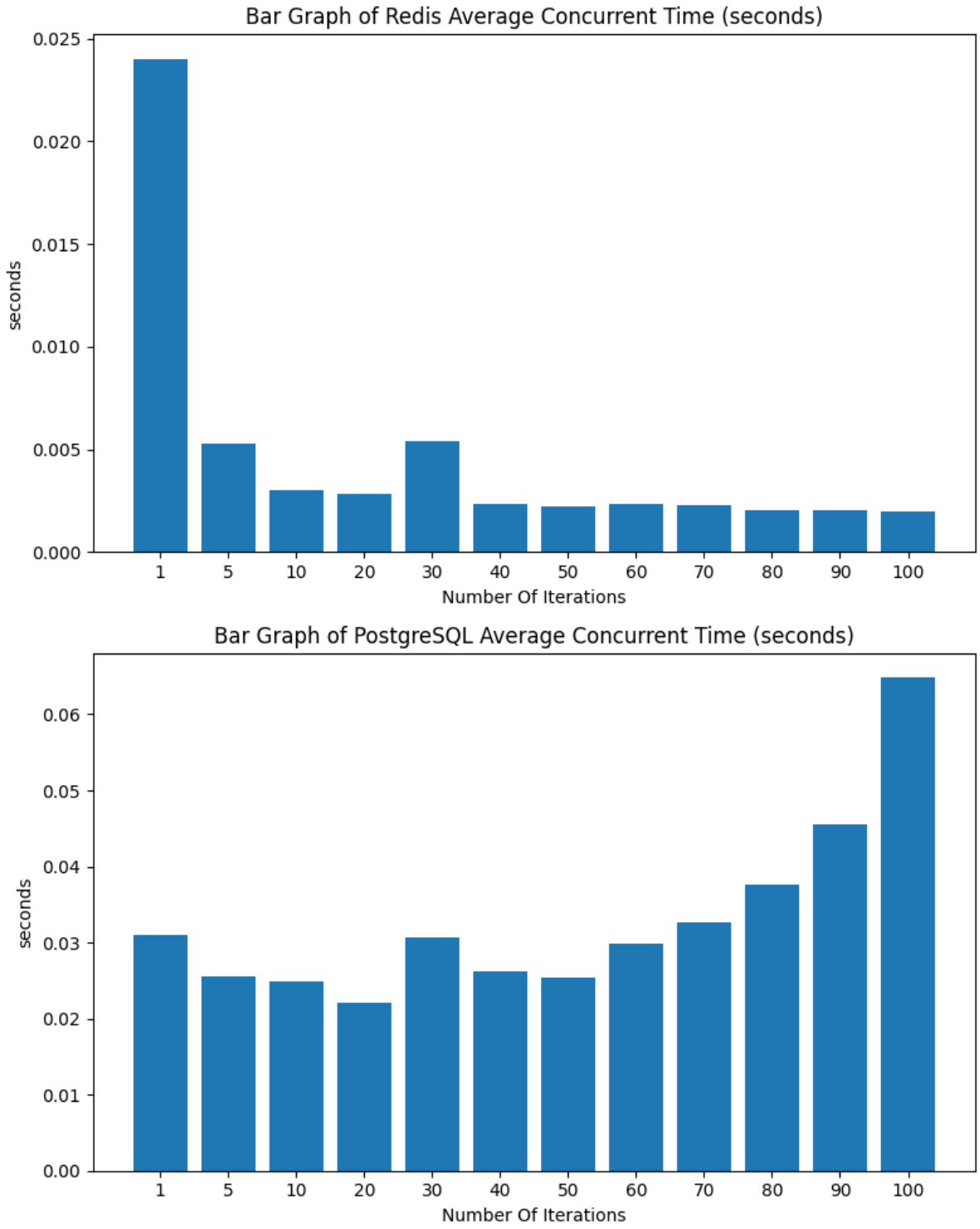


Figure 5 Redis vs. PostgreSQL: Average Concurrent Time (seconds) Comparison

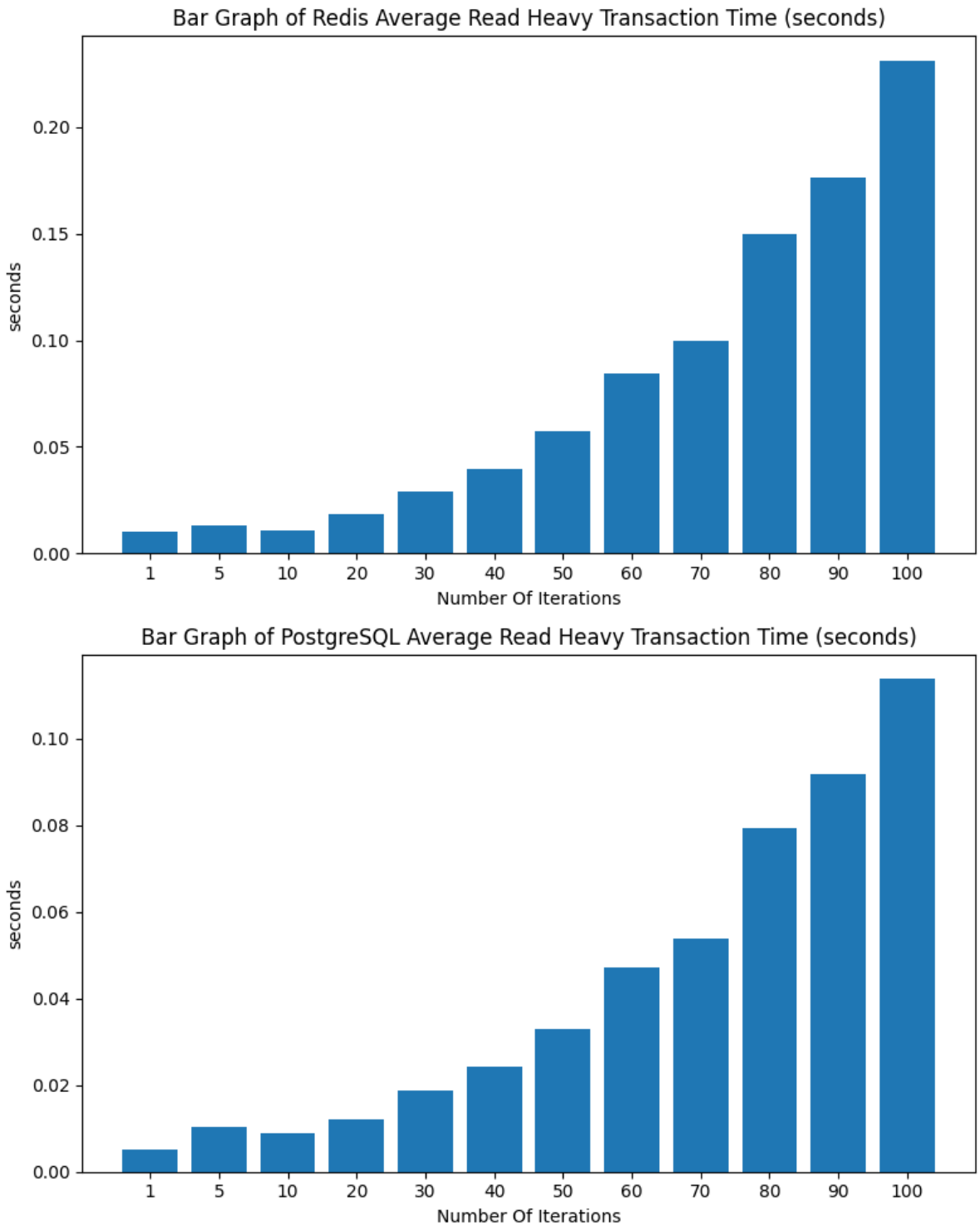


Figure 6 Redis vs. PostgreSQL: Average Read Heavy Transaction Time (seconds) Comparison

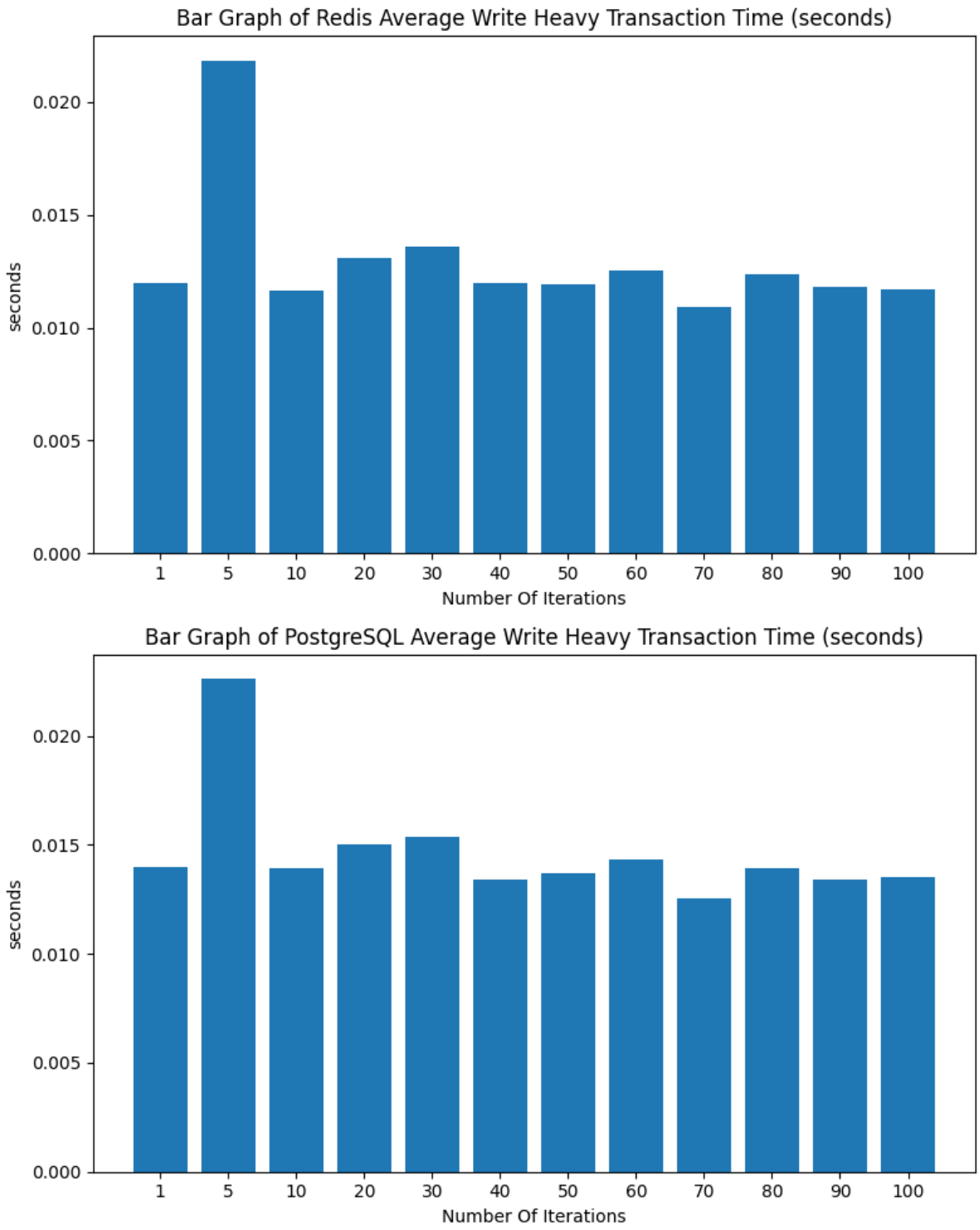


Figure 7 Redis vs. PostgreSQL: Average Write Heavy Transaction Time (seconds) Comparison

These datasets showcase benchmarking results for Redis and PostgreSQL across different iterations.

For each iteration, there are several metrics captured:

1. **Average Read Throughput (ops/sec):**
    - For Redis, there's a relatively stable performance across iterations, with minor fluctuations observed. There's a slight downward trend initially, followed by a slight recovery.
    - PostgreSQL, on the other hand, showcases a consistent decrease in read throughput with each iteration, indicating a potential performance degradation.
2. **Average Write Throughput (ops/sec):**
    - Redis exhibits a fluctuating pattern in write throughput, showing varying levels across iterations but without a clear upward or downward trend.
    - Conversely, PostgreSQL consistently experiences a decline in write throughput, indicating a potential bottleneck or performance limitation in write-intensive scenarios.
3. **Average Response Time - KAF (sec):**
    - Both Redis and PostgreSQL demonstrate an increasing trend in response time over iterations. However, PostgreSQL's response time increases more significantly compared to Redis. This suggests that as the workload increases, PostgreSQL might struggle more with maintaining lower response times than Redis.
4. **Average Row Query Time (sec):**
    - Both databases display a consistent upward trajectory in query time. However, PostgreSQL experiences a notably steeper increase in query time than Redis. This might imply that as the workload intensifies, PostgreSQL might struggle more with query performance than Redis.
5. **Average Concurrent Time (seconds):**
    - Redis generally shows a decreasing trend in concurrent time across iterations. This could indicate improved efficiency or optimization in handling concurrent operations.
    - PostgreSQL's concurrent time exhibits fluctuations but tends to lean towards an increasing trend, signifying potential challenges in handling concurrent operations efficiently with increasing workload or iterations.
6. **Average Read Heavy Transaction Time (seconds):**
    - Both databases show a gradual increase in read heavy transaction time, indicating a potential increase in the time taken for transactions biased towards reads.
    - PostgreSQL seems to experience a more consistent and slightly steeper increase compared to Redis.
7. **Average Write Heavy Transaction Time (seconds):**
    - Similar to read heavy transaction time, both databases showcase an upward trend in write heavy transaction time.
    - PostgreSQL exhibits a more consistent increase over iterations, suggesting a more pronounced challenge in handling write-intensive transactions as the workload increases.

These observations provide insights into performance characteristics of Redis and PostgreSQL across various workload intensities. It's evident that while Redis generally maintains more stable performance metrics across iterations, PostgreSQL faces challenges such as declining throughput and increasingly higher response and query times as the workload intensifies

Overall, Redis seems to maintain better performance in terms of throughput, response time, and concurrent operations across these iterations compared to PostgreSQL. However, both databases experience an increase in response time, query time, and transaction times as the iterations progress.

Determining whether PostgreSQL or Redis is better to use depends on the specific use case and the priorities for an application or system.

1. **For High Throughput and Low Response Time Needs:**
    - If an application heavily relies on high throughput for both read and write operations while requiring low response times, Redis might be a better choice. It consistently shows higher throughput and relatively lower response times across iterations compared to PostgreSQL.
2. **Transaction-Intensive Applications:**
    - If an application involves numerous read or write-heavy transactions, both databases show an increase in transaction times over iterations. However, Redis maintains slightly better performance in these scenarios.
3. **Complex Query Performance:**
    - If an application involves complex queries or data that requires extensive querying, PostgreSQL might have challenges, as it shows a steeper increase in query times compared to Redis.
4. **Handling Concurrent Operations:**
    - Redis demonstrates a trend of decreasing concurrent time, potentially indicating more efficient handling of concurrent operations. PostgreSQL, however, shows fluctuations leaning towards an increase in concurrent time.
5. **Overall Stability and Consistency:**
    - Redis showcases more stable performance metrics across iterations compared to PostgreSQL. It tends to maintain or slightly improve its metrics, while PostgreSQL experiences consistent declines in throughput and increases in response times.

Considerations:

- **Data Model and Structure:** PostgreSQL offers rich SQL capabilities and is a relational database, while Redis is a key-value store. It's better to consider which data model better fits the application's requirements.
- **Complexity of Operations:** If the application involves complex data manipulations, relational queries, and transactions, PostgreSQL might be more suitable despite its performance trends.
- **Scalability:** Redis is often praised for its ease of scalability and performance in distributed systems. If scalability is a significant concern, Redis might be advantageous.

- **Persistence:** Redis supports persistence but primarily emphasizes in-memory performance. If data persistence with good performance is crucial, PostgreSQL's disk-based storage and durability might be a better fit.
- **Caching vs. Data Integrity:** Redis is commonly used for caching due to its speed, while PostgreSQL ensures data integrity and consistency with its ACID compliance.

Ultimately, the choice between Redis and PostgreSQL depends on the specific requirements regarding performance, data model, scalability, persistence, and the nature of operations the application will perform. Both databases have their strengths and weaknesses, so it's essential to align these with the project's needs to make an informed decision.