- **Benchmark Usage Guide**

  Measure the performance of your Fake User Generator.

- **Quick Start**

  Run Full Benchmark Suite

  python benchmark.py full

  This runs all tests and saves results to benchmark_results.json.

  Output Example:

  ==========

==========

FAKE USER GENERATOR BENCHMARK

==========

==========

  Started at: 2024-12-10 15:30:45

  ==========

==========

=== TEST 1: QUICK PERFORMANCE TEST ===

Benchmarking: 100 users x 5 iterations

Locale: en_US, Seed: 12345

Iteration 1: 45.23ms | 2,211 users/sec | 100 users generated

Iteration 2: 43.87ms | 2,279 users/sec | 100 users generated

Iteration 3:   44.12ms | 2,266 users/sec | 100 users generated

Iteration 4:   43.95ms | 2,275 users/sec | 100 users generated

Iteration 5:   44.31ms | 2,257 users/sec | 100 users generated

SUMMARY STATISTICS

Average throughput: 2,258 users/sec

Peak throughput: 2,279 users/sec

...

- **Individual Tests**

  Quick Performance Test

  python benchmark.py quick

  Tests generation of 100 users over 10 iterations. Good for quick validation.

  Scaling Test

  python benchmark.py scaling

  Tests batch sizes: 1, 5, 10, 25, 50, 100, 250, 500, 1000 users.

  Output:

  SCALING COMPARISON

  Batch Size | Avg Time | Throughput | Time/User

  ------------+------------+----------------+-------------

  1 | 5.23ms | 191 u/s | 5.230ms

  5 | 12.45ms | 402 u/s | 2.490ms

10	18.67ms	536 u/s	1.867ms
25	35.12ms	712 u/s	1.405ms
50	62.34ms	802 u/s	1.247ms
100	118.45ms	844 u/s	1.185ms
250	287.23ms	870 u/s	1.149ms
500	565.12ms	885 u/s	1.130ms
1000	1,124.56ms	889 u/s	1.125ms

Findings: Time per user decreases as batch size increases (amortized overhead).

Locale Comparison

python benchmark.py locales

Compares performance across different locales.

Output:

LOCALE COMPARISON

Locale	Avg Time	Throughput
en_US	118.2ms	846 u/s
de_DE	121.5ms	823 u/s

Individual Function Benchmark

python benchmark.py functions

Measures performance of each stored procedure individually.

Output:

INDIVIDUAL FUNCTION BENCHMARK

prng_int | Avg: 0.045ms | Med: 0.043ms | 22,222 calls/sec

prng_float | Avg: 0.047ms | Med: 0.045ms | 21,277 calls/sec

prng_normal | Avg: 0.089ms | Med: 0.086ms | 11,236 calls/sec

prng_sphere_coords | Avg: 0.092ms | Med: 0.089ms | 10,870 calls/sec

generate_name | Avg: 2.345ms | Med: 2.312ms | 426 calls/sec

generate_address | Avg: 1.876ms | Med: 1.845ms | 533 calls/sec

generate_phone | Avg: 0.234ms | Med: 0.228ms | 4,274 calls/sec

generate_email | Avg: 0.567ms | Med: 0.551ms | 1,764 calls/sec

generate_physical_attributes | Avg: 1.123ms | Med: 1.098ms | 891 calls/sec

---

- **Understanding Results**

  Key Metrics

  Throughput (users/sec):

- How many users can be generated per second
- Higher is better
- Example: 2,000 users/sec means 1M users in ~8 minutes

  Latency (ms):

- Time to generate one batch
- Lower is better
- Example: 50ms for 100 users = 0.5ms per user

  Time/User:

- Average time to generate one user
- Decreases with larger batches (overhead amortization)

- Example: 1.2ms per user at batch size 100

Performance Expectations

Local Development:

- 1,000 - 3,000 users/sec
- 50-150ms for 100 users

Cloud Deployment (shared CPU):

- 500 - 1,500 users/sec
- 100-250ms for 100 users

Cloud Deployment (dedicated CPU):

- 2,000 - 5,000 users/sec
- 30-80ms for 100 users

- **Optimization Tips**

  If Performance is Slow

  1. Check Database Connection:

  -- Check for slow queries

  SELECT query, calls, mean_exec_time, max_exec_time

  FROM pg_stat_statements

  WHERE query LIKE '%generate_fake_users%'

  ORDER BY mean_exec_time DESC;

  2. Database Tuning:

  -- Increase work memory for complex queries

  SET work_mem = '64MB';

-- Check current settings

SHOW shared_buffers;

SHOW work_mem;

3. Index Optimization:

-- Ensure indexes exist

\d+ first_names

\d+ last_names

-- Add missing indexes

CREATE INDEX IF NOT EXISTS idx_first_names_locale_gender

  ON first_names(locale_code, gender);

4. Connection Pooling:

Use connection pooling in app.py

from psycopg2 import pool

connection_pool = psycopg2.pool.SimpleConnectionPool(

    1, 20,

    **DB_CONFIG

)

- **Comparing Hardware**

  Run benchmark on different systems:

Local Machine:

python benchmark.py quick > benchmark_local.txt

Cloud Server:

python benchmark.py quick > benchmark_cloud.txt

Compare:

Windows

Compare-Object (Get-Content benchmark_local.txt) (Get-Content benchmark_cloud.txt)

- **Sample Benchmark Results**

  High-Performance Setup

  (i7-10700K, 32GB RAM, NVMe SSD, PostgreSQL 14 local)

  Batch Size: 100 users

  Iterations: 10

  Average time: 42.3ms

  Throughput: 2,364 users/sec

  Time per user: 0.42ms

  Cloud Setup

  (Render $7/month, PostgreSQL shared)

  Batch Size: 100 users

  Iterations: 10

  Average time: 187.5ms

  Throughput: 533 users/sec

  Time per user: 1.88ms

Raspberry Pi 4

(4GB RAM, PostgreSQL 13)

Batch Size: 100 users

Iterations: 10

Average time: 342.8ms

Throughput: 292 users/sec

Time per user: 3.43ms

- **Continuous Benchmarking**

  Track Performance Over Time

  # Run benchmark daily

  $timestamp = Get-Date -Format "yyyy-MM-dd"

  python benchmark.py full

  Rename-Item benchmark_results.json "benchmark_$timestamp.json"

  Compare Results

  import json

  import glob

  files = sorted(glob.glob('benchmark_*.json'))

  print("Date | Throughput | Avg Latency")

  print("-----------+------------+------------")

```
for file in files:
    with open(file) as f:
        data = json.load(f)
        date = file.replace('benchmark_', '').replace('.json', '')
        throughput = data['quick_test']['avg_users_per_sec']
        latency = data['quick_test']['avg_time_ms']
        print(f"{date} | {throughput:>6.0f} u/s | {latency:>7.2f}ms")
```

- **Benchmarking Different Configurations**

  Test with Different Batch Sizes

```
# Custom benchmark
from benchmark import benchmark_generation

for size in [10, 50, 100, 500, 1000]:
    print(f"\n{'='*50}")
    print(f"Testing batch size: {size}")
    benchmark_generation('en_US', 12345, size, 5)
```

  Test with Different Locales

```
for locale in ['en_US', 'de_DE']:
    print(f"\n{'='*50}")
    print(f"Testing locale: {locale}")
    benchmark_generation(locale, 12345, 100, 10)
```

- **Interpreting Bottlenecks**

  Database is Bottleneck

  Symptoms:

- High latency even for small batches
- CPU usage low
- Network usage low

  Solutions:

- Upgrade database tier
- Enable connection pooling
- Optimize queries

  Network is Bottleneck

  Symptoms:

- Large difference between local and cloud benchmarks
- High latency variance

  Solutions:

- Deploy app closer to database (same region)
- Use internal networking (not public internet)
- Enable compression

  Application is Bottleneck

  Symptoms:

- High CPU usage
- Latency increases linearly with batch size

  Solutions:

- Scale horizontally (more instances)
- Upgrade CPU
- Enable caching

- **Production Monitoring**

  Add to Your App

  ```
  # In app.py
  import time

  @app.route('/api/generate', methods=['POST'])
  def api_generate():
      start = time.time()

      # ... existing code ...

      elapsed = time.time() - start
      app.logger.info(f"Generated {batch_size} users in {elapsed*1000:.2f}ms")

      return jsonify({
          'users': users,
          'generation_time_ms': elapsed * 1000,
          # ... other fields
      })
  ```

Dashboard Monitoring

Most platforms provide:

- Request duration
- Requests per second
- Database query time
- Error rate

Set up alerts for:

- Latency > 500ms
- Error rate > 1%
- Database CPU > 80%

- **Expected Results Summary**

Batch Size	Time (Local)	Time (Cloud)	Throughput
10	15-25ms	50-100ms	400-600 u/s
100	40-80ms	150-300ms	800-2000 u/s
1000	400-800ms	1-3s	1000-2500 u/s

Conclusion: Batch size 100-250 offers best balance of latency and throughput for most use cases.