PROCESSAMENTO DE IoT

stats.h – Este cabeçalho define a interface pública da biblioteca. Ele declara a estrutura `SensorSeries`, que contém o ponteiro para os dados numéricos, o tamanho `n` e os campos onde serão gravados `min`, `max` e `mean`. Também expõe o protótipo da função `compute_stats_batch`, permitindo que qualquer código (C, C++ ou Python via ctypes) saiba exatamente como invocá-la e como estão dispostos os dados em memória.

stats.c – É a implementação da lógica de alto desempenho. O arquivo obtém o número de núcleos disponíveis, cria um conjunto de *pthreads*, reparte o vetor de `SensorSeries` em blocos contíguos para cada thread e, dentro de cada bloco, percorre os valores de cada série para calcular mínimo, máximo e média. Os resultados são gravados diretamente nos campos da própria estrutura, evitando sincronização extra. Ao final, o código aguarda todas as threads com `pthread_join`, garantindo que o vetor esteja completamente preenchido antes de retornar ao chamador.

process_iot.py – Este script Python faz a orquestração de ponta a ponta. Ele lê o arquivo `devices.csv`, detecta o delimitador, normaliza carimbos de data/hora e valores numéricos, separa linhas regulares de payloads JSON, agrega os dados por (dispositivo × mês × sensor) e constrói um vetor `SensorSeries` compatível com a assinatura C. Em seguida carrega a biblioteca compartilhada criada a partir de `stats.c`, invoca `compute_stats_batch`, recupera as estatísticas preenchidas, completa combinações ausentes com zeros, monta um `DataFrame` ordenado e grava o relatório final em `resultado.csv`.

1. INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

1. Compile a biblioteca C que contém compute_stats_batch

   • Linux / WSL:

   ```
   gcc -O3 -fPIC -shared -pthread stats.c -o libstats.so
   ```

   • macOS (Clang):

   ```
   clang -O3 -fPIC -shared -pthread -dynamiclib stats.c -o libstats.dylib
   ```

2. Coloque o arquivo resultante na mesma pasta do script Python.
3. Instale as dependências do script: pip install numpy pandas.
4. Execute process_iot passando o CSV de entrada e o caminho de saída:

   ```
   python3 process_iot.py devices.csv resultado.csv
   ```

2. CARREGAMENTO DO CSV

O script lê o primeiro registro para descobrir o delimitador com csv.Sniffer, carrega o arquivo inteiro como texto e separa duas fontes de dados:

- linhas “normais”, já em formato tabular, entram em df_csv;
- linhas cujo campo *device* começa com “{” são tratadas como *payloads* JSON, parseadas por parse_json_rows() e colocadas em df_json.
  Em ambos os casos os carimbos de data/hora são convertidos para datetime em UTC e valores numéricos são normalizados (ponto decimal, float, remoção de “NULL”, “-”, etc.).

3. DISTRIBUIÇÃO DE CARGA ENTRE THREADS

Dentro da biblioteca C, o vetor SensorSeries contém uma série por **(dispositivo × mês × sensor)**. O programa calcula:

séries_totais = dispositivos × meses × sensores

O número de threads é o mínimo entre “CPUs disponíveis” e séries_totais. Cada thread recebe um intervalo contíguo de índices – não há critério semântico (por período ou dispositivo); o corte é simplesmente matemático:

```
size_t per   = total / nthreads;
size_t extra = total % nthreads;
```

A partilha é portanto uniforme, garantindo tamanhos quase iguais de trabalho por thread.

4. ANÁLISE DENTRO DAS THREADS

A função worker() percorre o seu bloco de séries. Para cada série:

1. Se n == 0, escreve 0 nos três campos-resultado.
2. Caso contrário percorre o vetor de amostras (já em memória contígua double[]) calculando:
   • min – menor valor da série;
   • max – maior valor;
   • mean – soma/ n.
   Essas estatísticas mensais são escritas de volta na mesma estrutura, portanto

cada thread grava apenas posições exclusivas, evitando contenção. Como nenhuma outra thread escreve nesses mesmos índices, não há disputa (ou "contenção") por área de memória compartilhada. Isso elimina a necessidade de mutexes ou outras travas de sincronização, porque cada thread só altera as suas próprias posições — não existe risco de duas threads tentarem escrever simultaneamente no mesmo campo.

5. GERACAO DO CSV

Depois que todas as threads terminam (pthread_join), o Python lê o vetor preenchido, transforma cada estrutura em um dicionário Python e cria um DataFrame. Faltas (pares *device/mês/sensor* sem dados) são completadas com zeros usando itertools.product. Por fim o script ordena por "ano-mês, dispositivo, sensor" e grava no arquivo de saída ("resultado.csv").

6. MODO DE EXECUÇÃO DAS THREADS

As threads criadas via **pthreads** são executadas **em modo usuário**. O sistema operacional só entra em cena para criar, escalonar e sincronizar (chamadas pthread_create, pthread_join, trocas de contexto). Depois disso, o código C que calcula min, max e mean roda inteiramente em espaço de usuário, sem privilégios de núcleo. Portanto, ainda que cada *pthread* corresponda a uma *kernel thread* na maioria dos Unix-likes, elas não "rodam em modo kernel"; apenas são gerenciadas pelo kernel.

7. POSSÍVEIS PROBLEMAS DE CONCORRÊNCIA

Chamadas simultâneas: se a mesma biblioteca for invocada por múltiplas threads Python ao mesmo tempo, cada instância criará seu próprio conjunto de *pthread*s. A função é reentrante, mas haverá competição por CPU.

Falsa partilha de cache: várias threads escrevem em elementos consecutivos de SensorSeries; dependendo do alinhamento, podem compartilhar a mesma *cache line* e degradar o desempenho.

Liberação de memória: array_refs em Python mantém os buffers vivos; esquecer essa lista levaria a *use-after-free*.

Carga desigual: se o número total de séries não for múltiplo do nº de threads, a diferença de 1 série entre algumas threads é mínima, mas com dados muito heterogêneos (séries vazias × séries longas) pode haver leve desequilíbrio