Equipe Repart-KV

Relatório Operacional Repart-KV

30 de janeiro de 2026

Equipe Repart-KV

Relatório Operacional Repart-KV

30 de janeiro de 2026

0.1 Introdução

Sistemas com estado local particionado estaticamente tendem a enfrentar desequilíbrios de carga e restrições de desempenho quando submetidos a padrões de acesso variáveis ou cargas de trabalho dinâmicas. A partir disso, foi desenvolvida a biblioteca Repart-KV, que busca explorar um rebalanceamento de baixo impacto, mantido o processamento eficiente e respeitando critérios de consistência preestabelecidos.

A biblioteca foi construída sobre o arcabouço das técnicas discutidas em [Luiz e Mendizabal 2024, Luiz e Mendizabal 2024, Luiz e Mendizabal 2025], e o documento descreve arquitetura, reconfiguração dinâmica e os experimentos que validam a abordagem.

Repart-KV foi projetada com extensibilidade em mente: além de suportar os backends permanentes documentados em 'storage/', a biblioteca permite adicionar rapidamente novos adaptadores para soluções comerciais de bancos de dados chave-valor, bastando implementar a interface pública e registrar o backend na CLI. Também fornecemos uma ferramenta de testes que automatiza a execução de workloads padronizados contra qualquer combinação de storage e configurações, facilitando a avaliação experimental em diferentes cenários.

Em termos de estratégia, a biblioteca implementa múltiplas estratégias de reparticionamento, e cada uma pode ser aplicada a qualquer backend que implemente as interfaces disponíveis. Os diferentes componentes da biblioteca contam com testes automatizados que validam o comportamento esperado e conferem maior confiabilidade quanto a corretude da implementação.

0.2 Metodologia

A implementação da biblioteca Repart-KV foi concebida como uma materialização das técnicas de reparticionamento dinâmico de estado apresentadas nos trabalhos publicados [Luiz e Mendizabal 2024, Luiz e Mendizabal 2024, Luiz e Mendizabal 2025]. Nesses estudos, foram propostas estratégias de balanceamento de carga baseadas em particionamento de grafos que permitem reconfigurar o esquema de partições durante a execução do sistema, sem interromper o processamento de requisições. A biblioteca foi projetada para encapsular essas técnicas em componentes modulares e extensíveis, possibilitando a sua aplicação em diferentes contextos e com diversos *backends* de armazenamento.

0.2.1 Linguagem e ferramentas

O desenvolvimento da biblioteca foi realizado na linguagem C++ utilizando o padrão C++20, que oferece recursos modernos de programação, como *concepts* para validação de tipos em tempo de compilação, e *templates* variádicos para a construção de componentes genéricos.

O sistema de compilação adotado foi o CMake (versão 3.20 ou superior), que permite a configuração multiplataforma e a integração com diversas bibliotecas externas. A biblioteca depende de componentes externos para funcionalidades específicas:

- **METIS** [Karypis e Kumar 1998]: biblioteca de particionamento de grafos utilizada para computar o esquema de partições a partir do grafo de carga de trabalho. METIS implementa heurísticas de corte mínimo em grafos multiníveis;

- **Tkrzw**: biblioteca de armazenamento chave-valor persistente de alto desempenho, utilizada como um dos *backends* de armazenamento. Oferece implementações baseadas em árvore B+ e tabelas de dispersão;
- **LMDB**: banco de dados embarcado baseado em árvore B+ com suporte a transações ACID, utilizado como alternativa de *backend* persistente;
- **TBB** (*Thread Building Blocks*): biblioteca da Intel para programação paralela, utilizada para estruturas de dados concorrentes como filas e mapas *thread-safe*;
- **unordered_dense**: biblioteca *header-only* que fornece implementações eficientes de mapas e conjuntos baseados em dispersão, utilizada para o armazenamento do grafo de carga de trabalho.

A arquitetura da biblioteca foi projetada utilizando *templates* C++ que permitem a composição de diferentes componentes. Isso possibilita, por exemplo, a combinação de diferentes bancos de dados chave valor com diferentes implementações de mapeamento de chaves, sem necessidade de modificar o código das estratégias de reparticionamento.

0.2.2 Testes automatizados

Para assegurar a corretude da implementação, os principais componentes da biblioteca contam com testes automatizados. Os testes foram desenvolvidos seguindo uma abordagem de verificação unitária, onde cada componente é testado isoladamente antes de ser integrado ao sistema. O sistema de compilação CMake configura automaticamente os executáveis de teste, que podem ser executados individualmente ou em conjunto através do *script* `run_tests.sh`.

0.2.3 Ferramenta de execução de experimentos

Além dos componentes da biblioteca, foi desenvolvida uma ferramenta de linha de comando (`repart-kv-runner`) que permite a execução de experimentos com diferentes configurações de armazenamento e estratégias de reparticionamento. A ferramenta lê arquivos de carga de trabalho em formato padronizado, executa as operações e coleta métricas de desempenho durante a execução.

0.3 Arquitetura e principais componentes

A biblioteca Repart-KV foi projetada de forma modular, com componentes que podem ser combinados de diferentes maneiras para atender a diferentes requisitos de armazenamento e reparticionamento. A arquitetura segue um fluxo onde operações são recebidas, executadas ou despachadas para o componente de armazenamento particionado, que por sua vez utiliza um ou mais motores de armazenamento subjacentes e, opcionalmente, rastreia os padrões de acesso para reconfiguração dinâmica das partições.

A seguir são descritos os principais componentes da biblioteca, e as estratégias de reparticionamento implementadas.

0.3.1 *Storage*

O componente *Storage* fornece a interface de armazenamento chave-valor utilizada pelas estratégias de reparticionamento. A implementação segue o padrão CRTP (*Curiously Recurring Template Pattern*), que permite polimorfismo em tempo de compilação sem o custo de funções virtuais. A classe base `StorageEngine` define os métodos `read`, `write` e `scan`, que são implementados pelas classes derivadas de acordo com o *backend* de armazenamento escolhido.

A operação `read` recebe uma chave e retorna o valor associado. A operação `write` recebe uma chave e um valor, armazenando o par no *backend*. A operação `scan` realiza uma varredura a partir de uma chave inicial, retornando até um número limite de pares chave-valor em ordem lexicográfica (para *backends* que suportam ordenação).

A classe base também fornece primitivas de travamento (`lock`, `unlock`, `lock_shared`, `unlock_shared`) que permitem controle manual de concorrência. As operações de leitura, escrita e varredura não adquirem travas automaticamente, delegando essa responsabilidade às camadas superiores da biblioteca.

As seguintes implementações de *StorageEngine* estão disponíveis:

- **MapStorageEngine**: armazenamento em memória utilizando `std::map`, com chaves ordenadas;
- **TkrzwHashStorageEngine**: armazenamento persistente baseado em tabela de dispersão, utilizando a biblioteca Tkrzw (HashDBM);
- **TkrzwTreeStorageEngine**: armazenamento persistente baseado em árvore B+, utilizando a biblioteca Tkrzw (TreeDBM), com chaves ordenadas;
- **LmdbStorageEngine**: armazenamento persistente utilizando LMDB, com suporte a transações ACID e chaves ordenadas;
- **TbbStorageEngine**: armazenamento em memória utilizando `tbb::concurrent_hash_map`, com suporte nativo a concorrência.

0.3.2 *Grafo*

O componente *Grafo* é responsável por representar o padrão de acesso à carga de trabalho e realizar o particionamento utilizando a biblioteca METIS. A representação do grafo de carga de trabalho segue a modelagem apresentada nos trabalhos publicados [Luiz e Mendizabal 2024, Luiz e Mendizabal 2025]: os vértices representam chaves acessadas, com pesos que acumulam a frequência de acesso, e as arestas representam co-acessos entre chaves, com pesos que acumulam a frequência de co-ocorrência em operações de varredura.

O grafo é implementado utilizando listas de adjacência baseadas em tabelas de dispersão. Essa estrutura garante complexidade $O(1)$ amortizada para operações de incremento de peso de vértices e arestas, que são as operações mais frequentes durante o rastreamento da carga de trabalho. Ao incrementar o peso de um vértice ou aresta, o elemento é criado caso não exista, ou tem seu peso incrementado caso já exista.

Para realizar o particionamento, o grafo é convertido para o formato CSR (*Compressed Sparse Row*) exigido pelo METIS, construindo os vetores de adjacência, os pesos dos vértices e os pesos das arestas. Também são mantidos mapeamentos bidirecionais entre os nomes das chaves e os índices inteiros utilizados pelo METIS.

O particionamento do grafo é realizado em um número especificado de partições. O resultado do particionamento é um vetor que associa cada vértice (chave) a uma partição, utilizado pelos componentes de KV-Store da biblioteca para atualizar seus esquemas de partição.

0.3.3 *KeyStorage*

O componente *KeyStorage* fornece estruturas de mapeamento utilizadas internamente pela biblioteca para associar chaves a valores de tipos integrais ou ponteiros. Diferentemente do *Storage*, que armazena pares chave-valor onde ambos são strings, o *KeyStorage* é utilizado para mapear chaves a índices de partições, ponteiros para instâncias de armazenamento, e outras informações de controle interno.

A interface segue o mesmo padrão CRTP utilizado pelo componente *Storage*, oferecendo polimorfismo em tempo de compilação. As operações disponíveis são: obter o valor associado a uma chave, inserir ou atualizar um par chave-valor, e buscar o primeiro elemento com chave não menor que uma chave especificada (operação de limite inferior). Esta última operação é essencial para a implementação de varreduras ordenadas nas estratégias de reparticionamento.

As seguintes implementações estão disponíveis:

- **MapKeyStorage**: implementação em memória utilizando `std::map`, com chaves ordenadas e suporte eficiente à operação de limite inferior;
- **TkrzwHashKeyStorage**: implementação persistente baseada em tabela de dispersão, utilizando a biblioteca Tkrzw;
- **TkrzwTreeKeyStorage**: implementação persistente baseada em árvore B+, utilizando a biblioteca Tkrzw, com chaves ordenadas;
- **LmdbKeyStorage**: implementação persistente utilizando LMDB, com chaves ordenadas;
- **UnorderedDenseKeyStorage**: implementação em memória baseada em tabela de dispersão, que constrói iteração ordenada coletando e ordenando as chaves sob demanda.

0.3.4 *KVStorage*

O componente *KVStorage* representa o núcleo da biblioteca Repart-KV, implementando as estratégias de armazenamento particionado com suporte a reparticionamento dinâmico. Este componente orquestra os demais módulos descritos anteriormente: utiliza instâncias de *Storage* como motores de armazenamento subjacentes, estruturas de *KeyStorage* para manter mapas de partições, e o componente de *Grafo* para rastrear padrões de acesso e computar novos esquemas de particionamento via METIS.

A arquitetura segue o mesmo padrão CRTP utilizado nos demais componentes. A classe base define as operações de leitura, escrita e varredura, que são delegadas às implementações

concretas. As estratégias de reparticionamento estendem essa interface adicionando funcionalidades de rastreamento de padrões de acesso, verificação de estado de reparticionamento em progresso, e acesso ao grafo de carga de trabalho.

A seguir são descritas as estratégias de reparticionamento implementadas, que diferem na forma como gerenciam a concorrência, a quantidade de instâncias de armazenamento, e o modelo de execução das operações.

0.3.4.1 *SoftThreadedRepartitioningKeyValueStorage*

Esta estratégia é a mais similar às propostas apresentadas nos trabalhos publicados [Luiz e Mendizabal 2024, Luiz e Mendizabal 2025]. Utiliza uma única instância de motor de armazenamento compartilhada entre múltiplas *threads* trabalhadoras, cada uma responsável por gerenciar uma partição lógica do armazenamento.

A estrutura principal é composta por: uma instância de motor de armazenamento, um mapa de partições que associa chaves a índices de partição, um conjunto de *threads* trabalhadoras (uma para cada partição), e uma *thread* de rastreamento que consome as chaves acessadas e constrói o grafo de carga de trabalho.

Quando uma operação é invocada, o mapa de partições é consultado para determinar qual trabalhador é responsável pela chave. Se a chave ainda não possui partição associada, uma partição é atribuída utilizando uma função de dispersão. A operação é então submetida à fila do trabalhador correspondente. A *thread* chamadora bloqueia até que a operação seja concluída, garantindo semântica síncrona para o cliente.

Cada trabalhador possui uma fila *lock-free* do tipo SPSC (*Single-Producer Single-Consumer*) implementada com a biblioteca Boost.Lockfree, e utiliza semáforos contadores para controlar a capacidade da fila e sinalizar disponibilidade de operações. Essa arquitetura objetiva baixa contenção entre produtores e consumidores, permitindo alto *throughput* de operações.

0.3.4.2 *SoftRepartitioningKeyValueStorage*

Esta estratégia é similar à anterior, porém sem utilizar *threads* trabalhadoras dedicadas. As operações são executadas diretamente pela *thread* chamadora, evitando a necessidade de instanciar objetos de operação e sincronizar a execução entre chamador e trabalhador.

A execução das operações ocorre em duas fases. Na primeira fase, o mapa de partições é bloqueado em modo de leitura (ou escrita, no caso de inserção de nova chave) e o índice da partição associada à chave é obtido. O bloqueio da partição correspondente é adquirido e o mapa de partições é liberado. Na segunda fase, a operação é executada sobre o motor de armazenamento e o bloqueio da partição é liberado.

Embora esta estratégia garanta que operações sobre a mesma chave não sejam executadas concorrentemente, como há uma única instância de motor de armazenamento, a execução ainda está sujeita a sincronizações inerentes ao motor de armazenamento que podem ocorrer mesmo ao acessar chaves diferentes.

A seguir são apresentados os algoritmos das operações de leitura e escrita. Seja M_p o mapa de partições, μ_m o *mutex* compartilhado do mapa, $P = \{\mu_0, \mu_1, \ldots, \mu_{n-1}\}$ o conjunto de *mutexes* das partições, S o motor de armazenamento, e $h : \mathcal{K} \to \mathbb{N}$ uma função de dispersão.

Algorithm 1 Operação de Leitura – *SoftRepartitioningKeyValueStorage*

1: **function** READ($k \in \mathcal{K}$)
2: LOCKSHARED(μ_m)
3: $p \leftarrow M_p[k]$
4: **if** $p =$ null **then**
5: UNLOCKSHARED(μ_m)
6: **return** NOT_FOUND
7: **end if**
8: LOCKSHARED(μ_p)
9: UNLOCKSHARED(μ_m)
10: $v \leftarrow S$.READ(k)
11: UNLOCKSHARED(μ_p)
12: **return** v
13: **end function**

Algorithm 2 Operação de Escrita – *SoftRepartitioningKeyValueStorage*

1: **function** WRITE($k \in \mathcal{K}, v \in \mathcal{V}$)
2: LOCK(μ_m)
3: $p \leftarrow M_p[k]$
4: **if** $p =$ null **then**
5: $p \leftarrow h(k) \bmod n$
6: $M_p[k] \leftarrow p$
7: **end if**
8: LOCK(μ_p)
9: UNLOCK(μ_m)
10: S.WRITE(k, v)
11: UNLOCK(μ_p)
12: **end function**

A operação de varredura (*scan*) apresenta maior complexidade por potencialmente acessar chaves em múltiplas partições. O algoritmo coleta as partições envolvidas e as bloqueia em ordem crescente de índice para evitar *deadlocks*. Seja L o limite de pares chave-valor a retornar.

Algorithm 3 Operação de Varredura – *SoftRepartitioningKeyValueStorage*

```
1: function SCAN(k_0 ∈ 𝒦, L ∈ ℕ)
2:     LOCKSHARED(μ_m)
3:     it ← M_p.LOWERBOUND(k_0)
4:     P_set ← ∅; K_arr ← []
5:     c ← 0
6:     while c < L ∧ ¬it.ISEND() do
7:         P_set ← P_set ∪ {it.VALUE()}
8:         K_arr.APPEND(it.KEY())
9:         it ← it.NEXT(); c ← c + 1
10:    end while
11:    P_sorted ← SORT(P_set)
12:    for all p ∈ P_sorted do
13:        LOCKSHARED(μ_p)
14:    end for
15:    UNLOCKSHARED(μ_m)
16:    R ← S.SCAN(k_0, L)
17:    for all p ∈ P_sorted do
18:        UNLOCKSHARED(μ_p)
19:    end for
20:    return R
21: end function
```

0.3.4.3 *HardRepartitioningKeyValueStorage*

Esta estratégia é similar à anterior no que diz respeito à execução das operações diretamente pela *thread* chamadora. Porém, diferentemente da estratégia *soft*, esta implementação utiliza uma instância de motor de armazenamento dedicada para cada partição, possibilitando um particionamento físico dos dados. O particionamento físico é realizado utilizando bloqueios leitor/escritor para controlar o acesso às partições, permitindo concorrência de leituras.

A cada reparticionamento, um novo conjunto de motores de armazenamento é criado, e os motores antigos tornam-se somente leitura. Quando uma chave é escrita, o valor é armazenado no motor mais recente correspondente à partição atual da chave. Para leituras, se a chave ainda não foi escrita após o reparticionamento mais recente, o valor será lido do motor antigo onde foi originalmente armazenado.

Para viabilizar esse comportamento, além do mapa de partições M_p, a implementação mantém um mapa de armazenamento M_s que associa cada chave à instância de motor onde seu valor está armazenado. Cada motor de armazenamento possui um atributo de nível ℓ que indica em qual geração de reparticionamento foi criado. Uma variável global ℓ_{curr} indica o nível atual do sistema.

Considere as chaves k_1 e k_2 atribuídas às partições p_1 e p_2, com valores armazenados em S_1 e S_2. Após um reparticionamento onde k_1 é reassociada à partição p_2, os novos motores são S_1' e S_2'. Até a próxima escrita, leituras de k_1 acessarão S_1. Quando uma escrita de k_1 ocorrer, seu mapeamento em M_s será atualizado para S_2', e leituras subsequentes lerão de S_2'.

Esta estratégia permite reassociar chaves no mapa de partições sem migrar dados entre motores. O projeto evita concorrencia entre escritas ou entre escrita e leitura na mesma instância de armazenamento, enquanto permite leituras concorrentes (pois partições diferentes podem ter

chaves em motores antigos compartilhados). Isso possibilita o uso de estruturas que não implementam sincronização interna, uma vez que leitura concorrente normalmente é naturalmente *thread-safe*. Além disso, essa estratégia permite distribuir os dados em dispositivos físicos diferentes facilmente, já que é possível criar instâncias de motores em diretórios distintos que podem residir em dispositivos separados, beneficiando-se de E/S paralela.

A seguir são apresentados os algoritmos das operações. Seja $M_s : \mathcal{K} \to \mathcal{S}$ o mapa de armazenamento, $M_p : \mathcal{K} \to \mathbb{N}$ o mapa de partições, μ_m o *mutex* compartilhado dos mapas, $P = \{\mu_0, \mu_1, \ldots, \mu_{n-1}\}$ o conjunto de *mutexes* das partições, $\mathcal{S} = \{S_0, S_1, \ldots, S_{n-1}\}$ o conjunto atual de motores de armazenamento, ℓ_{curr} o nível atual, e $h : \mathcal{K} \to \mathbb{N}$ uma função de dispersão.

Algorithm 4 Operação de Leitura – *HardRepartitioningKeyValueStorage*

```
1:  function READ(k ∈ K)
2:      LOCKSHARED(μ_m)
3:      S ← M_s[k]
4:      if S = null then
5:          UNLOCKSHARED(μ_m)
6:          return NOT_FOUND
7:      end if
8:      p ← M_p[k]
9:      if p = null then
10:         p ← h(k) mod n
11:     end if
12:     LOCKSHARED(μ_p)
13:     UNLOCKSHARED(μ_m)
14:     v ← S.READ(k)
15:     UNLOCKSHARED(μ_p)
16:     return v
17: end function
```

Algorithm 5 Operação de Escrita – *HardRepartitioningKeyValueStorage*

```
1: function WRITE(k ∈ 𝒦, v ∈ 𝒱)
2:     LOCK(μ_m)
3:     S ← M_s[k]
4:     if S ≠ null then
5:         p ← M_p[k]
6:         if p = null then
7:             p ← h(k) mod n
8:             M_p[k] ← p
9:         end if
10:        if S.ℓ ≠ ℓ_curr then
11:            S ← 𝒮[p]
12:            M_s[k] ← S
13:        end if
14:    else
15:        p ← h(k) mod n
16:        M_p[k] ← p
17:        S ← 𝒮[p]
18:        M_s[k] ← S
19:    end if
20:    LOCK(μ_p)
21:    UNLOCK(μ_m)
22:    S.WRITE(k, v)
23:    UNLOCK(μ_p)
24: end function
```

A operação de varredura coleta as partições envolvidas a partir do mapa M_p e as bloqueia em ordem crescente de índice para evitar *deadlocks*. Note que múltiplas chaves podem estar em um mesmo motor antigo, porém o bloqueio é feito no nível de partição conforme o esquema atual.

Algorithm 6 Operação de Varredura – *HardRepartitioningKeyValueStorage*

```
1: function SCAN(k_0 ∈ 𝒦, L ∈ ℕ)
2:     LOCKSHARED(μ_m)
3:     it ← M_s.LOWERBOUND(k_0)
4:     P_set ← ∅; S_arr ← []; K_arr ← []
5:     c ← 0
6:     while c < L ∧ ¬it.ISEND() do
7:         k ← it.KEY()
8:         p ← M_p[k]
9:         if p = null then
10:            p ← h(k) mod n
11:        end if
12:        P_set ← P_set ∪ {p}
13:        S_arr.APPEND(it.VALUE())
14:        K_arr.APPEND(k)
15:        it ← it.NEXT(); c ← c + 1
16:    end while
17:    P_sorted ← SORT(P_set)
18:    for all p ∈ P_sorted do
19:        LOCKSHARED(μ_p)
20:    end for
21:    UNLOCKSHARED(μ_m)
22:    R ← []
23:    for i ← 0 to |K_arr| − 1 do
24:        R.APPEND((K_arr[i], S_arr[i].READ(K_arr[i])))
25:    end for
26:    for all p ∈ P_sorted do
27:        UNLOCKSHARED(μ_p)
28:    end for
29:    return R
30: end function
```

0.3.4.4 *Tracker*

O componente *Tracker* é responsável por rastrear os padrões de acesso às chaves e construir o grafo de carga de trabalho utilizado pelo METIS para computar o particionamento. A arquitetura segue um modelo produtor-consumidor: as operações de leitura, escrita e varredura submetem as chaves acessadas a uma fila concorrente, e uma *thread* de rastreamento consome essa fila em segundo plano, atualizando o grafo.

O grafo de carga de trabalho $G = (V, E, w)$ é um grafo ponderado onde:

- $V \subseteq \mathcal{K}$ é o conjunto de vértices, cada vértice representando uma chave acessada;
- $w_v : V \to \mathbb{N}$ é a função de peso dos vértices, onde $w_v(k)$ acumula a frequência de acesso à chave k;
- $E \subseteq V \times V$ é o conjunto de arestas não direcionadas representando co-acessos entre chaves;
- $w_e : E \to \mathbb{N}$ é a função de peso das arestas, onde $w_e(k_i, k_j)$ acumula a frequência de co-ocorrência das chaves k_i e k_j em operações de varredura.

Quando uma operação de leitura ou escrita é executada, a chave acessada é submetida à fila do *Tracker*. A *thread* de rastreamento consome a chave e incrementa o peso do vértice correspondente no grafo. Para operações de varredura, todas as chaves acessadas são submetidas em conjunto. A *thread* de rastreamento incrementa o peso de cada vértice e também incrementa o peso das arestas entre todos os pares de chaves, refletindo a correlação de acesso entre elas.

A implementação do grafo utiliza listas de adjacência baseadas em tabelas de dispersão (`unordered_dense`), garantindo complexidade $O(1)$ amortizada para operações de incremento de peso de vértices e arestas. Essa estrutura é otimizada para o padrão de acesso do rastreamento, onde as operações predominantes são inserções e atualizações incrementais.

Para realizar o particionamento, o METIS requer o grafo no formato CSR (*Compressed Sparse Row*). O *Tracker* utiliza um componente auxiliar (`MetisGraph`) que converte o grafo de listas de adjacência para o formato CSR, construindo:

- Vetor `xadj`: índices de início de cada lista de adjacência;
- Vetor `adjncy`: lista concatenada de vizinhos;
- Vetor `vwgt`: pesos dos vértices;
- Vetor `adjwgt`: pesos das arestas.

Adicionalmente, são mantidos mapeamentos bidirecionais entre os nomes das chaves (strings) e os índices inteiros utilizados pelo METIS. Após o particionamento, o resultado é um vetor que associa cada índice de vértice a um índice de partição, que é então utilizado para atualizar o mapa de partições M_p do armazenamento.

0.3.5 Reparticionamento

O reparticionamento dinâmico é executado periodicamente por uma *thread* dedicada que opera em ciclos contínuos. Cada ciclo alterna entre duas fases: uma fase de rastreamento, durante a qual os padrões de acesso são coletados, e uma fase de intervalo, durante a qual o sistema opera sem rastreamento. Ambas as durações são configuráveis na inicialização do armazenamento.

O ciclo de reparticionamento segue os seguintes passos:

1. **Intervalo de reparticionamento**: A *thread* de reparticionamento aguarda por um período configurável (`repartition_interval`). Durante esse período, o rastreamento está desabilitado e o sistema opera normalmente com o esquema de partições atual.
2. **Período de rastreamento**: O rastreamento é habilitado e as operações executadas passam a submeter suas chaves à fila do *Tracker*. A *thread* de rastreamento consome a fila e constrói o grafo de carga de trabalho. Esse período tem duração configurável (`tracking_duration`).
3. **Execução do reparticionamento**: Ao final do período de rastreamento, o reparticionamento é executado:
    a) O rastreamento é desabilitado e a *flag* de reparticionamento é ativada;

b) A fila do *Tracker* é esvaziada para garantir que todas as chaves submetidas sejam processadas;

c) O grafo é convertido para o formato CSR e o METIS é invocado para computar o particionamento;

d) Se o particionamento for bem-sucedido, o mapa de partições M_p é bloqueado juntamente com todas as partições;

e) O mapa de partições é atualizado com as novas atribuições computadas pelo METIS;

f) Os bloqueios são liberados e o grafo é limpo para o próximo ciclo;

g) A *flag* de reparticionamento é desativada.

4. O ciclo retorna ao passo 1 e se repete indefinidamente até a destruição do armazenamento.

A Figura 1 ilustra o ciclo de reparticionamento. A fase de intervalo permite que o sistema opere com estabilidade sob o novo esquema de partições antes de iniciar uma nova coleta de padrões de acesso. A fase de rastreamento captura uma janela representativa da carga de trabalho atual, permitindo que o METIS compute um particionamento otimizado para os padrões de acesso observados.

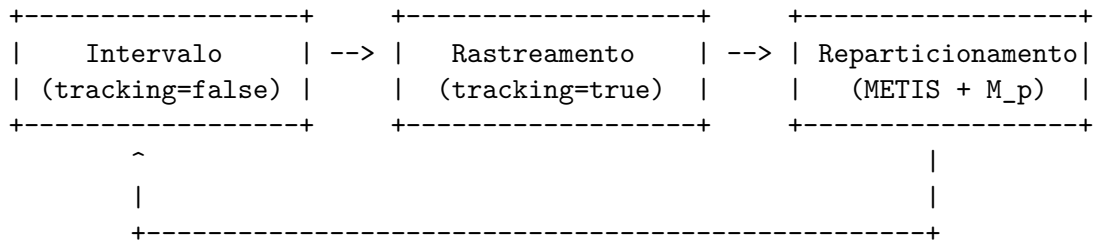

Figura 1 – Ciclo de reparticionamento dinâmico

Durante a atualização do mapa de partições, as operações de leitura e escrita são bloqueadas momentaneamente enquanto os bloqueios são adquiridos. Esse bloqueio é breve, pois a atualização do mapa consiste apenas em iterar sobre os resultados do METIS e atualizar as entradas correspondentes no mapa de partições. Os dados em si não são migrados durante o reparticionamento; a migração ocorre de forma preguiçosa (*lazy*) conforme as chaves são acessadas subsequentemente, especialmente na estratégia *Hard*.

0.3.6 Testes automatizados

0.3.7 Ferramenta de testes

Give examples of how to use the runner, and examples of output.

0.4 Execução de testes e experimentos

A biblioteca conta com testes automatizados que validam o comportamento esperado e conferem maior confiabilidade quanto a corretude da implementação. Enquanto isso, a ferramenta de

execução de experimentos permite a execução de experimentos com diferentes configurações de armazenamento e estratégias de reparticionamento a fim de permitir avaliações comparativas de diferentes abordagens de reparticionamento.

Os testes automatizados podem ser executados através do *script* `run_tests.sh` ou individualmente a partir dos executáveis gerados pelo CMake. Os testes verificam:

- Operações básicas de leitura, escrita e varredura (*scan*) nos motores de armazenamento;
- Funcionamento correto das estruturas de mapeamento de chaves (*KeyStorage*);
- Construção e manipulação do grafo de carga de trabalho;
- Integração com a biblioteca METIS para particionamento de grafos;
- Comportamento das diferentes estratégias de reparticionamento sob execução concorrente;
- Sincronização entre *threads* trabalhadoras durante a atualização do esquema de partições.

Os executáveis de teste incluem: `test_storage_engine`, `test_keystorage`, `test_graph`, `test_metis_graph`, `test_partitioned_kv_storage`, `test_repartitioning_storage`, entre outros.

A ferramenta de execução de experimentos aceita diferentes argumentos de configuração, além de um arquivo de carga de trabalho. É necessario que o arquivo de carga de trabalho siga o formato padronizado. A seguir é apresentado esse formato, e posteriomente são descritos os parâmetros de configuração.

0.4.1 Formato da carga de trabalho

O formato do arquivo de carga de trabalho segue a convenção:

- `0,<chave>`: operação de leitura;
- `1,<chave>`: operação de escrita;
- `2,<chave>,<limite>`: operação de varredura.

0.4.2 Parâmetros de configuração

A ferramenta aceita os seguintes parâmetros de configuração:

```
repart-kv-runner <arquivo_workload> [partições] [workers]
                 [tipo_storage] [motor_storage]
                 [warmup] [caminhos_storage]
```

O parâmetro `tipo_storage` pode assumir os valores:

- `soft`: reparticionamento leve com partições lógicas;
- `hard`: reparticionamento com múltiplas instâncias de armazenamento;

- `threaded`: versão com *threads* trabalhadoras;
- `hard_threaded`: reparticionamento rígido com *threads* trabalhadoras;
- `engine`: acesso direto ao motor de armazenamento sem reparticionamento.

O parâmetro `motor_storage` permite selecionar entre os *backends* disponíveis: `tkrzw_tree`, `tkrzw_hash`, `lmdb`, `map` (em memória) ou `tbb`.

0.4.3 Funcionamento da ferramenta

Durante a execução, a ferramenta instancia o armazenamento configurado e cria múltiplas *threads* trabalhadoras que executam operações concorrentemente. Cada *thread* processa uma porção das operações do arquivo de carga de trabalho, e os contadores de operações executadas são agregados ao final para o cálculo da vazão total do sistema.

Paralelamente, uma *thread* de métricas registra periodicamente em um arquivo CSV o tempo decorrido, número de operações executadas, uso de memória, uso de disco e estados de rastreamento e reparticionamento. Essas métricas permitem a análise detalhada do comportamento do sistema sob diferentes cargas de trabalho, facilitando a comparação entre as estratégias implementadas e a identificação de gargalos de desempenho.

0.5 Discussão

Referências

[Karypis e Kumar 1998]KARYPIS, G.; KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, v. 20, n. 1, p. 359–392, 1998.

[Luiz e Mendizabal 2024]LUIZ, D. P.; MENDIZABAL, O. M. Balanceamento de carga com reparticionamento contínuo em sistemas com estado particionado. In: *Proceedings of the Encontro Regional de Computação (ERAD IC) 2024*. [S.l.: s.n.], 2024. p. 31–38.

[Luiz e Mendizabal 2024]LUIZ, D. P.; MENDIZABAL, O. M. Lightweight asynchronous repartitioning for local state partitioned systems. In: *Proceedings of the 2024 Symposium on System-Level Performance and Availability (SSCAD)*. [S.l.: s.n.], 2024. p. 1–8.

[Luiz e Mendizabal 2025]LUIZ, D. P.; MENDIZABAL, O. M. Stall-free asynchronous state repartitioning with a proactive workload tracking window. *Computers and Concurrent Engineering*, v. 11, n. 2, p. 134–147, 2025.