Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Engenharia da Computação

Estrutura híbrida para melhoria de desempenho dos *Roaring Bitmaps* para consultas *rank*

Sérgio Victor Ramos da Silva

Trabalho de Graduação

Recife
5 de outubro de 2024

Universidade Federal de Pernambuco
Centro de Informática

Sérgio Victor Ramos da Silva

Estrutura híbrida para melhoria de desempenho dos *Roaring Bitmaps* para consultas *rank*

Trabalho apresentado ao Programa de Bacharelado em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Paulo G. S. da Fonseca*

Recife
5 de outubro de 2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Sergio Victor Ramos da.
Estrutura híbrida para melhoria de desempenho dos Roaring Bitmaps para consultas rank / Sergio Victor Ramos da Silva. - Recife, 2024.
44 p : il.

Orientador(a): Paulo Gustavo Soares da Fonseca
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2024.
Inclui referências.

1. Roaring Bitmaps. 2. Consultas rank. 3. Bitvectors. I. Fonseca, Paulo Gustavo Soares da. (Orientação). II. Título.

620 CDD (22.ed.)

SÉRGIO VICTOR RAMOS DA SILVA

ESTRUTURA HÍBRIDA PARA MELHORIA DE DESEMPENHO DOS ROARING BITMAPS PARA CONSULTAS RANK

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 29/08/2024

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Gustavo Soares da Fonseca (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Adenilton Jose da Silva (Examinador Interno)

Universidade Federal de Pernambuco

Dedico este trabalho a meus pais: Sergio e Ana.

Agradecimentos

Agradeço primeiramente a Deus por essa oportunidade de me desenvolver intelectualmente e por todas as experiências que me fizeram chegar até hoje, que me permitirão contribuir de forma positiva à sociedade com minhas faculdades mentais e habilidades adquiridas.

Eu sou grato também a meus pais por todo apoio e confiança e por me incentivarem a continuar estudando e perseverando, a despeito de todas as dificuldades e desafios que encontrei nessa jornada, e me permitirem continuar estudando quando as circunstâncias nos separaram fisicamente.

Agradeço ao professor Paulo Soares, popularmente conhecido como "paguso", por todo o cuidado e perspicácia no processo de orientação deste trabalho, pelas minuciosas revisões de texto, ideias construtivas, sugestões e pelos *feedbacks* necessários, sem os quais este projeto não teria a mesma qualidade.

E, por fim, quero agradecer a todos os profissionais da Universidade Federal de Pernambuco (UFPE) que, de alguma forma, por menor que seja, contribuíram para que minha experiência como aluno fosse tão significativa.

Quem olha para fora, sonha; Quem olha para dentro, desperta.
—C. G. JUNG

Resumo

Desde o desenvolvimento das primeiras estruturas de dados compactadas, uma variedade de estudos foi realizada, com o objetivo de alcançar um desempenho computacional próximo ao das estruturas não compactadas, relativamente a operações de consulta [1, 5, 9, 12, 16], . Dentre as abordagens propostas até o momento, a viabilidade matemática dessas estruturas nem sempre corresponde à realidade prática [17]. Nesse contexto, desenvolver algoritmos e estruturas de dados eficientes tem sido uma constante no mundo das operações de consulta.

Este trabalho avalia a viabilidade dos *Roaring Bitmaps* para a operação de *rank* (i.e. retornar a soma de bits 1, do início do vetor até uma dada i-ésima posição) e melhorar seu desempenho, além de compará-lo, antes e depois de algumas modificações, com o desempenho de estruturas especializadas em *ranks*, examinadas em [16] e [17].

Mostramos como é possível substituir um algoritmo que realiza contagem de bits, utilizando 2^{10} operações de *popcount* em um vetor de bits de tamanho fixo, por uma operação em tempo constante que realiza consulta a uma estrutura auxiliar, capaz de, em um dos cenários explorados, reduzir o tempo de execução em 95,65%. Em seguida, mostramos como tornar constante o tempo total de execução das consultas *rank* com os *Roaring Bitmaps* com praticamente o mesmo uso de memória.

Nós concluímos que utilizar uma estrutura auxiliar que utiliza 6,25% a mais de memória com os *Roaring Bitmaps* é superior a utilizar a versão original, por manter a complexidade de espaço e reduzir a complexidade de tempo. Concluímos também que, na versão final dos *Roaring Bitmaps* modificados, é necessário criar uma restrição sobre a adição de valores representativos dos conjuntos de inteiros, para que seja possível tornar as operações nos *Roaring Bitmaps* constantes no tempo, porque a otimização final foi possível considerando o seguinte caso: as consultas *rank* são realizadas após finalizada a adição de inteiros aos *Bitmaps*.

Palavras-chave: RRR, Vigna, SDSL, Roaring Bitmaps, estruturas de dados, compressão, bitvectors, rank

Sumário

1 Introdução **1**

2 Fundamentação Teórica e Trabalhos Relacionados **3**
2.1 Conceitos e Terminologia 3
2.2 História das operações de *rank* 4
2.3 *Bitvectors* comprimidos e os *Roaring Bitmaps* 8
2.4 Biblioteca SDSL 9

3 Desenvolvimento **10**
3.1 *Roaring Bitmaps* 10
3.2 Desenvolvimento 12
3.2.1 Análise de estruturas concorrentes 12
3.2.2 Modificações aos *Roaring Bitmaps* 13

4 Experimentos e Resultados **18**
4.1 Experimentos 19
4.2 Resultados Comparativos com os *Roaring Bitmaps* originais 21
4.3 Resultados com *CRoaring modificado (SVRS-Roaring)* 24

5 Conclusões **33**

Lista de Figuras

3.1 Roaring Bitmap [1] 12

3.2 Note que um único vetor *SDSL* é utilizado para armazenar todos os *bitvectors* que pertenciam aos *bitset containers*. O *rank* local é calculado utilizando a estrutura *rank-support-v5* gerada a partir do vetor *SDSL*. 15

4.1 Experimento 1.1, capacidade de compressão das diferentes estruturas, considerando um *bitvector* bruto inicial de 1MB (2^{23} bits). 22

4.2 Experimento 1.2, o *Roaring Bitmap* mostrou-se inicialmente inferior ao *RRR-vector* e *rank-support-v5* em tempo de execução. 23

4.3 Experimento 2. *O Roaring Bitmap* tem resultado superior em tempo de execução ao *RRR-Vector* e competitivo com *rank-support-v5* para densidades baixas. 24

4.4 Experimento 3. Resultado do mesmo experimento da Figura 4.3, mas com o *CRoaring* modificado. 25

4.5 Experimento 4. Experimento com um *bitvector* inicial de tamanho 32 vezes maior que o experimento anterior (128 *bitset containers*) 26

4.6 O tempo necessário para calcular os *ranks* varia de 1,8 segundos até 1,12 horas para o mesmo número de consultas. 27

4.7 Este gráfico mostra que o tempo de execução passou a ser constante, independentemente da posição do índice do *rank* no *Roaring Bitmap* 28

4.8 Compare a escala de tempo relativamente à Figura 4.7. Aqui a variação é de alguns segundos e podemos observar que, para posições abaixo de 2^{18}, o número de *containers* não influencia negativamente no tempo de execução. 29

4.9 Comparativo do tempo de execução final entre as estruturas competitivas elencadas no início do capítulo (*RRR-vector* e *Rank-support-v5*) com o *SVRS-Roaring2*. Note que apenas o *RRR-vector* mostrou desempenho variável de acordo com a posição do valor do *rank* calculado. 30

4.10 Mesmo resultado da figura anteior, mas adicionamos o tempo de execução do *Roaring* original em comparação ao *SVRS-Roaring2*. Note que foi necessário modificar a escala do gráfico para que o resultado do *Roaring Bitmap* original não tornasse os outros valores invisíveis. 31

4.11 Comparativo do uso de memória entre a versão final modificada do *Roaring Bitmap* em comparação com as estruturas já existentes antes deste trabalho. Comparando com as figuras anteriores, note que o novo *Roaring Bitmap* é capaz de realizar *ranks* em tempo constante com praticamente o mesmo uso de memória da estrutura original 32

CAPÍTULO 1

Introdução

O uso de estruturas de dados sucintas (estruturas de dados comprimidas que realizam operações equivalentes às não comprimidas, de forma eficiente em tempo de execução, para operações de consulta) tem crescido a partir dos anos 2000 em diante [17]. No entanto, apesar de muitas das soluções propostas serem matematicamente eficientes, os resultados destas soluções dependem de experimentos práticos que as viabilize [5]. Duas operações muito comumente estudadas são as consultas *rank* (contar o número de 1's até certa posição em um vetor de bits) e *select* (encontrar a posição do i-ésimo bit 1) [17].

Dentre algumas das propostas de compressão de dados, destacam-se os *Roaring Bitmaps*, cujo desempenho tem se mostrado superior a todas as propostas concorrentes em [1], para as operações união, interseção, adição e remoção de itens de *Bitmaps* comprimidos (que são, em geral, uma forma eficiente, no tempo e no espaço, de representar vetores de bits). Embora os resultados se mostrem eficientes para os *Roaring Bitmaps* para as diversas operações supracitadas, a solução para *rank* não foi explorada a fundo pelos autores em [1] e [2], pois a referência se dá em apenas algumas linhas, em que os autores apresentam a possibilidade de realizar consultas *ranks* de forma eficiente [1].

Diante da criticidade do problema e as aplicações de *Bitmaps* em sistemas de buscas (como a ferramenta de busca SQL do Youtube, Netflix Atlas, Apache Hivemall, dentre outras[4]),, este trabalho se propõe a estudar os *Roaring Bitmaps* e comparar seu desempenho para as consultas *rank* com o de outras estruturas de dados especializadas nesta operação, como as propostas de Sebastian Vigna [17], González et al. [5], baseadas nas soluções teóricas de Raman et al. [16] e Munro [14].

Nesse contexto, mostramos como é possível melhorar o desempenho dos *Roaring Bitmaps* para realizar consultas *rank* com algumas alterações nesta ferramenta, tornando possível, por exemplo, reduzir o tempo de execução de 8,4 milhões de operações de *rank* de 1,2 horas para 2 segundos em um *Bitmap* original contendo 2^{30} bits, como observado em um dos cenários explorados.

No capítulo 2, encontramos uma breve introdução ao tema, em que são apresentados a terminologia da área de estudo e conceitos importantes para compreensão dos capítulos seguintes. Além disso, descrevemos o desenvolvimento de estruturas de dados que suportam consultas *rank* desde 1974 até os dias de hoje.

No capítulo 3, mostramos como funcionam os *Roaring Bitmaps* e, em seguida, explicamos as alterações que foram realizadas com o objetivo de melhorar o desempenho para as operações de *rank*.

O capítulo 4 contém uma análise dos experimentos que foram realizados neste trabalho e como os *Roaring Bitmaps* se comparam às soluções para *rank* baseadas nas propostas de González et al., S. Raman et al. e S. Vigna [5, 16, 17].

Por fim, no capítulo 5, finalizamos com uma breve elucidação dos resultados obtidos e apresentamos perspectivas futuras.

CAPÍTULO 2

Fundamentação Teórica e Trabalhos Relacionados

Neste capítulo, damos uma base conceitual e terminológica necessária para a compreensão deste trabalho. Em seguida, apresentamos os primeiros estudos e trabalhos sobre as operações de *rank* cronologicamente. Então concluímos apresentando diferentes abordagens, que se ramificaram com o objetivo de propor melhores soluções, relativamente a eficiência de tempo e espaço. Algumas das propostas examinadas incluem trabalhos de Raman-Raman-Rao [16], Vigna [17], R. González et al[5], dentre outros.

2.1 Conceitos e Terminologia

Estrutura de dados sucinta é uma estrutura de dados comprimida (e que, portanto, utiliza uma quantidade próxima ao mínimo teórico necessário para armazenar a informação) que utiliza uma quantidade menor de memória quando comparada ao que se espera para armazenar os dados de forma não comprimida, mas que é capaz de realizar operações de consulta de forma eficiente, apesar da compressão. Um ***bitmap***, também conhecido como *bitset* ou ainda *bitvector*, é uma representação computacional de uma sequência de bits. No contexto deste trabalho, *bitvectors* são utilizados para representar conjuntos finitos de números naturais. A forma convencional para essa representação é que, para cada bit armazenado, um bit 0 indica a ausência do i-ésimo natural no conjunto, e o bit 1 representa a presença deste. Por exemplo, um *bitvector* b = [0, 1, 1, 0, 0, 1] é equivalente ao conjunto B = {1, 2, 5}. Esta representação é chamada de ***bitset* bruto**. A quantidade de bits 1 presentes em um *bitvector* bruto é chamada de **cardinalidade**.

É comum encontrarmos na literatura referências a *bitsets* comprimidos [10], ou seja, que são representados de forma a economizar espaço de memória comparativamente a um *bitset* **bruto** (i.e. não-comprimido).

Comprimento de um *bitset* é a quantidade total de bits em um *bitset* bruto. Nesse contexto, **Densidade** é o termo utilizado para descrever a frequência relativa de bits 1 em relação ao comprimento do *bitset*, ou seja, a razão entre a cardinalidade e o comprimento do *bitset*. Por exemplo, o *bitset* b = [0, 1, 1, 0, 0, 1] possui uma densidade de 3/6, 0,5 ou 50%.

Dentre as possíveis operações de consulta que são realizadas em um *bitset*, temos a operação de ***rank***, que retorna a soma de bits 1 presentes antes de uma dada posição no *bitvector*[16]. Em outras palavras, retorna a quantidade de naturais menores que aquele recebido como entrada, tais que pertençam a um dado conjunto. Por exemplo, para o *bitvector* [1, 1, 1, 1, 1, 1, 0, 1], *rank(7)* = *rank(6)* = 6.

Para tornar as consultas em *bitvectors* eficientes, algumas estruturas auxiliares são adicionadas de forma a realizar um *trade-off* de memória e tempo de execução. Esse espaço extra de memória utilizado recebe o nome de ***overhead***.

2.2 História das operações de *rank*

Elias (1974), em seu trabalho sobre armazenamento e recuperação de dados em arquivos estáticos [3], abordou conjuntos de arquivos com entradas em forma de palavras binárias de tamanho fixo ordenadas lexicalmente . Ele analisou algumas operações sobre esses arquivos, como por exemplo, encontrar o valor da i-ésima entrada, o que no contexto de *bitvectors* é equivalente a uma operação de *select*[16]. Outra operação consistia em encontrar o número de entradas de valor menor que i, equivalente à operação de *rank* para *bitvectors*. Ele considerou o número de bits armazenados e o número de bits acessados em cada uma das operações de busca como medidas relevantes para análise de eficiência. Sendo assim, apresentou limites inferiores para

medidas de *minimax* (i.e. o menor valor dentro de um conjunto de valores máximos), considerando o uso de memória para cada uma das questões envolvendo operações sobre esses arquivos, com o objetivo de obter um limitante inferior.

Nesse contexto, a primeira solução não-trivial para a consulta *rank* de que temos conhecimento (i.e. mais inteligente que contar bits sequencialmente em um vetor), por Jacobson (1989) [12], realiza três acessos à memória: 1. para um contador de superbloco, 2. para um contador de bloco, 3. uma visita a uma tabela com respostas de *popcount* (i.e. contagem de bits 1 em uma sequência binária) pré-computadas. Esses três acessos implicam, em geral, três faltas de cache [12]. Jacobson propôs também uma codificação de bits baseada na construção de sequências de Zaks [18], utilizadas para codificar árvores binárias.

Desde o trabalho de Jacobson [12], estruturas de dados sucintas, apresentadas na seção 2.1, que visam reduzir o espaço de armazenamento de dados mantendo a capacidade de realizar operações de forma eficiente sobre esse dados, são comumente utilizadas e têm sido propostas para árvores, grafos, *strings* e textos. Com a ajuda de uma estrutura auxiliar ocupando $o(n)$ bits é possível responder *rank* em tempo constante [12].

J. Ian Munro (1996) mostrou que é possível realizar consultas a valores em estruturas de dados estáticas, como árvores de sufixos e representações de conjuntos esparsos (i.e. análogos a *bitsets* com baixa densidade), [14] em tempo constante [14]. Esta foi a primeira solução de tempo constante, utilizando n + o(n) de espaço. Este resultado é relevante quando pensamos em conjuntos finitos relativamente grandes com, por exemplo, centenas de milhões de entradas, em que há a necessidade de eficiência tanto de espaço quanto de tempo. Para tal feito, utilizou-se de tabelas com todas as respostas possíveis de consultas aos valores dentro do domínio fornecido.[14].

Em 2002, Raman, Raman e Rao apresentaram uma proposta teórica para consultas *rank* em tempo constante, com uma abordagem de custo teórico menor de memória extra comparativamente à proposta de Jacobson [16, 12]. Para tal feito, utilizaram uma representação de um

conjunto de tamanho n e aplicaram uma função de *hash* aos t bits mais significativos dos números naturais de cada conjunto [16]. Aproveitando a solução em [12], Raman et al, dividiram o *bitset* em superblocos, que por sua vez foram divididos em blocos.

Desta forma, propuseram o uso de consultas a tabelas de permutações de bits associadas a *popcounts*. Assim, para obter *ranks* em tempo constante, utilizou-se em [16] de um método discutido por Munro em 1996 [14], em que é armazenado o *rank* global do *bitvector* em cada início de superbloco.

Portanto, para calcular *rank(i)*: 1) Calcula-se em qual bloco e superbloco o índice está localizado 2) Obtém-se o *rank* global no início do superbloco 3) Obtém-se o *rank* local utilizando a contagem de bits por bloco e soma-se ao valor do *rank* global do superbloco.

Em relação à solução clássica de tempo constante em [16], a tabela universal que contém valores de *rank* locais pode ser substituída por *popcounting* [5]. Na prática, o comprimento de cada bloco deve ser múltiplo de 8, porque as soluções de *popcount* funcionam no mínimo byte a byte [5]. Vale ressaltar que a taxa de acerto do cache diminui à medida que n cresce, o que explica o crescimento em tempos de consulta, que deveriam ser constantes, que ocorre na prática [5]. Consequentemente, soluções que são constantes no tempo passam a crescer quase que linearmente com $logn$ [5].

González et al. (2005) [5] consideraram a alternativa de utilizar o mesmo espaço de um vetor para se ter um único nível de blocos com uma entrada de tamanho múltiplo de 32 bits e utilizar um único inteiro de 32 bits para armazenar os *ranks* e realizar uma **varredura sequencial**, realizando *popcounts* de grupos de palavras de 32 bits até alcançar a posição desejada. Os autores em questão concluem que utilizar dois níveis de bloco mais um acesso a um *array* de bits é, em geral, melhor que utilizar o mesmo espaço de armazenamento para um único nível de blocos, mas a situação muda quando n torna-se grande o suficiente (o que não foi especificado pelos autores), devido ao efeito de cache.

Sucessivamente, a proposta de Sebastiano Vigna (2008) [17] abandona o *bitvector* bruto

completamente e armazena todas as posições com bits 1 em um dicionário completamente indexável, representando um conjunto de naturais, com o qual é possível obter o *rank* em tempo constante. Diferentemente de abordagens típicas de 32 bits, envolvendo tabelas pré computadas, Vigna et al. recomendaram a utilização de programação *broadword* (a.k.a. SWAR - "SIMD in a Register), técnica que realiza operações em paralelo de dados contidos em um registrador, uma única instrução - múltiplos dados [17]. A implementação concedida pelos autores endereça 2^{64} bits, utiliza menos espaço e é 30% mais eficiente em tempo de execução que a solução de González et al. [17].

Vigna argumentou que a notação de complexidade utilizada pelos autores anteriores a ele frequentemente esconde constantes tão grandes que antes que a vantagem assintótica tenha efeito, a estrutura de dados é demasiadamente grande [17]. Nesse sentido, pode-se dizer que o resultado é interessante matematicamente, mas tem pouco valor prático. Por este motivo, Vigna abandonou a abordagem clássica, argumentando que em uma arquitetura de 64 bits moderna, a programação *broadword* é muito mais eficiente [17]. *Broadword* utiliza registradores maiores (ex. registradores com mais que 64 bits) como pequenos computadores paralelos, processando diversos pedaços de informação de uma vez. A maior vantagem da programação *broadword* é que é possível ganhar mais velocidade à medida que o tamanho de uma palavra aumenta, enquanto que é possível processar mais dados em paralelo [17].

Com o objetivo de minimizar faltas de cache, Sebastiano Vigna propôs inserir dados intercalados de diferentes tabelas, de modo que uma única falta de cache seja suficiente para que se possa encontrar todas as informações necessárias que estejam associadas a uma porção do *bit array* [17]. Resumidamente, a estrutura com suporte para consultas *rank* de Vigna segue uma abordagem tradicional de utilizar dois níveis, com o diferencial de realizar somas utilizando *broadword* para realizar contagens dentro de uma palavra.

2.3 *Bitvectors* comprimidos e os *Roaring Bitmaps*

Dentre algumas das propostas de compressão de *bitmaps* recentes, citamos WAH (*Word Aligned Hybrid*) e Concise (*Compressed 'n' Composable Integer Set*) que utilizam o esquema de compressão RLE (*run-length encoding*) [1], em que sequências de valores repetidos são armazenadas como um único valor em uma contagem; por exemplo, a *string* 1011001010011011111 111111111111 poderia ser representada por *substrings* de tamanho *w* utilizando um bit 0 para indicar sequências binárias literais (contendo uma mistura de 0's e 1's) de tamanho fixo *w-1* e um bit 1 mais um bit indicativo de 0 ou 1 precedendo sequências binárias chamadas de *fill words*, que contêm representações de palavras contendo apenas 0's ou apenas 1's de tamanho *w-2* (1 bit foi utilizado para representar as *fill words* e 1 bit foi utilizado para indicar se a sequência era de 0's ou 1's), dando como resultado a *string* 010110010010011011010001 . Embora reduzam uso de memória, estes formatos ainda possuem um desempenho em tempo de execução relativamente lento para acesso aleatório quando comparados a *bitmaps* não comprimidos [1].

Nesse contexto, foi desenvolvido o *Roaring Bitmap* (2015), como proposta de esquema de compressão de *Bitmaps*, com o objetivo de ser superior em memória e tempo de execução aos formatos de codificação baseados puramente em *RLE* supracitados. Elucidamos detalhadamente no capítulo 3, mas ressalte-se que o *Roaring Bitmap* particiona o *Bitvector* original e o armazena em partes mutáveis, dependendo se a parte armazenada é densa ou esparsa [1]. Divisões esparsas são armazenadas em *arrays* contendo apenas inteiros de 16 bits, enquanto que partes densas são armazenadas utilizando *bitsets* brutos [1]. O *Roaring Bitmap* é capaz de realizar operações de *rank*, mas essa funcionalidade foi implementada de forma muito próxima à solução trivial (i.e. realizar meramente contagem de bits).

2.4 Biblioteca SDSL

Simon Gog et al. (2014) lançaram a biblioteca *SDSL* [7], especializada em estruturas de dados sucintas/compactas, baseada em C++, com a proposta de tornar possível reproduzir a complexidade de tempo e espaço teóricas das estruturas de dados em questão. Ela contém ferramentas baseadas em mais de 40 publicações de pesquisa na área [7].

CAPÍTULO 3

Desenvolvimento

Neste capítulo, detalhamos a estrutura de dados que foi utilizada como base experimental para realizar consultas *rank* (i.e. dado um vetor binário, desejamos saber quantos bits 1 estão presentes até a i-ésima posição). A estrutura abordada neste capítulo é o *Roaring Bitmap*. Também discutimos brevemente outras estruturas que realizam consultas *rank*, a saber, as estruturas baseadas nas propostas em [16, 17] e, por fim, como a versão do *Roaring Bitmap*, disponível em [11], foi modificada com o objetivo de torná-la mais eficiente em tempo de execução para a operação de foco deste trabalho.

3.1 *Roaring Bitmaps*

Em primeiro lugar, *bitvector* é uma sequência de bits utilizada para representar conjuntos de números naturais, cuja **cardinalidade** é sinônimo do número de bits 1 presentes. Um bit 0 na i-ésima posição do *bitvector* indica que o valor *i* não está presente no conjunto, enquanto que um bit 1 significa que o valor *i* pertence ao conjunto. Assim, o *bitvector* v = [0, 0, 1, 1, 0, 0, 0, 0, 1, 1] é equivalente ao conjunto {2, 3, 8, 9} e possui cardinalidade igual a 4 (note que *rank(9)* = 3, *rank(3)* = 1). A forma de representação de conjuntos de naturais por meio de *bitvectors* permite que sejam realizadas operações de união, interseção e checar pertencimento, dentre outras [1].

O *Roaring Bitmap* é uma estrutura que foi desenvolvida com objetivo de permitir a compressão de *bitvectors*, mantendo a possibilidade de realizar as operações supracitadas. Ressaltamos que ele é capaz de representar conjuntos de naturais até $2^{32} - 1$. Um *Roaring Bitmap*

armazena seus valores em diversos *containers*, que são subestruturas indexáveis pelos 16 bits mais significativos de um natural em notação binária. Isso se dá por meio de um cabeçalho.

Existem, em princípio, dois tipos de *container*: o primeiro tipo contém um *bitvector* bruto, que armazena 0's e 1's associados aos 16 bits menos significativos restantes de cada natural pertencente ao subconjunto indexado, e um *container* que contém um *array* de inteiros de 16 bits, em que cada posição armazena os valores equivalentes aos 16 bits menos significativos restantes de cada natural, uma vez que os 16 mais significativos estão sendo utilizados para indexar o *container*. Note que o primeiro tipo de *container* representa cada valor por meio de 0 ou 1, enquanto que o segundo armazena *int16*'s.

Consequentemente, é possível indexar até 2^{16} *containers* em cada *Roaring Bitmap*. Além do mais, cada *array container* tem capacidade de armazenar 4096 *int16*'s. Semelhantemente, cada *container* de *bitvector* bruto contém 2^{16} posições. Sendo assim, um *bitset container* é ideal para casos em que a quantidade de 1's é muito maior que a quantidade de 0's, ou seja, o *bitvector* possui alta densidade. Desse modo, o *array container* é utilizado para casos em que há baixa densidade local no intervalo coberto pelo *container*, ou em termos práticos, uma quantidade menor ou igual a 4096 números no *container* em questão.

A Figura 3.1 mostra uma representação da estrutura geral de um *Roaring bitmap*. Nesta figura, o *container* mais à esquerda possui uma densidade local abaixo de 6,25% (i.e. $4096/2^{16}$) e, portanto, está na forma de *array container*. Note que o valor máximo possível neste *container* específico é 65535 (ou seja, os 16 bits mais significativos são a *string* binária 0000000000000000). O segundo *container* mais à esquerda representa o conjunto de naturais cujos 16 bits mais significativos são a *string* binária 0000000000000001 e cujos 16 bits menos significativos representam valores de 0 a 99. Ou seja, são representados valores no intervalo $[2^{16}, 2^{16}+100)$. O terceiro *container* é do tipo *bitset container*, representando o conjunto de inteiros de 32 bits cujos 16 bits mais significativos são a *string* binária 0000000000000010. Note que há mais que 4096 valores neste *container*. Em resumo, temos, sucessivamente, 1) a lista dos primeiros 1000

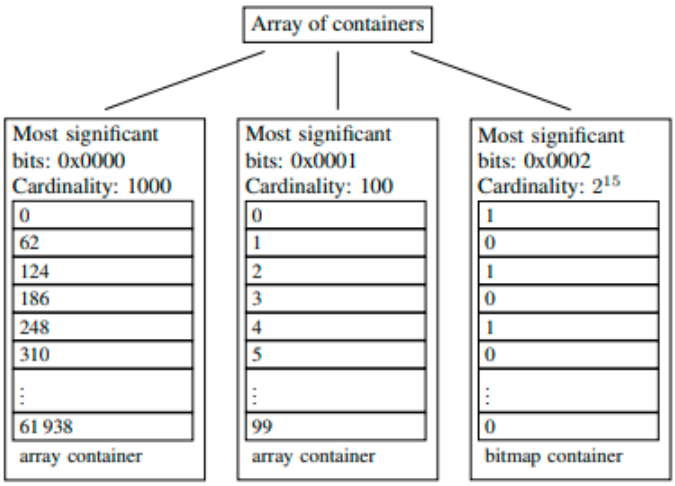

Figura 3.1 Roaring Bitmap [1]

múltiplos de 62, 2) todos os inteiros no intervalo [2^{16}, 2^{16}+100) e 3) todos os números pares no intervalo [2 x 2^{16}, 3 x 2^{16})

3.2 Desenvolvimento

3.2.1 Análise de estruturas concorrentes

De acordo com [1], o *Roaring Bitmap* mostrou-se superior em tempo de execução em comparação com estruturas de dados sucintas em estado da arte como *Concise* e *WAH* para operações de acesso, união e interseção, mas o desempenho para consultas do tipo *rank* não foi explorado.

Com o objetivo de avaliar o desempenho do *Roaring Bitmap* em relação a estruturas especializadas nesta operação, como [16, 17], foram escolhidas duas abordagens: 1) a primeira, cuja implementação é chamada RRR-vector, disponível na biblioteca *SDSL* [7], que foi baseada nas propostas de Rasmus Pagh [15], Rajeev Raman, V. Raman and S. Srinivasa Rao [16] e também Gonzalo Navarro e Eliana Providel [6][9], comprime o *bitvector* bruto e dá suporte à busca *rank*; 2) estrutura baseada na proposta de Vigna [17], que não comprime o *bitvec-*

tor bruto e adiciona uma estrutura auxiliar com *overhead* de 6,25% de memória [7], chamada *rank-support-v5* ou simplesmente *V5*. Esta baseia-se em uma abordagem não comprimida de um *bitvector* e, de acordo com a literatura, possui desempenho em tempo de execução superior à primeira estrutura [17], mas sua desvantagem é no uso maior de memória e, portanto, foi escolhida para representar um desafio de desempenho de tempo a ser alcançado pelo *Roaring Bitmap* modificado durante este trabalho.

Tanto o *RRR-vector* quanto o *Roaring Bitmap* são considerados estruturas de dados sucintas, e suportam operações de *rank* e, consequentemente, possuem necessariamente uma abordagem diferente de uma estrutura que utiliza um *bitvector* bruto, sem compressão. Ambas as estruturas em questão sofrem penalidade no desempenho em troca de uma melhor compressão, quando comparadas com a utilização de uma estrutura auxiliar sobre uma estrutura não comprimida, como na proposta de Vigna [17]. Assim, quisemos analisar o efeito da compressão no tempo de execução das buscas de *rank*.. Para tal, utilizando como parâmetros configuráveis a densidade do *bitvector* bruto inicial, tamanho do *bitvector* e número de operações, com o objetivo de posicionar o *Roaring Bitmap* em relação às outras duas estruturas concorrentes e analisar a eficiência deste na prática.

Tanto o RRR-vector como o V5 foram executados utilizando a biblioteca *SDSL*[7].

3.2.2 Modificações aos *Roaring Bitmaps*

Como visto na introdução deste capítulo, quando a densidade de 1's no *bitvector* é menor ou igual a 6,25% por *container*, o *Roaring Bitmap* armazena os 16 bits menos significativos do valor a ser guardado em um *array* de inteiros, de forma que os 16 bits mais significativos restantes ficam guardados no cabeçalho do *container*, uma vez que todos os valores dentro do *container* em questão possuem o mesmo índice e, consequentemente, são endereçados pelo mesmo cabeçalho. Além disso, o *Roaring Bitmap* salva a cardinalidade local dos *containers*. Quando a densidade local ultrapassa 6,25% por *container*, é necessário utilizar um *bitset container* em

vez de um *array container*, ou seja, um *container* que contém um *bitvector* bruto.

Ao gerar um *Roaring Bitmap* cuja densidade em todos os *containers* é maior que 6,25%, teremos apenas *bitset containers*. Como a estrutura do *bitset container* é não comprimida e realiza operações de *rank* de forma próxima à solução trivial de contar bit a bit, podemos concluir que não há vantagem em termos de compressão, por utilizar um vetor bruto, e não possui a vantagem em tempo de execução que se obtém com uma estrutura auxiliar, como visto em [17], então é razoável considerar que adicionar uma camada extra de complexidade pode melhorar seu desempenho sem perda de qualidade no quesito compressão da estrutura. Por este motivo, o *bitset container* recebeu foco neste trabalho como a primeira chave para melhorar o desempenho do *Roaring Bitmap* para consultas *rank*, como veremos a seguir.

Em primeiro lugar, ressaltamos que apenas a versão em C/C++ (chamada *CRoaring* [11]) foi explorada. Além disso, a primeira modificação ao *Roaring Bitmap* mantém a ideia do projeto inicial, mas aproveita o fato que, para cardinalidades acima de 4096 por *container*, o *Roaring* faz uso de *bitvectors* brutos, isto é, não comprimidos, além de não haver uma função que calcule o *rank* nestes *bitsets* de forma não-trivial. Na verdade, a solução proposta em [1] não é a mais "burra", pois utiliza 2^{10} operações de *popcount* sobre palavras de tamanho 64 bits para calcular os *ranks* locais.

Todos os *bitset containers* foram modificados para, em vez de apontarem para um vetor de palavras binárias, passarem a armazenar um índice para uma posição específica em um vetor da forma *SDSL vector*, que é um *bitset* bruto da biblioteca *SDSL* [7], de forma a tornar possível utilizar a estrutura auxiliar *rank-support-v5*. Os *bitset containers* foram concatenados e separados em *slots*, e passaram a ser endereçados por um número de bloco, com o objetivo de tornar possível realizar operações de *rank* locais em tempo constante, calculando o *rank* da posição atual e subtraindo do *rank* da posição final do *container* precedente. Note que, esta operação é necessária, pois não necessariamente todos os *containers* serão de tipo *bitset* e logo não serão transformados no formato de *bitvector SDSL* [7].

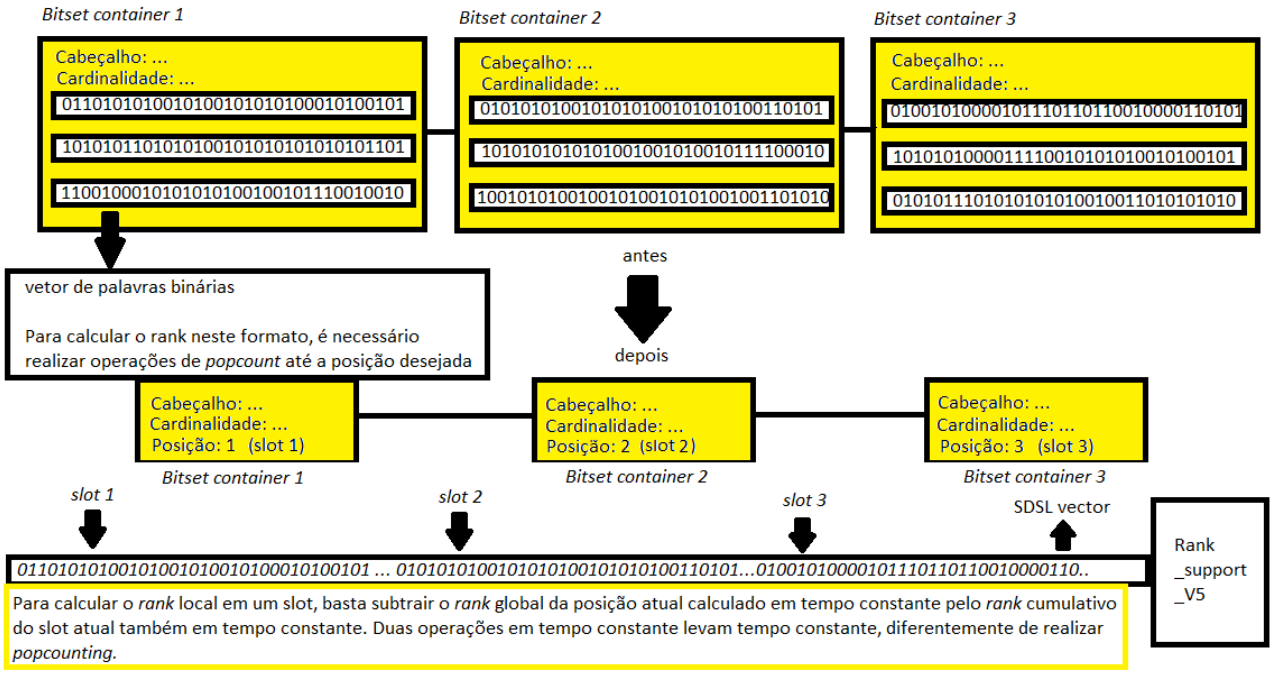

Figura 3.2 Note que um único vetor *SDSL* é utilizado para armazenar todos os *bitvectors* que pertenciam aos *bitset containers*. O *rank* local é calculado utilizando a estrutura *rank-support-v5* gerada a partir do vetor *SDSL*.

Na versão em [1] dos *Roaring Bitmaps*, cada *bitset container* é construído a partir de um *struct*, característico da linguagem C, que armazena a cardinalidade do *container* a partir de um inteiro de 32 bits e, além disso, utiliza um ponteiro para um vetor de *strings* binárias de 64 bits, que é o *bitvector* bruto associado ao *bitset container* "atual".. Neste projeto, com o objetivo de melhorar o desempenho da estrutura em questão para as operações de *rank*, foi adicionado ao *struct* representativo de cada *bitset container* um *int32* que serve como identificador para cada *bitset container*, e os vetores brutos foram concatenados. Dessa forma, as *strings* binárias dos *bitset containers* foram armazenadas em um único *bitvector* bruto da biblioteca *SDSL*, e este foi dividido de forma indexada em blocos de tamanho 2^{16} bits, associados a cada *bitset container*.

Em seguida, algumas funções foram criadas ou modificadas para otimizar o *CRoaring*. Essas funções estão detalhadas abaixo.

Criamos a função *roaring-bitmap-rank-start*, que recebe um *Roaring Bitmap* e converte os *containers* de tipo *bitset* para um *bitvector* bruto *SDSL* e, em seguida, cria uma estrutura auxiliar com uso de memória extra de 6,25% [7] sobre o vetor bruto recém criado. Isto é, se o *Roaring Bitmap* inicial continha 2 *bitset containers*, a função cria um novo vetor que contém a representação concatenada de ambos os *bitsets containers*, exclui a representação anterior, e, em seguida, adiciona a estrutura auxiliar (*rank-support-v5*), que permite realizar operações de *rank* em tempo constante [17]. Na versão original em [1] dos *Roaring Bitmaps*, as operações de *rank* em cada *bitset container* eram realizadas contando bits em um vetor de 2^{16} bits, utilizando 2^{10} *popcounts* sobre palavras de tamanho 64 bits.

As funções do *CRoaring* original alteradas para se adaptar à modificação que inclui um vetor *SDSL* [7] estão citadas abaixo:

1) ***bitset-container-create***: função chamada sempre que há a necessidade de criar um novo *bitset container*. Com a modificação, além de criar um novo *container*, um identificador e o *rank* cumulativo são adicionado ao *bitset container* para facilitar algumas operações, como por

exemplo, o cálculo do *rank* global até o *bitset container* em questão.

2) ***bitset-container-add***: adiciona um número ao conjunto representado pelo *bitset* e retorna *true* se o número adicionado não estava presente. Esta função utiliza o *bitvector* bruto da biblioteca *SDSL* [7].

3) ***bitset-container-set***: esta função modifica o i-ésimo bit do *bitset container* dado como entrada para 1. Esta função utiliza o *bitvector* da biblioteca *SDSL* [7] e permite o acesso a cada *bitset container* por meio de um identificador de *bitset container*.

4) ***bitset-container-rank***: função que realiza o *rank* local de determinado *bitset container*. Em vez de fazer uma contagem de bits, aproveita a estrutura auxiliar *rank-support-v5* [7, 17].

Por fim, utilizando uma tática proposta por Munro em 1996 [14], uma função de preparação dos *Roaring Bitmaps* foi criada para, além de converter a estrutura para o novo formato desejado, conforme explicado no início deste capítulo, também alterar a forma como as cardinalidades são armazenadas dentro de cada *container*. Assim, em vez de salvar a cardinalidade do *array* ou *bitset* locais, armazena a soma das cardinalidades anteriores, o que permite realizar a consulta das somas de cardinalidades em cada *container* em tempo constante.

CAPÍTULO 4

Experimentos e Resultados

Neste capítulo, enunciamos o desempenho do *Roaring Bitmap* e das versões modificadas neste trabalho, as quais chamamos *SVRS-Roaring e SVRS-Roaring2*, para a operação de *rank*, em três cenários, variando-se a densidade: 1) *Roaring Bitmap* e *SVRS-Roaring* contendo um número pequeno de *containers* (4 *containers*) .; 2) *Roaring Bitmap* e *SVRS-Roaring* contendo 128 *containers*. 3) *Roaring Bitmap* e *SVRS-Roaring2* contendo 16384 *containers*. Ressaltamos que o *SVRS-Roaring2* é uma versão incremental do *SVRS-Roaring*.

Inicialmente, realizamos 14 execuções de três programas escritos em C++, um para cada uma das estruturas exploradas em [1, 16, 17], combinando sete densidades de *bitsets* para duas quantidades de consulta, variando a densidade dos *bitsets* de 1% até 50% e estimamos a capacidade de compressão de cada estrutura e o tempo necessário para executar de 2^{18} a 2^{23} operações de *rank*, em posições escolhidas de forma pseudoaleatória.

O *Roaring Bitmap* mostrou-se inferior a *V5* e *RRR-vector* tanto em memória quanto em tempo de execução ao realizar operações de *rank* para densidades acima de 6,25%, para os casos em que o *bitvector* bruto de referência ultrapassa 2^{23} bits, ou 1MB. Quando o *bitvector* bruto de referência abarcava 2^{18} bits , o *Roaring Bitmap* apresentava melhor tempo de execução para densidades de *bitvector* abaixo de 6,25%, fator que correspondeu com o fato de densidades acima de 6,25% necessitarem de *bitset containers* em lugar de *array containers*, o que necessariamente implica o uso de operações de *popcount*. Além disso, salientamos que o número de *containers* correspondente a um melhor desempenho dos *Roaring Bitmaps* pode ser considerado pequeno para que a operação de soma de cardinalidades, passando por cada *container*, pudesse interferir no resultado global do tempo de execução. Sendo assim, surge a

seguinte hipótese: o número de *containers* decorrente do tamanho do *Roaring Bitmap* impacta na complexidade de tempo de execução de *ranks* mais do que realizar cálculos de *ranks* locais em cada *bitset container*.

Em seguida, avaliamos *bitvectors* densos (i.e. cuja densidade ultrapassa 6,25%) e variamos a magnitude da posição associada a uma operação de *rank* (de 2^{16} até 2^{30}), tendo como objetivo observar o efeito da complexidade de tempo das estruturas estudadas na prática em função do índice de entrada. Concluímos que é possível tornar constante o tempo para realizar as operações de *rank* independentemente do número de *containers* no *Roaring Bitmap*, utilizando uma técnica descrita em [14], armazenando valores de *rank* cumulativos em cada *container* em lugar de armazenar a cardinalidade local de cada *container*.

4.1 Experimentos

Para este trabalho, limitamos nosso estudo ao caso em que os *bitvectors* possuem uma distribuição uniforme de 1's, com o objetivo de tornar imparcial a análise e evitar resultados inconsistentes ou aleatórios.

Criamos programas para realizar testes em C/C++, que criam estruturas de *bitvectors* e realizam operações de *rank*. Para cada estrutura em [1, 17, 16], foi criado um programa que a reproduza. Para isso, utilizamos a biblioteca *SDSL* [7], que dá suporte a várias estruturas associadas a *bitvectors*. Além desta, utilizamos o repositório oficial dos *Roaring Bitmaps* para a linguagem C/C++, cuja versão é nomeada *CRoaring* [11].

Os *bitvectors* foram preenchidos de forma pseudoaleatória com uma distribuição uniforme. Ou seja, a probabilidade de que um valor esteja presente em cada posição era sempre constante. Desta forma, foi possível gerar *bitvectors* com densidades uniformes variando de 1% até 50%.

Os programas criados utilizando funções da biblioteca *SDSL* [7] utilizam uma estrutura que dá suporte à operação *rank* por cima do *bitvector* bruto, enquanto que o *Roaring Bitmap* já

possui esta operação disponível na própria classe. Uma vez que os *bitmaps* e as estruturas de suporte estejam criados, são realizadas diversas operações de *rank* em um loop . Nos cenários iniciais, as operações foram programadas para serem realizadas em posições previsíveis e com espaçamento inversamente proporcional ao número de operações desejado, de forma a cobrir todo o vetor. Por exemplo, se o número de operações de *rank* for igual a 10 em um vetor de tamanho 100, isto implica que seria realizada uma operação a cada múltiplo de 10 dentro do vetor (neste caso, as posições 0, 10, 20, 30, 40, 50, 60, 70, 80 e 90). Na prática, o número de operações de *rank* variou de 260 mil a 8 milhões por teste, e o tempo de execução para executar o conjunto de operações é saída nos programas de teste.

No entanto, para fins de analisar a influência dos *bitset containers* no cálculo dos *ranks* nos *Roaring Bitmap* para o caso em que o *bitvector* era de magnitude superior à capacidade de armazenamento em memória cache, um cenário adicional foi configurado com *bitvectors* de tamanho 128 MB, e alguns índices de busca foram selecionados com base em posições associadas aos *containers*, variando-se de 2^{16} até 2^{30}

Todas as operações de *rank* executadas foram realizadas sequencialmente sem interrupções de outras funções no programa de teste e, assim, foi possível medir o tempo total de execução do conjunto de operações utilizando a função *clock()* da biblioteca *time.h*. Além de medir o tempo de execução das operações, as bibliotecas associadas a cada estrutura possibilitaram medir o espaço de memória utilizando funções que recebem um *bitmap* e retornam o tamanho dele em bytes (ex. *roaring-bitmap-portable-size-in-bytes*, para o *Roaring Bitmap*), avaliando, por exemplo, funções de alocação de memória dinâmica ou estática.

Nos dois primeiros cenários, avaliamos os desempenhos do *CRoaring* padrão, *RRR-vector* e *rank-support-v5* para as operações de *rank* com densidades variando de 1% a 50%, para *bitsets* de tamanho 2^{18} a 2^{23} bits e, em cada uma, variando o número de operações de 2^{18} a 2^{23}. No terceiro cenário, avaliamos também os desempenhos do *CRoaring*, *RRR-vector* e *rank-support-v5*, com a diferença que a densidade foi fixada para 50%, que representa o pior caso geral em

termos de desempenho para estruturas com suporte a *ranks*, conforme as observações dos dois cenários anteriores. No terceiro cenário, o número de operações escolhido também foi de 8 milhões, com o objetivo de tornar os resultados equiparáveis aos dois primeiros.

Os experimentos foram executados em um computador com sistema operacional Bodhi Linux, com arquitetura x86-64, processador AMD E2-7110, placa de vídeo AMD Radeon R2, com cache L1 de 256 KB, L2 de 2 MB, 4 GB de RAM. Os programas foram criados em C/C++ utilizando o compilador GNU GCC, utilizando as configurações padrão do compilador em todos os cenários e testes.

4.2 Resultados Comparativos com os *Roaring Bitmaps* originais

A Figura 4.1 mostra o resultado do experimento que avaliou a capacidade de compressão dos *Roaring Bitmaps*, *RRR-vector* e *rank-support-v5*, considerando um *bitvector* bruto inicial de 1MB (2^{23} bits).

Observamos que o *Roaring Bitmap* possui uma taxa de crescimento do uso de memória maior que a estrutura *RRR-vector* para densidades até 6,25% e se estabiliza para densidades maiores, porque a partir de 6,25%, todos os *bitvectors* possuem a mesma quantidade de bits (i.e. temos um vetor de tamanho constante), enquanto que para menores densidades, temos um array de inteiros de tamanho variável, de acordo com a especificação em [1]. , enquanto que o *rank-support-v5* é não comprimido e, portanto, utiliza uma quantidade constante de memória independentemente do *bitvector*. Assim, era razoável que o *Roaring Bitmap* fosse capaz de justificar esta compressão inferior com um desempenho superior em tempo de execução comparativamente ao *RRR-vector*. Mas como mostra a Figura 4.2, o *Roaring Bitmap* mostrou-se também inferior ao *RRR-vector* e *rank-support-v5* em tempo de execução para *ranks*.

Ao analisar o código da função de *rank* do *Roaring Bitmap*, pudemos notar que há dois problemas principais que tornam o cálculo do *rank* ineficiente: 1. O algoritmo soma uma a

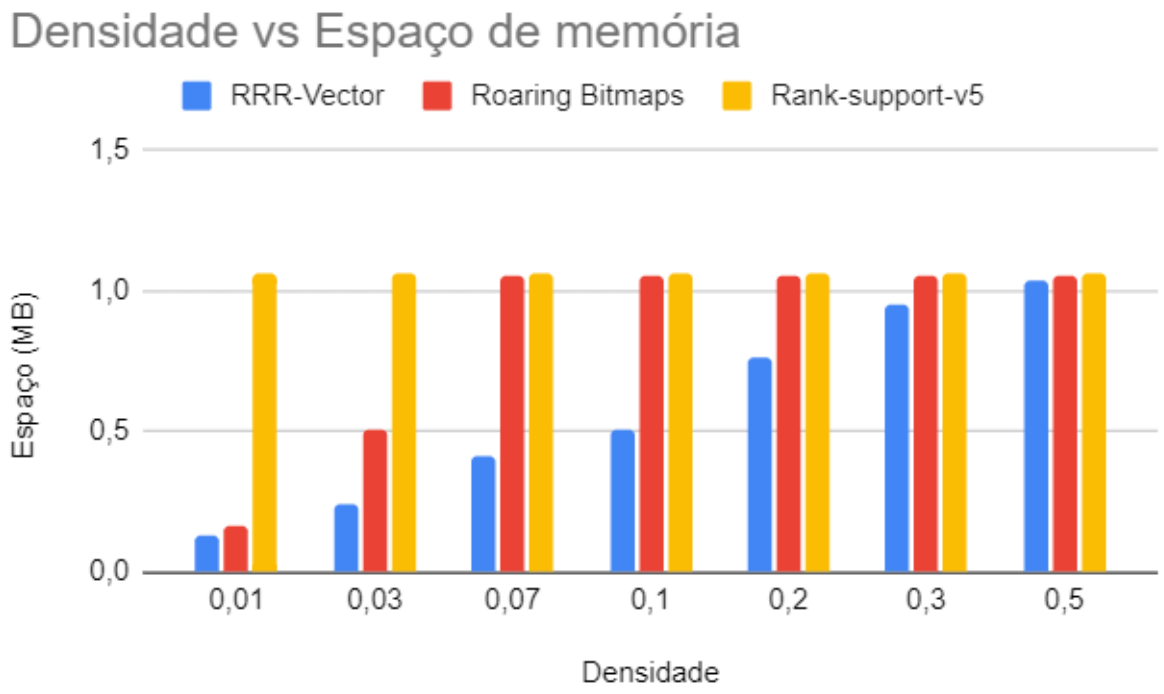

Figura 4.1 Experimento 1.1, capacidade de compressão das diferentes estruturas, considerando um *bitvector* bruto inicial de 1MB (2^{23} bits).

uma as cardinalidades de cada *container* precedente àquele que contém a posição desejada, ou seja, o *container* que possui a posição da qual se deseja calcular o *rank*; 2. O algoritmo, por fim, calcula o *rank* local do *container* da posição desejada fazendo uma busca binária no caso de um *array container* ou somando todos os bits 1 no caso de um *bitset container*. E este último caso é a forma mais simples e natural de calcular *rank*, mas é ineficiente quando comparado às propostas discutidas em [16] e [17] .

Então, com base nas informações supracitadas, podemos inferir que para um experimento em que o número de *containers* é mínimo e todos os *containers* são do tipo *array* teremos o melhor cenário de desempenho do *Roaring Bitmap* original, enquanto que o cenário se desfavorece à medida em que aumentamos o número de *containers* e aumentamos a densidade, comparativamente às duas propostas concorrentes.

A Figura 4.3 mostra um novo experimento, com o objetivo de verificar a redução do impacto

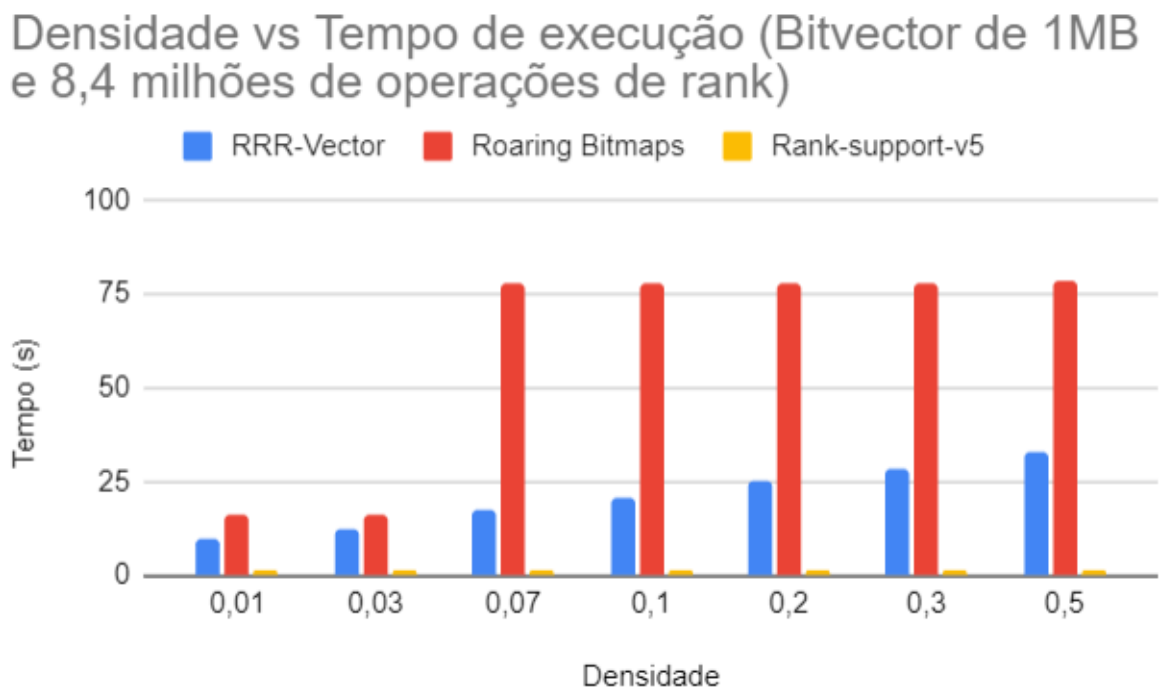

Figura 4.2 Experimento 1.2, o *Roaring Bitmap* mostrou-se inicialmente inferior ao *RRR-vector* e *rank-support-v5* em tempo de execução.

do número de *containers*, em que *bitvectors* foram criados de forma que cada *Roaring Bitmap* possua apenas 4 *bitset containers*. Podemos ver que, de fato, o *Roaring Bitmap* apresenta um desempenho melhor em tempo de execução que o *RRR-vector* para densidades pequenas, considerando que todos os *containers* são do tipo *array*. Podemos observar também que quando a densidade aumenta além de 6,25% e todos os *containers* passam a ser do tipo *bitset container*, o resultado é menos eficiente, o que é consistente com a mudança para um algoritmo que calcula o *rank* utilizando *popcounting*.

A partir desses resultados, alguns problemas a serem tratados tornaram-se proeminentes, a saber, o algoritmo que calcula *ranks* locais nos *bitset containers* e a função que soma cardinalidades de cada *container*. Os próximos experimentos tentaram verificar se a modificação na função *bitset-container-rank*, que por padrão realiza *popcounting* para calcular *ranks* locais, traria resultado satisfatório, com o objetivo de ultrapassar o desempenho do *RRR-vector* para

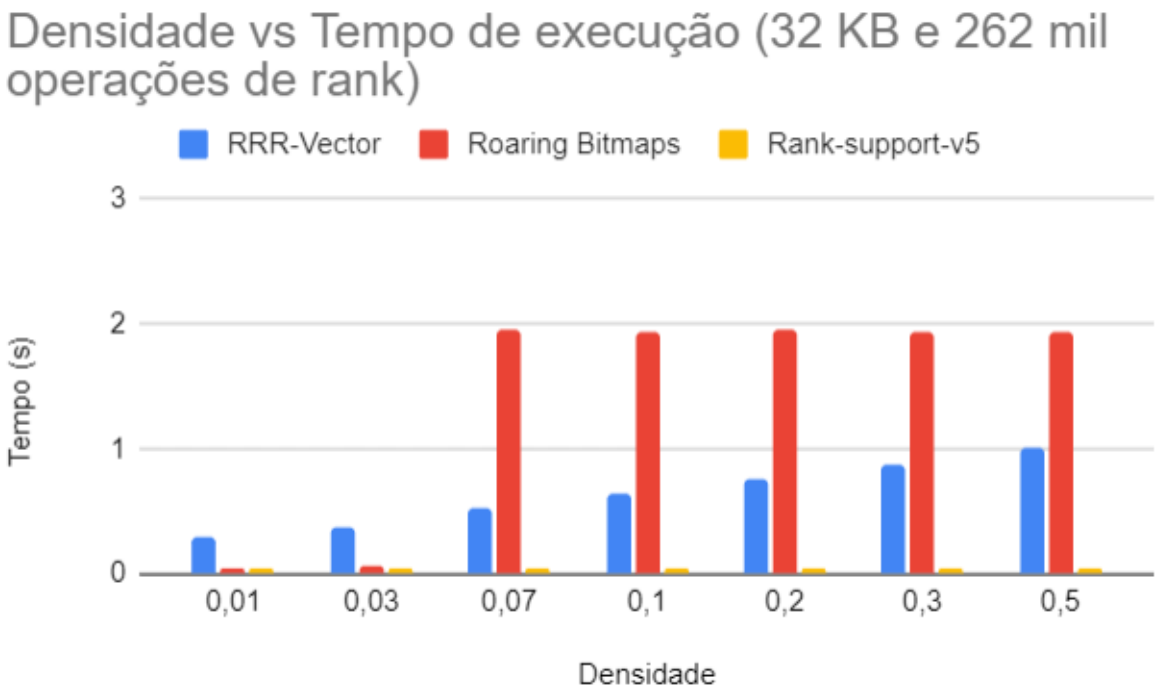

Figura 4.3 Experimento 2. *O Roaring Bitmap* tem resultado superior em tempo de execução ao *RRR-Vector* e competitivo com *rank-support-v5* para densidades baixas.

densidades acima de 6,25%.

4.3 Resultados com *CRoaring modificado (SVRS-Roaring)*

A Figura 4.4 mostra o resultado de um experimento idêntico ao da Figura 4.3, mas com o *CRoaring* modificado em lugar do *Roaring Bitmap* original. Após a otimização do *rank* dentro de um *bitset container*, este experimento mostrou que nossa estrutura foi capaz de realizar as mesmas operações de forma 23 vezes mais eficiente dentro do cenário explorado, uma vez que em vez de realizar *popcounting* no *bitset container*, a operação utilizou a função *rank* do *rank-support-v5* aplicada ao *bitset container* local. Vale ressaltar que neste cenário, o número total de *bitset containers* é no máximo 4, com o objetivo de minimizar o efeito do número de *containers* sobre o desempenho dos *Roaring Bitmaps*, o que será analisado em seguida.

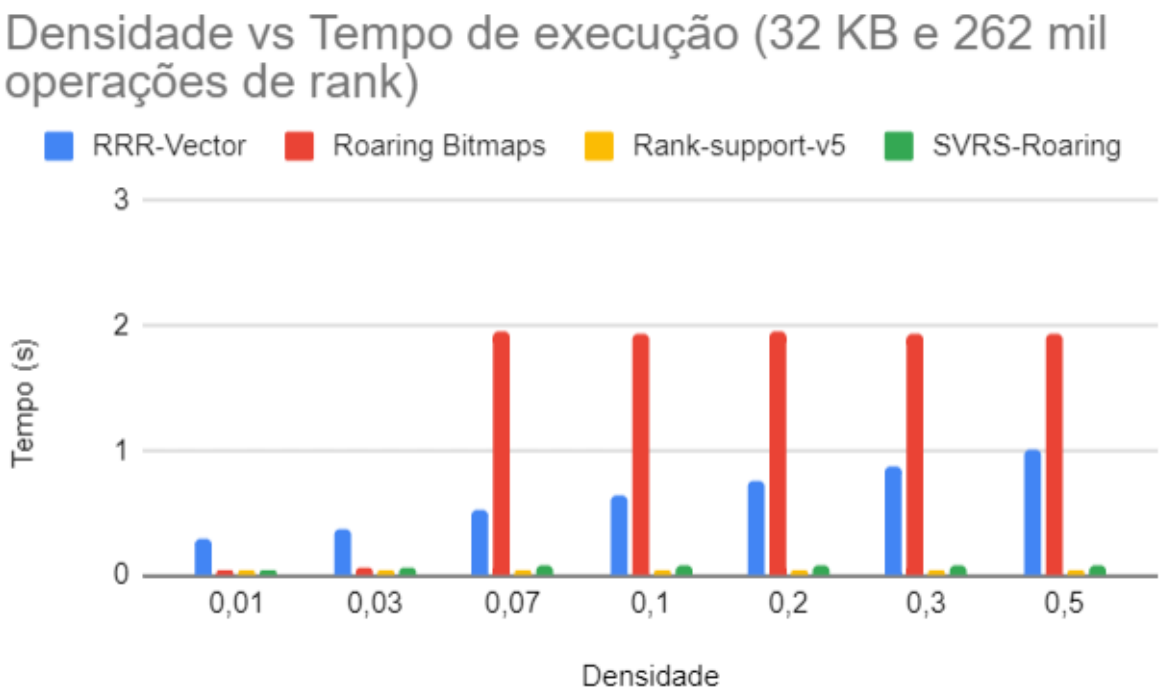

Figura 4.4 Experimento 3. Resultado do mesmo experimento da Figura 4.3, mas com o *CRoaring* modificado.

Para analisar a limitação do *Roaring Bitmap* de que a operação de *rank* se torna mais ineficiente à medida que o número de *containers* cresce, a Figura 4.5 mostra mais um experimento, desta vez com um *bitvector* inicial de tamanho 32 vezes maior que o do experimento anterior, com 128 *containers*.

Como podemos observar, a influência do número de *containers* fez com que o tempo total dos *ranks* aumentassem a ponto de o que antes estava 23 vezes mais rápido que a versão original do *Roaring Bitmaps* passasse a ser apenas 4 vezes mais eficiente.

Em seguida, um novo experimento foi realizado com o *Roaring Bitmap* para entender a variação no tempo de execução dos *ranks* da estrutura à medida que o índice cresce, pois, a cada busca em cada posição, é necessário realizar a soma de todas as cardinalidades dos *containers* anteriores. Assim, a Figura 4.6 mostra a variação do tempo de execução do *rank* de uma única posição à medida que este valor aumenta em diferentes testes, variando-se o valor

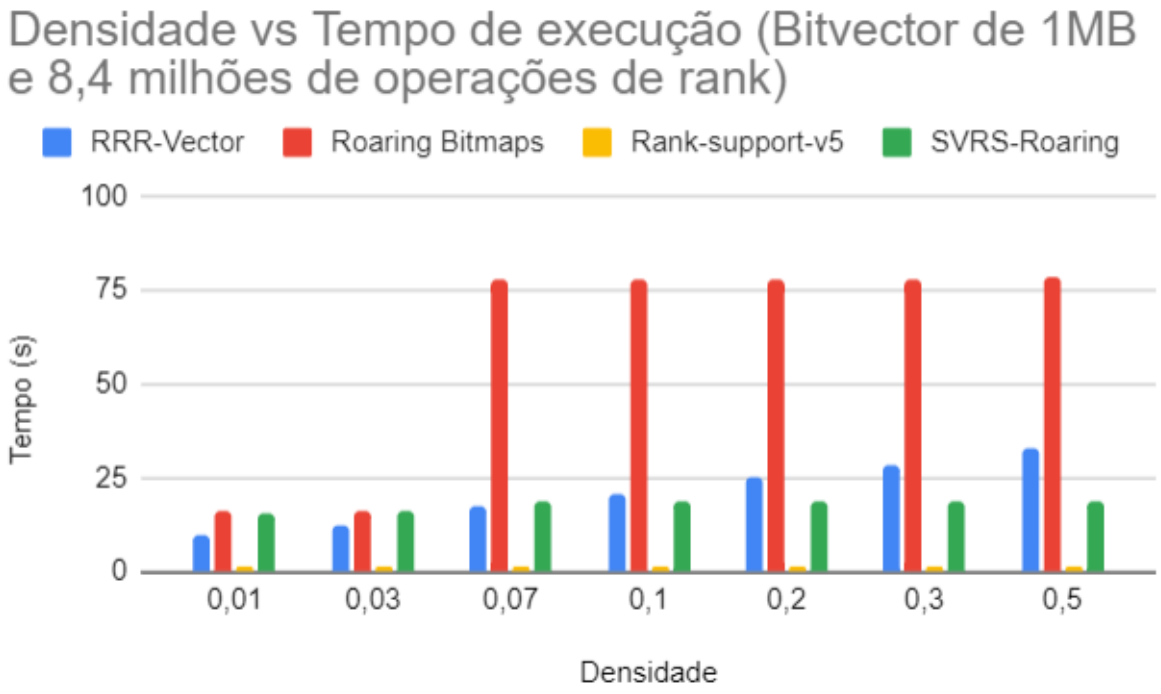

Figura 4.5 Experimento 4. Experimento com um *bitvector* inicial de tamanho 32 vezes maior que o experimento anterior (128 *bitset containers*)

do índice de 2^{16} até 2^{30}.

Com base no resultado observado na Figura 4.6, outra modificação foi feita na estrutura do *Roaring Bitmap*, para armazenar o valor cumulativo dos *ranks* até o *container* atual, abordagem proposta por Munro em [14]. Dessa forma, as Figuras 4.7 e 4.8 mostram o resultado da modificação ao *Roaring Bitmap* que permitiu que as operações de obter a soma dos *ranks* dos *containers* anteriores pudesse ser realizada em tempo constante. Note que o tempo de execução para cálculo dos *ranks* cresce em função do valor do índice, o que é explicado pelas somas das cardinalidades em um número crescente de *containers*, mas com a modificação, estes valores se tornaram constantes. As Figuras 4.7 e 4.8 se referem ao mesmo resultado em diferentes escalas.

A Figura 4.9 mostra um experimento comparativo do tempo de execução entre as estruturas competitivas analisadas e a última versão modificada dos *Roaring Bitmap*, a qual chamamos

Figura 4.6 O tempo necessário para calcular os *ranks* varia de 1,8 segundos até 1,12 horas para o mesmo número de consultas.

SVRS-Roaring2. Dessa vez, o índice dos *ranks* foi testado individualmente em 8 posições distintas executando a busca para cada posição, com o objetivo de estudar a variação dos resultados de cada estrutura para índices crescentes. . E a Figura 4.10 inclui o resultado do *Roaring Bitmap* original para o experimento em questão e mostra os mesmos resultados da figura anterior em uma escala diferente (porque a ineficiência do *Roaring* original estava tornando os outros resultados no gráfico difíceis de observar).

Por fim, a Figura 4.11 mostra um comparativo do uso de memória de cada uma das estruturas analisadas. Frise-se que *SVRS-Roaring* utiliza praticamente a mesma quantidade de memória que o *Roaring Bitmap* original em todos os cenários.

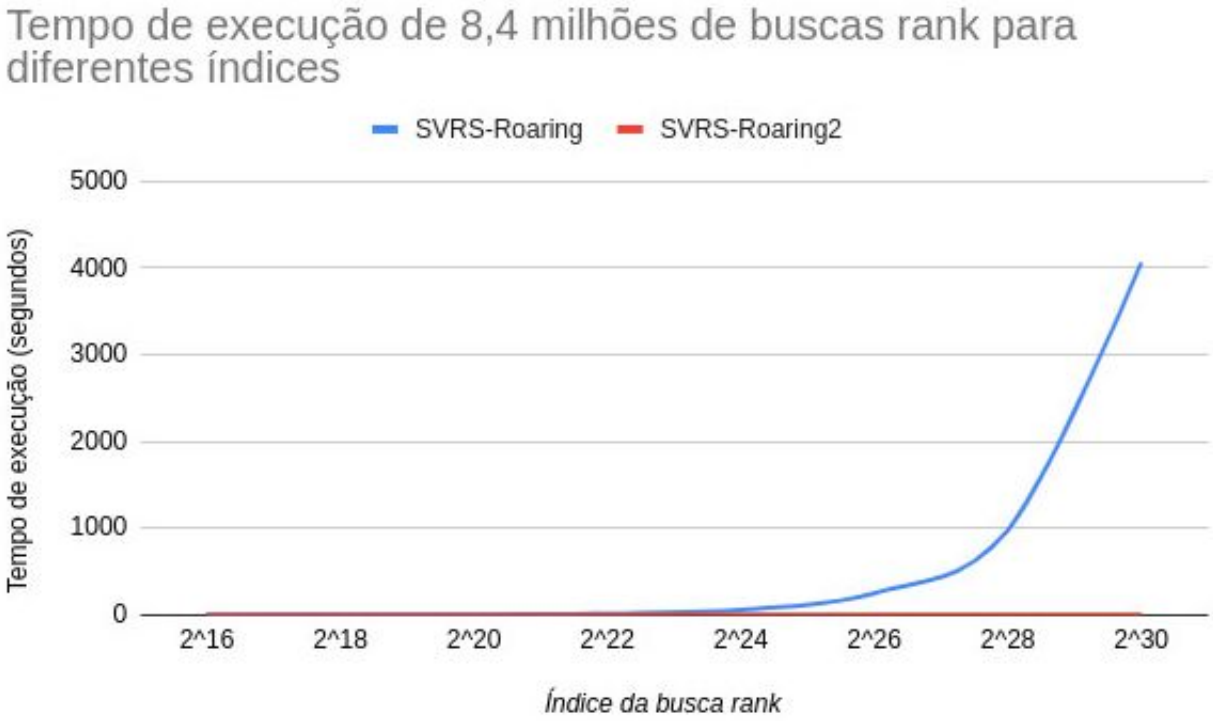

Figura 4.7 Este gráfico mostra que o tempo de execução passou a ser constante, independentemente da posição do índice do *rank* no *Roaring Bitmap*

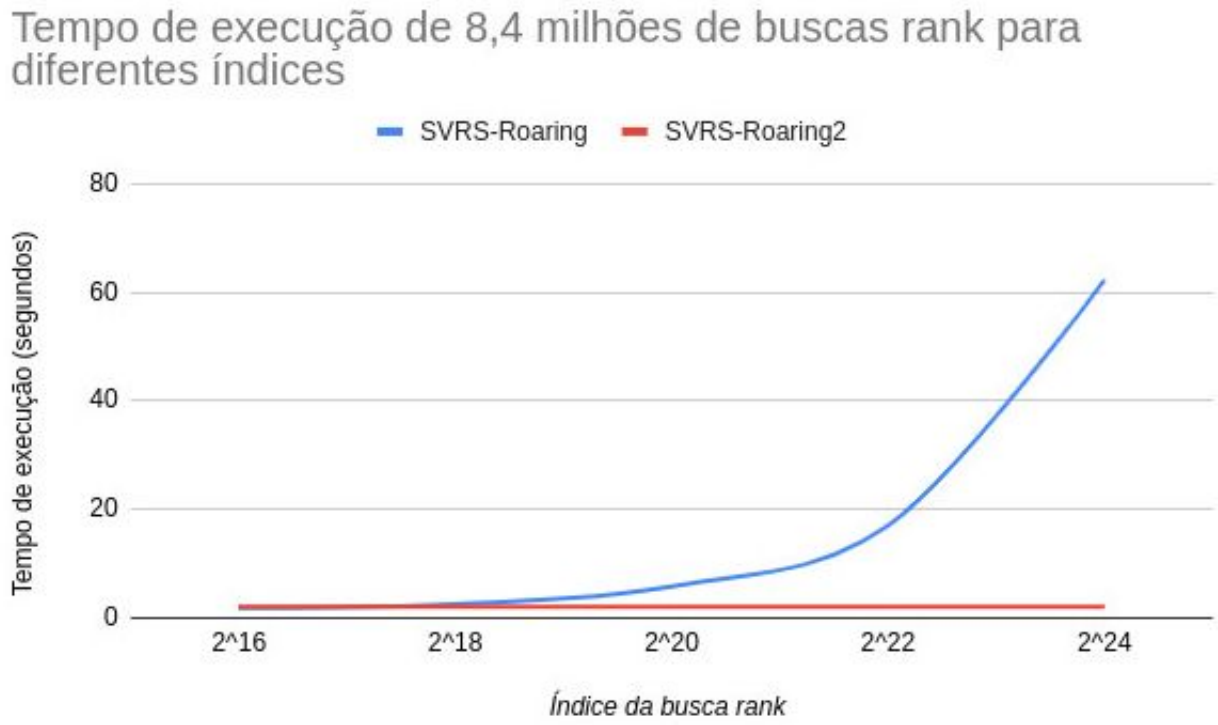

Figura 4.8 Compare a escala de tempo relativamente à Figura 4.7. Aqui a variação é de alguns segundos e podemos observar que, para posições abaixo de 2^{18}, o número de *containers* não influencia negativamente no tempo de execução.

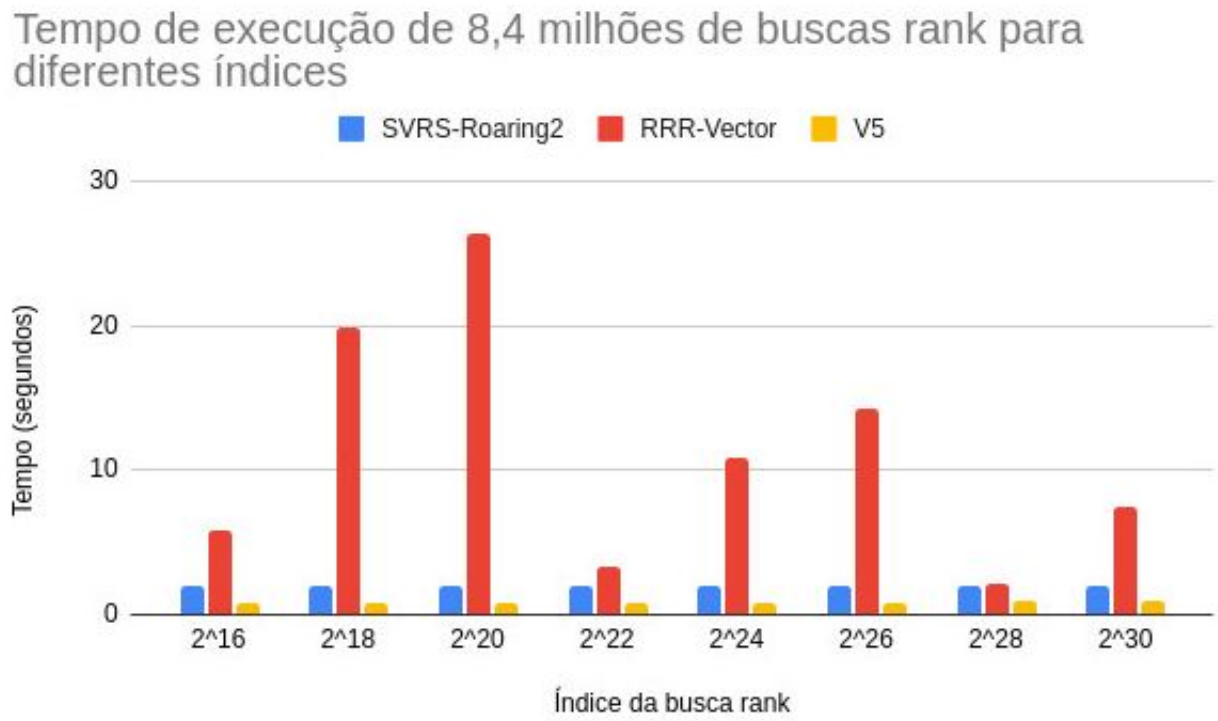

Figura 4.9 Comparativo do tempo de execução final entre as estruturas competitivas elencadas no início do capítulo (*RRR-vector* e *Rank-support-v5*) com o *SVRS-Roaring2*. Note que apenas o *RRR-vector* mostrou desempenho variável de acordo com a posição do valor do *rank* calculado.

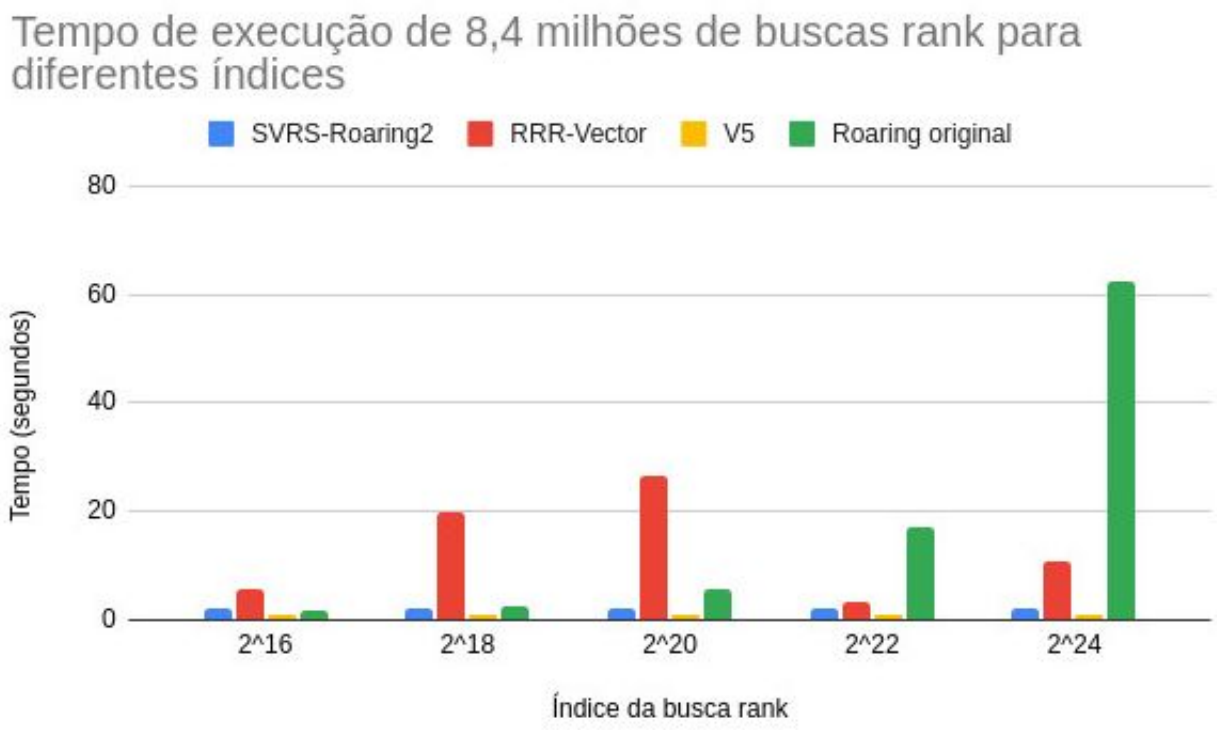

Figura 4.10 Mesmo resultado da figura anteior, mas adicionamos o tempo de execução do *Roaring* original em comparação ao *SVRS-Roaring2*. Note que foi necessário modificar a escala do gráfico para que o resultado do *Roaring Bitmap* original não tornasse os outros valores invisíveis.

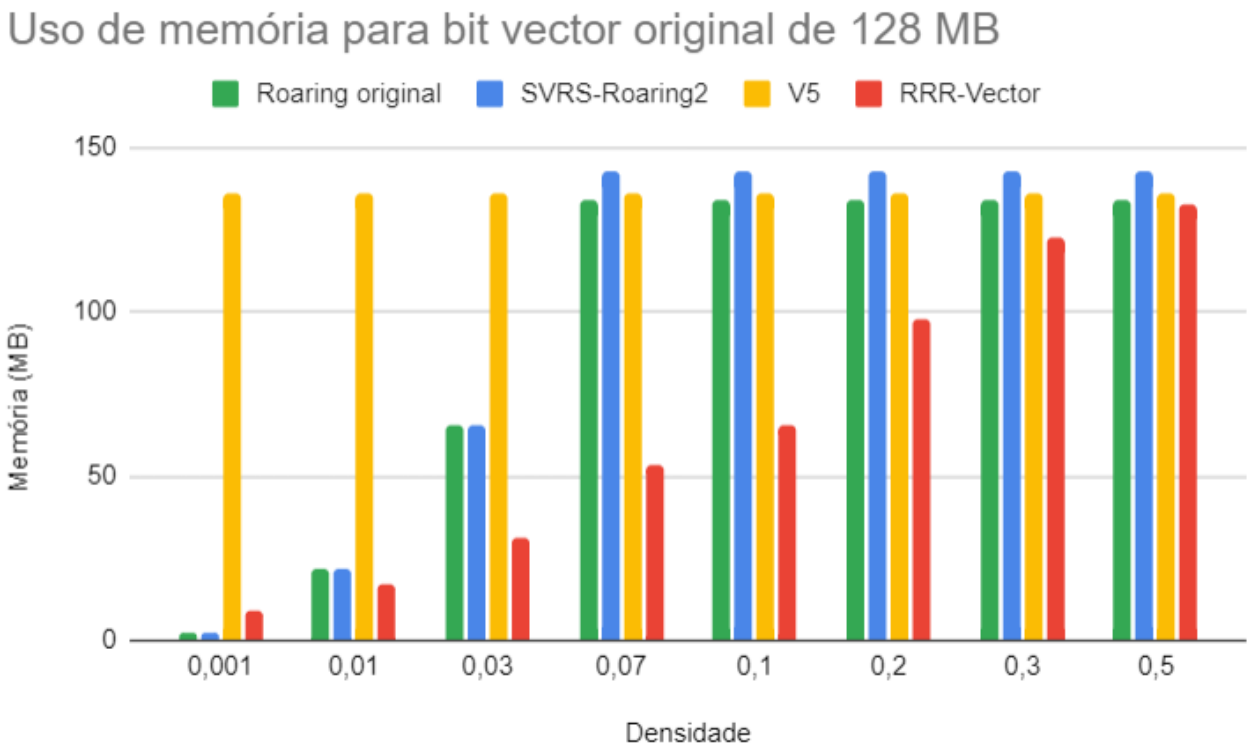

Figura 4.11 Comparativo do uso de memória entre a versão final modificada do *Roaring Bitmap* em comparação com as estruturas já existentes antes deste trabalho. Comparando com as figuras anteriores, note que o novo *Roaring Bitmap* é capaz de realizar *ranks* em tempo constante com praticamente o mesmo uso de memória da estrutura original

CAPÍTULO 5

Conclusões

Em primeiro plano, o *Roaring Bitmap* se mostrou inferior às estruturas *RRR-Vector* e *V5* tanto em memória quanto em tempo de execução para densidades acima de 6,25% em consultas *rank* quando o *bitvector* inicial era relativamente grande (a partir de 1MB em diante). Para o caso em que o *bitvector* inicial era de tamanho 2^{18} bits (32 KB), o *Roaring Bitmap* se mostrou superior às estruturas mencionadas em tempo de execução para densidades de *bitvector* abaixo de 6,25% apenas, fato que se relaciona ao número de *containers* ser pequeno (ex. 4 *containers*).

Dessa forma, utilizar a estrutura *V5*, baseada na proposta de S. Vigna [17], juntamente com os *bitset containers* dos *Roaring Bitmaps* melhorou a obtenção do *rank* local de cada *container* e a adaptação se mostrou eficaz para casos em que o número de *containers* não influenciava negativamente no cálculo do *rank*, ou seja, como no caso de 128 *containers*, fato que se explica pelo fato do algoritmo original que calcula o *rank* ser dividido em duas partes: 1) calcular o *rank* local do *container* utilizando busca binária para baixas densidades (em um *array container* e *popcounting* para altas densidades, em um vetor de palavras binárias de 64 bits. 2) somar todas as cardinalidades de todos os *containers* anteriores ao atual. Esta segunda etapa se torna um gargalo no tempo de execução à medida que o número de *containers* cresce. Dessarte, a segunda modificação, que possibilita o armazenamento da soma cumulativa de cardinalidades por *container* se mostrou capaz de eliminar este problema, e foi possível obter o *rank* global em tempo constante (figuras 4.7 e 4.8).

.

Nesse contexto, ressalte-se que a utilização da proposta de Munro [14] nos *Roaring Bitmaps* requer que as consultas *rank* sejam realizadas quando não há mais necessidade de realizar

atualizações (*updates*) no *Bitmap*. Pois cada novo *update* invalida a soma de cardinalidades por *container* e torna necessário o recálculo destes valores em todos os de *containers* para cada atualização (com possivelmente dezenas de milhares operações), o que inviabilizaria o uso da nova função criada, que prepara o *Bitmap* para as operações de *rank*.

Frise-se também que as estruturas obtidas com as modificações ocupam virtualmente a mesma quantidade de memória (6,25% de memória extra), como pode ser visto na figura 4.11, que os *Roaring Bitmaps* originais, enquanto que o desempenho melhorou de uma função linear para uma função constante, e uma melhora em calcular os ranks locais.

Destaque-se que o cenário de *bitvectors* com baixa densidade não foi explorado em profundidade neste trabalho, como foi o caso de [11], pelo fato de que, no cenário em questão, os *containers* são do tipo *array*, cuja qualidade de compressão é superior às estruturas concorrentes utilizadas para comparação (*V5* e *RRR-vector*). Mas, com base nos resultados envolvendo número elevado de *array containers*, é de se concluir que o algoritmo possui uma limitação em tempo de execução, que pode ser eliminada utilizando a técnica proposta por Munro [14] de acumular valores de *rank* espaçadamente ao longo do *bitvector*.

Uma perspectiva de trabalho futuro inclui a possibilidade de explorar a operação de *select*, que é comumente encontrada na literatura envolvendo operações de *rank*, por serem ambas operações requeridas no contexto de estruturas de dados sucintas. Outra possibilidade é a de melhorar a forma como os novos *Roaring Bitmaps* são preparados para realizar operações de *rank*, de forma a permitir múltiplas atualizações (i.e. operações de *update*) entre buscas *rank*, o que não foi obtido neste trabalho.

Referências Bibliográficas

[1] Kaser O Godin R. Chambi S, Lemire D. Better bitmap performance with roaring bitmaps. In Software: Practice and Experience, 2015. (document), 1, 2.3, 3.1, 3.1, 3.2.1, 3.2.2, 3.2.2, 4, 4.1, 4.2

[2] Owen Kaser Daniel Lemire, Gregory Ssi-Yan-Kai. Consistently faster and smaller compressed bitmaps with roaring. In Software: Practice and Experience Volume 46, Issue 11, 2016. 1

[3] Peter Elias. Efficient storage and retrieval by content and address of static files. 1974. 2.2

[4] Daniel Lemire et al. Site oficial dos roaring bitmaps. 1

[5] R. Gonzales et al. Practical implementation of rank and select queries. In Poster Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA), pages 27–38, 2005. (document), 1, 2, 2.2

[6] Gonzalo Navarro Francisco Claude. Practical rank/select queries over arbitrary sequences. In SPIRE 2008, pages 176–187, 2014. 3.2.1

[7] Simon Gog. Sdsl - succinct data structure library. 2016. 2.4, 3.2.1, 3.2.2, 3.2.2, 4.1

[8] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In 13th International Symposium on Experimental Algorithms, (SEA 2014), pages 326–337, 2014.

[9] Eliana Providel Gonzalo Navarro. Fast, small, simple rank/select on bitmaps. In SEA 2012. (document), 3.2.1

[10] Szymon Grabowski and Marcin Raniszewski. Rank and select: Another lesson learned. 2016. 2.1

[11] https://github.com/RoaringBitmap/CRoaring. Portable roaring bitmaps. 2015. 3, 3.2.2, 4.1, 5

[12] G. Jacobson. Space-efficient static trees and graphs. In In Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS), page 549–554, 1989. (document), 2.2

[13] M. MINSKY and S. PAPERT. Perceptrons. In MIT Press, Cambridge, Mass., pages 215–225, 1969.

[14] J. Ian Munro. Tables. In V. Chandru and V. Vinay, editors, Foundations of Software Technology and Theoretical Computer Science, pages 37–42, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. 1, 2.2, 3.2.2, 4, 4.3, 5

[15] Rasmus Pagh. Low redundancy in dictionaries with o(1) worst case lookup time. In Technical Report 1998. 3.2.1

[16] V. Raman Rajeev Raman and S. Srinivasa Rao. Succinct indexable dictionaries with applications to representations of k-ary trees and multi-sets. In SODA 2002. (document), 1, 2, 2.1, 2.2, 3, 3.2.1, 4, 4.1, 4.2

[17] S. Vigna. Broadword implementation of rank/select queries. 2008. (document), 1, 2, 2.2, 3, 3.2.1, 3.2.2, 3.2.2, 4, 4.1, 4.2, 5

[18] S. Zaks. Lexicographic generation of ordered trees. In Theor. Comput. Sci., page 10:63–82, 1980. 2.2