Técnicas de Otimização do Mergesort Externo num Ambiente de Banco de Dados

Elton Gustavo Fanelli

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Técnicas de Otimização do Mergesort Externo num Ambiente de Banco de Dados

Elton Gustavo Fanelli

Fevereiro de 2006

Banca Examinadora:

- Prof. Dr. Rogério Drummond (Orientador)
- Prof. Dr. João Eduardo Ferreira
  Departamento de Ciência da Computação – IME – USP
- Prof. Dr. Ricardo de Oliveira Anido
  Instituto de Computação – Unicamp
- Prof. Dr. Geovane Cayres Magalhães (Suplente)
  Instituto de Computação – Unicamp

UNIDADE BC
Nº CHAMADA T/UNICAMP
F213t
V ____ EX ____
TOMBO BC/ 68490
PROC 16.P.00123-06
C ____ ex ____
PREÇO 11,00
DATA 24/05/06
Nº CPD ____
BIBID-379648

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Fanelli, Elton Gustavo

F213t Técnicas de otimização do mergesort externo num ambiente de banco de dados / Elton Gustavo Fanelli-- Campinas, [S.P. :s.n.], 2006.

Orientador : Rogério Drummond

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Ordenação (Computadores). 2. Banco de dados. 3. MySQL (Linguagem de programação de computador). I. Drummond, Rogério. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: External mergesort optimization strategies in a database environment

Palavras-chave em inglês (Keywords): 1. Sorting (Electronic computers). 2. Databases. 3. MySQL (Computer programming language)

Área de concentração: Banco de dados, Análise de Algoritmos e Complexidade

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rogério Drummond (IC-UNICAMP)
Prof. Dr. José Eduardo Ferreira (IME-USP)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)
Prof. Dr. Geovane Cayres Magalhães (IC-UNICAMP)

Data da defesa: 23/02/2006

Técnicas de Otimização do Mergesort Externo num Ambiente de Banco de Dados

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Elton Gustavo Fanelli e aprovada pela Banca Examinadora.

Campinas, 23 de fevereiro de 2006.

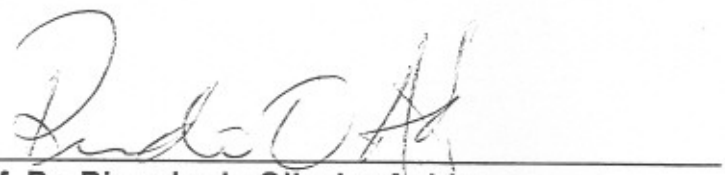
Prof. Dr. Rogério Drummond (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 23 de fevereiro de 2006, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. João Eduardo Ferreira
DCC / USP.



Prof. Dr. Ricardo de Oliveira Anido
IC / UNICAMP.

Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
IC / UNICAMP.

Aos meus pais,
que desde cedo me fizeram
acreditar que somente
o conhecimento, o trabalho
e a dedicação são capazes
de enobrecer o espírito humano.

"E debbasi considerare come non è cosa più difficile
a trattare, né più dubia a riuscire, né più pericolosa
a maneggiare, che farsi capo ad introdurre nuovi ordini."

*"E deve-se considerar não haver coisa mais difícil
de negociar, nem mais duvidosa de conseguir, nem mais perigosa
de manejar, do que tornar-se líder e introduzir uma nova ordem."*

Niccolo Machiavelli – O Príncipe, 1513.

Prefácio

A ordenação é uma tarefa frequente em bancos de dados. É utilizada em diversas ações realizadas pelo banco, como em consultas com saída ordenada, criação de índices, operações de consulta com agrupamento, remoção de dados duplicados, união, intersecção, diferença, junção relacional, e algumas outras. Quando os dados a serem ordenados não cabem no espaço de memória disponível, torna-se necessário recorrer a algoritmos de ordenação externa. Dentre estes, o mais largamente estudado e utilizado é o algoritmo de mergesort. Este algoritmo se divide em duas fases. Na primeira ocorre a criação dos blocos, que são conjuntos menores ordenados de dados. Na segunda fase ocorre a intercalação destes blocos. Quase em sua totalidade, o custo desta fase provém da realização de E/S. Ambas as fases podem ser otimizadas. A fase de intercalação pode ser paralelizada, o que torna o problema ainda mais atrativo. Muitos trabalhos de otimização para mergesort externo foram realizados prevendo-se modelos simplificados ou implementações em simuladores de bancos de dados. O foco deste trabalho é o estudo, implementação e análise de técnicas para otimização do mergesort externo dentro de um ambiente real de aplicação: o banco de dados MySQL.

Agradecimentos

Primeiramente, eu gostaria de agradecer ao Prof. Rogério Drummond. O Prof. Rogério não foi apenas um professor e orientador. O Prof. Rogério tornou-se um grande amigo. O clima de liberdade que proporcionou para que eu pudesse expressar minhas idéias, mesmo quando dissonantes das suas, foi um grande motivador da minha criatividade durante todo este trabalho. Nossas discussões científicas, políticas, econômicas, filosóficas e gastronômicas no Laboratório A-HAND jamais serão esquecidas. Com ele aprendi muito, não apenas sobre o assunto tratado neste trabalho, nem apenas sobre computação. O Prof. Rogério é um grande motivador da criatividade, valorizador das boas idéias e entusiasta do conhecimento. O período em que tive o prazer de conviver com ele diariamente foi um período de intenso enriquecimento cultural. Tenho certeza que a bagagem adquirida neste período será extremamente importante para trilhar um bom caminho agora e no futuro.

Gostaria de agradecer também a todas as pessoas do Laboratório A-HAND, em especial à Sônia, por ter sido praticamente uma segunda mãe durante todo este tempo. Além de prover uma excelente infra-estrutura para o trabalho no laboratório, a Sônia foi uma confidente dos problemas, frustrações e incertezas. Muitas de suas palavras me ajudaram a seguir em frente quando o caminho era árduo e complicado. Gostaria de agradecer também ao Marcos, Carlos, Paulo, Pedro, Fábio, Daniel, Montu e João pelas conversas, divagações, almoços e diversas outras atividades que realizamos em conjunto. O clima de descontração e alegria criado por esses amigos foi essencial para o bom prosseguimento deste trabalho.

Gostaria de agradecer a todas as pessoas da república em que resido. Nossa república transformou-se em uma família já há alguns anos, e tenho todos como grandes irmãos. Ao Higa, Wilson, Fred, Menardi, Ricardo, Edson, Gustavo, Zé e ao mais novo integrante Fabiens. Morar com todos vocês durante este período foi e é extremamente gratificante. O clima de alegria e irmandade que impera em nossa casa deixará muitas saudades.

Agradeço ainda aos grandes amigos que fiz e mantive durante toda a vida. A lista é extensa, e pronunciá-la aqui devoraria algumas páginas. Aos amigos de infância na rua, aos da escola de primeiro e segundo grau e aos amigos da universidade. Todos vocês sempre

estiveram prontos para ajudar direta ou indiretamente quando os problemas pessoais e profissionais apareceram. É reconfortante poder contar com vocês.

Gostaria de agradecer também a diversas pessoas do Instituto de Computação da Unicamp, em especial aos professores e ao pessoal da secretaria.

Para finalizar, e em especial, eu gostaria de agradecer aos meus pais e ao meu irmão. Aos três devo muito do que sou. Vocês foram a base de formação da minha personalidade, e em diversas ações e atitudes que tomei e tomo durante toda a vida, me espelho em vocês. À vocês dedico este trabalho.

Sumário

Prefácio **ix**

Agradecimentos **x**

1 Introdução **1**
1.1 Motivação 1
1.2 O problema e os objetivos 2
1.3 Estrutura da dissertação 3

2 Conceitos **5**
2.1 Ordenação de dados 5
2.2 Algoritmos de ordenação interna 6
2.3 Algoritmos de ordenação externa 7
2.3.1 O Mergesort 7
2.4 Classes de técnicas para melhoria do mergesort externo 10
2.4.1 Formação de blocos 11
2.4.2 Leitura antecipada de dados dos blocos 15
2.4.3 Padrões de intercalação 17
2.5 Termos e notação utilizada 19

3 Trabalhos Relacionados **23**
3.1 Double buffering 24
3.1.1 Intercalação de um nível com sobreposição perfeita 28
3.1.2 Intercalação de vários níveis com sobreposição perfeita 30
3.2 Forecasting 32
3.3 Forecasting estendido 35
3.4 Equal buffering 40

4 A ordenação no MySQL **45**
4.1 O banco de dados MySQL 45

4.2 O processo original de ordenação 46
4.3 Modificações introduzidas no processo de ordenação do MySQL 50
4.3.1 Cálculo do novo padrão de intercalação 51
4.3.2 O mergesort paralelo com buffers flutuantes 53
4.4 Testes comparativos entre as versões 55
4.4.1 Descrição dos testes 55
4.4.2 Resultados dos testes 56

5 Conclusões **66**
5.1 Contribuições 66
5.2 Trabalhos futuros 68

A Especificações do hardware e software utilizados **70**
A.1 Máquina 70
A.2 Discos 70
A.3 Sistema operacional 70
A.4 Banco de dados 70

B Especificações das tabelas e consultas realizadas **72**
B.1 Tabelas 72
B.2 Consultas 72
B.3 Dados obtidos 72

Bibliografia **79**

Lista de Tabelas

2.1 Exemplo de *replacement selection* de largura $w = 4$ 14
2.2 Notação utilizada na dissertação 22

A.1 Especificação da máquina 71
A.2 Especificação dos discos 71

B.1 Descrição das tabelas 74
B.2 Tempos: 10 MB / Com mem. 74
B.3 Tempos: 20 MB / Com mem. 75
B.4 Tempos: 30 MB / Com mem. 75
B.5 Tempos: 10 MB / Sem mem. 76
B.6 Tempos: 20 MB / Sem mem. 76
B.7 Tempos: 30 MB / Sem mem. 77
B.8 Tempos: 10 MB / Acesso a disco obrigatório 77
B.9 Tempos: 20 MB / Acesso a disco obrigatório 78
B.10 Tempos: 30 MB / Acesso a disco obrigatório 78

Lista de Figuras

2.1 Ordenação de dados através da criação de blocos (fatias ordenadas dos dados) e consecutiva intercalação 8
2.2 Esquema de execução de mergesort externo 9
2.3 Árvore de torneio: árvore de vencedores 12
2.4 Árvore de torneio: árvore de perdedores 12
2.5 Diferentes padrões de intercalação para seis blocos 18
2.6 Termos utilizados no trabalho 20
2.7 Notação utilizada 21

3.1 Mergesort utilizando double buffering 25
3.2 Um buffer por bloco. Um buffer vazio, os demais quase vazios. 26
3.3 Um buffer por bloco. Um buffer foi lido, mas rapidamente outro esvaziou. . 26
3.4 2 buffers por bloco, a intercalação já iniciou e a leitura continua 27
3.5 2 buffers por bloco, antes do preenchimento dos segundos buffers, 1 dos primeiros é esvaziado 28
3.6 Mergesort utilizando forecasting 32
3.7 Forecasting: comparação da última chave para realizar leitura antecipada . 33
3.8 Forecasting: fase inicial da execução 34
3.9 Forecasting: leitura de um novo buffer 34
3.10 Mergesort utilizando forecasting estendido 36
3.11 Fase inicial do forecasting estendido com algoritmo padrão do mergesort . . 37
3.12 Fase intermediária do forecasting estendido com algoritmo padrão do mergesort 38
3.13 Forecasting estendido com algoritmo de intercalação com leituras atrasadas 39
3.14 Mergesort utilizando equal buffering 41
3.15 Execução do equal buffering 41
3.16 Momento de disparo de leitura no equal buffering 42
3.17 Bloco acabou e seus buffers estão inutilizados 43

4.1 Arquitetura do SGBD MySQL 46

4.2 Estrutura do buffer na formação dos blocos 47
4.3 Fase de intercalação no MySQL original 49
4.4 Fase de intercalação utilizando forecasting estendido no novo MySQL . . . 53
4.5 Mergesort com 10 megabytes de memória para ordenação e memória livre para o SO 58
4.6 Mergesort com 20 megabytes de memória para ordenação e memória livre para o SO 59
4.7 Mergesort com 30 megabytes de memória para ordenação e memória livre para o SO 60
4.8 Mergesort com 10 megabytes de memória para ordenação e restante da memória ocupada 61
4.9 Mergesort com 20 megabytes de memória para ordenação e restante da memória ocupada 62
4.10 Mergesort com 30 megabytes de memória para ordenação e restante da memória ocupada 63
4.11 Mergesort com 10 megabytes de memória para ordenação e acesso a disco em cada gravação 63
4.12 Mergesort com 20 megabytes de memória para ordenação e acesso a disco em cada gravação 64
4.13 Mergesort com 30 megabytes de memória para ordenação e acesso a disco em cada gravação 65

Capítulo 1

Introdução

1.1 Motivação

Desde o ínicio do estudo dos problemas computacionais, a ordenação de dados mantém posição de destaque. Foi estudada durante o último meio século e continua sendo. O tipo de problema que apresenta é tão adequado ao ambiente computacional que há evidência de que o primeiro programa escrito para um computador que poderia armazenar programas foi uma rotina de ordenação [11].

A ordenação é uma tarefa frequente em bancos de dados. É utilizada em diversas ações realizadas no banco, como em consultas com saída ordenada, na criação de índices, em operações de consulta com agrupamento, remoção de dados duplicados, união, intersecção, diferença, junção relacional, e outras [7], [17].

Um SGBD[1] não executa as consultas como um bloco monolítico gigantesco. Para executar uma consulta, ele provê um número de rotinas especializadas chamadas operadores de consulta. A cada um destes operadores cabe a execução de uma tarefa de forma especialmente bem feita, sendo eficiente em tempo ou espaço de execução. Os operadores podem ser conectados, formando um plano de consulta, capaz de computar uma consulta específica. Um dos mais básicos e importantes operadores em um plano de consulta é o operador de ordenação [17].

A ordenação de dados é provavelmente um dos problemas mais exaustivamente estudados em ciência da computação. É vasta a literatura a respeito da ordenação de dados interna. Com relação à ordenação de dados externa ela é um pouco menor. A ordenação de dados externa é definida como a ordenação de uma quantidade de dados maior do que a memória é capaz de armazenar. O algoritmo de mergesort é o algoritmo mais utilizado para a realização da ordenação externa. Ele consiste de duas fases. Na primeira fase, o

[1] Sistema gerenciador de banco de dados

conjunto de dados a serem ordenados é divido em conjuntos menores ordenados, chamados blocos. Os blocos são gravados em um meio de armazenamento temporário, em geral nos discos. A segunda fase consiste em ler pedaços de blocos em memória, comparar suas chaves e prover um resultado final já ordenado. O tempo utilizado pela E/S na execução do algoritmo vem a ser a maior fração do tempo total de execução do mergesort externo.

Ambas as fases do mergesort podem ser paralelizadas, tornando o estudo do algoritmo ainda mais interessante. Com o paralelismo dos processos de leitura, comparação de chaves e gravação, estas tarefas podem ser sobrepostas e diminuírem o tempo total de execução da intercalação. O paralelismo requer a ponderação de diversas variáveis. Entre elas estão o tamanho dos buffers, o momento do disparo de leitura de um buffer, a quantidade de níveis de execução da intercalação, a largura da intercalação, etc. A maior parte dessas variáveis tem responsabilidade direta no tempo de execução de E/S.

Diversos trabalhos analizam formas de diminuir o tempo necessário para realizar ordenação externa com o mergesort [20], [11], [34], [33]. As técnicas introduzidas por estes trabalhos podem ser agrupadas em três tipos. O primeiro deles diz respeito à criação dos blocos. Algumas técnicas foram propostas para criar blocos maiores e em menos tempo. O segundo aspecto diz respeito à leitura e tamanho dos buffers. Técnicas de leitura antecipada de buffers permitem uma sobreposição de leitura, processamento e gravação dos dados. O terceiro tipo se concentra em elaborar padrões de intercalação que tenham custo mínimo, onde a soma do tempo gasto por seeks e transferência de dados seja o menor possível. Em todos estes trabalhos, as técnicas descritas foram estudadas através de modelos simplificados ou pela criação de aplicações que simulavam o funcionamento de um banco de dados.

Este trabalho tem por iniciativa implementar e analisar o funcionamento de algumas destas técnicas de otimização do mergesort externo num ambiente real de funcionamento: um banco de dados. O SGBD MySQL foi utilizado na implementação.

Neste trabalho foi elaborada e implementada uma heurística para tentar prever qual seria a largura ótima que otimizaria o tempo total de intercalação dos blocos. Foi também implementada no MySQL a técnica de forecasting estendido com algoritmo padrão de mergesort na fase de intercalação.

1.2 O problema e os objetivos

Num ambiente real, os recursos de memória são limitados. Para garantir a ordenação de um conjunto grande de dados em um ambiente com recursos limitados de memória é necessário recorrer a algoritmos de ordenação externa. O algoritmo mais estudado e utilizado para este propósito é o algoritmo de mergesort.

Quando um algoritmo de ordenação externa é necessário para ordenar os dados, o

tempo gasto em E/S para transferência dos dados entre a memória e o meio de armazenamento externo costuma dominar o tempo total de execução da ordenação. Portanto, a quantidade de memória disponível e a forma de uso desta memória para a ordenação podem ter um efeito grande no tempo total de ordenação. Por isso, o uso de técnicas avançadas de ordenação externa se torna vital para um banco de dados trabalhar com um grande volume de dados.

O objetivo deste trabalho é investigar o funcionamento dos algoritmos de ordenação externa – em especial o mergesort – numa aplicação real, um banco de dados. O primeiro passo é descobrir como são implementados na prática. Quais as diferenças encontradas na prática e na teoria descrita na literatura e, principalmente, se utilizam técnicas avançadas para realização da intercalação, explorando todo o potencial da CPU e E/S. Para explorar todo o potencial da CPU e E/S é preciso que sejam no mínimo implementados de forma paralela. O segundo passo é implementar uma das técnicas estudadas neste trabalho em um SGBD e analisar seu desempenho comparativamente à implementação original. Existem diversos bancos de dados com código fonte aberto que poderiam ser utilizados nesta implementação. Dentre eles, destacam-se o SAPDB, SQLite, Firebird, PostgreSQL e MySQL. O MySQL foi escolhido para realizar esta implementação de forma um tanto arbitrária. É provável que uma das principais motivações que levaram à escolha do MySQL para a implementação foi o fato de ele possuir características que o qualificam como uma boa escolha para um ambiente empresarial de grande porte. Algumas delas são a replicação, independência de sistema operacional, simplicidade nas tarefas administrativas e apoio à transações (com InnoDB e Berkeley DB) [26]. O MySQL se autodenomina o mais popular banco de dados de código aberto do mundo.

1.3 Estrutura da dissertação

O restante dessa dissertação é divida em quatro capítulos.

No capítulo 2 encontram-se os conceitos básicos relativos ao problema de ordenação. Nele é dado um rápido resumo sobre algoritmos de ordenação interna e externa. O mergesort externo é explicado em mais detalhes. Algumas diferenças principais entre um mergesort externo serial e um mergesort externo paralelo são explicitadas. Classes de técnicas de otimização para o mergesort externo também são mostradas neste capítulo, como a criação dos blocos, a leitura antecipada dos buffers e os padrões de intercalação.

No capítulo 3 são analisados alguns trabalhos correlatos. Nele são aprofundados os conceitos de leitura antecipada de buffers. As técnicas de leitura antecipada de buffers podem ser aplicadas apenas quando a intercalação é realizada por processos distintos de leitura, ordenação e gravação. Neste capítulo são estudadas as técnicas de double buffering, forecasting, forecasting estendido e equal buffering.

O capítulo 4 mostra como é realizada a ordenação de dados no banco de dados MySQL. O novo processo de ordenação utilizando o forecasting estendido e uma heurística para cálculo do padrão de intercalação também são explicados. Neste capítulo são comparados o mecanismo de ordenação original com o novo mecanismo de ordenação implementado no banco de dados pelo autor.

No capítulo 5 tem-se algumas contribuições à area, as conclusões relativas a este trabalho e alguns possíveis projetos futuros.

Capítulo 2

Conceitos

2.1 Ordenação de dados

A ordenação de dados é um problema clássico em computação, tanto teórica quanto prática. Possui uma diversa gama de aplicações e é peça fundamental para o funcionamento de diversos algoritmos e sistemas.

Este capítulo se compõe dos processos e conceitos básicos relativos à ordenação de dados, em especial à ordenação externa. Ordenação externa é o termo utilizado para designar a ordenação de dados que são maiores que a memória disponível. É chamada ordenação interna a ordenação dos dados que cabem na memória disponível. Apesar de ambas serem formas de ordenação, o estudo e as técnicas utilizadas em cada uma delas são bastante diferentes.

Na ordenação interna todos os dados estão em memória, que é considerado um dispositivo rápido. Mas ainda assim não é mais rápido que a CPU e seus registradores. Portanto as técnicas dos algoritmos de ordenação interna visam minimizar as operações mais custosas em relação ao processamento, que são as comparações e movimentações entre os dados, já que ambas exigem o acesso à memória, caso estes não estejam nos registradores ou na memória cache.

Na ordenação externa, como os dados não cabem em memória, é necessário utilizar como meio de armazenamento temporário um dispositivo de maior capacidade de armazenamento. No ínicio do estudo das técnicas de ordenação, estes dispositivos eram as fitas. Várias técnicas e algoritmos foram desenvolvidos para ordenar dados que se encontravam em fitas [11]. Atualmente, os dispositivos mais utilizados como meio de armazenamento temporário para estes dados são os discos. Quando um algoritmo de ordenação utiliza um disco como meio de armazenamento temporário, as técnicas utilizadas são um pouco distintas das técnicas utilizadas na ordenação interna, já que o disco é diversas ordens de grandeza mais lento que a memória.

No cálculo do custo da ordenação externa, o processamento dos dados – basicamente comparações – é irrisório quando comparado ao custo da movimentação dos dados entre a memória e o disco. O custo dos algoritmos de ordenação externa é baseado no tempo gasto em seeks[1] em disco e o tempo utilizado para transferência dos dados. As técnicas utilizadas nos algoritmos de ordenação externa visam minimizar estes custos.

2.2 Algoritmos de ordenação interna

A ordenação interna trata de dados que podem ser ordenados inteiramente em memória. Diversos algoritmos foram elaborados para realizar esta tarefa, como o quicksort, bublesort, distribution sort, radixsort, heapsort, mergesort, e outros [3], [11].

Apesar de na prática o quicksort ser o mais utilizado pela sua velocidade e necessidade de pouco espaço de memória, todos estes algoritmos tem sua importância. Diferentes aplicações podem demandar diferentes algoritmos de ordenação. O radixsort, por exemplo, é apropriado para chaves pequenas ou que possuem sequências lexicográficas não usuais.

No início do estudo dos algoritmos de ordenação interna, a maioria das técnicas utilizadas buscavam diminuir a complexidade e o tempo de ordenação. Com o passar do tempo, outras áreas foram exploradas para otimizar a ordenação, como a melhor utilização dos processadores, memória, memória cache e E/S.

Os algoritmos de ordenação interna são tipicamente realizados em três fases: entrada, ordenação e saída dos dados. O algoritmo AlphaSort utiliza o quicksort para ordenar os dados em buffers menores. Depois de ordenados, é realizada a intercalação dos buffers, e o resultado é direcionado para a saída. Este algoritmo é capaz de realizar sobreposição de entrada dos dados com ordenação em buffers menores pelo quicksort; e sobreposição da intercalação dos buffers com a saída dos dados [14]. Este é um exemplo de algoritmo que visa utilizar melhor utilização da CPU e da E/S.

Na fase de ordenação dos dados, o maior custo provém dos acessos à memória realizados pelas comparações e movimentações. Portanto torna-se importante o desenvolvimento de técnicas que visem o melhor aproveitamento da memória cache. O algoritmo quicksort tem bom aproveitamento da memória cache, já que acessa a memória de forma sequencial. Além disso, sua estratégia de "dividir para conquistar" faz com que o uso do cache seja potencializado, já que os dados são divididos em pedaços cada vez menores, com mais chances de caber inteiramente na memória cache.

O estudo dos algoritmos de ordenação interna não faz parte do escopo deste trabalho.

[1] O seek é o movimento do braço do disco para leitura ou gravação de outra trilha

2.3 Algoritmos de ordenação externa

A ordenação externa trata de dados que são maiores que a mémoria disponível. Diversos algoritmos foram desenvolvidos para resolver o problema, sendo grande parte deles derivados de algoritmos de ordenação interna.

Dentre os algoritmos de ordenação externa, o algoritmo de mergesort externo é o mais estudado em detalhes. O algoritmo de mergesort externo consiste de duas fases. Na primeira fase ocorre a divisão dos dados em conjuntos de dados menores. Em detrimento do tamanho do conjunto criado, estes dados podem ser ordenados em memória. Depois de ordenados, eles são gravados em um meio de armazenamento externo. Cada um desses conjuntos é chamado de bloco. Na segunda fase é realizada a intercalação destes blocos.

Outro algoritmo bastante utilizado na prática é o algoritmo de distribution sort externo. Este algoritmo tem vários outros nomes, sendo também conhecido como radixsort, columnsort, digitalsort, pocketsort, separationsort, etc.

Ele também é derivado de um algoritmo de ordenação interna. Assim como o mergesort, o distribution sort também consiste de duas fases. Durante a primeira fase ocorre a distribuição dos dados em um conjunto de buckets relativamente ordenados em intervalos disjuntos. Este processo é repetido nos buckets até que cada bucket seja pequeno o bastante para ser ordenado em memória. Na segunda etapa, os dados de cada bucket são ordenados internamente e o resultado final é formado através da combinação de todos os buckets. O maior problema relativo ao distribution sort é encontrar formas de distribuir os dados de maneira uniforme entre os buckets, reduzindo assim as distribuições subsequentes.

Há uma certa correspondência entre o mergesort e o distribution sort. Ambos são formados por duas fases. Confrontam-se blocos ordenados por um lado e buckets ordenados pelo outro, além da fase de intercalação em comparação à fase de distribuição. Apesar disso, os problemas a serem pesquisados em relação a ambos são bastante distintos. O maior problema do distribution sort é o fato de que é necessária uma enorme quantidade de memória para que seja possível concluir a distribuição em apenas um passo quando os dados a serem ordenados são muito grandes. Isto faz com que o distribution sort seja usualmente inferior ao mergesort [11]. Por esta razão, o distribution sort não será estudado neste trabalho.

2.3.1 O Mergesort

O algoritmo de mergesort é composto por duas fases. Na primeira fase são criados conjuntos de dados menores, já ordenados, chamados blocos. Os blocos são conjuntos de dados temporários e ficam armazenados em um meio de armazenamento externo. Na segunda fase é realizada a intercalação destes blocos.

A figura 2.1 mostra como é realizado um mergesort externo.

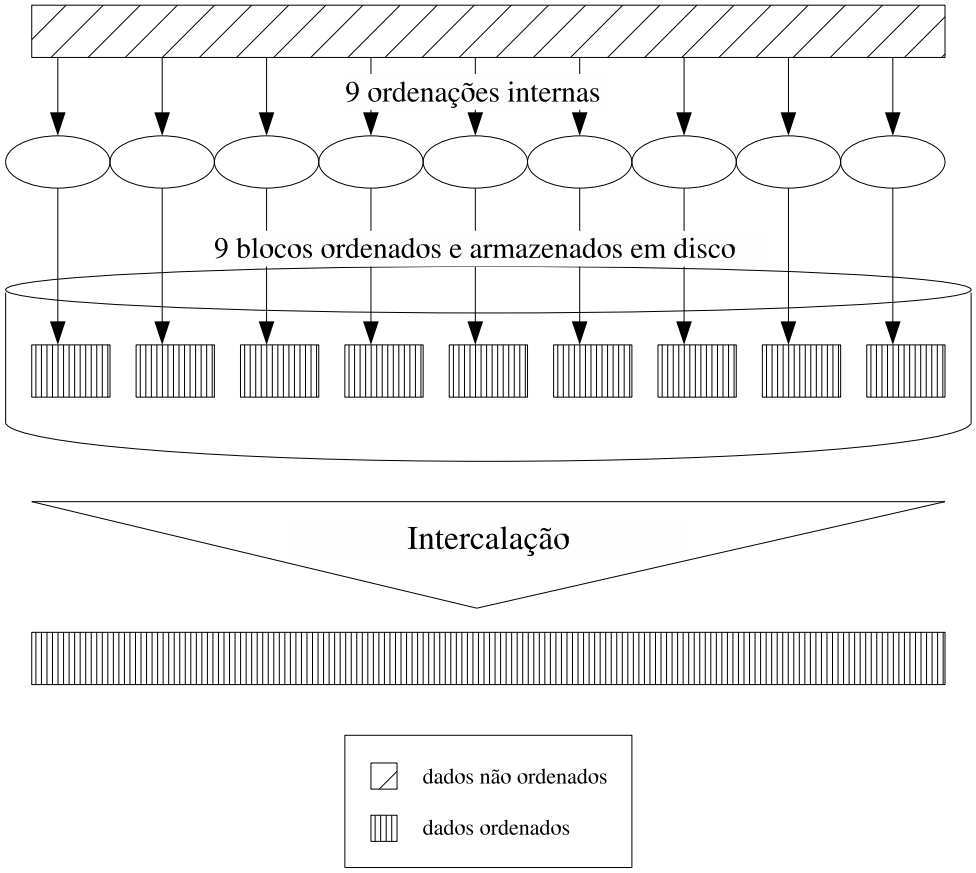


Figura 2.1: Ordenação de dados através da criação de blocos (fatias ordenadas dos dados) e consecutiva intercalação

Como o conjunto dos dados a serem ordenados é maior que a memória, ele não pode ser ordenado através dos algoritmos de ordenação interna usuais. Portanto, os dados são divididos em conjuntos menores que possam ser ordenados em memória. Cada um desses conjuntos (no exemplo da figura 2.1, temos 9 conjuntos) é ordenado em memória, um a um, e gravado em disco. A cada um desses conjuntos de dados menores já ordenados gravados em disco dá-se o nome de bloco. Para ordenar estes conjuntos de dados menores em memória, podem ser utilizados diversos algoritmos de ordenação interna. A escolha do algoritmo depende do conjunto de dados e das características das chaves a serem ordenadas. Os mais utilizados para realizar esta ordenação são o quicksort, o radixsort, o heapsort e algumas de suas variações. O quicksort e heapsort são interessantes por necessitarem de pouco espaço de memória para realizar a ordenação. Dependendo das características das chaves a serem ordenadas, o radixsort pode realizar a ordenação dos dados mais rapidamente que os demais. Algumas variações do heapsort, vistas mais

adiante na seção 2.4.1, podem criar blocos de tamanho até duas vezes maior que o tamanho da memória disponível, em média.

Quando todos os blocos terminam de ser produzidos, uma nova fase se inicia. Nesta fase, ocorre a intercalação das chaves de cada um dos blocos pré-ordenados. As chaves são comparadas e gravadas em disco. Para realizar a intercalação destas chaves, utilizam-se árvores de torneio. As árvores de torneio também são variações do heapsort e serão analisadas posteriormente na seção 2.4.1.

O tempo total para realização do mergesort compreende as duas fases: a formação de blocos e a intercalação dos blocos já ordenados. Durante a intercalação dos blocos já ordenados, pode ocorrer ainda a gravação dos dados ordenados, no caso do dispositivo de saída do processo de ordenação ser um meio de armazenamento externo. No caso de bancos de dados, é comum ter-se entradas e saídas referentes aos processos de ordenação tanto em meios de armazenamento externos, como discos, quanto geradas e consumidas por outros processos. Neste caso, a ordenação seria um processo intermediário entre outros operadores da consulta no banco.

Muitas etapas dentro de um processo de intercalação podem ser paralelizadas. Em especial, depois do término da formação dos blocos, pode-se paralelizar leitura de dados, processamento de chaves pela intercalação e gravação dos dados em disco. Esta otimização se torna ainda mais interessante quando os blocos são lidos de um disco e o resultado da intercalação (seja ele novos blocos ou o resultado final) é gravado em outro disco. Com o uso de dois discos é possível impedir a competição entre acessos a disco realizados para a leitura e os realizados para gravação de dados. Com isso, a gravação se torna sequencial.

Na figura 2.2, tem-se o esquema de execução de um mergesort externo.

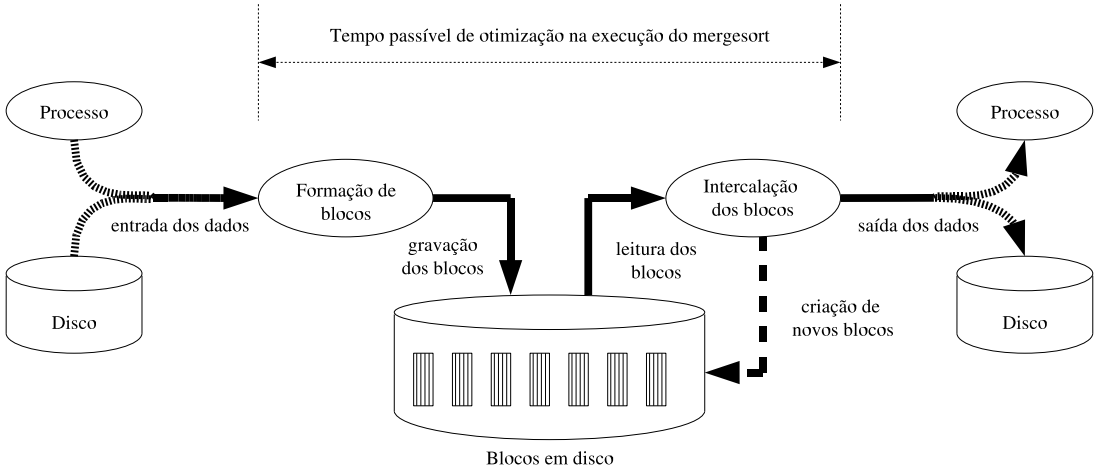


Figura 2.2: Esquema de execução de mergesort externo

Os dados a serem ordenados podem vir tanto de um processo quanto do disco.

A primeira fase, composta pela formação dos blocos, pode ser otimizada. Utilizando-se replacement selection, podem ser gerados blocos com até o dobro do tamanho da memória. Esta otimização na fase de geração dos blocos é uma melhoria para a próxima fase do mergesort. Como os blocos são maiores, tem-se menos blocos gravados em disco. Dessa forma é possível realizar uma intercalação com menos níveis. Realizando-se a intercalação com menos passos, economiza-se transferência de dados entre o disco e a memória. Além disso, quanto menos blocos forem necessários para o processo de intercalação, maiores poderão ser os buffers de leitura dos dados do disco. Com buffers maiores, é possível economizar seeks no disco.

Na fase de intercalação dos blocos, temos duas vertentes passíveis de otimização. A primeira delas estuda os padrões de intercalação. Sempre que não puder haver a intercalação dos blocos em apenas um passo, mais passos serão necessários. Isto pode ocorrer sempre que o tamanho dos dados a serem ordenados for muito grande em relação à memória. Neste caso, não é possível manter uma quantidade mínima de dados por bloco em memória. Torna-se necessário então realizar a intercalação de apenas alguns blocos em cada passo. Então serão gerados novos blocos. E no próximo nível, será realizada a intercalação destes novos blocos. Pode ser que ainda não seja possível finalizar o mergesort neste passo. Então novos blocos são criados novamente. Isto pode ocorrer diversas vezes. O estudo dos padrões de intercalação será analisado adiante, mas apenas superficialmente.

A segunda vertente que pode ser otimizada na fase da intercalação dos blocos diz respeito à leitura antecipada de buffers. Esta otimização só faz sentido quando os processos de leitura dos blocos, ordenação pela intercalação e gravação dos dados ou de novos blocos ordenados ocorrem em paralelo. A análise do momento em que uma nova leitura deve ser disparada e de qual bloco se devem ler os dados, pode economizar diversos seeks. Quando esses processos funcionam em paralelo, podem ser sobrepostas leitura, comparação e gravação dos dados. Neste caso, a transferência de dados se torna uma constante que não pode ser baixada e portanto a diminuição de seeks é o único caminho para conseguir diminuir o tempo de intercalação.

As técnicas de leitura antecipada de buffers serão analisadas neste e nos demais capítulos deste trabalho.

2.4 Classes de técnicas para melhoria do mergesort externo

Ao longo dos últimos anos, diversas técnicas foram criadas a fim de prover a melhoria no desempenho da ordenação externa pelo mergesort. Entre elas, algoritmos para criação de

blocos maiores, leitura antecipada de blocos para a intercalação, e a análise de padrões de intercalação. A leitura antecipada de blocos para a intercalação só faz sentido quando aplicada num contexto onde a leitura, processamento e gravação funcionam em paralelo.

2.4.1 Formação de blocos

A maneira mais simples de criar blocos é preencher toda a memória disponível com parte dos dados não ordenados, utilizar um algoritmo de ordenação interna para ordená-lo e gravar os dados, agora ordenados, em disco. Para ordenar os dados, utilizam-se de preferência algoritmos que exijam pouco uso de memória além da necessária para os dados, como o quicksort, o heapsort ou o radixsort. Partindo deste pressuposto, o tamanho do bloco formado será igual ao tamanho da memória disponível. Os blocos são formados um a um, todos com o mesmo tamanho. Exceto o último, que pode ser igual ou menor. Se as chaves forem pequenas o bastante para permitirem o uso do radixsort, a ordenação pode ser bem rápida. Caso o quicksort seja utilizado, também tem-se um bom desempenho, já que ele tem boa localidade, ou seja, utiliza bem a memória cache. De qualquer forma, a entrada dos dados, o processamento (comparação) das chaves e a gravação dos dados em novos blocos são realizadas em fases distintas, não sobrepostas. Portanto, não são completamente aproveitados os recursos de E/S e processamento que o sistema pode prover.

Existe uma forma de criar blocos com o dobro do tamanho da memória e sobrepor E/S e processamento das chaves. É através da utilização do algoritmo de replacement selection. Este algoritmo utiliza um heap como estrutura básica, organizado como uma árvore de torneio. Podem ser utilizados tanto a árvore de vencedores como a árvore de perdedores. Na árvore de vencedores, o nó pai guarda o vencedor da comparação entre seus dois nós filhos.

No caso de uma ordenação crescente, todo nó pai na árvore terá o menor de seus dois filhos. A figura 2.3 apresenta uma árvore de vencedores. As folhas foram preenchidas com as chaves a serem ordenadas e os nós internos com os vencedores da comparação entre seus dois filhos. Quando a menor chave da árvore (061) tiver de ser reposta por outra chave, apenas as chaves 512, 087 e 154 deverão ser analisadas. Estas chaves foram as chaves perdedoras quando disputaram o torneio com a chave 061. Portanto, nota-se que é mais interessante armazenar no nó pai a chave perdedora.

Dessa análise surgiu a idéia da árvore de perdedores. A figura 2.4 representa a mesma árvore da figura 2.3, porém armazenando em cada nó o perdedor da comparação ao invés do ganhador. Um nó extra é adicionado no topo da árvore. Este nó indica o vencedor do torneio.

De maneira diferente do que ocorre na árvore de vencedores, cada chave é representada

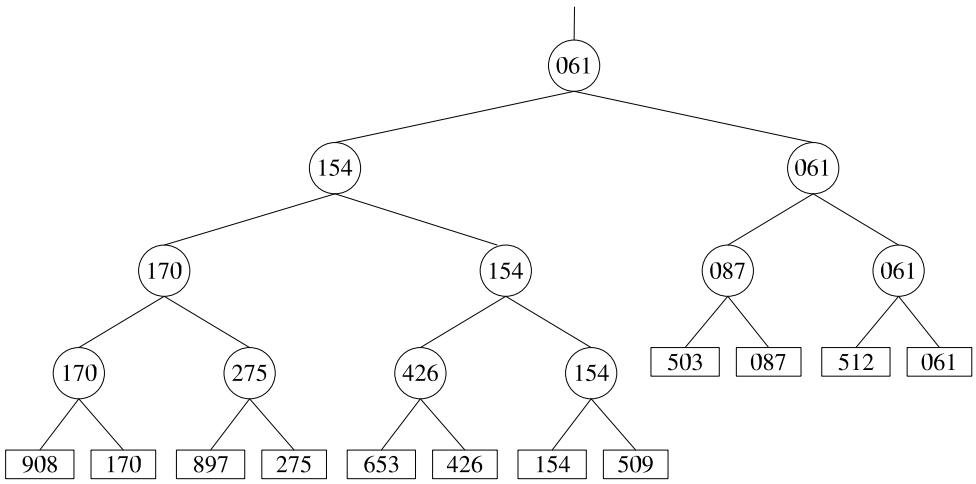


Figura 2.3: Árvore de torneio: árvore de vencedores

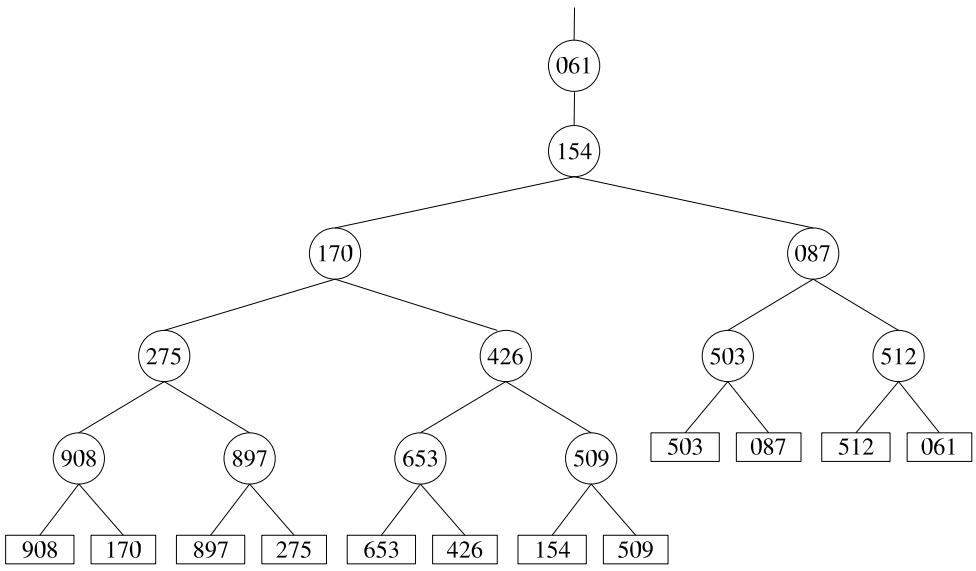


Figura 2.4: Árvore de torneio: árvore de perdedores

apenas uma vez nos nós internos da árvore. Na prática, as folhas representam as chaves, que podem ser registros grandes. Portanto, os nós internos da árvore são apontadores que apontam para estes registros. Como os apontadores ocupam pouco espaço em memória, movimentá-los na memória custa muito menos que movimentar registros grandes.

Sendo n o número de blocos e mantendo qualquer uma destas árvores de torneio em memória, reduz-se o número de comparações necessárias para a ordenação de uma chave de $(n-1)$ comparações para $\lceil \log_2 n \rceil$ comparações.

As árvores de torneio podem ser utilizadas nas duas fases do mergesort. Na primeira fase, em conjunto com replacement selection, utiliza-se para criar blocos maiores e sobrepor processamento, leitura e gravação. Na segunda fase, elas são o procedimento padrão para realizar a comparação das chaves e a intercalação dos blocos. Utilizando-as, são necessárias apenas $\lceil \log_2 n \rceil$ comparações para enviar uma chave para a saída.

Quando o replacement selection é utilizado na primeira fase do mergesort, os dados estão desordenados e não há nenhum bloco formado. É necessário então fazer uma intercalação de largura w dos dados com eles mesmos. Escolhe-se um w grande, de modo que a memória possa ser completamente preenchida. Quando qualquer registro é retirado do topo da árvore (o vencedor das comparações), ele é substituído pelo próximo registro do bloco. Se o novo registro for menor que a chave que ele substituiu, ele não pode ser incluído no bloco corrente. Se ele for maior, pode entrar na árvore de torneio pela forma usual e fazer parte do bloco em formação. Portanto cada bloco pode ter mais de w registros, mesmo que nunca se tenha mais do que w registros na memória em qualquer instante. A tabela 2.1 apresenta o processo de replacement selection para $w = 4$.

As chaves entre colchetes não podem ser incluídas no bloco corrente. Elas estão aguardando para ser incluídas no próximo bloco. No exemplo da tabela 2.1, o primeiro bloco formado tem 8 registros, ou seja, $2w$ registros. Quando este método foi inventado por Seward, ele acreditava que para dados aleatórios os blocos gerados teriam tamanho de $1,5w$ em média. Depois de uma análise mais profunda, descobriu-se que para dados completamente aleatórios o tamanho médio dos blocos gerados seria de $2w$ [11].

Em algoritmos de mergesort externo, a técnica de replacement selection é bastante utilizada para geração de blocos. Quando se utiliza um método simples para geração de blocos – como a leitura dos dados e ordenação através de um algoritmo de ordenação interna como o heapsort – os blocos formados tem o mesmo tamanho (w), com exceção do último bloco, que pode ser menor. Com a utilização de replacement selection, os blocos formados podem ter diferentes tamanhos. Para realizar a intercalação de blocos de tamanhos diferentes de forma eficiente, é necessário o estudo de padrões de árvores de intercalação.

As maiores vantagens da utilização de replacement selection na formação dos blocos em comparação ao método simples de ordenação, com leitura dos dados e uso de um

Tabela 2.1: Exemplo de *replacement selection* de largura $w = 4$

Conteúdo da árvore de torneio				Chave vencedora
503	087	512	061	061
503	087	512	908	087
503	170	512	908	170
503	897	512	908	503
[275]	897	512	908	512
[275]	897	653	908	653
[275]	897	[426]	908	897
[275]	[154]	[426]	908	908
[275]	[154]	[426]	[509]	bloco acabou
275	154	426	509	154
275	612	426	509	275
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

algoritmo interno de ordenação são duas:

1. Os blocos gerados são maiores.

   Em média, os blocos gerados têm o dobro do tamanho. Quanto maiores os blocos, menos blocos são gerados. Quanto menos blocos são gerados, menor é a largura de intercalação necessária na próxima fase do algoritmo. Com uma largura menor de intercalação, os buffers de leitura dos blocos na fase de intercalação podem ser maiores, economizando seeks. Além disso, em alguns casos, se a largura da intercalação superar a largura mínima para realização de apenas um nível de intercalação, haverá menos transferência de dados entre o disco e a memória. Isto ocorre porque cada nível a mais no algoritmo de mergesort aumenta a transferência total dos dados. Sendo D o tamanho dos dados, uma intercalação de nível 1, tem um fator de D dados transferidos entre o disco e a memória. Uma intercalação de nível 2 tem um fator de $2D$ dados transferidos. Uma intercalação de nível i, tem um fator de iD dados transferidos.

2. Há sobreposição de leitura, processamento e gravação.

   Com replacement selection é possível sobrepor leitura, processamento e gravação dos dados. Um processo pode se encarregar de ler os dados e colocá-los em memória. Outro processo pode se encarregar de fazer a comparação das chaves através das árvores de torneio. A um terceiro processo caberia a tarefa de gravar os dados em

disco. Esta sobreposição de tarefas e melhor aproveitamento dos recursos de CPU e E/S seria de difícil implementação se utilizadas técnicas simples de geração de blocos, como a ordenação com um quicksort em dados preenchidos na memória.

Apesar disso, nenhum banco de dados comercial utiliza em seu algoritmo de mergesort externo blocos maiores que o tamanho da memória. Quando um banco de dados possui registros de tamanho variável, torna-se difícil implementar estes algoritmos de maneira eficiente [17].

2.4.2 Leitura antecipada de dados dos blocos

Quando a fase de intercalação é realizada por um único processo, de forma serial, esta fase se divide em três subfases que se intercalam no decorrer do processo:

1. Leitura de dados dos blocos
2. Comparação dos dados para se descobrir a chave vencedora
3. Gravação da chave em disco

Neste caso, é necessário apenas um buffer de leitura alocado para cada bloco para que seja possível comparar as chaves e realizar a intercalação. Com o mergesort externo realizado desta forma, não há sobreposição de leitura de dados, processamento da chave e gravação. Tudo é feito de forma serial. Quando o processo envia uma requisição de leitura para o disco, o processador fica esperando os dados para poder prosseguir com a intercalação. E só depois do processamento da chave é possível realizar a gravação do dado processado. O tempo total da fase de intercalação de um mergesort externo simples ou serial de um nível que utiliza um disco para leitura (onde estão os blocos) e outro para escrita (onde os dados ordenados serão gravados) é dado por[2]

$$T = \frac{D}{B} \cdot s + p + 2 \cdot \frac{D}{t} \tag{2.1}$$

sendo T o tempo total, D o tamanho dos dados, B o tamanho do buffer, s o tempo de seek e latência médios, p o tempo gasto no processamento das chaves pela CPU e t a taxa de transferência do disco. (Estes símbolos serão utilizados no decorrer deste trabalho e se encontram na tabela 2.2).

O primeiro termo da fórmula 2.1 representa o número de seeks em disco. Como B é a quantidade mínima que pode ser transferida do disco para a memória e todo o volume

[2] O início da fase de gravação exige um seek adicional que foi desprezado para maior clareza. A fórmula completa seria $T = \frac{D}{B} \cdot s + p + 2 \cdot \frac{D}{t} + s$

de dados deverá ser lido pelo menos uma vez, temos que (D/B) é o número de seeks na leitura dos dados. Não estão sendo considerados o número de seeks da gravação porque esta utiliza um outro disco. Supõe-se que a gravação dos dados se dará de forma sequencial neste outro disco, ou seja, haverá apenas um seek. Como não há sobreposição de leitura, processamento e gravação, deve ser computado o custo gasto na ordenação das chaves (p). O tempo de transferência dos dados é representado no terceiro termo. Como não há sobreposição de tarefas, a transferência deve ser computada tanto na leitura quanto na gravação dos dados (2 vezes).

Se, ao invés de um único processo houver três processos (ou processos leves, dependendo da implementação), sendo um encarregado de ler os dados, outro encarregado de comparar as chaves e um terceiro tendo como tarefa gravar o dado ordenado de volta ao disco, podem ser sobrepostos os trabalhos de leitura, processamento e escrita. Neste caso, é impossível atingir uma sobreposição da leitura dos dados com o processamento das chaves se for alocado apenas 1 buffer para cada bloco. São necessários mais buffers. Porém, quanto maior o número de buffers alocados para cada bloco, mais vezes deverão ser transferidos os dados entre disco e memória. Sendo assim, quanto maior o número de buffers, mais seeks em disco. Supondo que haja sobreposição completa entre as tarefas de leitura, processamento e gravação dos dados, o tempo total da fase de intercalação de um mergesort externo de um nível que utiliza um disco para leitura (onde estão os blocos) e outro disco para escrita (onde os dados ordenados serão gravados) é dado por[3]

$$T = \frac{D}{B} \cdot s + \frac{D}{t} \tag{2.2}$$

O primeiro termo da fórmula 2.2 representa o número de seeks em disco e sua análise é semelhante a análise para a equação 2.1. Como ocorre a comparação das chaves enquanto os dados estão sendo lidos do disco, o processamento não é computado. O segundo termo da fórmula representa o tempo de transferência dos dados. Como ocorre a leitura dos dados ao mesmo tempo que ocorre a gravação dos dados já ordenados, é computada apenas uma vez o tempo de transferência dos dados.

Analisando-se as fórmulas 2.1 e 2.2, nota-se que a abordagem do mergesort com processos paralelos é mais interessante, por não apresentar custo de processamento e de transferência na gravação dos dados. Porém, como não é possível atingir uma boa sobreposição com um buffer por bloco, são necessários mais buffers. Como a memória é fixa, quanto maior o número de buffers, menor o tamanho de cada um. Analisando o primeiro termo da fórmula 2.2, nota-se que quanto menor o tamanho dos buffers, maior o número de seeks em disco.

[3] O início da fase de gravação exige um seek adicional que foi desprezado para maior clareza. A fórmula completa seria $T = \frac{D}{B} \cdot s + \frac{D}{t} + s$

O estudo das técnicas de leitura antecipada dos dados dos blocos em buffers visa diminuir o número de seeks necessários para transferência, obtendo a máxima sobreposição de tarefas possível. A diminuição de seeks torna-se vital para a diminuição do tempo total da intercalação. Num ambiente onde os processos de leitura, processamento e gravação dos dados executam em paralelo, o processamento das chaves (ordenação dos dados) não é levado em consideração no cálculo do tempo total, já que a intercalação dos dados é sobreposta pela leitura e gravação. Portanto, os custos a serem minimizados dizem respeito ao número de seeks e transferência dos dados. Quando é definida a quantidade de níveis de intercalação, pouco pode ser feito para diminuir a transferência dos dados. Ela é uma constante dependente apenas do tamanho do conjunto de dados a ser ordenado.

Diversas técnicas de leitura antecipada de buffers foram estudadas. Uma das mais utilizadas e difundidas na literatura é o *double buffering*. Nesta técnica dois buffers são alocados de forma fixa para cada bloco. Enquanto um deles está sendo utilizado na comparação de chaves, o outro está sendo utilizado para leitura dos dados. Dessa forma é possível atingir sobreposição completa de leitura e processamento [20].

O *forecasting* é uma técnica proposta por Knuth onde existe um buffer alocado para cada bloco, e mais um buffer adicional. Este buffer adicional é utilizado para leitura antecipada dos dados. Para saber de qual buffer alocado para determinado bloco os dados acabarão primeiro, basta descobrir a menor entre as últimas chaves de cada buffer já lido. O próximo pedaço do bloco à qual pertence este buffer deverá ser lido no buffer adicional [11].

Ambas as técnicas de forecasting e double buffering podem atingir sobreposição completa dos dados. Uma análise mais aprofundada mostrou que a sobreposição dos dados no forecasting pode ser melhorada se houver mais de um buffer adicional. Da derivação desta técnica surgiu o *forecasting estendido* [32].

Entre as formas propostas de reduzir o tempo total de seek, duas se destacam: o aumento do tamanho do buffer e a mudança na ordem de leitura. Com a mudança na ordem de leitura, é possível agrupar buffers de um mesmo bloco e disparar apenas uma leitura para preencher estes buffers.

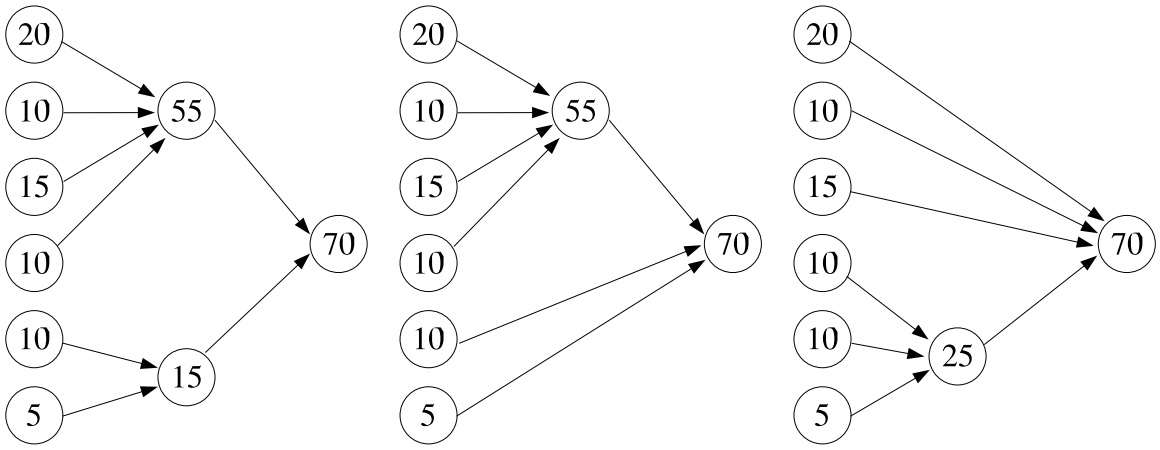
Estas técnicas de leitura antecipada de buffers visando a diminuição do tempo total de seeks será analisada de forma mais profunda no decorrer deste trabalho.

2.4.3 Padrões de intercalação

Dado um certo número de blocos, a intercalação pode ser realizada de várias formas. Pode ser feita em um, dois ou mais níveis. A largura da intercalação pode variar. Além disso, não é obrigatório que todos os blocos participem de todos os níveis, já que o principal objetivo da intercalação é diminuir o número de blocos.

Quando não é possível realizar a intercalação em apenas um nível, o tempo gasto na transferência dos dados aumenta. A cada novo nível, todos os dados deverão ser lidos e gravados em disco novamente. Quando os blocos tem tamanho variável, o estudo dos padrões de intercalação pode reduzir de forma significativa o tempo gasto na transferência de dados.

Os padrões de intercalação são normalmente representados por árvores. É construída uma árvore de forma que cada folha represente um bloco inicial. Cada nó interno da árvore representa um bloco que passou por um processo de intercalação, de forma que seus descendentes sejam os blocos dos quais ele foi fabricado. O valor contido dentro do nó representa o tamanho do bloco formado.



(a) Todos os blocos participam de todos os níveis

(b) Alguns blocos participam de todos os níveis

(c) Alguns blocos participam de todos os níveis

Figura 2.5: Diferentes padrões de intercalação para seis blocos

A figura 2.5 representa três padrões de intercalação diferentes para os mesmos blocos. Estes blocos tem tamanho variável. A máxima largura permitida da intercalação é 4. Cada um destes três padrões de intercalação resultam num custo de transferência de dados diferente.

Na figura 2.5(a), a intercalação é realizada de forma convencional. Os primeiros 4 blocos resultam num novo bloco de tamanho 55. A intercalação dos outros 2 blocos restantes resulta num bloco de tamanho 15. Uma nova intercalação é necessária, resultando em dados ordenados de tamanho 70. Analisando esta árvore, nota-se que foram transferidos $(55 + 15 = 70)$ registros no primeiro nível de intercalação, e mais 70 registros no segundo nível. Portanto, foram lidos e gravados 140 registros ao todo.

A figura 2.5(b) representa um padrão distinto de intercalação, onde os dois últimos

blocos participam apenas do último nível. Portanto, no primeiro nível de intercalação ocorre a participação dos 4 primeiros blocos apenas, sendo transferidos 55 registros. No segundo nível, é realizada a intercalação do bloco recém formado com o quinto e sexto blocos. Neste nível, são transferidos $(55 + 15 = 70)$ registros. Portanto, foram lidos e gravados 125 registros ao todo. Nota-se que este padrão de intercalação é melhor que o primeiro, pois realiza menos transferência de dados $(125 < 140)$.

A figura 2.5(c) representa o padrão de intercalação ótimo para estes blocos. Nele, são lidos e gravados 25 registros no primeiro nível de intercalação e 70 registros no segundo, sendo transferidos 95 registros ao todo.

Knuth descreveu uma solução bastante simples para calcular o padrão ótimo de intercalação, utilizando o método de Huffman [11]. Sendo r o número inicial de blocos e w a máxima largura de intercalação, adicionam-se $(1 - r)mod(w - 1)$ blocos fictícios de tamanho 0. Então, realiza-se uma intercalação dos w menores blocos existentes até que reste apenas um bloco. A solução de Knuth comprova que o padrão denotado na figura 2.5(c) é o padrão ótimo.

O estudo dos padrões de intercalação tenta minimizar apenas a quantidade de dados transferida. Isso não significa que o padrão ótimo de intercalação resulte em seu custo ótimo. Afinal, está sendo analisado apenas o custo de transferência, quando os seeks ainda representam um custo vital na realização da intercalação.

Dado uma quantidade fixa de memória, o número de buffers utilizado é inversamente proporcional ao tamanho do buffer. Quanto maior a largura da intercalação, mais buffers serão necessários, e menor será seus tamanhos. Com uma largura avantajada ocorre a minimização de transferência de dados, mas um aumento no número de seeks. Portanto, podem haver casos em que realizar dois níveis de intercalação pode diminuir o custo total da intercalação apesar de dobrar o custo de transferência. Isso depende basicamente da relação entre o tempo de transferência e o tempo de seek.

2.5 Termos e notação utilizada

Nesta seção serão definidos alguns termos e símbolos utilizados neste trabalho.

Utilizando-se um algoritmo de mergesort externo para ordenar os dados, o conjunto de dados inicial não ordenado é divido em conjuntos de dados menores. Estes dados podem ser ordenados em memória. Depois de ordenados eles são gravados em um meio de armazenamento externo. No escopo deste trabalho, este meio de armazenamento será sempre um disco. Cada um desses conjuntos ordenados gravados em disco é chamado de *bloco*.

Os dados só podem ser comparados em memória. A memória disponível é dividida em buffers. Neste contexto, um buffer é uma unidade mínima de transferência entre a

memória e o disco. Os *buffers de entrada ou leitura* são os buffers utilizados para a leitura dos dados dos blocos ou para a comparação das chaves pelo algoritmo de mergesort. Os *buffers de saída ou gravação* são os buffers que recebem as chaves que foram comparadas pelo mergesort e que são gravados em disco. Sua gravação pode formar o resultado final ou um novo bloco. Um *pedaço* é uma fração do bloco que corresponde exatamente ao tamanho de um buffer. Estes termos estão representados na figura 2.6.

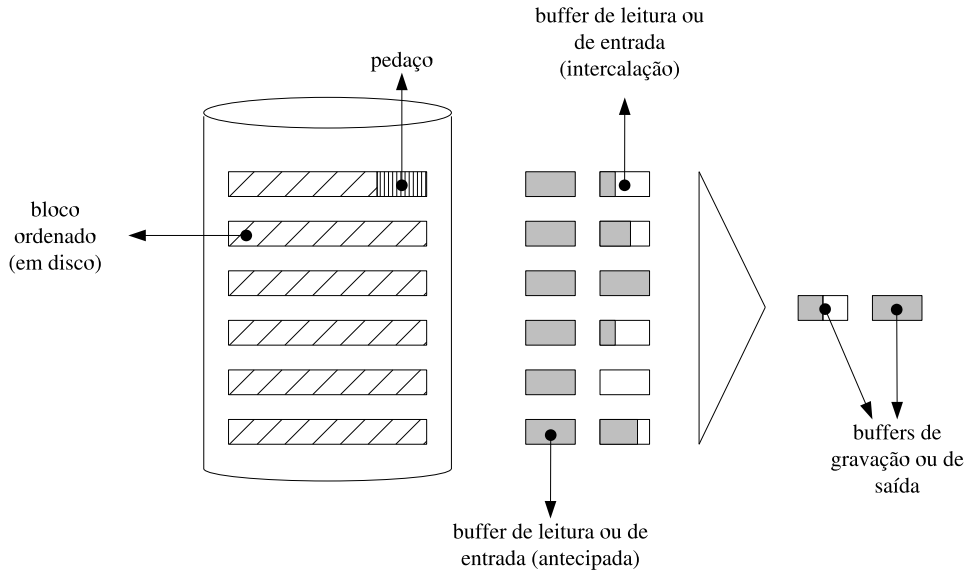


Figura 2.6: Termos utilizados no trabalho

Os buffers podem ser fixos ou flutuantes. Um buffer é dito fixo quando é alocado para apenas um bloco. Quando é esvaziado, ele será utilizado para a leitura do mesmo bloco ao qual foi alocado. Ele não pode ser alocado para outro bloco. Os buffers de saída também podem ser fixos, quando são utilizados apenas para armazenar chaves já ordenadas, não podendo se tornarem buffers de entrada. Os buffers são chamados flutuantes quando podem ser alocados para qualquer bloco. Quando um buffer flutuante é consumido, ele pode ser alocado para um bloco diferente do anterior ou ainda ser alocado como um buffer de saída, caso haja necessidade.

Um processo de intercalação pode se dar em um ou mais níveis. Isto pode ocorrer por necessidade, quando a memória (M) é pequena demais em relação aos dados a serem ordenados (D), ou para diminuir o tempo total de execução da intercalação. Dados n blocos, para uma intercalação de apenas um nível tem-se uma intercalação de largura $w = n$, ou seja, todos os blocos participam do único nível da intercalação. A largura de uma intercalação é dada pela quantidade de blocos que participam daquele passo da intercalação.

No exemplo da figura 2.7, tem-se que o número de blocos iniciais é $n = 8$. Então

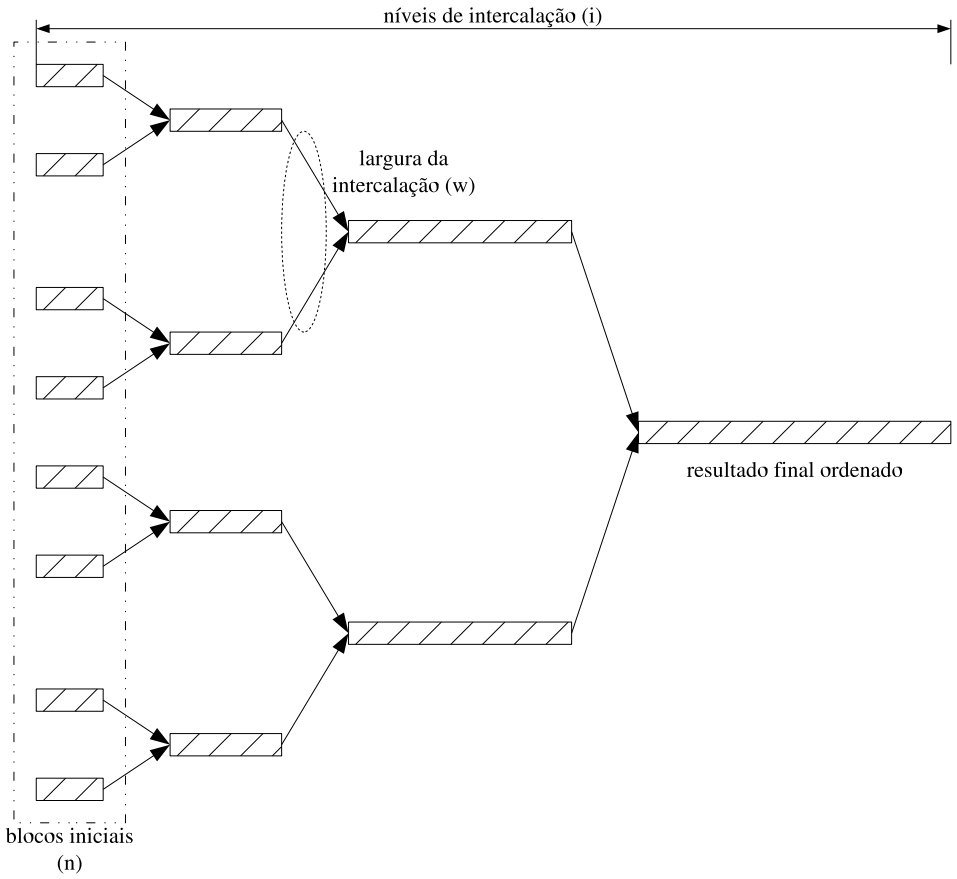


Figura 2.7: Notação utilizada

Tabela 2.2: Notação utilizada na dissertação

Símbolo	Definição	Exemplo
s	Tempo de seek + latência médios	12 ms
t	Taxa de transferência	60MB/s
B	Tamanho do buffer	2560 bytes
b	Número de buffers	202
M	Tamanho da memória disponível para o mergesort	524288 bytes
D	Tamanho dos dados a serem ordenados	104857600 bytes
n	Número de blocos ordenados	200
T	Tempo total para realizar a intercalação dos blocos	5800 ms
w	Largura da intercalação (blocos de entrada para a intercalação)	26
i	Níveis da intercalação	2

são realizadas quatro intercalações de dois blocos, ou seja, a largura dessa intercalação é $w = 2$. No próximo nível de intercalação, tem-se quatro blocos ($n = 4$). Estes blocos têm o dobro do tamanho dos blocos iniciais. São realizadas mais duas intercalações de largura $w = 2$, resultando em dois blocos. No nível final, é realizado mais uma intercalação de dois blocos. Desse passo surge o resultado final ordenado. Vários padrões de intercalação podem ser calculados e utilizados. Isto se torna ainda mais necessário quando os blocos têm tamanho variável. Quando os blocos têm tamanho fixo, é usual utilizar $i = \lceil log_w n \rceil$.

A notação da tabela 2.2 será utilizada no decorrer deste trabalho.

Capítulo 3

Trabalhos Relacionados

Nesta seção serão discutidas algumas técnicas para otimização da fase de intercalação do mergesort. Nas técnicas de mergesort apresentadas, o processo completo divide-se em 2 etapas:

1. Criação dos blocos ordenados
2. Intercalação dos blocos criados

A primeira etapa compõe-se de uma primeira leitura dos dados. Os dados estão desordenados. Como eles não cabem em memória, é necessário ordenar pequenas partes dos dados e regravá-las em disco para posterior realização da intercalação. Dá-se o nome de blocos a estas fatias ordenadas dos dados que foram regravadas em disco. Diversas técnicas para criação de blocos foram propostas. Algumas permitem criar blocos de tamanho maior que a memória disponível para a realização da intercalação [11].

Na segunda etapa são realizadas a leitura destes blocos, o processamento ou ordenação – a fase em que consiste a intercalação propriamente dita – e a gravação do resultado ordenado.

Esta etapa pode ser invocada diversas vezes, tomando blocos como entrada e criando novos blocos como saída. Isto pode ser desejável para diminuição do tempo total de realização da intercalação (ponderação entre número de seeks e quantidade de dados transferidos), ou por necessidade, pelo fato da relação entre o tamanho dos dados e o tamanho da memória ser muito grande. Quando isso ocorre, nem sempre é possível realizar a intercalação em apenas um nível.

Todas as técnicas apresentadas neste capítulo têm por objetivo otimizar a segunda etapa. Em todas elas, a leitura dos blocos, seu processamento ou ordenação e a gravação do resultado ordenado – seja ele um novo bloco ou o resultado final – são realizadas por processos paralelos.

3.1 Double buffering

Supondo que haja n blocos, o double buffering consiste em dividir a memória em $2n$ buffers de entrada, sendo 2 buffers alocados para cada bloco. Estes buffers são utilizados apenas pelo bloco ao qual foram alocados. São buffers fixos. Todos os buffers possuem o mesmo tamanho. É necessário ainda que sejam alocados ao menos 2 buffers de saída.

Três processos são disparados. Um processo tem como tarefa ler os dados que estão nos blocos e armazená-los em buffers. Este processo armazenará os dados de determinado bloco em um dos buffers relativos à este bloco. Ao segundo processo compete realizar a intercalação dos dados já lidos. Este processo compara uma chave de cada buffer de cada bloco. Ele retira a chave escolhida do buffer a qual pertencia e a coloca num buffer adicional, utilizado apenas para gravacao. O terceiro processo é responsável por gravar este buffer adicional no disco.

No processo de intercalação, cada bloco utiliza um buffer para processamento da chave pela intercalação. O outro buffer é utilizado para leitura antecipada (read-ahead) pelo processo de leitura.

O processo de intercalação inicia após a leitura de um buffer de cada bloco. Então, de forma concomitante com a intercalação, o segundo pedaço de cada bloco é lido e colocado no segundo buffer. Sempre que um buffer de um determinado bloco é esvaziado pelo processo de intercalação, o próximo pedaço deste bloco é lido do disco para este buffer.

O fato de cada bloco possuir um buffer sobressalente para leituras antecipadas permite que sejam intercalados o processamento e a escrita (mais rápidos) com a leitura. Deste modo, tenta-se evitar que o processador necessite esperar a leitura de um novo buffer de um determinado bloco quando o buffer de entrada (intercalação) já tenha esvaziado. Como houve uma leitura antecipada, o buffer a ser utilizado já foi lido. Este comportamento é explicitado pelos buffers utilizados no penúltimo bloco da figura 3.1.

O double buffering foi uma das primeiras técnicas de buffering visando aumentar a sobreposição da leitura e gravação para realização do mergesort [20].

Quando o tempo total de realização da intercalação iguala exatamente o tempo de leitura dos blocos, temos *sobreposição perfeita* [20].

Em 1973, Knuth escreveu [11] *"Se para uma intercalação de largura w forem utilizados $2w$ buffers de entrada ao invés de w, é possível realizar uma boa sobreposição com a leitura, mas estaremos dobrando o tempo de seek, portanto podemos não ganhar nada"*. Nesta época, as memórias eram pequenas. Baseado no trabalho de Knuth, a maioria das técnicas de intercalação [1], [2], [28] utilizava apenas um buffer por bloco até meados de 1990. Nesta época, as memórias já eram maiores e tornavam a constatação de Knuth obsoleta.

No início da fase de intercalação é necessária a leitura de pelo menos um buffer de

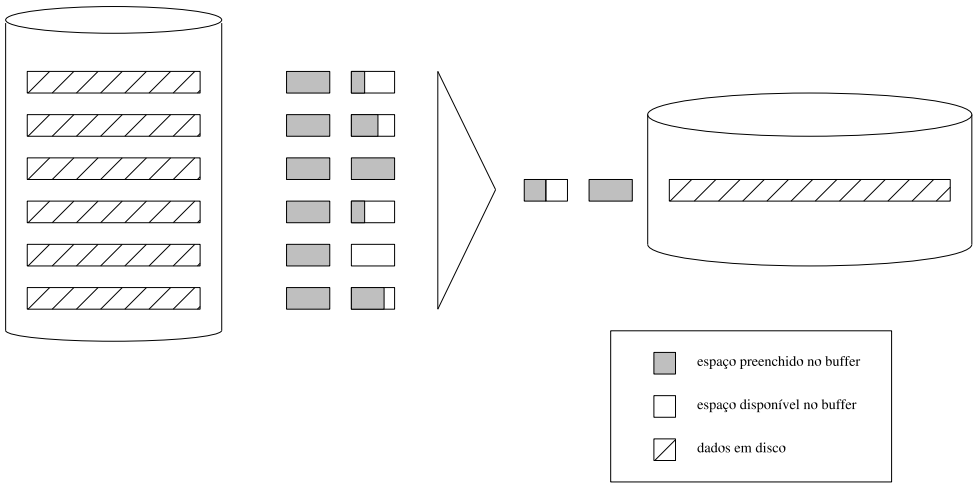


Figura 3.1: Mergesort utilizando double buffering

cada bloco para que as chaves ou dados possam começar a ser comparados. Durante este período, não há sobreposição de leitura com processamento ou escrita. Supondo que os dados estejam distribuídos de maneira uniforme pelos blocos – de forma que as chaves finais de cada buffer lido sejam próximas – e que tenhamos apenas um buffer, teremos em determinado momento um cenário próximo ao da figura 3.2.

Quando o primeiro buffer é esvaziado, é necessário parar a comparação das chaves e aguardar que o processo de leitura dos blocos leia o próximo buffer. Quando a comparação de chaves para, o processo de gravação continua apenas se tiver um buffer cheio (já ordenado pela intercalação) para gravar. Portanto, se há apenas um buffer por bloco, quando ocorre leitura, muito pouco ou nada é gravado no disco.

Logo após a leitura do próximo buffer, o processo de intercalação continua (conforme expresso na figura 3.3). Então, um novo buffer se esvazia, uma nova leitura deve ser disparada e a comparação de chaves é interrompida novamente. Este processo ocorrerá diversas vezes em um ciclo.

O processo de leitura trabalha até a memória ficar completamente ocupada. Depois fica um longo tempo aguardando o próximo buffer esvaziar. Já o processo de escrita começa a trabalhar quando a memória está completamente ocupada. Depois que o primeiro buffer se esvazia, caso não haja nenhum buffer cheio já ordenado pela intercalação, ele fica um longo tempo aguardando que a intercalação lhe forneça um novo buffer para gravação. Durante os períodos de leitura, a comparação de chaves fica bloqueada e a gravação dos buffers funciona pouco ou nada. Temos então que na maior parte do tempo, a leitura e a gravação dos buffers não é sobreposta.

Para sobrepor leitura e gravação, é necessário que o buffer de saída seja preenchido mesmo quando estão havendo leituras. Portanto, é necessário que a comparação das

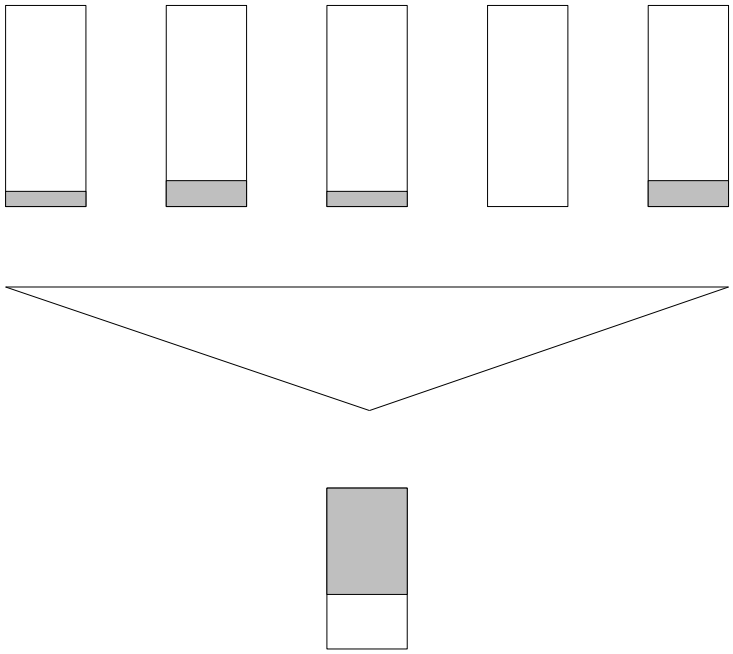


Figura 3.2: Um buffer por bloco. Um buffer vazio, os demais quase vazios.

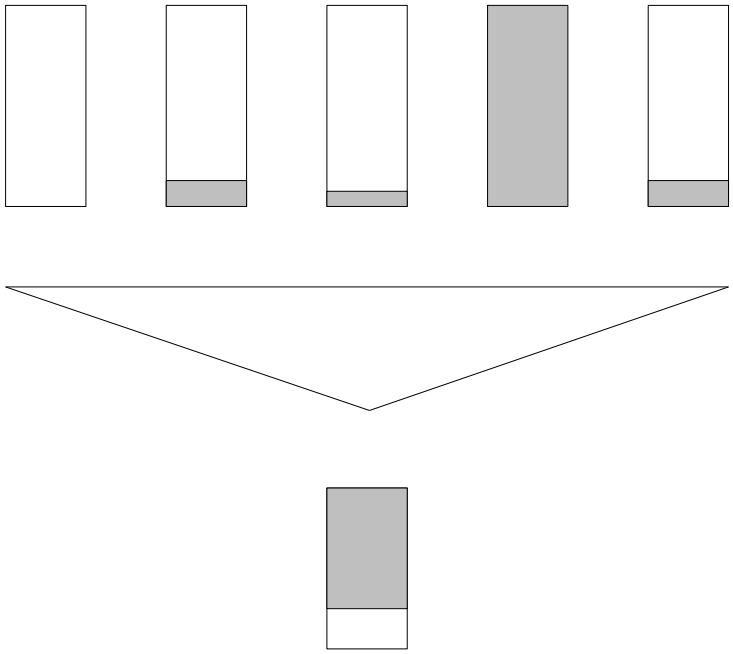


Figura 3.3: Um buffer por bloco. Um buffer foi lido, mas rapidamente outro esvaziou.

chaves não seja interrompida quando um buffer seja esvaziado. Para que isto seja possível, é necessário alocar mais de um buffer por bloco.

Da mesma forma que com apenas um buffer por bloco, quando se alocam 2 buffers por bloco é necessário aguardar a leitura de pelo menos um buffer de cada bloco para iniciar a intercalação. Depois de iniciada a intercalação, o processo de leitura dos blocos pode continuar lendo nos buffers sobressalentes, conforme figura 3.4.

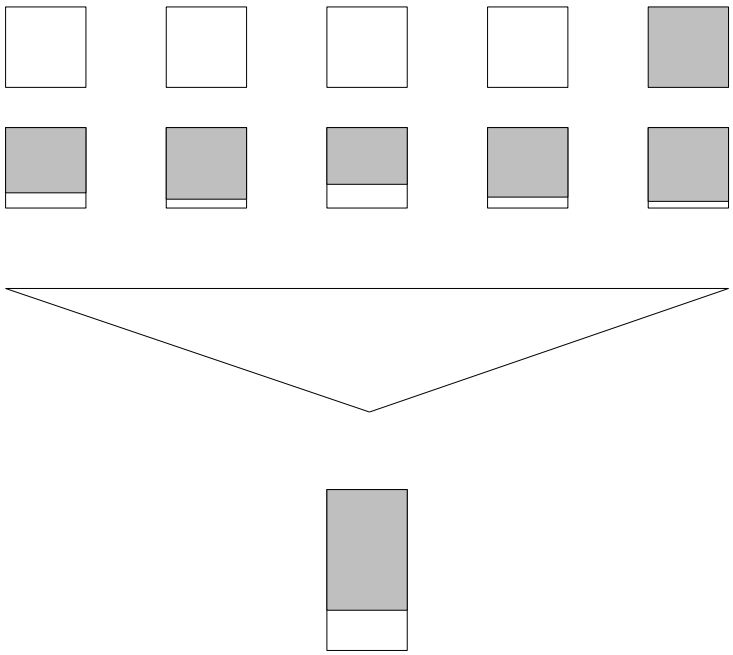


Figura 3.4: 2 buffers por bloco, a intercalação já iniciou e a leitura continua

Quando os buffers lidos estão quase sendo esvaziados, temos um comportamento similar ao da figura 3.5.

Como o processo de leitura lê blocos de um disco e o processo de gravação escreve os buffers em outro, temos que a leitura é mais lenta que a gravação. Isso ocorre porque a leitura dos blocos dá-se de forma não sequencial enquanto a gravação dos buffers de saída dá-se de forma sequencial, portanto sem seeks. (No modelo apresentado os discos são dedicados para a ordenação).

No modelo com dois buffers, a gravação também é interrompida no momento em que não há mais buffers preenchidos de um bloco não acabado. Então o processo de leitura lê os segundos buffers remanescentes. Quando todos os buffers remanescentes forem preenchidos, volta-se a um estado parecido ao estado inicial. Este é um processo cíclico. Nota-se que desde o início até que os blocos acabem, o processo de leitura está lendo pedaços dos blocos e preenchendo buffers. Algumas vezes o processo de gravação fica bloqueado. Quando se apresenta esta conjuntura, temos a sobreposição perfeita [20].

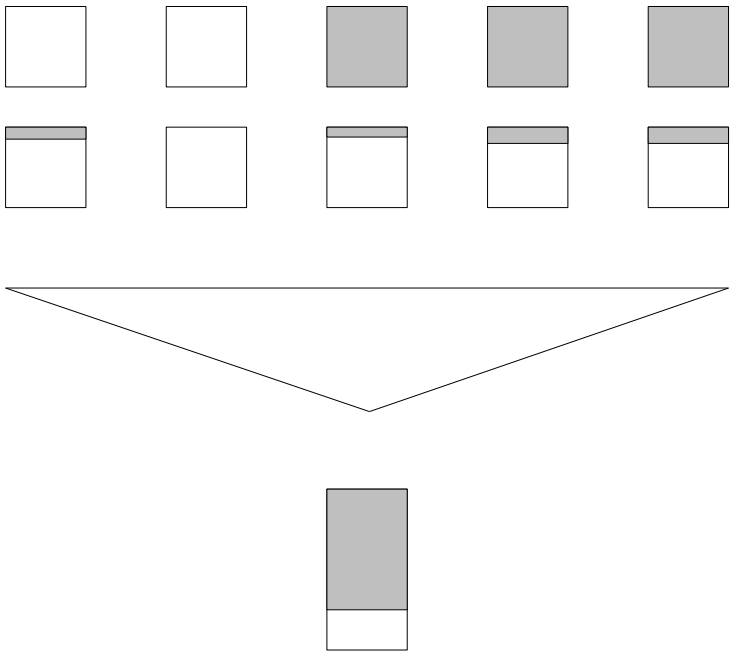


Figura 3.5: 2 buffers por bloco, antes do preenchimento dos segundos buffers, 1 dos primeiros é esvaziado

Com sobreposição perfeita, o tempo total da fase de intercalação é o mesmo tempo de leitura dos blocos (desprezando-se o tempo de gravação dos últimos buffers depois que a leitura acabou).

O custo para atingir a sobreposição perfeita é dobrar o número de seeks em disco, já que com double buffering o tamanho do buffer foi diminuído pela metade. Torna-se necessária uma análise para avaliar quando é mais eficiente utilizar o double buffering com sobreposição perfeita do que realizar a intercalação com 1 buffer por bloco sem sobreposição. Esta análise é divida em duas partes, a seguir.

3.1.1 Intercalação de um nível com sobreposição perfeita

Nesta sessão é assumido que haja sobreposição perfeita e que seja realizada uma única intercalação de um nível [20].

Sendo n o número de blocos, tem-se $2n$ buffers de entrada na memória. Foi utilizado replacement selection (descrito na seção 2.4.1) na criação dos blocos. Portanto o tamanho dos blocos é em média o dobro do tamanho da memória. Logo, cada bloco comporta $4n$ unidades de tamanho B. Chamaremos estas unidades de pedaços. Portanto, para cada um dos n blocos é necessária a leitura de $4n$ pedaços. Logo, lemos ao todo $4n^2$ pedaços. Sendo assim, o tempo total da fase de intercalação é:

$$T = 4n^2 \cdot \left(s + \frac{B}{t}\right) \tag{3.1}$$

De forma equivalente

$$T = 4n^2 \cdot s + 4n^2 \cdot \frac{B}{t} \tag{3.2}$$

O primeiro termo de 3.2 representa o custo total de seeks na intercalação. O segundo termo representa o tempo de leitura sequencial para o volume completo dos dados, D. Sempre que o primeiro termo é menor que o segundo termo, o tempo de intercalação é menor do que o tempo de duas leituras sequenciais e este método é melhor do que qualquer método em que não haja sobreposição de escrita e leitura.

Portanto, quando

$$s < \frac{B}{t} \tag{3.3}$$

ou seja, o tempo de acesso (seeks) é menor que o tempo de transferência de dados de um buffer de entrada, a intercalação de um nível com sobreposição perfeita é melhor do que qualquer outro método onde não haja sobreposição de leitura e escrita. Isto ocorre porque neste caso o tempo de intercalação é menor que o tempo gasto para ler e escrever os dados em disco.

Além disso, quando a equação 3.3 é válida, não há motivos para se considerar a realização de uma intercalação com mais de um nível.

Como cada bloco tem em média o dobro do tamanho da memória, temos que o número de blocos em média é

$$n = \frac{D}{2M} \tag{3.4}$$

Como temos $2n$ buffers de entrada em memória, usando a estimativa em 3.4, tem-se que o número de buffers de entrada é

$$2n = \frac{D}{M} \tag{3.5}$$

O tamanho de cada buffer é dado por

$$B = \frac{M}{D/M} = \frac{M^2}{D} \tag{3.6}$$

Ao substituir-se 3.6 em 3.3, tem-se

$$s < \left(\frac{M^2/D}{t}\right) \tag{3.7}$$

De forma equivalente,

$$D < \frac{M^2}{s \cdot t} \tag{3.8}$$

Logo, quando 3.8 é válido, utilizar a técnica da intercalação de um nível com dois buffers de entrada e sobreposição perfeita é mais rápido que qualquer padrão de intercalação sem sobreposição e é mais rápido que utilizar uma intercalação de dois níveis com sobreposição perfeita.

3.1.2 Intercalação de vários níveis com sobreposição perfeita

Quando 3.8 não é válido, torna-se necessário considerar a realização de dois ou mais níveis de intercalação com sobreposição perfeita.

Sendo w a largura da intercalação e n o número de blocos, a quantidade de níveis de intercalação (i) realizados é dada por $\log_w n$. A largura w deve ser considerada entre n, $\lceil\sqrt{n}\rceil$, $\lceil\sqrt[3]{n}\rceil$, $\lceil\sqrt[4]{n}\rceil$, e assim por diante.

Assume-se que há sobreposição perfeita para leitura, escrita e comparação. Sempre que i é maior que 1, os dados deverão ser lidos mais de uma vez. Ao realizar uma intercalação de largura w utilizando-se n blocos, sendo $w = \lceil n^{1/i}\rceil$ e i um inteiro, o tempo total da intercalação será dado por

$$T = i \cdot 4w \cdot n \cdot \left(s + \frac{B}{t}\right) \tag{3.9}$$

Tem-se então $2w$ buffers de entrada na memória. O tamanho do buffer é $B = \frac{M}{2w}$. A quantidade de níveis da intercalação é dada por $i = \log_w n = \frac{\ln n}{\ln w}$. Substituindo em 3.9 tem-se

$$T = \left(\frac{\ln n}{\ln w}\right) \cdot 4w \cdot n \cdot \left(s + \left(\frac{M/2w}{t}\right)\right) \tag{3.10}$$

simplificando 3.10, tem-se

$$T = 2n \cdot \ln n \cdot \left(\frac{2w \cdot s + \frac{M}{t}}{\ln w}\right) \tag{3.11}$$

Os únicos valores que devem ser considerados para w são os que satisfazem $w = \lceil n^{1/i}\rceil$, sendo i um inteiro positivo.

Ao comparar-se uma intercalação de dois níveis com sobreposição perfeita (com $i = 2$) com uma intercalação de um nível sem nenhuma sobreposição, nota-se que o tempo total para cada método é composto pelo tempo de duas leituras sequenciais dos dados, além do seek. Na intercalação de um nível sem sobreposição, temos que considerar o tempo de

transferência da gravação, que é o mesmo tempo de transferência dos dados na leitura. Por isso em ambos temos que o tempo de transferência é o tempo de duas leituras. Como o tempo de transferência de ambas as técnicas é o mesmo, para compará-las basta comparar-se o tempo de seek. Numa intercalação de um nível (onde $w = n$) com apenas um buffer, o tempo de seek é dado por

$$TS1 = 2 \cdot n^2 \cdot s \tag{3.12}$$

assumindo-se que haja n buffers de entrada e que cada bloco tenha o dobro do tamanho da memória.

Na intercalação de dois níveis ($i = 2$) com sobreposição total de leitura e escrita, haverá 2 buffers por bloco. Como $i = 2$, tem-se que $w = \lceil \sqrt{n} \rceil$. Como os blocos têm o dobro do tamanho da memória e tem-se $2w$ buffers de entrada em memória, cada bloco comporta $4w$ buffers. Como são dois níveis de intercalação, serão necessárias duas leituras. Sendo assim, o tempo de seek na intercalação de dois níveis com sobreposição completa é dado por

$$TS2 = 2 \cdot 4w \cdot n \cdot s \tag{3.13}$$

Analisando 3.13 e 3.12, tem-se que a intercalação de um nível com um buffer por bloco (sem sobreposição) será melhor que a intercalação de dois níveis com dois buffers por bloco (com sobreposição completa), quando

$$2 \cdot n^2 \cdot s < 2 \cdot 4w \cdot n \cdot s \tag{3.14}$$

de forma análoga,

$$n < 4w \leq 4\sqrt{n} \tag{3.15}$$

Assim,

$$n < 16 \tag{3.16}$$

Como é utilizado $n = D/2M$, substituindo-se em 3.16, tem-se

$$D \geq 32M \tag{3.17}$$

Logo, quando tem-se $D \geq 32M$, realizar uma intercalação de dois níveis com dois buffers por bloco e sobreposição completa é mais rápido que realizar uma intercalação de um nível sem nenhuma sobreposição de leitura e escrita.

Analisando 3.8 e 3.17, tem-se

$$D < 32M \quad \text{e} \quad D \geq \frac{M^2}{s \cdot t} \tag{3.18}$$

A equação 3.18 implica as condições para que uma intercalação de um nível sem sobreposição possa ser mais rápida que uma intercalação de um ou dois níveis com sobreposição perfeita.

A técnica de double buffering foi uma das primeiras onde se provou a viabilidade de diminuir o tempo de intercalação mesmo que seja necessário aumentar o número de seeks. A abordagem elevou a sobreposição dos processos de leitura e escrita à uma condição tão vital a ser analisada quanto o número de seeks realizado no processo. A visão que priorizava aumentar ao máximo o tamanho do buffer para assim diminuir o número de seeks e por conseguinte o tempo total da intercalação passou a ser melhor analisada depois da análise e criação desta técnica.

3.2 Forecasting

Supondo que haja n blocos, a memória disponível para a intercalação é dividida em $n + 1$ buffers de entrada [11]. São alocados também ao menos 2 buffers de saída. De maneira diferente do que foi apresentado na seção 3.1, estes buffers não são fixos para cada bloco. Eles são alocados para um determinado bloco de acordo com a necessidade. São chamados de buffers flutuantes. A figura 3.6 mostra um mergesort utilizando forecasting em andamento.

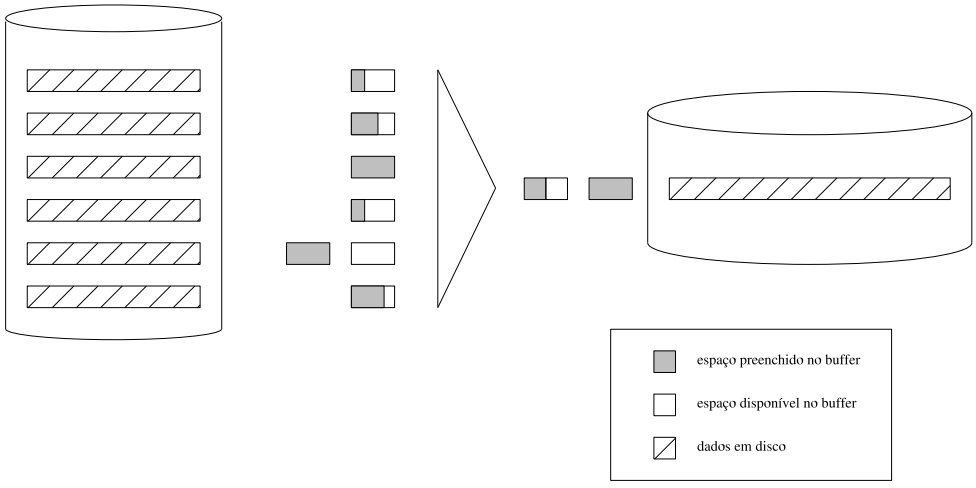


Figura 3.6: Mergesort utilizando forecasting

O buffer sobressalente é utilizado para leitura antecipada. A técnica do forecasting consiste em comparar a última chave de cada buffer (em memória) de cada bloco. Dessa

forma é possível predizer que o buffer que apresenta a menor última chave será o primeiro a ser esvaziado. Então o próximo pedaço deste bloco será lido no buffer extra. Na figura 3.7, a última menor chave presente nos buffers é a chave 21, que se encontra num buffer alocado para leitura do terceiro bloco.

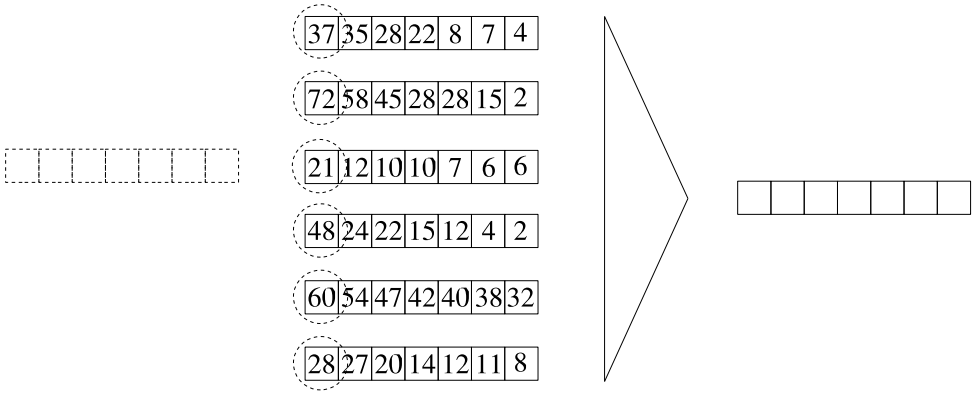


Figura 3.7: Forecasting: comparação da última chave para realizar leitura antecipada

Como 21 é menor que as demais chaves, sabe-se de antemão que o buffer deste bloco acabará mais cedo que os demais, e portanto pode-se disparar antecipadamente a leitura deste bloco e alocar o buffer disponível que foi utilizado na leitura no local apropriado. Esta comparação pode ser realizada no momento da criação dos blocos, criando-se assim uma sequência de consumo. Na técnica de forecasting, ela é realizada pelo processo de leitura no momento em que a intercalação estiver sendo executada. Neste caso, a comparação das chaves pode ser realizadas de formas variadas, sendo a mais comum uma árvore de seleção de perdedores (seção 2.4.1).

No decorrer da execução, as chaves vão sendo comparadas e consumidas pelo processo de intercalação. A figura 3.8 mostra os buffers de entrada, o buffer disponível para leitura antecipada e um buffer de saída logo após o preenchimento de 5 chaves no buffer de saída pelo processo de comparação.

Como o processo de comparação está lidando apenas com dados em memória, sua execução ocorre de forma mais rápida que a execução dos processos de leitura e gravação, que lidam com dados no disco. Dessa forma, o processo de comparação vai consumindo as chaves que estão no buffer de entrada até que se chega ao cenário apresentado na figura 3.9.

Neste instante temos duas situações possíveis:

1. O buffer sobressalente já foi lido.

   Neste caso, o processo de comparação pode continuar sua execução. Como há no mínimo dois buffers alocados para a gravação (buffers de saída), o buffer cheio em

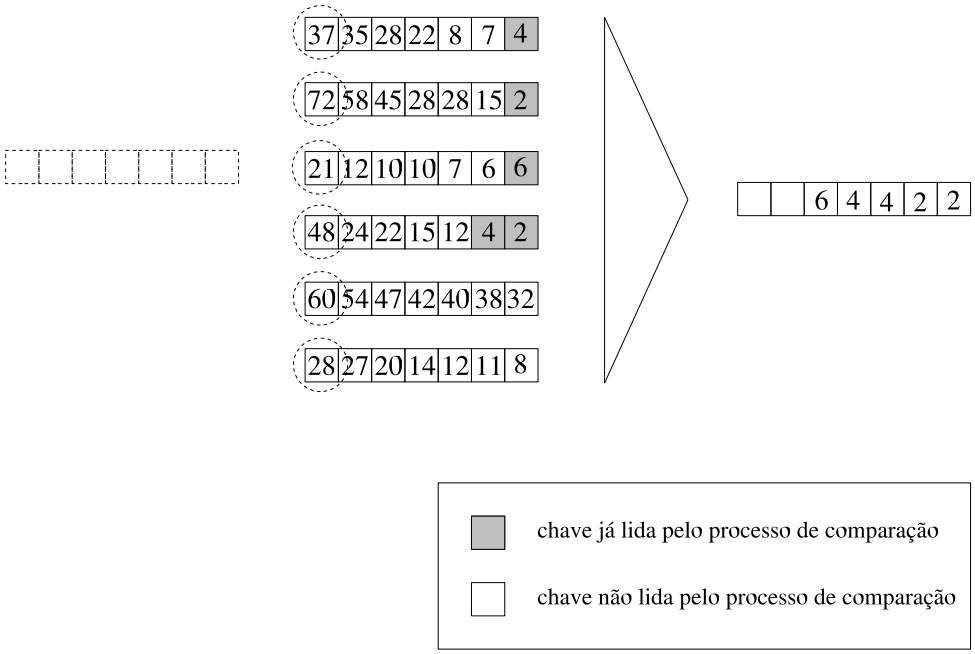


Figura 3.8: Forecasting: fase inicial da execução

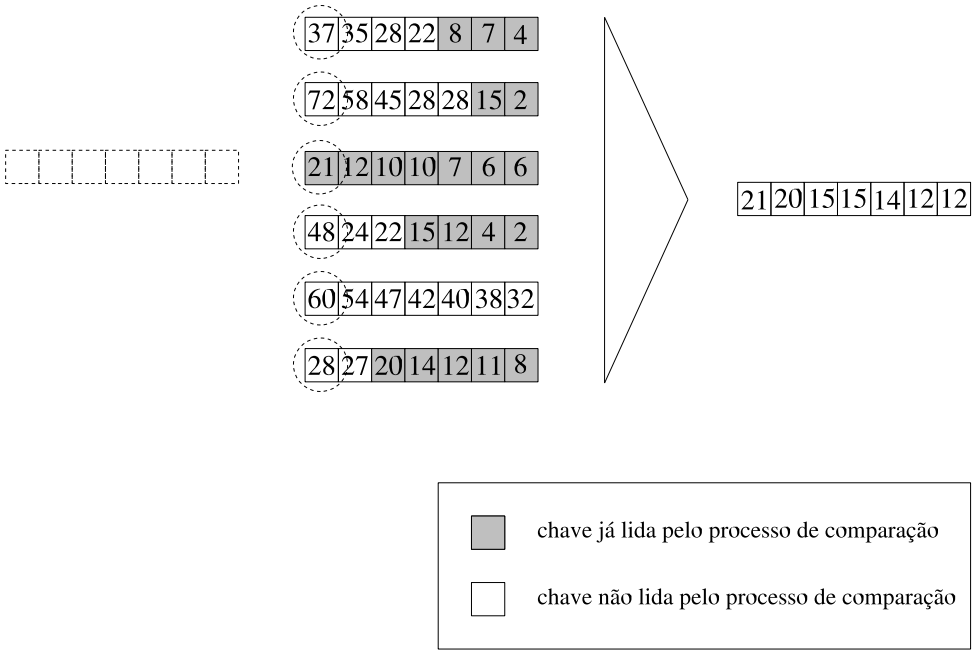


Figura 3.9: Forecasting: leitura de um novo buffer

que estão sendo colocadas as chaves já ordenadas pela intercalação é colocado na fila para gravação em disco e um outro buffer de saída começa a ser preenchido pelo processo de comparação. Se esta situação ocorrer em um ciclo, teremos sobreposição muito próxima à sobreposição completa. Se esta situação não ocorrer num ciclo, teremos sobreposição de leitura e gravação em alguns momentos do mergesort.

2. O buffer sobressalente não foi lido.

   Neste caso, o processo de intercalação será bloqueado e deverá aguardar que o processo de leitura leia um pedaço do terceiro bloco no buffer sobressalente. O processo de escrita continuará executando, e gravará o buffer de saída cheio no disco. Mas ele gravará apenas este buffer. Enquanto o processo de leitura não preencher o buffer sobressalente, não pode continuar executando e, consequentemente, não pode preencher o buffer de saída. Logo, o processo de gravação ficará bloqueado também. Se esta situação ocorrer num ciclo, teremos pouca sobreposição de leitura e escrita, mas ainda assim haverá alguma sobreposição.

Assim como o double buffering, o forecasting pode atingir a sobreposição praticamente completa. Mas nem sempre isso é possível. Calcular o quanto pode haver de sobreposição de leitura e gravação quando se utiliza o forecasting é uma tarefa complexa, dependente do conjunto, tamanho e pré-ordenação dos dados; além do tamanho da memória e a capacidade de processamento.

A maior vantagem desta técnica em relação ao double buffering é tornar possível uma boa sobreposição da gravação e leitura e diminuir a quantidade de seeks em relação ao double buffering quase pela metade. Afinal, desprezando-se os buffers de saída, no double buffering são utilizados $2n$ buffers, enquanto no forecasting são utilizados apenas $n + 1$. Desse modo, os buffers tem quase o dobro do tamanho que teriam no double buffering. Portanto, no forecasting, toda vez que os dados são lidos do disco lêem-se o dobro de dados do que seria lido no double buffering. Logo, tem-se a metade de seeks.

3.3 Forecasting estendido

Conforme visto na seção 3.2, um processo de intercalação não consegue produzir buffers vazios a uma taxa constante. Esta produção depende de diversas variáveis, como o conjunto, tamanho e pré-ordenação dos dados, além do tamanho da memória e a capacidade de processamento. No caso desta ordenação ser realizada num ambiente multiusuário com alta carga de processamento e E/S – como um banco de dados –, deve-se acrescentar ainda a variação da carga de E/S e processamento do sistema. Portanto, no forecasting, depois de ler o buffer adicional, pode ser que não haja buffers livres para que a leitura possa

continuar. Ainda que consiga produzir buffers vazios à uma taxa acima da requerida pelo processo de leitura, o próximo pedaço a ser lido do bloco é determinado no decorrer da execução. Portanto o processo de leitura tem sempre um tempo de espera depois de efetivar a última requisição de leitura. No caso de necessitar esperar apenas pela computação das chaves esse tempo é praticamente desprezível, mas ainda assim existe. Portanto, o processamento, gravação e leitura não são completamente sobrepostos. Para que seja possível realizar a sobreposição completa, é necessário que sempre haja uma requisição de leitura na fila de E/S. Para que isto seja possível, é necessário:

1. Ter mais de um buffer sobressalente para leitura
2. Saber qual bloco deve ser lido antes dos buffers acabarem

O forecasting estendido é uma extensão do forecasting tradicional [32],[33]. No forecasting estendido são utilizados $n+k$ buffers, sendo $k \in Z$, $k > 1$ e n o número de blocos. Na figura 3.10, temos um exemplo de forecasting estendido, com $k = 2$.

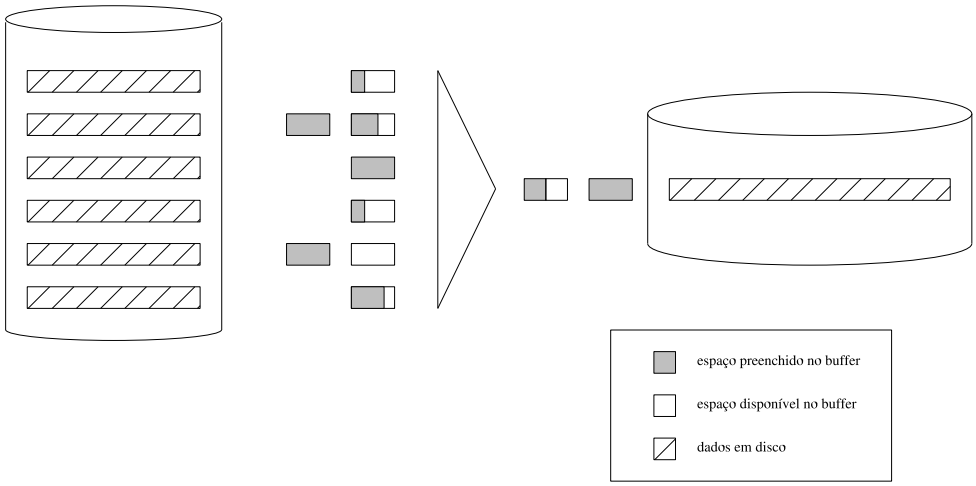


Figura 3.10: Mergesort utilizando forecasting estendido

A utilização de mais de um buffer adicional no forecasting estendido potencializa a sobreposição de leitura, processamento e gravação. Com o forecasting estendido pode-se atingir sobreposição completa.

Tanto no forecasting quanto no forecasting estendido, a leitura dos pedaços dos blocos pode ser antecipada através da comparação da última chave de cada buffer já lido. A ordem na qual os pedaços dos blocos são consumidos é chamada de sequência de consumo. Esta sequência pode variar de acordo com o algoritmo de mergesort. O algoritmo padrão de mergesort pressupõe que sempre que os buffers de entrada relativos a um bloco acabam, deve ocorrer uma leitura de um pedaço deste bloco para que o processo de intercalação

possa continuar. Sempre deve haver pelo menos um buffer relativo a cada bloco (seja ele fixo ou flutuante) preenchido. O processo não continua se para determinado bloco todos os buffers de leitura estiverem vazios. O problema é que nem sempre todos os buffers de entrada dos blocos estão sendo utilizados. No caso extremo, supondo que o arquivo a ordenar já esteja completamente ordenado, por exemplo, teremos que apenas os buffers relativos ao primeiro bloco estarão ativos até o término deste. Então apenas os buffers relativos ao segundo bloco estarão ativos. Quando o segundo bloco acabar, teremos apenas os buffers relativos ao terceiro bloco ativos, e assim por diante. Esse comportamento pode ser averiguado nas figuras 3.11 e 3.12. Nestas figuras temos uma intercalação com forecasting estendido e $k = 2$.

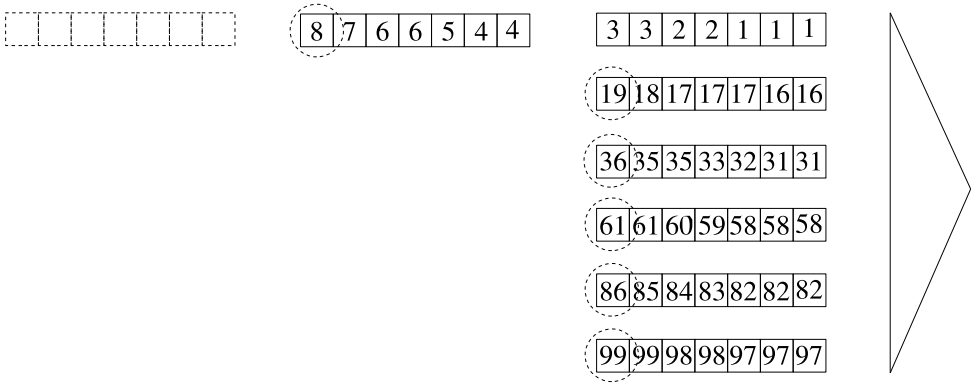


Figura 3.11: Fase inicial do forecasting estendido com algoritmo padrão do mergesort

Na figura 3.11, deu-se o início da intercalação. Foi lido um pedaço de cada bloco para um buffer, de modo que a intercalação pudesse começar. Então, através da árvore de seleção das últimas chaves do buffer, descobriu-se que o buffer alocado no primeiro bloco iria acabar. Uma leitura para este bloco foi disparada e um novo buffer foi alocado. Ao inserir a última chave do novo buffer do primeiro bloco na árvore de seleção, tem-se que será novamente o buffer relativo a este bloco o próximo a acabar.

Continuando o processo, na figura 3.12 tem-se que o primeiro bloco acabou. Portanto a chave 16 não particpa mais da árvore de seleção. Sendo assim, a próxima leitura a ser disparada deve ser para o segundo bloco, já que a chave 19 é a menor pertencente à árvore de seleção.

Como o arquivo já estava previamente ordenado (é um caso extremo), nota-se que este comportamento irá se repetir até o final da intercalação. Tem-se portanto, que durante todo o tempo teremos apenas três buffers ativos na intercalação.

Pode ser raro o caso do arquivo estar completamente ordenado, mas pode ocorrer com frequência algumas situações onde certas partes grandes do arquivo estejam quase ordenadas. E neste caso, haverá algumas fases em que o comportamento da intercalação

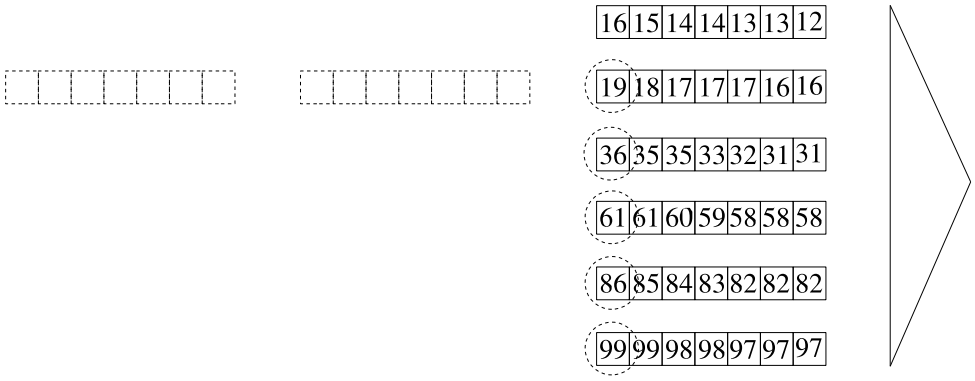


Figura 3.12: Fase intermediária do forecasting estendido com algoritmo padrão do mergesort

será bastante similar ao comportamento visto nas figuras 3.11 e 3.12.

Para cada bloco, a leitura de seus pedaços pode ser atrasada até que todos os blocos com chaves menores tenham terminado. A leitura de cada pedaço pode ser atrasada até que todos os pedaços com chaves menores que a primeira chave do bloco tenham sido lidos. Assim é possível realizar a intercalação de uma forma diferente da forma padrão. Esta forma de ler os pedaços é chamada de *intercalação com leituras atrasadas* [32]. Na intercalação com leituras atrasadas, sempre que um buffer fica vazio, o próximo pedaço requerido é o que possui a menor primeira chave dentre todos os blocos. Desta forma, é possível proceder mesmo que não hajam buffers de entrada alocados para determinados blocos. A sequência de consumo para a intercalação com leituras atrasadas não pode ser criada em tempo de execução da intercalação. Ela deve ser criada antes. Pode ser criada na fase de formação dos blocos, por exemplo. Basta realizar uma ordenação de todas as primeiras chaves de todos os pedaços dos blocos e tem-se a sequência de consumo. Se aplicarmos a intercalação com leituras atrasadas com forecasting estendido nos mesmos dados que foram utilizados nas figuras 3.11 e 3.12, teremos um comportamento similar ao da figura 3.13.

Neste algoritmo também está sendo utilizado $k = 2$. Como há 6 blocos, temos 8 buffers. Assim como o mergesort tradicional realizado nas figuras 3.11 e 3.12, este é um exemplo fictício onde cada bloco tem apenas três pedaços. O primeiro bloco acaba logo depois da leitura de seu terceiro pedaço, e assim sucessivamente.

Diferentemente do mergesort tradicional, estes buffers são alocados de acordo com a primeira chave de cada buffer, que nos dá uma sequência de consumo mais precisa. Esta sequência não pode ser computada em tempo de execução da intercalação. Deve ser realizada anteriormente. Se o próximo buffer a ser lido pertence ao n-ésimo bloco e o buffer de leitura alocado para este bloco ainda não terminou, o processo de intercalação pode continuar sua execução mesmo que outros blocos não possuam buffers de leitura

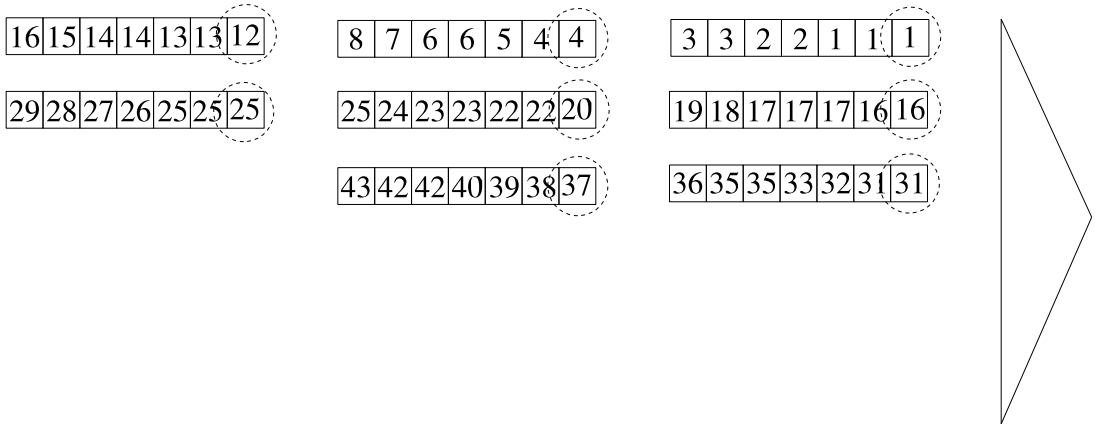


Figura 3.13: Forecasting estendido com algoritmo de intercalação com leituras atrasadas

alocados. Estes blocos sem buffers de leitura alocados não participam do processo de intercalação até o momento em que seus pedaços sejam os próximos na sequência de consumo. É interessante notar que esta técnica diminui o número de comparações para realizar a intercalação além de permitir uma maior sobreposição de leitura e gravação. Num processo de intercalação, o maior custo de CPU está relacionado às comparações entre as chaves para realização da intercalação. Portanto, tem-se também uma diminuição do custo de processamento. Há um gasto adicional de tempo apenas para criação da sequência de consumo, que no mergesort tradicional pode ser realizada concorrentemente à intercalação.

Supondo que o forecasting estendido ocorra com sobreposição completa, temos a seguir a derivação das fórmulas de tempo para um ou mais níveis de intercalação.

O tamanho do buffer, B, é dado por

$$B = \frac{M}{w+k}, \quad \text{sendo } M \text{ o tamanho da memória, } w \text{ a largura da intercalação, } k > 1 \text{ e } k \in Z \tag{3.19}$$

Sabe-se que o tempo total da intercalação com sobreposição completa é dado pelo número de seeks aliado à taxa de transferência. Sendo assim, para i níveis de intercalação, temos

$$T = i \cdot \frac{D}{B} \cdot s + i \cdot \frac{D}{t} \tag{3.20}$$

Substituindo 3.19 em 3.20, tem-se

$$T = i \cdot \frac{D}{\frac{M}{w+k}} \cdot s + i \cdot \frac{D}{t} \tag{3.21}$$

De forma equivalente, temos

$$T = i \cdot \frac{D(w+k)}{M} \cdot s + i \cdot \frac{D}{t} \tag{3.22}$$

Sendo w a largura da intercalação, e $w = \sqrt[i]{n}$, temos

$$T = i \cdot \frac{D(\sqrt[i]{n}+k)}{M} \cdot s + i \cdot \frac{D}{t} \tag{3.23}$$

Note que para o caso especial em que $k = 1$, a fórmula 3.23 vale para um forecasting simples com sobreposição completa.

Estudos experimentais para vários tipos de dados mostraram que $k = 2$ provê uma ótima sobreposição para o forecasting estendido [32],[33].

A vantagem de se utilizar o forecasting estendido em comparação ao double buffering é diminuir o custo de seeks (utiliza-se $n+k$ buffers ao invés de $2n$, normalmente com $k < n$). Sendo assim, tem-se buffers maiores, e para transferir a mesma quantidade de dados gastam-se menos seeks. Além disso, o forecasting estendido pode prover sobreposição completa dos dados. Se ao invés da intercalação tradicional for utilizada a técnica de intercalação com leituras atrasadas, a sobreposição é potencializada. O custo adicional para utilização da intercalação com leituras atrasadas é a criação da sequência de consumo realizando-se a ordenação da primeira chave de cada bloco antes do ínicio do processo de intercalação. Isto pode ser realizado durante a criação dos blocos, por exemplo.

A principal vantagem do forecasting estendido em relação ao forecasting simples visto na seção 3.2, é o aumento da sobreposição da leitura, processamento e gravação. A única desvantagem é o aumento do número de seeks, que é baixo quando se utiliza apenas mais um buffer adicional ($k = 2$) em relação ao forecasting simples.

.

3.4 Equal buffering

De forma diferente das técnicas anteriores, o equal buffering é uma técnica que utiliza leituras em grupos. Nas leituras em grupos, supõe-se que um subsistema de E/S é encarregado de realizar as leituras de pedaços dos blocos em disco para os buffers. Dados y buffers vazios alocados de forma fixa para um mesmo bloco, sendo $y > 1$, o subsistema de E/S pode preencher os y buffers com dados adjacentes deste bloco realizando apenas um seek. Para que isto seja possível, é necessário que os blocos sejam gravados de forma sequencial no disco.

O objetivo da técnica de equal buffering é economizar seeks, mantendo uma boa sobreposição de leitura, processamento e gravação. De forma diferente do forecasting e do

forecasting estendido, esta técnica utiliza buffers fixos por bloco. Na figura 3.14 nota-se como os buffers são distribuídos. A técnica de equal buffering pressupõe que o número de buffers alocado para cada bloco é sempre igual.

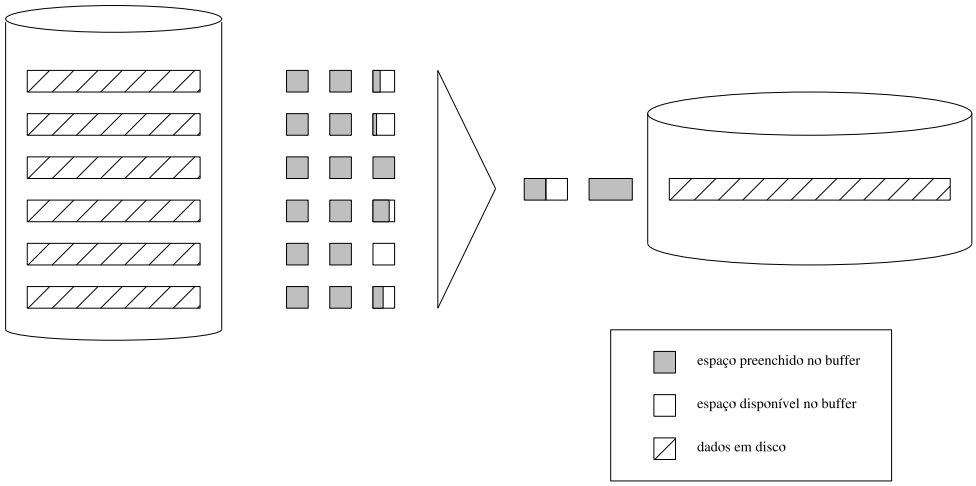


Figura 3.14: Mergesort utilizando equal buffering

O equal buffering supõe que cada bloco tenha y buffers, sendo um buffer para realização da intercalação e $y - 1$ buffers para leituras antecipadas.

Ao invés de ler um novo buffer sempre que um buffer de determinado bloco é esvaziado, espera-se que haja apenas um buffer não vazio alocado para determinado bloco. Então, dispara-se uma leitura de $y - 1$ pedaços deste bloco. Este comportamento é explicitado na figura 3.15.

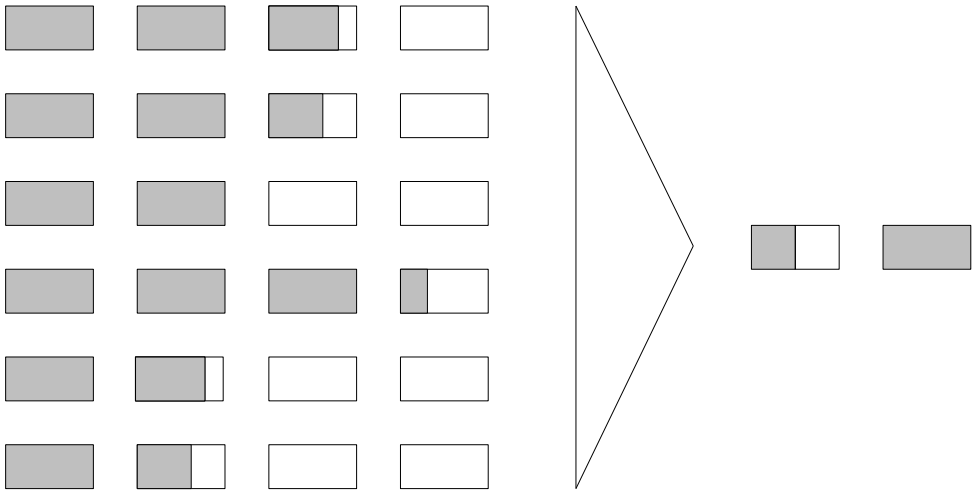


Figura 3.15: Execução do equal buffering

Nesta figura, nota-se que todos os blocos tem quatro buffers alocados. Portanto, tem-se $y = 4$. Com execeção do conjunto de buffers alocados para o quarto bloco, em todas as outras pelo menos um buffer inteiro foi consumido pelo processo de intercalação. Mas não foram disparadas novas leituras. O processo de leitura aguarda que haja apenas um buffer cheio (ou $y - 1$ buffers vazios) para disparar uma leitura. Na figura 3.16, esta nova leitura será disparada quando o penúltimo buffer do quinto bloco for consumido por completo.

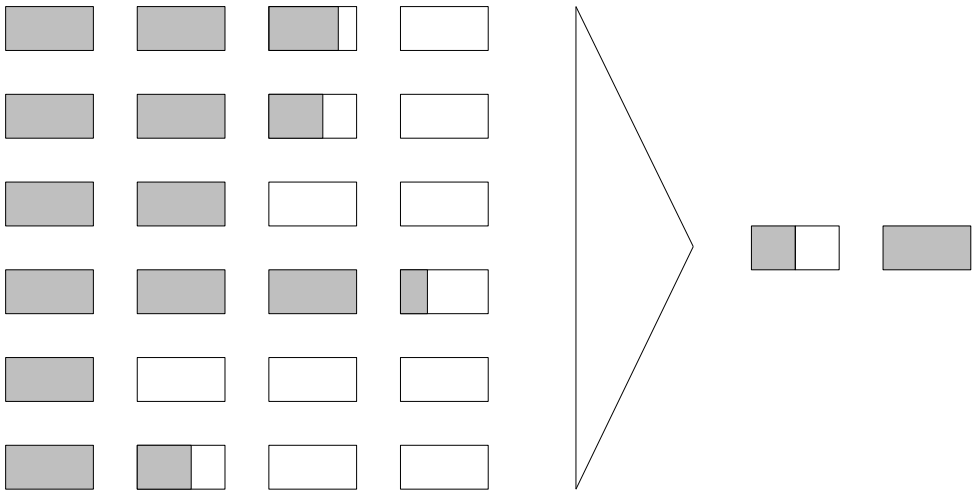


Figura 3.16: Momento de disparo de leitura no equal buffering

Então serão disparadas $y - 1$ leituras para o quinto bloco. Como todos estes $y - 1$ buffers estão alocados para o mesmo bloco, todas estas leituras serão sequenciais. O subsistema de E/S tem o poder de agrupar estas leituras de forma atômica permitindo que apenas um seek seja realizado. Portanto, ao invés de gastar-se $y - 1$ seeks, será gasto apenas um seek.

Para prover sobreposição de leitura, processamento e gravação, técnicas de ordenação que utilizam um esquema de buffers fixos devem alocar ao menos dois buffers por bloco, sendo $2n$ buffers no total. Estas técnicas podem proceder com menos buffers, mas sem atingir sobreposição completa.

Na técnica de equal buffering, como se dá uma redução do números de seeks por um fator de $(y - 1)$, uma maior quantidade de buffers é capaz de reduzir o número de seeks.

O maior problema da utilização do equal buffering é o fato de que buffers fixos não utilizam a memória ao máximo possível. Quando são esvaziados $y-1$ buffers, o processo de leitura pode enviar requisições de leitura destes buffers para o subsistema de E/S apenas se estes $y - 1$ buffers pertencerem ao mesmo bloco. Se os dados são pouco aleatórios e mal distribuídos ou os blocos não possuem o mesmo tamanho, alguns buffers podem ficar inutilizados durante um longo período de tempo. O bloco que tiver todos os seus

dados consumidos pelo processo de intercalação ainda terá y buffers alocados até o final do processo de intercalação. Esta situação pode ser vista na figura 3.17.

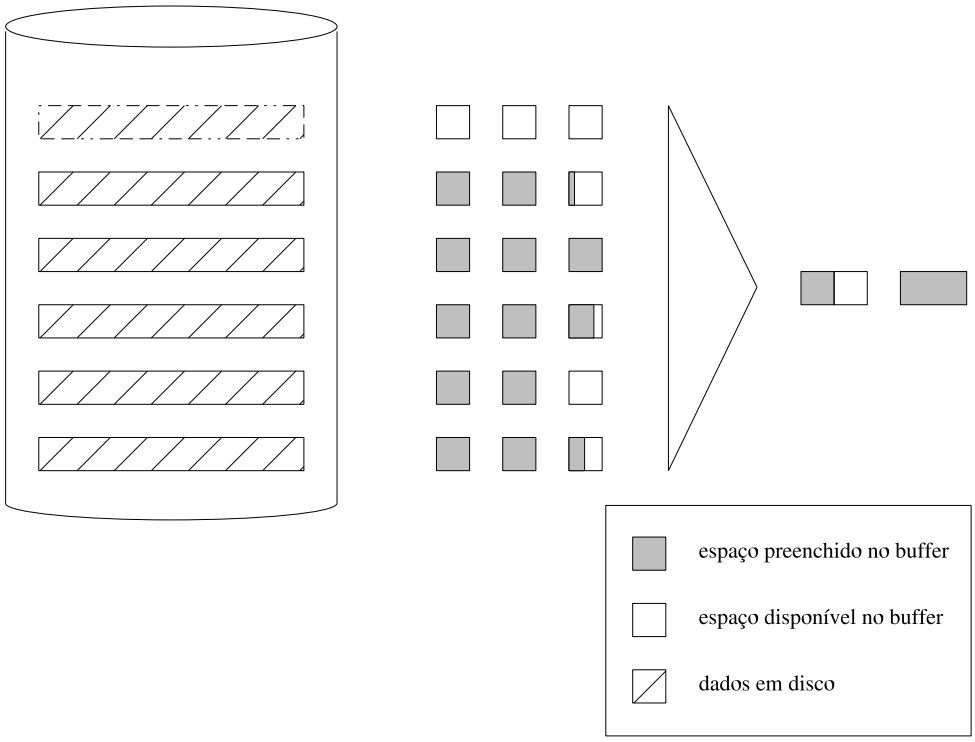


Figura 3.17: Bloco acabou e seus buffers estão inutilizados

No exemplo da figura 3.17, temos dois conjuntos de buffers alocados para blocos que estão pouco ativos. No caso do primeiro bloco, vê-se que ele já foi completamente consumido pelo processo de intercalação. Como este bloco acabou e a alocação de buffers para os blocos é fixa, os buffers alocados para o primeiro bloco não serão mais utilizados até o final do processo de intercalação. Já no terceiro bloco, nota-se que nenhum de seus buffers foram utilizados. Nenhuma chave deste bloco foi consumida pelo processo de intercalação. Portanto, seus buffers também não estão ativos. O ideal num processo de intercalação é que todos os buffers estejam ativos o tempo todo. Afinal, se temos buffers inativos durante algum período de tempo, estes buffers poderiam estar alocados em outros blocos aumentando a sobreposição de leituras e escritas. Um outro método a ser utilizado seria alocar um número de buffers para cada bloco proporcionalmente a seu tamanho em disco. Esse método é bastante heurístico, já que nem sempre os blocos menores são consumidos previamente. Para utilizar os buffers de forma mais eficiente, o ideal seria que os buffers não fossem dedicados a nenhum bloco em especial, como no esquema de buffers flutuantes visto nas técnicas de forecasting e forecasting estendido.

Supondo que o equal buffering ocorra com sobreposição completa, temos a seguir a

derivação das fórmulas de tempo para um ou mais níveis de intercalação.

Para alguns blocos, tem-se alocados $\lfloor b/w \rfloor$ buffers, enquanto que para outros tem-se alocados $\lceil b/w \rceil$. Portanto, o número médio de buffers alocados por bloco é dado por b/w. Se um bloco possui y buffers alocados, sendo $y \geq 2$ e $y \in Z$, as suas leituras serão sempre de $y-1$ buffers. Portanto, se cada bloco tem mais de dois buffers alocados para leitura, ou seja, se $b \geq 2w$, a leitura média será de $b/w - 1$ buffers. Sendo assim, para i níveis de intercalação, tem-se que o número de seeks é dado por

$$i \cdot \frac{D}{B\left(\frac{b}{w} - 1\right)} \tag{3.24}$$

e o tempo total da intercalação, incluindo a taxa de transferência é dada por

$$T = i \cdot \frac{D}{B\left(\frac{b}{w} - 1\right)} \cdot s + i \cdot \frac{D}{t} \tag{3.25}$$

De forma equivalente,

$$T = i \cdot \frac{D}{B\left(\frac{b}{\sqrt[i]{n}} - 1\right)} \cdot s + i \cdot \frac{D}{t} \tag{3.26}$$

A maior vantagem da utilização do equal buffering reside no fato das leituras serem agrupadas. Reunindo as leituras em grupos é possível diminuir o número de seeks por um fator de $(y-1)$, sendo y o número de buffers alocados por bloco. O maior problema desta técnica é o fato de utilizar buffers fixos. Com o uso de buffers fixos não é possível otimizar ao máximo a utilização da memória. Isto só ocorre com a utilização de buffers flutuantes. Algumas modificações poderiam ser introduzidas nesta técnica, como a realocação de buffers pouco ativos para outros blocos. Isso permitiria um melhor uso da memória e uma diminuição de seeks por um fator maior que $(y-1)$. Poderia ainda se promover uma distribuição desigual dos buffers, de acordo com o tamanho dos blocos formados. Utilizando-se de qualquer das modificações, o algoritmo estaria desfigurado de suas qualificações principais, e portanto, estaria se criando novo algoritmo. Com a realocação dos buffers pouco ativos em outros blocos, estariam sendo criadas novas questões a serem analisadas.

Capítulo 4

A ordenação no MySQL

A ordenação é uma tarefa realizada frequentemente em bancos de dados. É utilizada na criação de índices, em consultas com saída ordenada, operações de consulta com agrupamento, união, intersecção, diferença, junção relacional, remoção de dados duplicados e outras.

4.1 O banco de dados MySQL

O MySQL é um banco de dados relacional que se autodenomina o mais popular SGBD de código aberto do mundo. Entre suas qualificações se encontram o apoio à replicação incluso na versão padrão, controle de transação (através do InnoDB e BerkeleyDB), independência de plataforma e sistema operacional, apoio à recuperação de dados através de logs e apoio a uso do disco de forma direta (raw devices).

Uma das características mais interessantes do MySQL é a forma que lida com os sistemas de armazenamento. Diferentemente de outros SGBDs, o MySQL pode lidar com múltiplos sistemas de gerenciamento e armazenamento de tabelas (table handlers). Os mais utilizados são o InnoDB, o BerkeleyDB e o MyISAM. O MyISAM é o sistema padrão, mas não apresenta apoio à transações. O InnoDB e o BerkeleyDB apresentam.

A figura 4.1 representa a arquitetura do MySQL.

O gerenciador de banco de dados MySQL é responsável por realizar o controle de autorização de acesso do usuário, processar os comandos componentes da consulta, otimizá-la e enviar as requisições para um table handler. Na figura 4.1 há dois table handlers representados: o InnoDB e o BerkeleyDB. No table handler, o gerenciador de transações, o escalonador e o gerenciador de recuperações em conjunto com o gerenciador de buffers são responsáveis por permitir a execução concorrente das transações preservando as propriedades ACID. O gerenciador de buffers é responsável por transferir dados entre a memória principal e a memória secundária, que normalmente é o disco. Um table handler

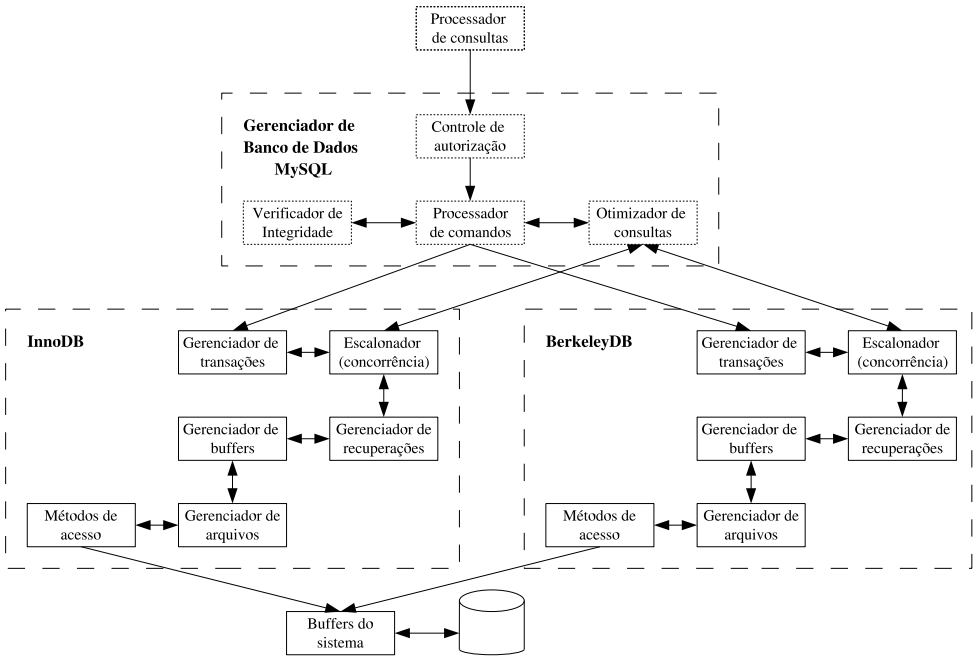


Figura 4.1: Arquitetura do SGBD MySQL

MyISAM é bem mais simples, pelo fato de não apresentar controle de transações. Ele também não possui um gerenciador próprio de buffers, confiando portanto na bufferização realizada pelo sistema operacional.

4.2 O processo original de ordenação

O processo original de ordenação do MySQL é realizado pelo programa *filesort* e mais um extenso subconjunto de programas que funcionam como bibliotecas. Este subconjunto de programas fornecem funções auxiliares para realização de E/S, controle da memória, controle do cache de E/S, comparação otimizada de chaves, manipulação de funções e dados dependendo da arquitetura da máquina e sistema operacional, entre diversas outras.

Analisando a figura 4.1, o programa filesort seria um dos componentes do processador de comandos. Na versão original, o programa filesort funciona da seguinte forma:

1. Lê todos os registros de acordo com a chave ou sequencialmente. As linhas que não satisfazem a cláusula *where* são descartadas.

2. Para cada linha, armazena num buffer uma tupla de valores correspondendo ao valor das chaves que serão ordenadas, um apontador para a tupla que contém estas chaves na tabela original e talvez as demais colunas que serão solicitadas na consulta e que não fazem parte da ordenação.

O tamanho deste buffer é fixo e escolhido como um parâmetro na inicialização do MySQL. Cada conexão ao servidor do banco de dados que utilizará o algoritmo filesort para ordenar os dados alocará um buffer deste tamanho. A tupla de valores armazenada para realização da ordenação nem sempre inclui as demais colunas que são solicitadas na consulta mas não fazem parte da ordenação. Estas colunas são incluídas apenas se a soma de seus tamanhos não exceder o tamanho de uma variável configurada na inicialização do banco. A vantagem de se incluir as colunas extras é não ser necessário fazer um segundo acesso à tabela (utilizando os apontadores armazenados na tupla) para obter o resultado completo após a ordenação. Mas como as colunas extras requerem mais espaço dentro do buffer, menos tuplas serão ordenadas a cada passo. Portanto, é possível que esta estratégia que consome mais E/S na ordenação para não necessitar de um novo acesso à tabela torne o processo mais lento, e não mais rápido. Por isso tem-se a possibilidade de determinar qual seria o maior tamanho das colunas extras. A figura 4.2 representa a estrutura de um buffer pronto para a alocação das tuplas de valores.

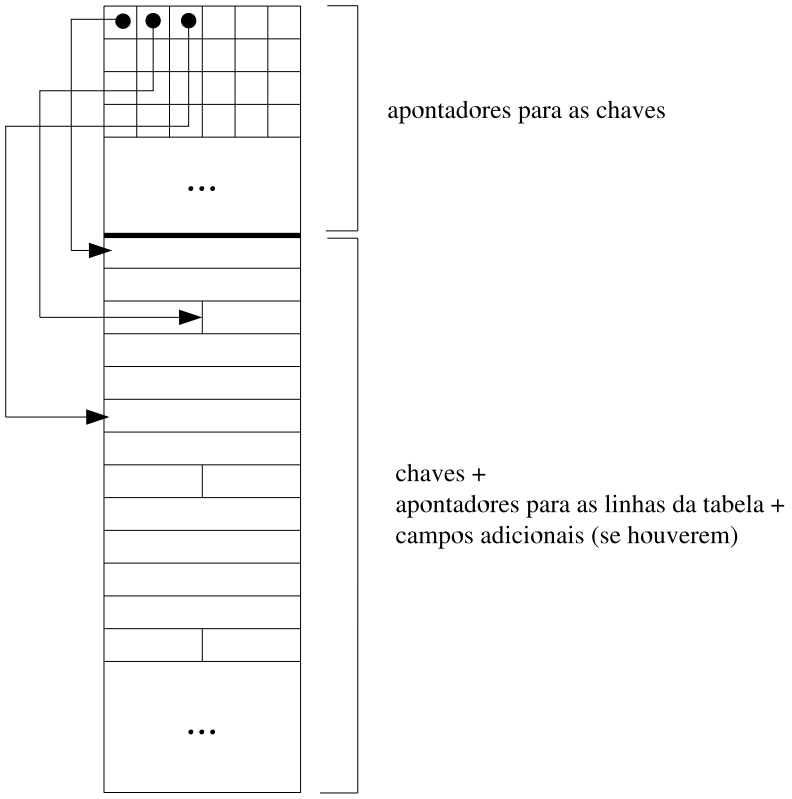


Figura 4.2: Estrutura do buffer na formação dos blocos

Na parte inicial do buffer, é reservada uma área para guardar apontadores para os

dados. A motivação desta implementação é economizar tempo de movimentação dos dados em memória, já que é mais vantajoso movimentar apontadores de tamanho fixo e pequeno (neste caso com 4 bytes) do que movimentar os dados que geralmente são maiores que 4 bytes e nem sempre de tamanho fixo.

Na parte final do buffer são armazenados os dados propriamente ditos. Eles são compostos dos campos a serem ordenados, do apontador para a sua linha na tabela original – nota-se que estes dois dados tornam essa chave única, que é um requisito fundamental – e dos campos adicionais que não serão ordenados, mas participarão do resultado final. Os campos adicionais serão colocados apenas se o seu tamanho não for maior que um valor pré-determinado na inicialização do MySQL. Antes de serem colocados no buffer, os dados propriamente ditos passam por uma normalização. Nesta normalização são uniformizados os diferentes conjuntos de caracteres em decorrência da língua utilizada. Além disso, se necessário, é trocada a ordem de cada byte de forma que bytes com maior valor relativo dentro da chave sejam comparados primeiro.

3. Após o buffer ser preenchido completamente com os dados, é rodado um radixsort ou um quicksort para ordená-los.

   A decisão de qual algoritmo será utilizado depende da arquitetura da máquina, do tamanho do conjunto de campos a serem ordenados e da quantidade de ítens. No geral, quando o tamanho é pequeno (≤ 20) e há poucos ítens (menos de 100.000), o radixsort se apresenta como um método mais vantajoso. Caso contrário, o quicksort é utilizado. Se todos os dados a serem ordenados couberem no espaço de memória destinado para a ordenação (parâmetro definido na inicialização do banco de dados), mantém-se estes dados em memória e não será necessário realizar a fase de intercalação. Se eles não couberem, o conteúdo do buffer é salvo num arquivo temporário e alguns dados com a localização deste bloco dentro do arquivo temporário são salvos em um outro arquivo temporário. O primeiro arquivo conterá todos os blocos, o segundo conterá uma espécie de tabela de índices para estes blocos. O objetivo é criar um arquivo único, com vários blocos.

4. Repetem-se todos os passos anteriores até que todas as linhas da tabela que satisfaçam as condições requeridas sejam lidas.

5. Neste instante todos os dados a serem ordenados foram lidos e parcialmente ordenados em blocos, gravados num arquivo temporário.

   Realiza-se uma intercalação de até 7 blocos gravando-se o resultado num outro arquivo temporário. Repete-se esta intercalação de largura 7 para todos os blocos

criados nos passos anteriores. Sendo n o número de blocos criados nos passos anteriores, no final de todas estas intercalações, haverá $\lceil n/7 \rceil$ novos blocos ordenados. Nota-se que estes blocos, com exceção do último, serão 7 vezes maiores que os anteriores.

6. Repete-se o passo anterior até que haja menos de 15 blocos a serem ordenados.

   A cada repetição do passo anterior, tem-se a realização de mais um nível de intercalação. Esquematicamente, haveria algo próximo à figura 4.3. Nota-se que todo o processo de intercalação, incluindo a leitura dos blocos em buffers, a ordenação através da árvore de torneio e a gravação dos dados em disco é realizado de forma não sobreposta, por um único processo. Primeiramente, é dividido o espaço de memória reservado para a ordenação (o mesmo utilizado na criação dos blocos) em 7 buffers fixos por bloco e de mesmo tamanho. São lidos o ínicio de cada um dos 7 primeiros blocos nestes buffers.

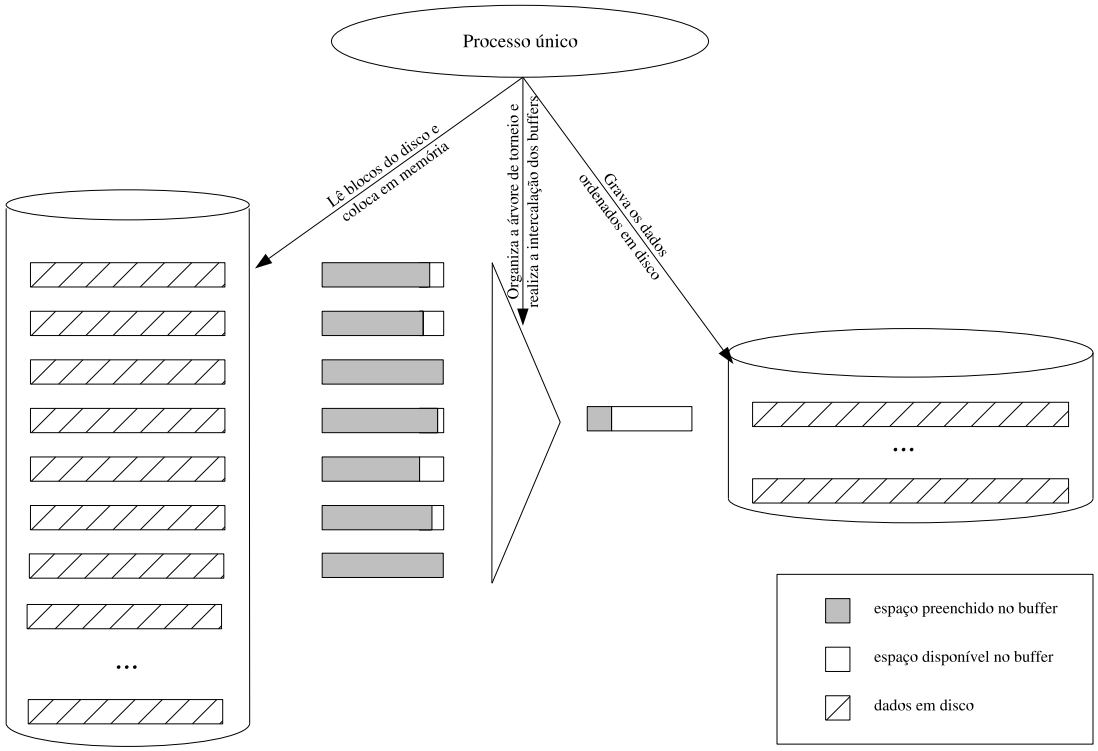


Figura 4.3: Fase de intercalação no MySQL original

É montada uma árvore de torneio, em forma de fila de prioridade [21]. Esta fila é construída num heap, e é basicamente uma árvore de torneio de ganhadores. A árvore é alimentada pelos buffers de cada um dos 7 blocos. A menor chave é colocada num buffer, que é adicional e não faz parte do espaço previamente alocado para a

ordenação. Este buffer adicional tem 64 Kbytes. Quando este buffer é completado, o mesmo processo é incumbido de gravá-lo em disco. Nota-se que quando este processo lê dados dos blocos para os buffers, ou quando grava os dados em disco, não ocorre processamento. O processo faz apenas uma coisa de cada vez. E sempre são realizadas intercalações de largura 7, sendo que a intercalação final pode ser de até 15 blocos.

7. Caso os campos adicionais que participam da consulta mas não participam da ordenação não tenham sido incluídos nos buffers por serem maiores que o padrão pré-determinado, é realizada a última intercalação onde é gravado apenas o apontador para a linha da tabela original. Neste caso, todas as linhas da tabela original que satisfazem a cláusula *where* serão relidas novamente. Esta leitura tem um custo alto, já que os apontadores estão ordenados, mas as linhas na tabela à qual referenciam não estão. Portanto, o acesso ao arquivo será aleatório.

   No caso dos campos adicionais terem sido inclusos nos buffers, o último bloco apresenta o resultado final da ordenação, e nenhuma releitura da tabela original será necessária.

4.3 Modificações introduzidas no processo de ordenação do MySQL

Conforme já descrito, a ordenação é composta por duas fases. Na primeira ocorre a leitura dos dados e criação dos blocos ordenados. Na segunda ocorre a intercalação dos blocos, criando novos blocos (de maior tamanho) ou o resultado final, dependendo da quantidade de níveis e largura da intercalação.

No novo processo de ordenação, o processo de leitura das chaves e criação dos blocos foi mantido conforme o original. Foi alterado essencialmente o processo de intercalação.

A largura da intercalação, antes fixa em 7 ou 15 blocos, tornou-se dinâmica. Foi elaborada uma heurística para tentar prever qual seria a largura ótima que otimizaria o tempo total de intercalação dos blocos.

A fase de intercalação, antes realizada por apenas um processo, de forma linear e com 7 buffers fixos, tornou-se paralela. Foi aplicada a técnica de forecasting estendido com algoritmo padrão de mergesort na realização desta fase. A memória disponível para a ordenação foi dividida em buffers flutuantes, que são alocados conforme a necessidade. O objetivo principal desta mudança foi conseguir sobreposição de processamento, leitura e escrita, já que cada uma dessas etapas são realizadas por processos paralelos, ao mesmo tempo.

4.3.1 Cálculo do novo padrão de intercalação

Depois que todos os dados a serem ordenados foram lidos e parcialmente ordenados em blocos gravados em um arquivo temporário, é aplicada uma heurística para tentar otimizar o tempo total gasto na intercalação. No processo original, o padrão de intercalação é fixo. Eram realizadas intercalações de largura 7 (7 blocos), até que restassem menos de 15 blocos. Então era realizada uma última intercalação de largura 15, no máximo. Fixar a largura certamente não resulta no padrão de intercalação ótimo. Para se calcular o padrão de intercalação ótimo, deve ser levado em consideração o tamanho do conjunto de dados a serem ordenados, o número de blocos gerados, o tamanho da memória disponível para o mergesort, o tempo médio de seek do disco e a taxa de transferência de dados entre o disco e memória.

Admitindo-se que haverá sobreposição completa de leitura e escrita, o tempo total gasto na fase de intercalação é dado pela fórmula 4.1.

$$T = i \cdot \frac{D(\sqrt[i]{n} + k)}{M} \cdot s + i \cdot \frac{D}{t} \tag{4.1}$$

A fórmula 4.1 foi deduzida na seção 3.3, e as variáveis da fórmula encontram-se na tabela 2.2.

Lembrando-se que $w = \lceil \sqrt[i]{n} \rceil$ e $i = \lceil \log_w n \rceil$, a heurística utilizada tem por objetivo descobrir o número de níveis de intercalação (i) e consequentemente a largura da intercalação (w) de tal forma que o tempo total gasto na fase de intercalação (T) seja mínimo.

Sendo B o tamanho do buffer, M o tamanho da memória disponível para o mergesort, w a largura da intercalação e k o número de buffers adicionais, o tamanho do buffer é dado por

$$B = \frac{M}{w + k} = \frac{M}{\lceil \sqrt[i]{n} \rceil + k} \tag{4.2}$$

Para que a intercalação seja possível, é necessário que uma chave a ser ordenada (composta pelos campos a serem ordenados, o apontador da linha na tabela original e os campos adicionais) caiba dentro de um buffer. Sendo assim, o menor grau possível de uma intercalação é dado por um i_{min} mínimo que satisfaça

$$B = \frac{M}{\lceil \sqrt[i_{min}]{n} \rceil + k} \geq TamanhoDaChave \tag{4.3}$$

É possível também calcular o grau máximo da intercalação. Supondo que para uma intercalação ser possível é necessário no mínimo dois blocos, o número mínimo de buffers é cinco, sendo dois buffers de leitura utilizados em cada bloco, dois buffers de saída

ou gravação e um buffer adicional para evitar problemas de deadlock. Sendo assim, o tamanho do buffer (B) deve ser

$$B \leq \frac{M}{5} \tag{4.4}$$

substituindo a fórmula 4.2 em 4.4, tem-se

$$\frac{M}{\lceil \sqrt[i_{max}]{n} \rceil + k} \leq \frac{M}{5} \tag{4.5}$$

de forma análoga,

$$\lceil \sqrt[i_{max}]{n} \rceil + k \geq 5 \Rightarrow \lceil \sqrt[i_{max}]{n} \rceil \geq 5 - k \Rightarrow \frac{1}{i_{max}} \geq \lceil \log_n (5-k) \rceil \Rightarrow \frac{1}{i_{max}} \geq \left\lceil \frac{\log (5-k)}{\log n} \right\rceil \tag{4.6}$$

portanto, o maior número de níveis possível é dado por

$$i_{max} \leq \lceil \log_{5-k} n \rceil \tag{4.7}$$

Como $k = 2$ apresentou a melhor sobreposição de processamento, leitura e escrita, temos que

$$i_{max} \leq \lceil \log_3 n \rceil \tag{4.8}$$

Sendo assim, calcula-se o tempo total gasto na intercalação através da fórmula 4.1, substituindo-se i por todo o intervalo de inteiros existente entre i_{min} e i_{max} ($i_{min} \leq i \leq i_{max}, i \in Z$). O grau de intercalação a ser utilizado é dado pelo i que minimiza o tempo total da fase de intercalação (T). Nota-se que, no caso de existirem 5.000 blocos, $i_{max} = 8$. Portanto, poucas iterações serão necessárias para descobrir o número de níveis de intercalação (i) ótimo, já que, na prática, 5.000 é um número considerável de blocos.

Com o número de níveis (i) em mãos, calcula-se o grau da intercalação, que será dado por $w = \lceil \sqrt[i]{n} \rceil$

Logo, ao invés de realizarmos intercalações de grau 7, como no original, serão realizadas intercalações de um grau dependente das diversas variáveis relativas ao problema, com o objetivo principal de minimizar o tempo total gasto na intercalação.

É interessante notar que há uma tendência em acreditar que o menor número de níveis resultará na intercalação mais rápida, já que será realizada menos transferência de dados. Nem sempre isso é verdade. O número ideal de graus de uma intercalação dependerá do tamanho dos dados com relação ao tamanho da memória, e da relação entre o tempo de seek e o tempo de transferência de dados em disco. Ao utilizar a equação 4.1, faz-se uma tentativa de ponderar estas variáveis e calcular o que seria o número ideal de graus de uma intercalação.

4.3.2 O mergesort paralelo com buffers flutuantes

Foi aplicada a técnica de forecasting estendido na realização da intercalação.

A memória disponível para a ordenação foi dividida em $(w + k + 3)$ buffers, todos flutuantes. O tamanho destes buffers é adequado para ser um múltiplo de uma página (4 kbytes), com o intuito de otimizar o acesso à memória e ao disco, onde um bloco tem 512 bytes.

Estes buffers podem ser alocados para qualquer bloco como buffers de entrada (intercalação), buffers de leitura (antecipada), ou ainda como buffers de gravação. A alocação destes buffers ocorre conforme a necessidade. A figura 4.4 representa o funcionamento do novo algoritmo de mergesort implementado no MySQL.

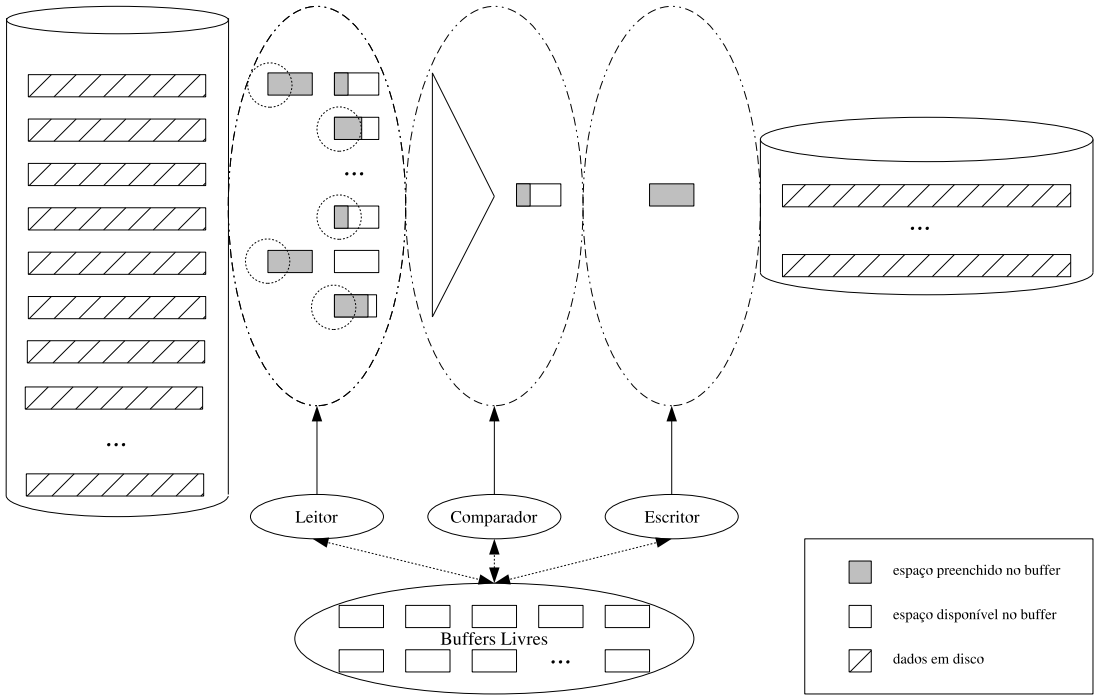


Figura 4.4: Fase de intercalação utilizando forecasting estendido no novo MySQL

São disparadas três threads:

- A primeira delas é a thread leitora. Ela é reponsável por alocar buffers de entrada (intercalação ou leitura antecipada) para os blocos, ler pedaços de blocos nos buffers de entrada e manter uma árvore de torneio contendo cada última chave do último buffer de entrada alocado para cada bloco. Esta árvore informa à thread leitora qual o próximo buffer que será completamente consumido. Com essa informação, a thread leitora é capaz de alocar um buffer livre e disparar uma leitura de um pedaço do bloco relativo ao buffer que acabará mais cedo.

- A segunda thread disparada é a thread comparadora. Ela é responsável por manter uma árvore de seleção com a menor chave não consumida de cada buffer de entrada (intercalação) para realizar a intercalação. Cada nova menor chave descoberta é escrita num buffer de saída. Quando este buffer é completado, a thread comparadora é responsável por colocar este buffer numa fila de saída (ou gravação) e alocar um novo buffer livre para que o processo de comparação de chaves para a intercalação possa continuar.

- A terceira thread criada é a thread escritora. Esta thread é responsável por gravar os buffers que se encontram na fila de saída em disco. Além disso esta thread é responsável por armazenar informações pertinentes aos novos blocos criados.

Sempre que um buffer é consumido por qualquer thread, ele é devolvido para a lista de buffers livres. Desta forma ele poderá ser realocado por outra thread como um buffer de leitura para qualquer bloco ou como um buffer de saída. Esta alocação ocorre dinamicamente, conforme as necessidades que surgem a partir da interação das três threads. Toda a memória utilizada pelos buffers é alocada dinamicamente. Seu tamanho é dado pelo tamanho máximo de memória utilizado por uma ordenação, que é um parâmetro definido na inicialização do MySQL. Esse trecho de memória constituído pelos buffers é compartilhado pelas três threads.

A principal vantagem da implementação deste método é conseguir realizar sobreposição de leitura, processamento e gravação. Depois da fase inicial de leitura, onde é necessário ler um buffer de cada bloco para que o processo de intercalação possa começar, a thread leitora realiza na maior parte do tempo leituras antecipadas. Desta forma, enquanto a thread leitora estiver lendo dados do disco, o processamento das chaves realizado pela thread comparadora pode continuar. E sempre que um buffer de saída é preenchido completamente, ele poderá ser gravado em disco, sem parar a leitura ou processamento dos dados, já que a gravação pode ser realizada em um disco diferente do disco de leitura. No caso ideal, onde os blocos que estão sendo lidos ficam em um disco e a gravação dos buffers ocorre em outro disco, pode ocorrer sobreposição completa de leitura, processamento e gravação. E tudo isso com um custo baixo de seeks, já que foram adicionados poucos buffers a mais ($k+3$). Vale lembrar que como a memória é fixa, quanto maior é o número de buffers, menor é o tamanho de cada um. Quanto menor é o tamanho de cada buffer, menos dados são lidos por vez. Portanto mais seeks são necessários para ler a mesma quantidade de dados. Como o forecasting estendido pode permitir sobreposição completa de leitura, processamento e gravação, o custo no aumento de seeks pela diminuição do tamanho dos buffers é relativamente baixo quando analisado sob a ótica do tempo total gasto na fase de intercalação.

4.4 Testes comparativos entre as versões

Foram realizados alguns testes para comprovar a eficácia das alterações no método de intercalação do MySQL. Em todos os testes realizados, o novo algoritmo de intercalação mostrou-se mais eficiente ou tão eficiente quanto a implementação original. No caso mais eficiente, a fase de intercalação chegou a ser completada em 6,25% do tempo que levou a fase de intercalação original. Isso significa que o algoritmo de mergesort original chegou a ser 16 vezes mais lento que o novo algoritmo implementado.

4.4.1 Descrição dos testes

Para testar a nova versão implementada, foram criadas algumas tabelas com dados aleatórios. Nessas tabelas foram realizados *selects* com a cláusula *order by* de modo que um mergesort externo fosse disparado para ordenar os dados. Foram medidos os tempos gastos na fase de intercalação dos blocos. O estudo dos dados deu-se de forma comparativa entre o MySQL original e a nova versão implementada.

O tamanho das tabelas utilizadas segue os fatores de escala definidos pelo benchmark TPC-H. São eles: 1, 10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000. O tamanho mínimo definido em TPC-H é 1 GB. Como não é possível manter o tamanho absoluto conforme o padrão definido no TPC-H em decorrência das características dos discos disponíveis para os testes, foram utilizados tamanhos de tabela com 1, 10, 30, 100, 300 e 1000 MB. Pelo fato da escala ser um tanto disforme (com um salto de 300 para 1000 MB, por exemplo), foram incluídos também os seguintes valores intermediários: 200, 400, 600, 800 e 1200 MB.

Os *selects* utilizados ordenam dados de todas as linhas das tabelas. O TPC-H é indicado para realizar métricas em sistemas que apresentam um grande volume de dados [27].

Nos testes foram utilizados três possíveis tamanhos de memória disponível para realização da ordenação: 10, 20 e 30MB. Este tamanho é um tamanho fixo configurado na inicialização do MySQL. O SGBD em si não aloca espaço adicional para realização do mergesort além deste espaço de memória pré-definido na inicialização.

Logo no início dos testes, notou-se um comportamento bastante aleatório com relação aos tempos gastos em ordenações de mesmo tamanho de dados e memória disponível. Percebeu-se que o responsável por tal comportamento era o cache do sistema de arquivos do sistema operacional. Quando há memória disponível para o sistema operacional, ele a utiliza para cache de arquivos conforme a necessidade. No caso do linux, a política de cache é bastante agressiva, chegando a ocupar toda a memória da máquina e até mesmo a realizar swap de processos para utilizar mais memória para o cache. Este comportamento pode ser alterado apenas em parte. Não é possível delimitar a quantidade máxima de memória que o cache do sistema de arquivos pode ocupar.

Na tentativa de estabilizar este comportamento, optou-se por uma segunda alternativa ao invés de alterar parâmetros do funcionamento da memória virtual do sistema operacional. Para não haver resquícios de possíveis dados úteis no cache do sistema de arquivos, em todos os intervalos de testes a máquina foi reinicializada para que fosse comprovada a limpeza da memória.

Foram então realizados três grupos de testes:

- Com toda a memória livre.
- Com a memória completamente ocupada, simulando alta carga de uso de memória, situação comum no uso de bancos de dados.
- Com memória livre, mas obrigando o MySQL a efetivamente gravar em disco todas as alterações que realiza nos arquivos temporários.

Em todos os testes realizados, temos três discos em operação. Num primeiro disco (A) ficam o sistema operacional, a área de swap e a base de dados. O segundo disco (B) armazena os blocos que são formados na primeira fase do mergesort. Durante o processo de intercalação, os blocos são lidos do disco B e os novos blocos gerados são armazenados num terceiro disco (C). Se for necessário mais um nível de intercalação, os papéis se invertem. Serão lidos os blocos do disco C e os novos blocos serão gravados no disco B. Isso ocorre num ciclo, até que ocorra o nível final de intercalação. Assim é possível permitir que a gravação e leitura dos dados ocorra de forma independente, sem interferências de ambas as partes. Obviamente, este modelo de não-interferência só seria garantido no caso de apenas um processo estar utilizando arquivos temporários em disco.

O testes foram realizados utilizando-se um Power Mac G4 com um processador de 1,25 Ghz e 1,5 gigabytes de memória RAM, rodando o sistema operacional Linux. As especificações da máquina, do sistema operacional, dos discos e da versão de MySQL utilizada para implementação e para os testes encontram-se no apêndice A.

A estrutura das tabelas e as consultas realizadas para disparo do mergesort externo encontram-se no apêndice B.

4.4.2 Resultados dos testes

Para melhor visualização e possibilidade de quantificação das melhorias introduzidas no MySQL, optou-se por colocar ao lado de cada gráfico formado pela relação entre o tamanho dos dados ordenados (D) e o tempo gasto pela fase de intercalação, um segundo gráfico. Este gráfico explicita a relação entre o MySQL original e a nova versão implementada. Ele mostra em porcentagem a quantidade de tempo que o novo MySQL gastou para realizar

a intercalação dos blocos em relação ao MySQL original. Este tempo, chamado de *Tempo R*, é dado pela equação:

$$TempoR = \frac{TempoMysqlNovo}{TempoMysqlOriginal} \cdot 100 \tag{4.9}$$

Os dados que alimentaram os gráficos podem ser consultados em tabelas contidas no apêndice B.3.

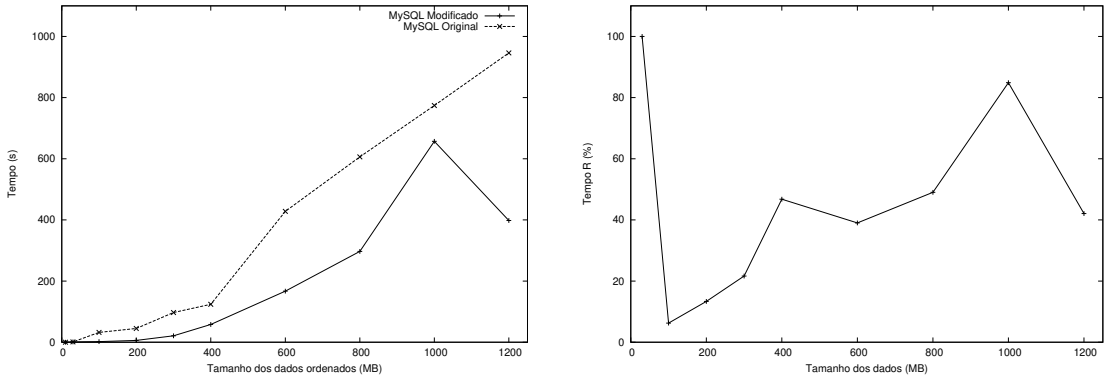
Testes com toda a memória livre

Quando toda a memória está desocupada, com exceção da memória utilizada pelos processos básicos do kernel do sistema operacional e alguns poucos serviços necessários para o funcionamento da máquina, o sistema operacional pode alocar toda a memória disponível para realizar cache de arquivos frequentemente acessados. Sendo assim, não faz sentido considerar que cada leitura ou gravação em disco realize de fato um acesso ao disco. As leituras são realizadas com *read-ahead*, e nas gravações os dados ficam em buffers do cache do sistema de arquivos. Uma chamada de sistema requisitando uma escrita retorna antes da gravação dos dados.

Portanto, quando o sistema operacional tem bastante memória disponível, a ordenação não utiliza o disco conforme o modelo simplificado. Mesmo que seja alocado para o banco de dados MySQL uma pequena quantia de memória para ordenação, o MySQL faz chamadas de escrita em disco que ficam efetivamente em buffers do cache do sistema operacional. Quando é necessário ler os blocos, o MySQL faz uma chamada de leitura ao sistema. Se o arquivo lido foi recentemente "gravado" e se encontra em cache, nenhum acesso de leitura é realizado. Este é um comportamento frequente quando há bastante memória disponível para o sistema operacional. Este comportamento acaba permitindo inclusive que o processo de ordenação original, que é único, sobreponha alguma leitura, processamento e gravação, já que uma chamada de sistema invocando uma escrita em disco escreve apenas em memória (no cache) e retorna.

Na figura 4.5, temos o comportamento das fases de intercalação no MySQL original e na nova implementação do MySQL. É definido como parâmetro que a memória disponível a ser utilizada pelo processo de ordenação é de 10 megabytes ($M = 10$).

Analisando o gráfico 4.5(a), nota-se que quando o volume de dados a ser ordenado é pequeno, o custo absoluto de ordenação é pequeno e próximo em ambas as implementações. Com mais dados a ordenar, mais acessos a disco são necessários, e o tempo absoluto das ordenações sobe consideravelmente. Esse crescimento não é linear, já que o espaço reservado em memória para ordenação continua o mesmo e um maior volume de dados significa possivelmente a necessidade de mais níveis de intercalação. Esse comportamento é comum a todos os outros experimentos. O gráfico 4.5(b) mostra a relação do tempo



(a) Custo da fase de intercalação

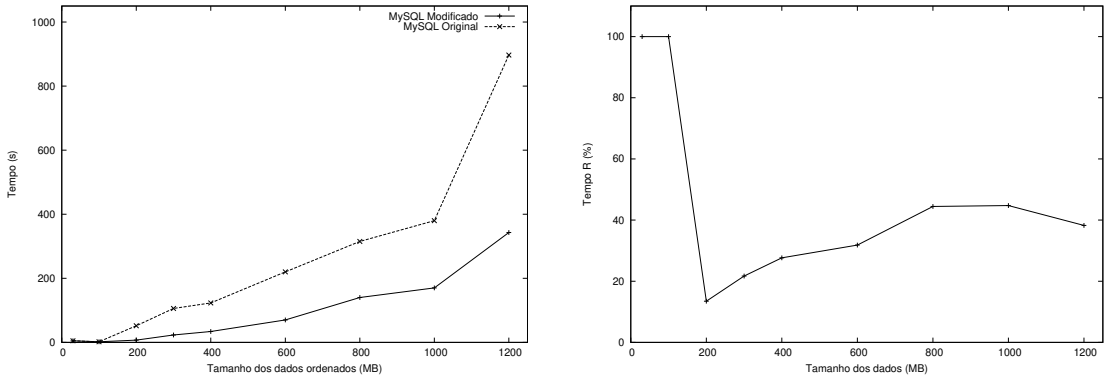
(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.5: Mergesort com 10 megabytes de memória para ordenação e memória livre para o SO

gasto para realização da fase de intercalação entre o novo MySQL e o MySQL original. De modo geral, espera-se que o novo MySQL seja em média 50% mais rápido que o MySQL original, pelo menos. Esta expectativa decorre do fato de que como o MySQL original não realiza sobreposição de leitura, processamento e gravação, o custo de sua ordenação abrange estas três fases linearmente. Espera-se também que o novo MySQL apresente picos de excelente desempenho quando comparado ao MySQL original. Estes picos decorrem da heurística utilizada. Com uma largura de intercalação que contempla a relação entre o tamanho dos dados e da memória e o custo dos seeks em relação à transferência de dados, o novo MySQL se sobressai em determinados instantes já que, por manter uma largura de intercalação fixa, o MySQL original pode realizar um nível a mais de merge sem que haja necessidade.

Supondo que o novo MySQL sobreponha leitura, processamento e gravação completamente, o custo gasto para ordenação será apenas o custo de leitura. Em média, isso daria pouco mais de 50% do custo de ordenação do MySQL original. No gráfico 4.5(b), há um ponto em que o custo da intercalação do novo MySQL é apenas 6,25% do custo no MySQL original. Enquanto o MySQL original realizou a intercalação em 32 segundos, o novo MySQL conseguiu a mesma façanha em apenas 2 segundos. Esse comportamento é explicado pela heurística no cálculo do padrão de intercalação a ser realizado. Com a largura da intercalação fixada em 7 blocos, o padrão de intercalação torna-se muito restrito. Fixando a largura, a intercalação é feita a "olhos vendados", ou seja, não é ponderada a relação entre o tamanho dos dados a serem ordenados e o tamanho da memória em relação ao custo de seeks e custo de transferência em disco. Com a largura fixa,

o padrão de intercalação se torna uma loteria onde os acertos não modificam muito o comportamento do mergesort, mas os erros podem torná-lo lento demais.



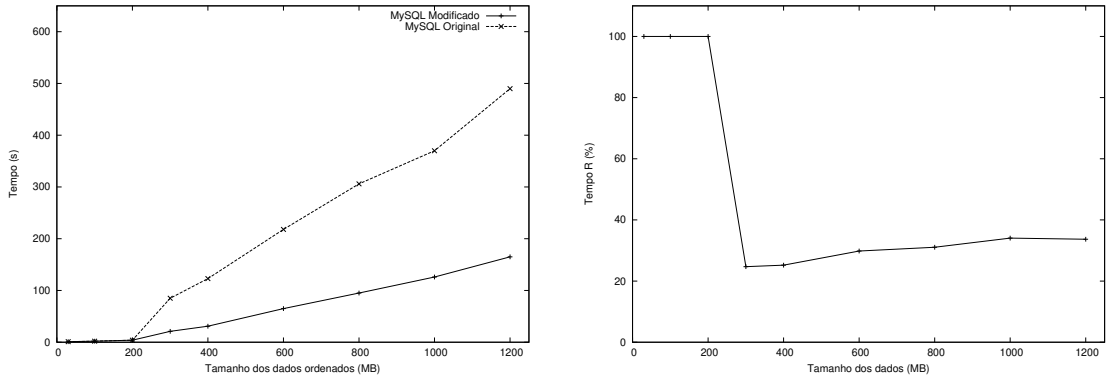
(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.6: Mergesort com 20 megabytes de memória para ordenação e memória livre para o SO

Os gráficos 4.6 e 4.7, que representam ordenações com memória disponível para ordenação de 20 e 30 megabytes, respectivamente, apresentam comportamentos bastante parecidos com o encontrado no gráfico 4.5. É importante ressaltar dois aspectos relativos a estes gráficos com relação ao gráfico 4.5. O primeiro deles é que, quando a memória utilizada para ordenação tem 20 ou 30 MegaBytes e os dados a serem ordenados tem 100 megabytes, o padrão de intercalação fixo apresentado no MySQL original acaba sendo igual ao padrão apresentado no novo MySQL. Sendo assim, com 20 ou 30 MegaBytes, tanto o MySQL original quanto o novo gastam o mesmo tempo (2 segundos) para realizar a intercalação de 100 megabytes de dados. O outro aspecto importante notado nos gráficos 4.5(b), 4.6(b), e 4.7(b), é o fato de estar ocorrendo um gradativo aumento da diferença de tempo gasto entre as intercalações conforme o volume de dados a ser ordenado aumenta, a partir de 200 megabytes, com vantagem para o novo MySQL.

Não é possível prever se este comportamento terá prosseguimento, já que a heurística adotada na criação do padrão de intercalação acaba se tornando um diferencial de peso na ordenação. Pode ocorrer o mesmo que ocorreu quando o volume de dados ordenados era de 100 megabytes, sendo que ao aumentar em 10 megabytes o espaço disponível para ordenação, o tempo relativo gasto pelo novo MySQL subiu de 6,25% para 100%. O que é possível concluir é que, quando há um aumento do tamanho de memória disponível para a ordenação, o novo MySQL é capaz de tirar um maior proveito já que realiza o cálculo do padrão de intercalação de forma dinâmica.



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

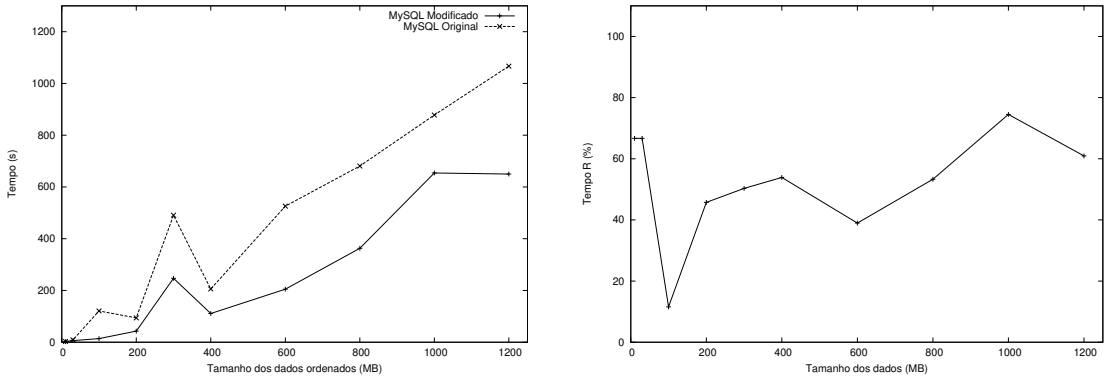
Figura 4.7: Mergesort com 30 megabytes de memória para ordenação e memória livre para o SO

Testes com toda a memória ocupada

A motivação principal desta bateria de testes é observar o comportamento do MySQL quando há pouca memória RAM disponível para uso na máquina. Num ambiente de produção este comportamento pode ocorrer com frequência. Também é usual o grande uso de CPU e disco. Quanto à CPU, o processo de intercalação requer pouquíssimo uso. Nos testes realizados, a CPU ficava com utilização em torno de 3% a 5%. Portanto, dificilmente um uso intenso de CPU traria alterações significativas na fase de intercalação. A fase de criação de blocos costuma apresentar picos de uso de CPU de até 100%, quando os dados são ordenados. Mas as alterações foram implementadas e analisadas apenas na fase de intercalação. A maior parte do tempo nesta fase é gasta em E/S. O objetivo de ocupar a memória disponível é limitar o uso de cache dos arquivos temporários para poder analisar o funcionamento do mergesort num ambiente em que o uso de memória não permitisse o cacheamento de uma grande fatia dos arquivos temporários.

Para isso, foi implementado um processo que consome toda a memória disponível. Para garantir que o conteúdo de memória consumido não seja transferido para a área de swap, este processo é incumbido de sujar em um intervalo de tempo pré-definido 1 byte de cada página de memória consumida. Assim é possível consumir a memória e mantê-la residente ao processo consumidor durante toda a fase de intercalação, com um gasto baixo de CPU.

A análise do gráfico 4.8 mostra um comportamento similar ao encontrado no gráfico 4.5. Novamente surge a grande diferença entre os custos para ordenação do volume de



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

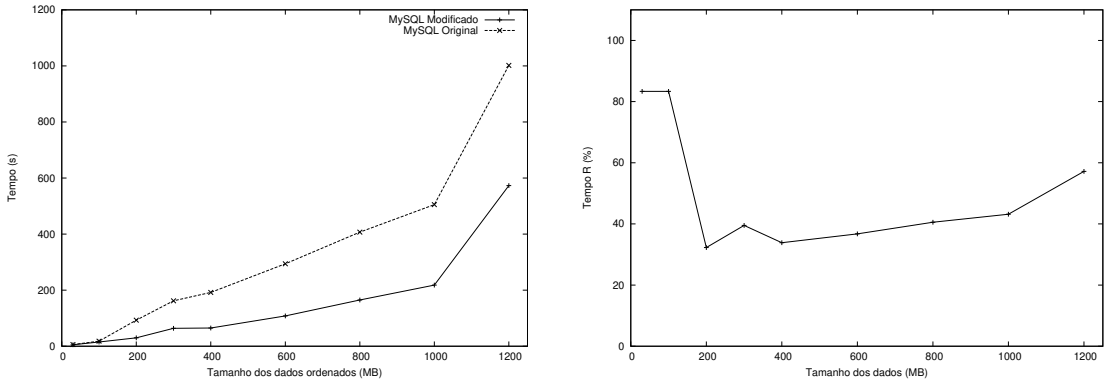
Figura 4.8: Mergesort com 10 megabytes de memória para ordenação e restante da memória ocupada

100 MegaBytes de dados no MySQL original e no novo MySQL. Essa diferença se dá pelo uso da heurística de padrão de intercalação.

O gráfico 4.8(b) mostra que para um volume pequeno (10 e 30 megabytes) e grande (1000 megabytes) de dados ordenados, o novo MySQL apresentou um custo relativo menor do que o apresentado quando o sistema operacional tinha bastante memória disponível e poderia utilizá-la para cache dos arquivos temporários (gráfico 4.5(b)). Isso ocorre porque a nova fase de intercalação implementada é mais independente do sistema operacional do que a fase de intercalação antiga. Quando o sistema operacional não tem a possibilidade de realizar cache dos arquivos temporários a vontade, o MySQL original é obrigado a aguardar a gravação dos dados no disco quando estão sendo gerados novos blocos. Sendo assim, ele fica incapaz de sobrepor processamento e gravação com a mesma voracidade. Como o novo MySQL implementa um mecanismo de sobreposição de leitura, processamento e gravação independente do sistema operacional, através do uso de três threads distintas para cada atividade na fase de intercalação, ele consegue se sobressair quando os recursos disponíveis para o sistema operacional são escassos.

Com 20 megabytes de memória disponíveis para ordenação, constata-se um declínio nos tempos de intercalação (gráfico 4.9(a)) e uma certa estabilização do tempo relativo gasto pelo novo MySQL (gráfico 4.9(b)) entre 40% e 50%.

Comparando-se os tempos gastos na fase de intercalação no novo MySQL, a suposição de que uma maior quantidade de memória disponível para a ordenação resultaria em menores tempos de intercalação, por haver menos acessos a disco, é comprovada. A cada aumento do tamanho de memória disponível para o mergesort (vide gráficos 4.8(a), 4.9(a)



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.9: Mergesort com 20 megabytes de memória para ordenação e restante da memória ocupada

e 4.10(a)) nota-se que o tempo gasto para realizá-lo descresce gradativamente tanto no novo MySQL quanto no MySQL original, com uma certa vantagem para o novo MySQL.

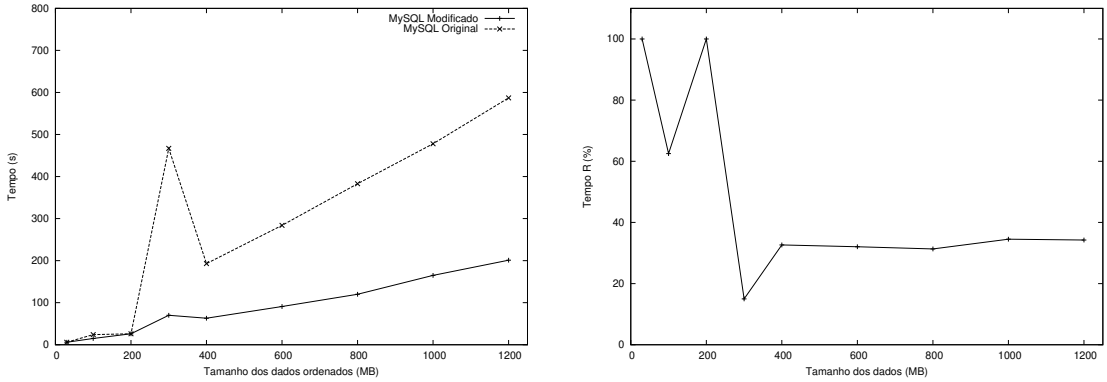
Ao utilizar 30 megabytes de memória para ordenação, o novo MySQL gasta menos de 40% do tempo para intercalar os dados do que gastaria o MySQL original (vide gráfico 4.10(b)).

Testes com obrigatoriedade de acesso a disco em cada gravação

Quando ambas as implementações da fase de intercalação são obrigadas a gravar dados em disco, tem-se um comportamento mais próximo ao modelo simplificado. Para realizar este experimento obrigou-se que cada acesso de gravação em disco efetivamente fizesse um seek e gravasse os dados. Isso sem atualizar metadados do arquivo, o que ocasionaria um segundo seek em disco para atualização do inode. Para realizar esta façanha, foi utilizada a chamada *fdatasync* após cada chamada *write*.

Tem-se um comportamento bastante similar utilizando-se 10, 20 ou 30 megabytes de memória disponível para ordenação (vide gráficos 4.11(b), 4.12(b) e 4.13(b)). Nota-se uma ligeira queda nos tempos absolutos tanto no MySQL original quanto no novo MySQL (vide gráficos 4.11(a), 4.12(a) e 4.13(a)) conforme esperado, já que a cada intercalação representada por cada gráfico foram disponibilizados 10 megabytes a mais de memória (10, 20 e 30 megabytes).

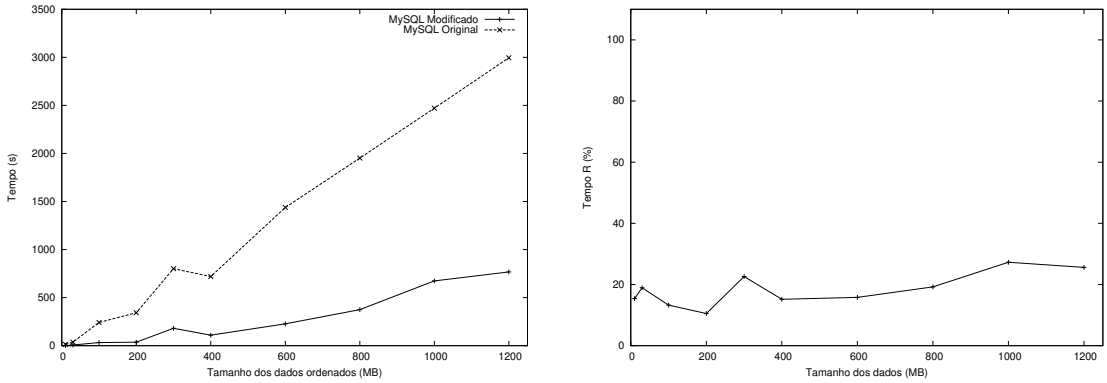
O tempo relativo gasto pelo novo MySQL tende a se estabilizar torno de 10 a 20%, denotando uma grande vantagem para o MySQL.



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

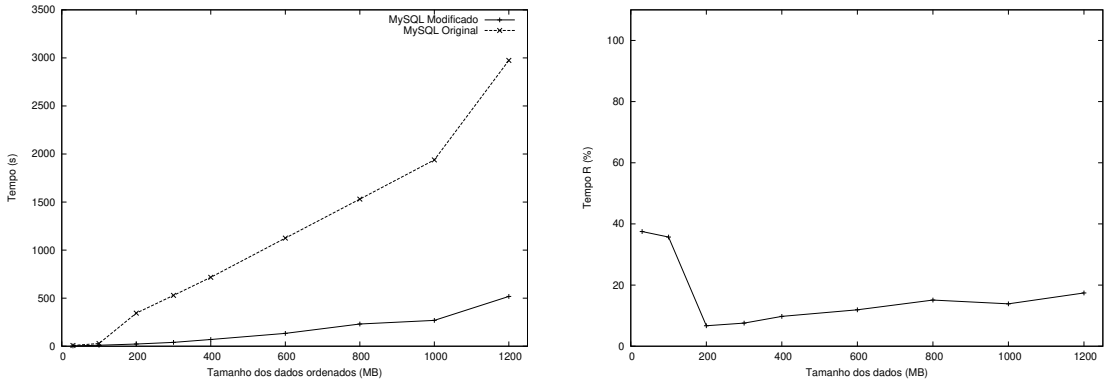
Figura 4.10: Mergesort com 30 megabytes de memória para ordenação e restante da memória ocupada



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.11: Mergesort com 10 megabytes de memória para ordenação e acesso a disco em cada gravação



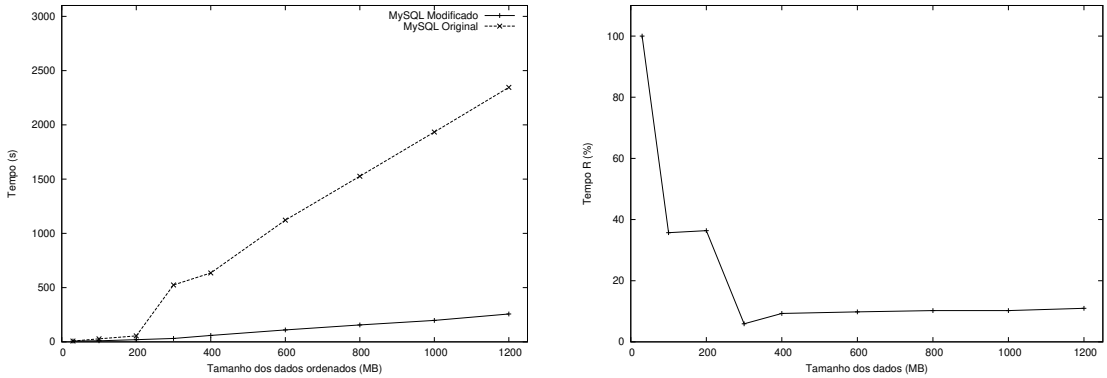
(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.12: Mergesort com 20 megabytes de memória para ordenação e acesso a disco em cada gravação

Os testes em que há obrigatoriedade de acesso a disco em cada gravação mostram o caso extremo de se realizar a ordenação quando é impossível que o sistema operacional realize cache do sistema de arquivos.

A cada conjunto de testes apresentado – com memória livre, com memória ocupada, com gravação obrigatória – a interferência do sistema operacional nas atividades de ordenação diminui. Analisando os gráficos referentes a todos os experimentos, nota-se que conforme diminui a interferência do sistema operacional nas atividades de ordenação, aumenta o desempenho do novo MySQL com relação ao MySQL original. Além de mostrar que o novo MySQL é mais independente da infraestrutura do sistema operacional do que o original, a análise destes gráficos mostra que o novo MySQL se torna cada vez mais atrativo conforme ocorre a degradação dos recursos disponíveis na máquina.



(a) Custo da fase de intercalação

(b) Relação do custo entre o novo MySQL e o MySQL original

Figura 4.13: Mergesort com 30 megabytes de memória para ordenação e acesso a disco em cada gravação

Capítulo 5

Conclusões

Muitos trabalhos de otimização de ordenação externa se basearam em modelos simplificados ou em simuladores de bancos de dados. Quando é utilizado um modelo simplificado, pressupõe-se que toda gravação ou leitura realiza um acesso ao disco, o que nem sempre é verdade. O sistema operacional provê mecanismos para adiar ao máximo o acesso a disco, já que este é diversas ordens de grandeza mais lento que o acesso à memória. O modelo simpificado pode ser mais adequado à implementação prática de forma transparente quando é realizado acesso direto ao disco, o que não é praxe na utilização de arquivos temporários no MySQL. Mesmo quando utiliza table handlers com acesso direto ao disco (InnoDB e BerkeleyDB), os dados temporários são gravados em arquivos num sistema de arquivos comum. Nesses arquivos temporários são criadas e lidas as fatias parciais ordenadas do conjunto de dados, os blocos. Como há mais uma camada (o sistema de arquivos) entre o SGBD e o disco, o comportamento de um mergesort pode ser alterado por diversos fatores, entre eles o tamanho da memória disponível na máquina, mesmo que ela não seja efetivamente utilizada pelo mergesort.

O principal objetivo deste trabalho foi realizar um estudo de técnicas de otimização de algoritmos de ordenação externa, em especial do mergesort externo e fazer uma avaliação prática de uma das técnicas. Com isso foi possível diminuir o tempo total gasto na realização do mergesort através de um melhor uso dos recursos de memória e de E/S no banco de dados MySQL.

5.1 Contribuições

A ordenação é realizada com frequência num ambiente de bancos de dados. É utilizada em consultas com saída ordenada, na criação de índices, em operações de consulta com agrupamento, remoção de dados duplicados, união, intersecção, diferença, junção relacional, e outras [7], [17]. Sendo assim, otimizar a ordenação num banco de dados significa não

otimizar apenas a requisição direta do usuário por um resultado ordenado, mas otimizar a vazão geral do banco.

Na realização deste trabalho foi implementada e analisada a técnica de forecasting estendido com o algoritmo padrão de mergesort na fase de intercalação dos blocos durante uma ordenação de dados externa no banco de dados MySQL. Foi também desenvolvida e implementada uma heurística para otimizar o padrão de intercalação no MySQL, que antes era realizado com largura fixa de blocos.

Através de alguns experimentos foi possível notar que a fase de intercalação do novo MySQL obteve um comportamento até 16 vezes mais rápido que o MySQL original. Esse comportamento decorreu basicamente de dois fatores, sendo um deles o uso de uma heurística para cálculo do padrão de intercalação e o outro a divisão do processo em threads para atingir sobreposição de leitura, processamento e gravação dos dados. Ambos os fatores tiveram por objetivo otimizar o uso de memória e principalmente o uso de disco. Através da utilização do forecasting estendido foi possível realizar sobreposição de leitura, processamento e gravação com um baixo aumento de seeks, quando comparado à utilização da técnica de double buffering, por exemplo.

Em linhas gerais, a principal contribuição deste trabalho é mostrar que mesmo SGBDs que se encontram solidificados no mercado podem ser otimizados. O MySQL provê quase todas as características e funcionalidades de um banco de dados comercial de grande porte, sendo utilizado por grandes empresas, como o Yahoo!, por exemplo [31].

Além disso, através deste trabalho foi possível analisar a implementação de uma técnica não convencional de mergesort num ambiente real. Na literatura padrão que cobre a implementação de ordenações externas em bancos de dados [6], [17], são exemplificadas apenas as técnicas convencionais. Talvez este seja um dos pilares que sustentam a implementação do mergesort do MySQL original da forma que é realizada hoje, sem grandes otimizações na fase de intercalação. Já a fase de formação dos blocos é um pouco mais otimizada, com a normalização das chaves para diminuição do custo de comparação e a ordenação de forma indireta (sempre utilizando apontadores) para conter o custo de movimentação de grande quantidade de dados em memória. Ainda assim, algumas otimizações na fase de formação dos blocos também seriam possíveis.

Como última grande contribuição, o resultado deste trabalho será enviado para o pessoal da MySQL AB para uma possível incorporação em futuras versões do MySQL. Na prática, talvez essa seja a maior contribuição deste trabalho, já que poderá beneficiar um grande número de pessoas e entidades que utilizam o banco de dados MySQL em todo o mundo.

5.2 Trabalhos futuros

O problema de ordenação externa é um problema cativante. Muitas análises, ponderações de parâmetros e implementações são possíveis. A relação entre o tamanho dos dados a serem ordenados e o tamanho da memória disponível para a ordenação, a relação entre o custo de seeks no disco em comparação ao custo de transferência de dados do disco para a memória, o grau de pré-ordenação das chaves, o tamanho do buffer utilizado, entre outros, são variáveis num problema de solução ótima ainda não desvendada e que possibilita diversas análises. Portanto, muitas análises e contribuições ainda são possíveis.

Dando continuidade a este trabalho, poderia ser implementado para avaliação uma técnica de agrupamento de buffers para diminuir efetivamente o número de seeks em disco. Para que essa implementação obtivesse sucesso, seria necessário repensar em todo o subsistema de E/S do MySQL, criando arquivos temporários com os blocos diretamente no disco, através de partições *raw* ou acesso direto a arquivos. Com um subsistema de E/S temporária direta no banco, a técnica de equal buffering poderia ser implementada na fase de intercalação. Outras técnicas de agrupamento tão interessantes quanto o equal buffering também poderiam ser implementadas, como o agrupamento com leituras atômicas [32].

Na fase de criação dos blocos, poderia ser desenvolvido um novo algoritmo de replacement selection para gerar blocos com tamanho maior que o tamanho disponível em memória. Isto não é usual de ser implementado em bancos de dados pela dificuldade de implementação quando o tamanho das chaves a serem ordenadas é variável. Alguns trabalhos provém algumas idéias para elaboração deste tipo de algoritmo [12]. Para tirar melhor proveito de uma intercalação com blocos maiores que o tamanho da memória e com um tamanho variável, seria necessário calcular os padrões de intercalação de forma diferente, conforme apresentado na seção 2.4.3. A heurística utilizada neste trabalho para determinar o número de níveis que otimiza o tempo total da intercalação daria lugar ao cálculo do padrão de intercalação que minimiza transferência de dados.

Com uma nova fase de criação de blocos, com blocos maiores que o tamanho de memória disponível, e um novo algoritmo que explora leituras em grupos minimizando seeks na fase de intercalação dos blocos, resultados bastante promissores poderiam ser conseguidos. Ambas as tarefas apresentam um alto grau de dificuldade e seria necessário uma maior quantidade de tempo para implementar as idéias e atingir bons resultados.

Além disso, durante o desenvolvimento deste trabalho o autor se deparou com diversas outras possíveis áreas promissoras de otimização num SGBD. Elas não dizem respeito à ordenação no banco, mas assim como a ordenação, são utilizadas com frequência e poderiam ser otimizadas. O cache de dados realizados pelo SGBD segue uma política de paginação de dados baseada no algoritmo LRU. No geral, a maioria dos SGBDs utiliza

políticas de cache de dados baseados no LRU. O sucesso da política de cache dos dados depende basicamente do tipo de acesso que é realizado com maior frequência. Para ler dados sequencialmente e de forma constante, o algoritmo LRU se apresenta como uma má opção de política de cache. Alguns algoritmos tentam melhorar este comportamento mantendo a idéia inicial do LRU, entre eles [9], [15]. Um melhor estudo poderia ser realizado a respeito de políticas de cache de dados de tabelas em disco e algumas novas técnicas poderiam ser implementadas.

Com a evolução dos sistemas de arquivos disponíveis e novas ferramentas para facilitar a administração dos mesmos, como os gerenciadores de volumes, além do aumento do uso de SGBDs de código aberto, o uso de sistemas de arquivos para armazenamento de tabelas tem crescido em relação ao uso de partições raw. Estudos mostram uma pequena vantagem (de 2% a 5%) de desempenho na utilização de partições raw ao invés de sistemas de arquivos [31]. Como a administração de sistemas de arquivos é mais simples que a administração de partições raw e o desempenho não é tão díspar, os sistemas de arquivos tem se mostrado como uma boa opção para o armazenamento de bases de dados. Quando um sistema de arquivos é utilizado para armazenar as tabelas ao invés do SGBD realizar acesso direto, diversos parâmetros configuráveis no sistema operacional podem ser decisivos no aumento de desempenho do banco de dados. Estes parâmetros são relativamente distintos entre os diversos sistemas operacionais, o que torna seu estudo prático bastante extenso. Apesar destes parâmetros não terem sido alterados nos testes realizados neste trabalho, notou-se que um ajuste fino nos mesmos pode trazer extensas contribuições no desempenho geral do banco.

Apêndice A

Especificações do hardware e software utilizados

A.1 Máquina

A especificação da máquina utilizada encontra-se na tabela A.1

A.2 Discos

Os três discos utilizados tem a mesma especificação. Ela se encontra na tabela A.2

A.3 Sistema operacional

Foi utilizado o sistema operacional Yellow Dog Linux, versão 4.0, com kernel 2.6.8. Este kernel utiliza threads NPTL. O Yellow Dog Linux é uma distribuição de Linux para PowerPC baseada no Fedora Core.

A.4 Banco de dados

Foi utilizado para desenvolvimento e testes o MySQL versão 5.0.4, compilado em Linux com threads NPTL (Native POSIX Threading Library).

Tabela A.1: Especificação da máquina

Componente	Característica
cpu	7455, altivec supported
clock	1249MHz
revision	3.3 (pvr 8001 0303)
bogomips	1245.18
machine	PowerMac3,6
motherboard	PowerMac3,6 MacRISC2 MacRISC Power Macintosh
detected as	129 (PowerMac G4 Windtunnel)
pmac flags	00000000
L2 cache	256K unified
memory	1536MB
pmac-generation	NewWorld

Tabela A.2: Especificação dos discos

Característica	Valor
Marca	Maxtor
Modelo	92049U3
Capacidade	20.490 MB
Cilindros	16.383
Seek médio	9 ms
Latência média	5,55 ms
Velocidade rotacional	5.400 rpm
Taxa de transferência	66 MB/s
Bytes por bloco	512
Controladora Integrada	ATA-5/Ultra DMA 66

Apêndice B

Especificações das tabelas e consultas realizadas

B.1 Tabelas

Foram criadas 11 tabelas para a realização dos testes. Todas com os mesmos campos. Elas foram chamadas de t1, t10, t30, t100, t200, t300, t400, t600, t800, t1000 e t1200. Em cada tabela, o número de linhas multiplicado pelo tamanho das chaves utilizadas para ordenação resulta em 1, 10, 30, 100, 200, 300, 400, 600, 800, 1000 e 1200 megabytes, respectivamente. O propósito dos tamanhos 1, 10, 30, 100, 300 e 1000 megabytes é fazer com que D, que é o tamanho dos dados a serem ordenados, seja proporcional as distribuições definidas no TPC-H: 1, 10, 30, 100, 300 e 1000. Os tamanhos restantes são utilizados para uma melhor observação dos gráficos. Todos os campos, inclusive os de tipo não numérico, foram alimentados com números aleatórios.

A descrição das tabelas se encontra na tabela B.1.

B.2 Consultas

Em todas as consultas, o seguinte select foi realizado:

select period from tabela order by period,name

Esta consulta obriga a ordenação da chave composta pelos campos *period* e *name*.

B.3 Dados obtidos

Os dados obtidos nos experimentos encontram-se distribuídos da seguinte forma:

- Testes com toda a memória livre
  - Custos e Tempo R com 10 megabytes de memória disponível para ordenação. Ver tabela B.2
  - Custos e Tempo R com 20 megabytes de memória disponível para ordenação. Ver tabela B.3
  - Custos e Tempo R com 30 megabytes de memória disponível para ordenação. Ver tabela B.4
- Testes com toda a memória ocupada
  - Custos e Tempo R com 10 megabytes de memória disponível para ordenação. Ver tabela B.5
  - Custos e Tempo R com 20 megabytes de memória disponível para ordenação. Ver tabela B.6
  - Custos e Tempo R com 30 megabytes de memória disponível para ordenação. Ver tabela B.7
- Testes com obrigatoriedade de acesso a disco em cada gravação
  - Custos e Tempo R com 10 megabytes de memória disponível para ordenação. Ver tabela B.8
  - Custos e Tempo R com 20 megabytes de memória disponível para ordenação. Ver tabela B.9
  - Custos e Tempo R com 30 megabytes de memória disponível para ordenação. Ver tabela B.10

Quando a consulta realizada não disparou um mergesort externo, seus tempos e dados não fizeram parte da tabela.

Tabela B.1: Descrição das tabelas

Campo	Tipo	Tamanho em bytes
period	int	4
name	char	32
address1	char	32
address2	char	32
company	int	4
price1	int	4
price2	double	8
price3	double	8
price4	double	8

Tabela B.2: Tempos: 10 MB / Com mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
10	0	0	100,0
30	1	1	100,0
100	2	32	6,3
200	6	45	13,3
300	21	97	21,6
400	58	124	46,8
600	167	428	39,0
800	297	606	49,0
1000	657	774	84,9
1200	398	946	42,1

Tabela B.3: Tempos: 20 MB / Com mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	5	5	100,0
100	2	2	100,0
200	7	52	13,5
300	23	106	21,7
400	34	123	27,6
600	70	220	31,8
800	140	315	44,4
1000	170	380	44,7
1200	343	897	38,2

Tabela B.4: Tempos: 30 MB / Com mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	1	1	100,0
100	2	2	100,0
200	4	4	100,0
300	21	85	24,7
400	31	123	25,2
600	65	218	29,8
800	95	306	31,0
1000	126	370	34,1
1200	165	490	33,7

Tabela B.5: Tempos: 10 MB / Sem mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
10	2	3	66,7
30	6	9	66,7
100	14	121	11,6
200	43	94	45,7
300	247	491	50,3
400	111	206	53,9
600	205	526	39,0
800	363	681	53,3
1000	654	878	74,5
1200	650	1067	61,0

Tabela B.6: Tempos: 20 MB / Sem mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	5	6	83,3
100	15	18	83,3
200	30	93	32,3
300	64	162	39,5
400	65	192	33,9
600	108	294	36,7
800	165	407	40,5
1000	218	505	43,2
1200	573	1002	57,2

Tabela B.7: Tempos: 30 MB / Sem mem.

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	6	6	100,0
100	15	24	62,5
200	26	26	100,0
300	70	467	15,0
400	63	193	32,6
600	91	284	32,0
800	120	383	31,3
1000	165	478	34,5
1200	201	587	34,2

Tabela B.8: Tempos: 10 MB / Acesso a disco obrigatório

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
10	2	13	15,4
30	7	37	18,9
100	32	241	13,3
200	36	343	10,5
300	181	801	22,6
400	109	719	15,2
600	227	1438	15,8
800	375	1952	19,2
1000	674	2471	27,3
1200	767	2995	25,6

Tabela B.9: Tempos: 20 MB / Acesso a disco obrigatório

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	3	8	37,5
100	10	28	35,7
200	23	344	6,7
300	40	529	7,6
400	70	717	9,8
600	134	1125	11,9
800	231	1531	15,1
1000	269	1939	13,9
1200	518	2974	17,4

Tabela B.10: Tempos: 30 MB / Acesso a disco obrigatório

(D) (MB)	Novo MySQL(s)	MySQL original(s)	Tempo R(%)
30	8	8	100,0
100	10	28	35,7
200	20	55	36,4
300	31	524	5,9
400	59	635	9,3
600	110	1122	9,8
800	156	1526	10,2
1000	198	1934	10,2
1200	257	2345	11,0

Referências Bibliográficas

[1] Alok Aggarwal and Scott Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] Micah Beck, Dina Bitton, and W. Kevin Wilkinson. Sorting large files on a backend multiprocessor. *IEEE Trans. Comput.*, 37(7):769–778, 1988.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[4] Michael J. Folk, Greg Riccardi, and Bill Zoellick. *File Structures: An Object-Oriented Approach with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[5] Ricardo Galli. Journal file systems in linux. http://130.206.130.95/impresion.phtml?nIdNoticia=1154, dezembro 2005.

[6] Hector Garcia-Molina, Jennifer Widom, and Jeffrey D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[7] Goetz Graefe. Implementation of sorting in database systems, submetido em 2003.

[8] Christoffer Hall, Bjarke Mortensen, Philippe Bonnet, Heikki Tuuri, and Peter Zaitsev. Do linux asynchronous i/o really matter?, abril 2003.

[9] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[10] The linux kernel archives. http://www.kernel.org/, janeiro 2006.

[11] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Publishing Co., Inc., Reading, Massachussets, USA, 1973.

[12] Per-Åke Larson. External sorting: Run formation revisited. *IEEE Trans. Knowl. Data Eng.*, 15(4):961–972, 2003.

[13] Mysql ab - developer zone. http://dev.mysql.com/, janeiro 2006.

[14] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: a risc machine sort. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 233–242, New York, NY, USA, 1994. ACM Press.

[15] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD Conference*, pages 297–306, 1993.

[16] Patrick E. O'Neil. Database performance measurement. In *The Computer Science and Engineering Handbook*, pages 1078–1092. 1997.

[17] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill Science/Engineering/Math, 2002.

[18] Kay A. Robbins and Steven Robbins. *UNIX Systems programming: communication, concurrency, and threads.* Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 2003.

[19] Marc J. Rochkind. *Advanced UNIX programming.* Addison-Wesley, 2. ed. edition, 2004.

[20] B. Salzberg. Merging sorted runs using large main memory. *Acta Inf.*, 27(3):195–215, 1989.

[21] Robert Sedgewick. *Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[22] Dennis Shasha. Tuning databases for high performance. *ACM Comput. Surv.*, 28(1):113–115, 1996.

[23] Dennis Shasha. Lessons from wall street: case studies in configuration, tuning, and distribution. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 498–501, New York, NY, USA, 1997. ACM Press.

[24] Dennis Shasha and Philippe Bonnet. Database tuning: principles, experiments, and troubleshooting techniques (part i). In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 637–637, New York, NY, USA, 2002. ACM Press.

[25] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[26] Toni Strandell. Open source database systems: systems study, performance and scalability. Master's thesis, University of Helsinki, Helsinki, Finland, 2003.

[27] Tpc - transaction processing performance council. http://www.tpc.org, janeiro 2006.

[28] A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. Fast sort: An external sort using parallel processing. *Tandem Technical Report 86*, 1986.

[29] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[30] Jeffrey Scott Vitter. External memory algorithms and data structures. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society Press, Providence, RI, 1999.

[31] Jeremy D. Zawodny and Derek J. Balling. *High Performance MySQL*. O'Reilly, 2004.

[32] W. Zhang. *Improving the Performance of Concurrent Sorts in Database Systems*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1997.

[33] Weiye Zhang and Per-Åke Larson. Buffering and read-ahead strategies for external mergesort. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 523–533, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[34] LuoQuan Zheng and Per-Åke Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.