上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目：通过分析应用场景优化键值数据库的性能

学生姓名：苏春智

学生学号：5080309068

专　　业：计算机科学（ACM 班）

指导教师：过敏意

学院(系)：电子信息与电气工程学院

通过分析应用场景优化键值数据库的性能

——为生产者-消费者数据缓冲仓库设计专一型键值数据库

摘要

功能完备、结构复杂、适用范围广泛的通用型数据库，曾长期占据数据库领域的统治地位。然而，随着并行计算和分布式系统的不断发展，现代大型计算集群的数据处理能力迅速增加，以至于通用型数据库的性能无法满足其需求。因此，人们开始利用每个具体应用性质设计专一型的数据管理系统。

在本课题中，我针对键值数据库的一个重要的应用场景——分布式系统中的生产者-消费者数据缓冲仓库，设计了专一型键值数据库。本文首先分析了该应用场景对键值数据库的功能需求、性能需求。经过深入探索，本文揭示了该场景中一个重要的数据访问模式，即“几乎有序”的数据访问模式。依据该应用场景的需求和“几乎有序”数据访问模式的特点，我设计了一种新型的数据索引结构——FBTree，以及基于 FBTree 的、专门用于该应用场景的键值数据库 FBTS。

实验数据表明，在“几乎有序”的数据访问模式下，FBTS 在一秒内可以插入 200K 个键值对，是 Berkeley DB 的三倍；利用区域读取的功能， FBTS 可在一秒内读取 400K 个键值对。另外，FBTS 对变长数据、并行读取都有较好的支持。FBTS 对乱序访问也有一定的容忍能力，可以在损失不超过 25%性能的前提下，容忍多达 5%由数据丢失引起的乱序插入。

关键词： 键值数据库，数据访问模式，几乎有序，FBTree，适应性分裂策略，FBTS

BUILDING APPLICATION-SPECIFIC KEY-VALUE DATABASES BY EXPLOITING DEMANDS AND USAGE PATTERNS

ABSTRACT

General-purpose databases, with rich functions, complex structures and wide scope of applications, have dominated the area of DBMS for a long time. However, as we are making progress in parallel computing and distributed systems, general-purpose databases cannot catch up with the increasing amount of data processed by modern computing clusters, and people start to build application-specific databases by exploiting unique characters and patterns in their applications.

In this thesis, I designed an application-specific key-value database for producer-consumer warehouses in distributed systems, which is an important and common application of key-value databases. I analyzed the functional and non-functional requirements in this application, and revealed an important data access pattern that key-value pairs are inserted, retrieved, and deleted in "almost" ascending order. With those demands and patterns, I designed a brand-new data structure for indexing, which is called "Flat BTree" (FBTree), and an application-specific key-value database FBTS based on FBTree.

Experiments show the inserting speed of FBTS under almost ordered pattern is over 200 K/s, which is as 3 times as that of Berkeley DB. With range queries, the retrieving speed of FBTS is over 400 K/s. FBTS supports variable length data very well, and is immune to concurrent readings. FBTS also tolerates over 5% out-of-order insertions caused by data loss with an acceptable performance reduction of 25%.

Key Words: key-value database, data access pattern, almost ordered pattern, FBTree, adaptive split strategy, FBTS

目 录

第一章　绪论和背景

1.1　键值数据库与专一型数据库

Web2.0 时代，互联网的数据量日益增长，传统的关系型数据库过于笨重，已不能满足应用需求，键值数据库（Key-Value Database）和其他非关系型数据库（NoSQL）在处理海量数据方面的优势日益体现，于是大批键值数据库应运而生，如 Google 的 BigTable[1], LevelDB, Amazon 的 Dynamo[2]，Berkeley DB[3] 和 Scalaris[4]。相较于传统的 SQL 数据库，键值数据库是一种轻量、高性能、易扩展（scalable）的数据库，大多数键值数据库放弃了对复杂查询、结构化数据、强一致性、甚至 ACID transaction[5]的支持，转而提高数据吞吐量、降低操作延时、支持超长数据[6][7][8],是一种典型的高性能数据库。

大部分键值数据库的数据模型都是简单的键值映射，也有例外，如 BigTable 则根据应用场景采用了三元组“行号，列号，时间戳”到值域的映射，但几乎所有键值数据库都不区分值域的内部结构。相较于关系数据库高度结构化的数据模型，键值映射是一种弱结构，允许用户更自由地存储不同类型的数据库，也简化了数据库系统的维护开销。由于系统不再知晓数据的内部结构，大多数键值数据库只提供主键索引。索引结构同时又是影响数据库访问方法的重要因素，如果数据库采用 BTree 索引，则可完整支持键查询和区间查询，而使用哈希表索引的数据库一般不支持区间查询。根据应用需求，大部分键值数据库只支持简单的插入操作，而不支持复杂的修改操作。对于不频繁进行并发读写的应用，可以通过锁机制实现强一致性，如 Berkeley DB 即支持 ACID transaction；但在频繁并发读写的场景下，强一致性会引入性能瓶颈，所以部分键值数据库不保证强一致性，只保证最终一致性（Eventually Consistency），并只提供有限的 transaction，如 BigTable；另外，Dynamo 使用可变参数 Quorum 算法[9]，不同的参数可以实现不同强度的一致性。

键值数据库与关系数据库的理念差异很大。关系数据库提供了丰富、广谱的功能集，期望为各种数据管理需求提供通用的解决方案；键值数据库则认为必须为不同的应用场景设计轻量级、专一型的解决方案，因为大多数应用场景都不需要完整的广谱功能集（如 join 操作、严格一致性），而这些复杂的功能严重影响了数据库的性能，得不偿失[10]。因此，许多键值数据库都具有专一而强烈的应用背景，只为特定的场景提供最小的功能支持，并针对特定的应用场景优化系统架构，以提高系统性能。不同的键值数据库会提供不同的数据模型、索引服务、访问方法、transaction 支持和一致性保证；同一个数据库在不同应用场景下的性能也有差异。

由此可见，通过发掘需求、量身定制，可以大幅提高数据库的性能。本文以一个典型的键值数据库应用场景——生产者-消费者数据缓冲仓库为例，分析该应用场景对数据库服务的需求，发掘数据访问模式，并为其设计了专一型键值数据库——FBTS。借助该项目需求分析、创新设计、开发优化的过程，我们可以管中窥豹地了解专一型数据库在当今计算机系统科学领域的重要价值。

1.2　问题来源和研究动机

为生产者-消费者数据缓冲仓库设计专一型键值数据库的问题来源于微软亚洲研究院的 Grape 项目[11]。Grape 是一个分布式流处理系统，需要在每一对生产者、消费者之间部署一个数据缓冲仓库。由于 Grape 系统的数据吞吐量很大（以及其他原因，见 2.1），这个缓冲仓库无法在内存中实现的，必须让数据在硬盘中暂存，因此这个仓库的实质便是一个键值数据库。

最初我们使用成熟的 Berkeley DB 和 Level DB 作为数据缓冲仓库，但发现他们的性能无法满足 Grape 的需求。我们认为 Berkeley DB 等通用数据库性能较低的原因有：

（1） 它们包含了事务处理、并发访问控制、日志等复杂逻辑，然而 Grape 的应用场景并不需要这些功能，这些功能反而会拖慢数据库效率；

（2） 它们没有利用 Grape 系统的数据访问模式进行性能优化；

总体而言，使用通用数据库的问题在于“牛刀杀鸡”：我们仅需要其中一小部分的功能，通用数据库却捆绑了太多我们不需要的功能。因此，我决定为生产者-消费者数据缓冲仓库的应用实现一个拥有最小功能集的高效键值数据库，这便是课题的来源。

1.3 论文结构

针对生产者-消费者仓库这个应用场景，本文第二章首先分析该应用场景对键值数据库的功能性需求和非功能性需求，随后在第三章探索并揭示此应用场景中一组十分有价值的数据访问模式。

第四章着重展示一个全新的索引数据结构 FBTree，FBTree 的基本设计理念与 B-Tree 接近，但它使用了与 BTree 不同的节点分裂策略，使 FBTree 可以充分利用该场景中的数据访问模式，提高了索引性能。第五章则描述了专门为该应用场景设计的专一型键值数据库——FBTS（FBTree Storage），该数据使用 FBTree 进行数据索引，以优化在该应用场景下的性能。

最后，第六章提供了实验数据以证明 FBTS 在该应用场景下的性能优势。

第二章　应用场景介绍与需求分析

在本章，我们以 Grape 分布式计算系统为例，来解读“生产者-消费者仓库”这一应用场景的特性，尤其是分析分布式系统对键值数据库的各种需求。

2.1　应用场景介绍

第二代互联网、传感器网络都会产生大量连续的数据流，但我们缺少分析这些海量数据的工具：StreamInSight[12]等流处理引擎迄今为止只能运行于单机，不能扩展到商业集群，当数据流量增大时便无法及时处理；传统的分布式计算系统，如 MapReduce[13]和 MadLinq[14]，主要面向矩阵运算等分阶段计算，并不善于处理数据流和连续询问（continuous query），更不能保证结果的实时性。Grape 系统很好地弥补了这一缺陷。该系统是微软亚洲研究院于 2011 年至 2012 年新开发的一个分布式计算系统，可以在大规模商业集群中对流数据进行实时、连续的处理。Grape 系统面临的主要挑战，亦是该系统与 MapReduce 等传统分布式系统的主要区别在于：（1）Grape 需要处理无限长的数据流，因此计算没有“开始”和“结束”的界限，而是增量式、连续进行的，各个计算节点需要以流水线的形式收发并处理数据；（2）Grape 是一个实时系统，必须降低数据处理和数据传输的延迟，以满足用户对终端延迟的要求。

为了缓冲上下游运算节点的吞吐速率不匹配，Grape 系统在上游节点（生产者）与下游节点（消费者）的数据传输通路上部署一个键值数据库作为缓冲仓库，以缓冲生产者和消费者之间的吞吐速率不匹配；另外，Grape 系统需要面临各种运行时动态事件，如故障恢复、计算模型的动态重配置等，为此 Grape 系统要求键值数据库在把数据推送至消费者后，将数据继续保留一段时间，以作为故障恢复时的“上游数据备份”（upstream backup）[15]，避免产生严重影响性能的溯往重计算（cascade re-computation）。

我们的键值数据库工作于 Grape 每一对相邻计算节点之间的流水传输线上，自然继承了 Grape 所面临的这两个挑战：传统（分阶段、离线）分布式系统的数据传输通路在整个生命周期内只需进行数次传输，而流水线计算则需连续、频繁地传输增量数据；同时，这些数据传输的延时必须得到控制。例如，Grape 的一个应用是通过实时收集的推特数据分析人们对特定话题的态度变化，根据 2011 年的推特统计数据[16]，该系统需要处理的峰值流量为每秒 10000 条推特，即使将每 100 条推特（10 毫秒当量）打包传输，键值数据库每秒仍需处理 100 次传输。在另一个网络连通性探测的应用中，键值数据库每秒更是需要处理数千次传输。

2.2　需求分析

图 2-1 描绘了 Grape 系统中每对生产者-消费者之间数据通路的结构：

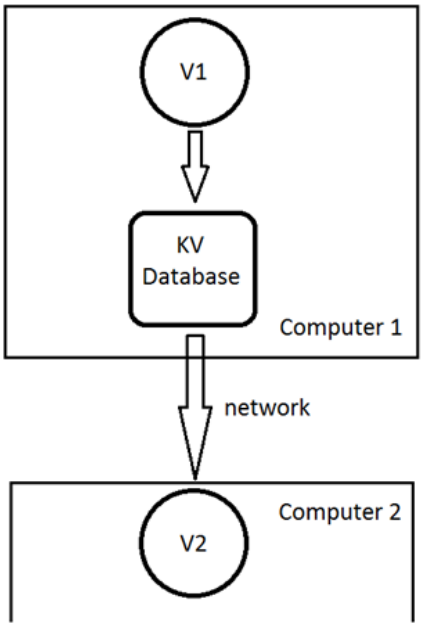

图 2-1　Grape 的数据通路

在系统正常运行时，计算节点 v1 输出一个数据流（无限长的键值对序列），该数据流即作为计算节点 v2 的输入流，在此过程中，键值数据库作为一个数据缓冲区调节生产者与消费者的吞吐速率波动。但该处的键值数据库与传统意义的缓冲区仍有不同：当一个数据被推送给消费者 v2 之后，键值数据库并不把该数据删除，而是像数据库一样继续保存该数据，直到分布式系统明确地发来数据删除指令。键值数据库为数据流提供更长的生存周期的主要目的是提供上游数据备份：如果在随后的计算中 v2 出现故障，分布式系统可以直接命令键值数据库将备份数据重新推送给 v2，恢复 v2 的内部状态，而不需要 v1 重新计算这些数据。数据删除的时机则由 v2 对输入数据的依赖关系决定，当 v2 的恢复不再需要某数据时，该数据即可被删除。

在某些罕见情况下，如生产者内部错误、数据丢失、进行故障恢复、对计算模型进行重配置、系统压力过大时，v1 也会产生乱序数据，这时就需要键值数据库依据数据的键对数据进行有序索引，无论何时，v2 总可以从数据库中获得有序的输入流。

总结上述场景，我们在数据传输通路上部署键值数据库以用于：

(1)　调节生产者与消费者的速度不匹配；

(2)　对数据进行上游备份，防止数据丢失或下游计算节点崩溃；

(3)　利用有序索引对乱序数据进行排序。

另外，键值数据库还需提供区域查询（range query）等批量操作（bulk operation）。这不仅是为了处理大吞吐量数据，更是“有序索引”蕴含的功能性需求：例如“取出某点之后的所有数据”这样的操作是无法通过单点查询实现的，这个询问的本质是在不知道数据库中存在哪些键的情况下，要求数据库将某段数据排序。

在大多数情况下，每个仓库只连接一个生产者和一个消费者。但某些特殊计算节点，例如广播节点，会将同一份数据发给多个消费者，因此其仓库会连接多个消费者。无论对于哪种场景，键值数据库必须引入合适的并行访问控制机制，因为生产者和消费者可能会同时访问仓库。

综上所述，该应用场景对键值数据库有如下功能性需求：

(fr.1) 在键上建立有序索引；

(fr.2) 支持批量插入、区域查询、区域删除；

(fr.3) 提供合适的并发访问控制机制，支持一个生产者与多个消费者同时访问数据库，但该用例并不需要数据库提供事务处理（transaction）。

相对于上述功能性需求，该应用场景对键值数据库的非功能性需求更为苛刻：

(pr.1) 高吞吐量、低延迟，由于 Grape 是一个增量计算的实时系统，并需要处理海量数据，键值数据库必须提供快速的数据插入和数据访问；

(pr.2) 内存使用控制，正如大多数后台服务一样，数据库服务也需要降低系统资源占用。数据库是输入输出密集型的，必须小心地利用有限的内存处理流经的海量数据。如果数据库服务占用了太多内存空间，则会影响计算节点的效率。

该应用场景只要求数据库提供最小的功能集，却对数据库的性能有着苛刻的要求，所以我们实际是在“牺牲”复杂的功能（如事务管理）换取性能的提升。这正是专一型数据库与通用型数据库的重大理念差异：通过舍弃特定应用场景不需要的功能，往往能提高数据库的性能。

2.3 本章小结

本章以 Grape 系统为例，介绍了“生产者-消费者仓库”应用场景，指出了流计算与传统计算在数据传输方面的重大区别和新的挑战：流计算需要更频繁的数据传输，并要求更低的传输延时。

随后，本章通过 Grape 系统在正常运行、异常处理时的行为，分析了键值数据库在该系统中的三个主要功能：速率匹配、数据备份和索引排序。根据这些功能，我们明确地写出了该应用场景对数据库的三个功能性需求和两个非功能性需求。后文中的数据访问模式分析、索引数据结构设计，以及整个数据库系统的设计和优化，都以实现这些需求为目的。

第三章　数据访问模式与特征

在上一章我们已经揭示了该应用场景中的第一个数据访问特征，即数据库不需要提供复杂的事务处理机制、多生产者并发写入控制。舍弃事务处理不仅可以带来数据库性能的提高，同时也简化了数据库本身的复杂程度。并发写入控制则是容易被忽略的一点：相比于单一写入并发读取的模式，维护并发写入的一致性对数据库系统是很大的负担。

该应用场景中最重要的数据访问模式是：数据的插入是**几乎有序**的。在正常运行时，生产者按顺序输出增量数据，数据库不会遇到乱序的数据；只有当网络异常、生产者内部错误（丢失数据）、故障恢复或系统负荷过高时，才会在局部产生乱序数据。尽管故障在整个计算集群中是常见的，针对一个特定的计算节点来说却是较少见的情况，所以对每个仓库来说，乱序数据的比例是很少的，整个数据插入过程是几乎有序的。实验数据表明，Grape 系统向仓库插入的数据中，乱序数据的比例小于千分之一。

需要注意的一点是，Grape 系统通过管道和 TCP 连接进行数据传输，而其他一些分布式系统会采用 UDP 收发数据。相比于 TCP 连接，UDP 传输不能保证数据的有序性，所以会引入更多的乱序数据（尤其是网络压力较大时），针对这些场景，需要首先测量乱序数据的比例，再决定能否使用 FBTS，关于 FBTS 对乱序访问的容忍能力请参阅 6.7 节。

在 Grape 系统中，不仅数据插入是按顺序进行的，数据的读取和删除也严格遵循“几乎有序”的模式。数据读取的有序性是显而易见的，因为消费者在正常运行时会按顺序索取输入数据。然而“几乎有序的删除”却更加隐蔽：数据库延长数据的生存周期以防止数据丢失或下游节点崩溃，一个数据的生存周期从他被插入到数据库开始，直至下游节点的修复工作不再需要此数据参与，所以“几乎有序的删除”需要数据的后效性也是单调，即先到数据对计算的影响比后续数据的影响更早结束。幸运的是，在大多数流处理工作中（例如窗口操作、Join 操作，参见[11][12]），数据的后效性的确是单调的，所以大部分删除操作也是几乎有序的。

图 3-1 总结了上述“几乎有序”的访问模式：在数据轴上有三个“几乎”单调后移的的指针，第一个指针代表了生产者将新数据插入到数据库的边界，第二个指针代表了消费者从数据库中获取数据的边界，第三个指针则代表了数据后效性的终止边界，亦即数据从数据库中删除的位置。只有在极少数情况下，指针会前后跳跃，对应了少量乱序操作。

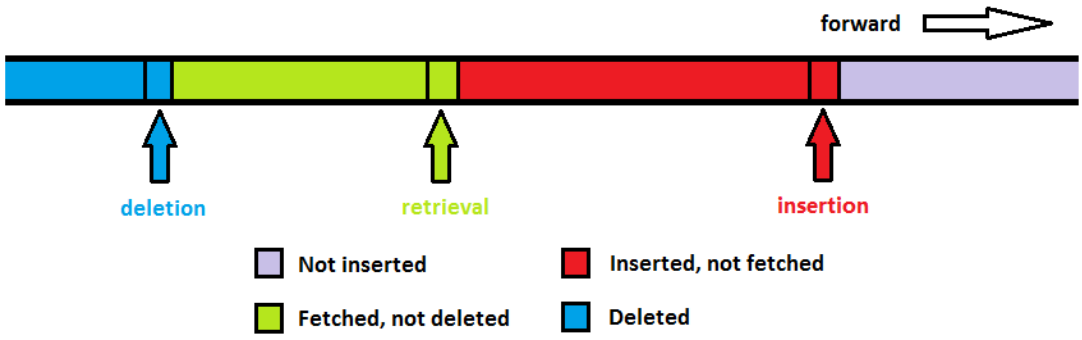

图 3-1　几乎有序的访问模式

第四章　索引数据结构的设计

本章将展示我为“生产者-消费者仓库”这一应用场景设计的新型索引数据结构——Flat BTree（FBTree），并分析为什么 FBTree 比传统的索引数据结构（如 BTree、哈希表）更适合该应用场景。

4.1　为什么不使用 BTree、线性表和哈希表

BTree、线性表、哈希表都是在键值数据库设计中常用的索引结构，但他们都不能胜任该应用场景，他们中的一部分不能提供足够的功能，另一些则因为没有利用前述的数据访问模式而无法满足性能需求。

哈希表无法满足功能性需求(fr.1)(fr.2)，因为它不是个有序的索引，也无法支持区域查询。线性表将数据按照线性序存储，可以提供高效的区域查询，事实上，如果数据的插入是完全有序的，线性表是性能最高的数据结构（O(1)的插入时间，O(logn)的快速检索）。但是线性表却不能容忍哪怕极少量的乱序数据，考虑以下例子：插入 1,2,3 后，一个乱序数据 1000 被插入，则随后的每次插入操作都会导致 1000 的移动，如果插入点前方有不只一个乱序数据，数据移动的代价会更高。如果使用链表实现线性表，虽然避免了乱序数据的频繁移动，却无法实现快速检索，因为在链表中进行数据检索的时间开销是线性的。线性表的其他变种，都是在插入速度与检索速度之间进行平衡，如块状链表的插入时间开销和检索时间开销都是 O(sqrt(n))，并不令人满意。

平衡二叉树（Balanced Search Tree）及其变种 BTree[17]从根本上解决了这一问题：BTree 可以对任意的访问模式提供 O(logn)的快速插入、快速检索，正因如此，大多数键值数据库都采用 BTree 进行索引。但 BTree 却并不擅长处理 “几乎有序”的访问模式，尽管插入和检索的时间复杂度仍是 O(logn)，原因如下文所述。

一棵分支系数为 k 的 BTree 要求每个节点至少包含 k/2 个数据，以保证可以在 logn 级别的高度内存储 n 个数据。为满足这一性质，BTree 采用了“平衡分裂策略”（Balanced Split Strategy，BSS），该策略将一个全满节点分裂为两个半满节点。以完全升序的插入为例，由于平衡分裂的影响，当插入操作完成后，我们得到的是一棵半满的 BTree：每个节点都仅有 k/2 个数据，这恰是 BTree 约束条件下的最坏情况！半满 BTree 的高度增长比我们的预期更快，更严重的是，它浪费了一半的磁盘空间，而且每个分裂操作都要把 k/2 个数据从一个节点移动到另一个节点，其代价是十分高昂的。

从 BTree 中删除数据也会触发其重平衡操作，例如数据移动和节点合并，这些操作的代价也是十分高昂的。但从几乎有序的删除模式来看，这样的重平衡意义不大，因为一旦某节点内的数据开始被删除，整个节点将很快被完全删除，没有必要时刻保证其数据量下限。这些都是 BTree 不适应“几乎有序”数据访问模式的例子。

4.2　FBTree：Flat BTree

哈希表、线性表和 BTree 的教训说明，我们必须专门为“几乎有序”的数据访问模式设计一种专门的数据结构，这种数据结构要支持批量操作，对几乎有序的数据操作提供类似线性表的性能，同时还能容忍一定数量的乱序操作。FBTree 正是这样一种数据结构。

FBTree 与 BTree 很像，甚至可以看作一种“简化”的 BTree，这也是其名字的来源：“Flat”指音乐中的降调号，FBTree 就是一种削弱了 BTree 约束条件的搜索树。但这种“简化”却很好的适应了几乎有序的数据访问模式，同时还能容忍一定数量的乱序访问，所以 FBTree 也可以称为“Flexible BTree”。

一棵 k-分叉 FBTree 的非叶节点（或称为内部节点）包含不超过 k 个孩子，叶节点包含不超过 k-1 个数据。在非叶节点上，每个指向子树的指针都附带一个区间，表示该子树中数据的键范围。各个子树的区间从左到右连续、递增且不相交，孩子节点的区间是其父节点区间的子区间。

在 FBTree 上进行检索的过程与 BTree 完全相同，并且他们的插入、删除操作都是类似的，但 FBTree 与 BTree 也有如下重大差异：

(1) BTree 要求每个节点拥有至少 k/2 个数据或子树，但 FBTree 没有此约束条件。

(2) 当数据插入操作使 BTree 中某个节点的宽度超过 k（或 k-1）时，会触发 BTree 的平衡分裂策略（BSS）：一个新节点被插入，旧节点中的数据被平均分配给两个节点，使他们都满足 BTree 的约束条件。在 FBTree 中，BSS 策略被“**适应性分裂策略**”（Adaptive Split Strategy, ASS）代替：**只有位于插入数据之后的数据（或子树）被移动到新节点中，其余数据全部留在旧节点中**。一个例子是，假设新数据从某节点的末尾溢出，则只有该新数据本身被移动到了新节点中，旧节点中原有的数据全部留在了原位置。

(3) 在 FBTree 中，数据删除不触发任何重平衡操作，FBTree 既不尝试从相邻节点移动数据，也不试图合并相邻节点，只有当一个节点因删除操作变成空节点时，才将该节点从 FBTree 中整体删除。

下面的例子展示了适应性分裂策略与传统策略的区别。图 4-1 中 BTree 和 FBTree 的分支系数都为 4，数据 1, 2, 4, 5, 7, 9 已经位于树中，当一个新数据 10 被插入时，BTree 将“5,7,9,10”分裂成了“5,7”和“9,10”，但 FBTree 却在新数据（10）处分裂，变为“5,7,9”和“10”。这种分裂方法的合理性建立在“几乎有序插入”的预期之上：如果插入是有序的，那么后续的数据都会比 10 大，不会有数据比 7 大比 9 小，所以 BTree 的平衡分裂实际上浪费了一个空间，而 FBTree 却没有这个问题，虽然现在看起来节点“10”的利用率很低，但我们可以预期后续的数据将其填满。

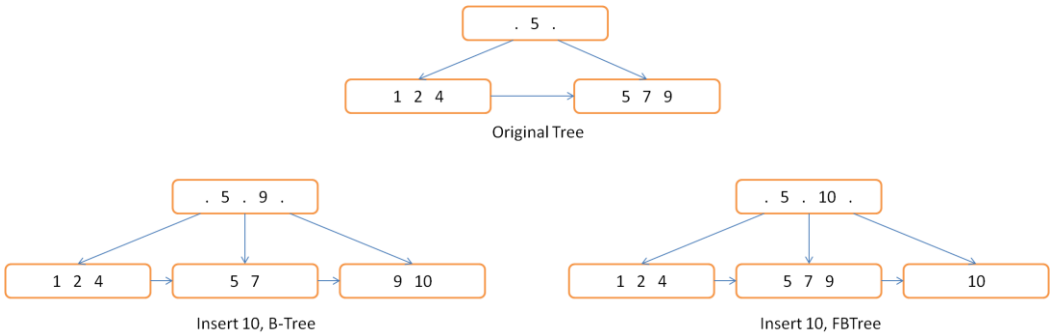

图 4-1 ASS 与 BSS 的区别，例 1

图 4-2 的例子则展示了对乱序插入应用适应性分裂策略的效果。假设数据 1, 2, 4, 5, 7, 9 已经存在，随后一个乱序数据 3 被插入，BTree 将“1,2,3,4”分裂为“1,2”和“3,4”，但 FBTree 却将其分裂为“1,2,3”和“4”。

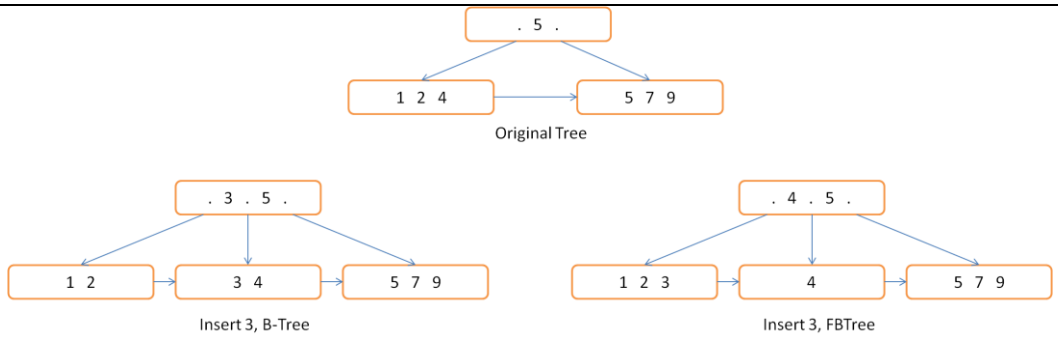

图 4-2　ASS 与 BSS 的区别，例 2

适应性分裂策略是否合理？如果数据插入是完全乱序的，则适应性分裂策略显然要比 BSS 差。最坏情况是数据完全倒序插入，ASS 会导致 FBTree 退化成为一个链表，空间利用率只有 1/k。但这并不意味着 ASS 任何情况下都比 BSS 差，尤其是在“几乎有序”的插入模式下！我们首先从完全有序的插入入手，图 4-3 的例子中，数据 1-10 顺序插入，正如前面提到的，我们得到了一颗半满的 BTree，最终树高为 3 层，但 FBTree 却是全满的，树高仅有 2 层。即：在完全有序的插入模式下，FBTree 几乎像线性表一样，性能要远优于 BTree。

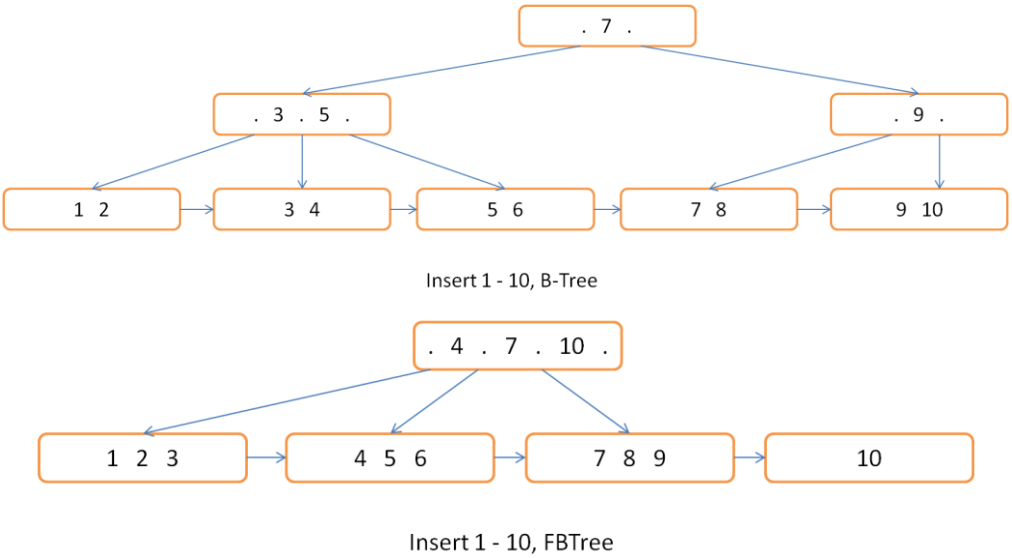

图 4-3　ASS 与 BSS 的区别，例 3

第二个例子则证明 ASS 可以容忍一定数量的乱序插入。在这个例子中我们将 FBTree 与线性表作比较，回忆在上节我们提到的例子：数据 1,2,3,4 插入后，有 2 个乱序数据 1000 和 1001 被提前插入，之后再回到 5 有序插入到 999。通过图 4-4 我们可以发现，在一次 ASS 分裂之后，乱序部分（1000 和 1001）与有序部分就被“隔离”到了不同的节点中，在随后的操作中，乱序部分既不会影响到有序部分的操作，也不再参与分裂操作，就像不存在一样。

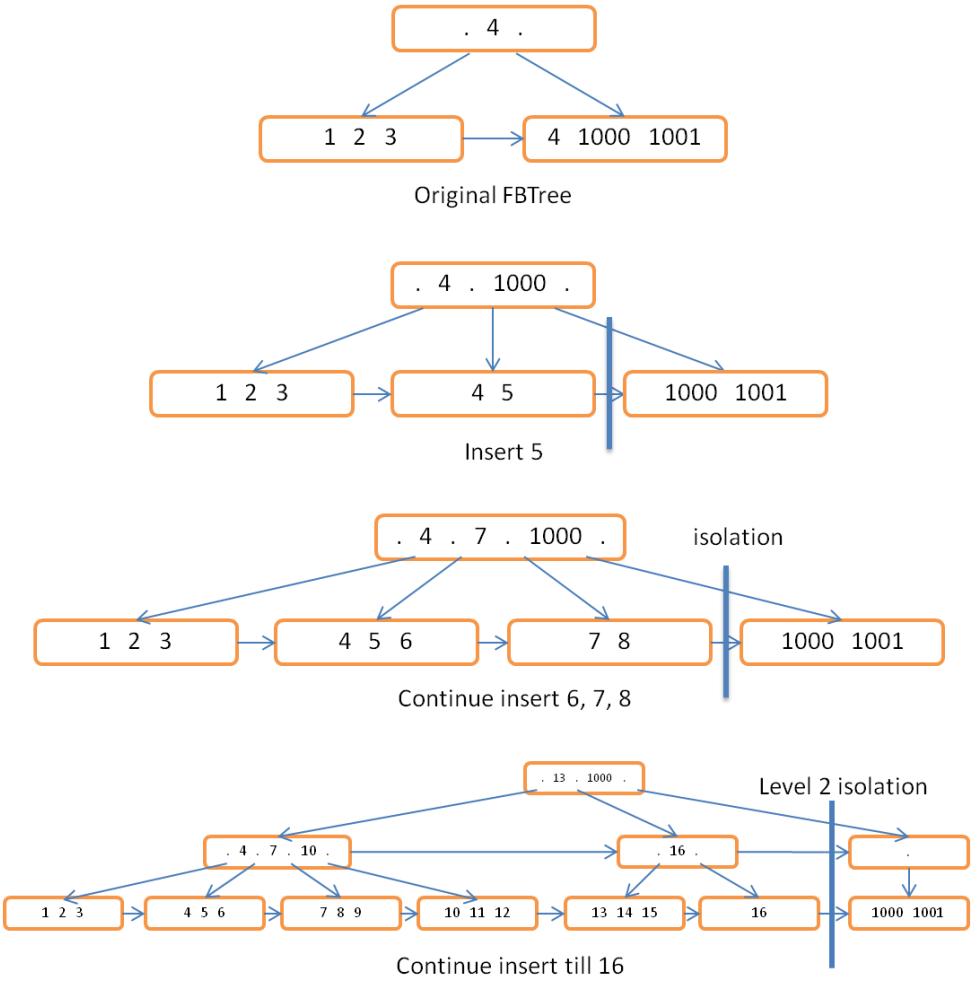

图 4-4　ASS 对乱序数据的隔离

上述两个例子都体现了适应性分裂策略的内涵：由于我们假设几乎升序的插入顺序，则新插入数据（或子树）的位置有极大的概率是“有序部分”的末尾，将来的“有序插入”理应从该位置向后延拓。所以，当插入操作触发节点分裂时：（1）如果插入数据位于树上某节点的末尾，则我们直接创建一个新节点将这个“有序部分”延拓下去；（2）如果插入数据位于某节点的中间，则说明这个节点中混有“有序部分”和“乱序部分”，并且该位置正是这两部分的分界线，所以我们在这个位置分裂，将“有序部分”和“乱序部分”隔离到两个不同的节点上。这种隔离首先发生在最底层的叶节点，随着新数据不断插入、节点分裂的位置越来越高，在更高层上也会实现类似的隔离。最终我们可以期望“有序部分”和“乱序部分”在各层（除根以外）都能通过节点边界自然隔离，而不会共存于某一节点内部。所谓的“适应性”，正是指这一点。

4.3　相关设计：Oracle 的 90-10 分裂策略

一些文献，如[18]，介绍了 Oracle 数据库针对有序插入的优化。当 Oracle 数据库发现新插入的数据是索引中最大的一个，且此次插入引起了 BTree（最右）叶节点的分裂时，会采取 90-10 分裂的策略，与 FBTree 类似地将新插入的数据单独放到新节点中，旧数据不挪动位置。

Oracle 引入该策略的动机与我们类似，是为了避免 BSS 策略在升序插入的情况下浪费一

半的空间。但 90-10 策略和 FBTree 的策略仍有诸多不同：（1）90-10 策略的非平衡分裂只针对插入最大数据的操作，仅发生于 BTree 的最右叶节点的末尾，而 FBTree 的非平衡分裂则可发生于任何节点的任何位置；（2）90-10 策略与删除操作无关，BTree 在删除数据时仍需进行重平衡，而 FBTree 则完全抛弃了删除时的重平衡操作。FBTree 的 ASS 策略可以看作是 90-10 策略的扩展，使 FBTree 能适应含有少量乱序插入的模式。相比之下，Oracle 的策略在乱序插入下极易失效：图 4-4 便是一个明显的例子，假设一个很大的数据被预先插入，则随后的数据插入都不位于索引末端，也就不会触发 90-10 分裂策略。

事实上，90-10 分裂策略是针对完全升序插入模式的优化，而非针对几乎升序插入模式的优化。并不是 Oracle 数据库忽略了“几乎有序”的插入模式，而是它不敢、也无法针对这个模式进行有效的优化。从开发目的来看，Oracle 数据库是一个综合型数据库，需要应对各种应用场景下迥异的数据访问模式，所以它必须保证每个针对特定模式的优化都不会在其他场景下被触发（回忆 FBTree 在乱序插入或倒序插入时会发生严重的退化，我们便能理解这其中蕴含的巨大风险）。因此 Oracle 数据库只能针对某些可以被严格界定的数据访问模式进行优化，而不能针对难以严格界定的、边界模糊的模式进行优化，而“几乎有序”这一模式正是难以界定、边界模糊的，所以 Oracle 数据库无法针对这一模式进行优化。

而我们的数据库则不然，他是高度专用化的，有着明确的应用场景。在生产者-消费者仓库中，可以肯定不会出现严重的、长时间的乱序数据访问，因此，FBTree 和 FBTS 采取的各种优化，包括 ASS 策略、以及下一章提到的更多系统优化，都是有效的、安全的。

4.4 本章小结

索引数据结构是键值数据库的核心，使用不同索引结构的数据库，其提供的功能以及性能都不尽相同。哈希表、线性表、BTree 是三种常用的索引数据结构，本章首先分析了为何这三种索引结构都不适合生产者-消费者仓库的应用场景：哈希表无法提供有序索引；线性表在处理乱序插入时需要进行大量的数据移动；BTree 虽然可以适应各种数据访问模式，但“几乎有序”的插入模式恰是 BTree 的最坏情况，此时 BTree 会浪费一半的空间，并引起许多不必要的数据移动。

随后，本文提出了一个全新的索引数据结构 FBTree，该数据结构是本课题的第一个主要成果。FBTree 是与 BTree 相似的检索树，但不再对节点内数据的数量提出强制性的下限要求，采用适应性分裂策略（ASS）替换 BTree 的平衡分裂策略，并取消了数据删除时的重平衡操作。适应性分裂策略的内涵是：不断尝试将“有序部分”与“乱序部分”用节点的自然边界隔离，保证有序部分的插入、检索、删除效率。FBTree 是专门针对“几乎有序”数据访问模式设计的专一型索引数据结构，在该访问模式下的性能要优于 BTree。

本章最后比较了一个相关的设计，即 Oracle 的 90-10 分裂策略，90-10 策略可以看作是 ASS 策略的一个特例，只针对完全有序的数据插入模式进行优化。

第五章　数据库的设计、实现及优化

5.1　FBTS 综述

我设计并实现了基于 FBTree 索引结构的数据管理系统 FBTS（FBTree Storage）。FBTS 是专门用于生产者-消费者仓库的专一型键值数据库，在设计过程中，我进一步运用应用需求和数据访问模式对 FBTS 进行优化，使 FBTS 更加适应于该应用场景。FBTS 实现于.Net 平台，使用 C#语言完成，总代码量约 3200 行。

每个 FBTS 实例可以管理多张数据表，每个数据表都是一个键、值二元组的集合，可作为一个生产者-消费者仓库。FBTS 使用泛型实现，允许用户使用任意类型作为键、值，但一个 FBTS 实例下辖的所有表要有一致的键、值类型。用户需要提供键比较函数、键和值的序列化方法、反序列化方法。5.2 节描述了 FBTS 的参数配制方法，5.3 节提供了 FBTS 的 API 说明。

FBTS 中的每个表都由一棵 FBTree 实现，即：FBTS 不仅用 FBTree 进行索引，更直接使用 FBTree 组织和存储数据，大部分键值对都直接存放于 FBTree 的叶节点中（除了某些超长的键值对）。5.4 节提供了 FBTree 实现的细节以及几个重要的优化。

为了提高性能，FBTree 将近期访问过的节点从硬盘读入内存以加速访问，这片内存区域就是内存缓冲池。但由于非功能性需求(pr.2)要求 FBTS 能够明确地控制总内存使用量，所以多个表必须共享同一个内存缓冲池，并需要一个内存管理系统来管理缓存、协调内存使用。正因为如此，虽然一个 FBTS 管理的不同表在逻辑上是独立的，在物理上却是相关的。

由于 FBTree 的结构特征，分页管理（paging）是 FBTS 内存管理的天然解决方案。我们可以把 FBTree 的节点看作页，把磁盘看作虚拟内存，把内存池看作物理内存。正如计算机程序通过虚页号来访问一个页，FBTree 也通过节点的磁盘地址来访问一个节点。FBTS 的内存管理子系统维护一个磁盘地址到内存池地址的页表映射（page table），并通过这个页表进行地址翻译。如果目标节点的不在内存池中（页表中查无此项），则说明发生了缺页错误（page fault），我们需要在内存池中选择一个节点并将其写回硬盘，再从硬盘中读取目标节点并放入内存池。内存管理子系统的详细情况以及重要优化位于 5.5 节中。

成熟的数据库系统必须支持变长数据，更何况 FBTS 允许泛型，系统无法事先获悉数据长度的信息。一个简单的解决方案是把所有数据都放入 FBTree 的叶节点中，但这样会导致叶节点的大小在数据插入前不可预测，从而违背了(pr.2)对内存控制的需求；另外，即使是同一个叶节点，其大小也会因为插入、删除操作而发生改变，这给叶节点的磁盘布局带来了困难。FBTS 解决变长数据问题的大致思路是：（1）每个节点的大小在创建时即确定，使用溢出堆保存超长的键或值；（2）动态预测数据存放于堆中还是叶节点中的阈值。5.6 节论述了 FBTS 如何支持变长数据。

如何在磁盘上存储 FBTree 节点看似不是一个大问题，但是如果我们想重用磁盘空间，就会产生外部碎片（External Fragmentation）。页外碎片不但浪费了磁盘空间，更会引起磁盘寻道，降低数据库性能。5.7 节论述了 FBTS 的磁盘管理子系统如何管理磁盘布局、重用磁盘空间。

最后，5.8 节描述了 FBTS 的并发控制机制。

5.2 FBTS 的参数与配置

FBTStorage<K, V>是 FBTS 的主类，它是一个泛型，其中 K 表示键类型，V 表示值类型，用户创建 FBTStorage<K, V>时需要将参数与配置通过类 FBTConfig<K, V>传入。本节将描述其中几个重要的参数。

BranchFactor，即前文中的 FBTree 的分支系数 *k*。FBTree 中的节点将最多拥有 *BranchFactor* 个数据或子树。

MaxKeyLen 与 *MaxValueLen* 是 FBTree 的叶节点所能储存的最大键、值长度。如果一个键或值的长度超过本参数，则其必然会被存放于堆中，但这并不意味着长度小于此参数的键值一定会被存放于叶节点中。FBTS 并不将这两个参数直接应用于全局的每个叶节点，因为如果大部分键值的长度都比较短，单一的全局阈值会导致大量的页内碎片（或称为内部碎片，Internal Fragmentation）。事实上，FBTS 在每个叶节点被创建时动态地预测适用于该叶节点的局部阈值，而这里的两个参数就是局部阈值的上限。

NBlock，该参数指定 FBTS 的全局内存缓冲池大小，FBTS 的内存管理子系统将最多在内存池中缓冲 *NBlock* 个 FBTree 节点。这些缓存空间将被所有表共享。

在 5.1 节我们提到 FBTree 会把超长的键值存放于堆中，为了优化系统性能，我们还需要为堆中的数据建立一个缓存。FBTS 管辖的所有表共享一个全局堆，参数 *HeapBufferSize* 指定了堆缓存的大小。

有了上述参数，我们即可计算 FBTS 的最大内存开销。单个 FBTree 叶节点的最大可能大小为 *(MaxKeyLen + MaxValueLen) * BranchFactor*，而非叶节点的大小更小。FBTS 的内存池最多缓存 *NBlock* 个节点，再加上堆缓存的大小，我们得出 FBTS 的最大内存开销为：

*MemSize = (MaxKeyLen + MaxValueLen) * BranchFactor * NBlock + HeapBufferSize*

FBTConfig<K, V>中还包括其他重要的参数，例如键类型比较函数，键类型和值类型的序列化与反序列化方法。FBTS 仅在一个 FBTree 节点被换出到硬盘、换入到内存时才调用序列化与反序列化方法。但 FBTS 仍需要在数据被插入到数据库时就获悉数据的序列化长度，以便及时把超长键值存入堆中、控制内存使用，因此 FBTS 要求用户提供计算键值大小的方法。在大多数用例中，用户可以不经序列化而直接给出键值的长度，这将进一步提高 FBTS 的性能；反之，用户亦可以进行一次序列化并返回其序列化长度，这样做仍然是可以接受的。

5.3 应用程序界面（API）

FBTS 实现了下列应用程序界面：

(1) 创建、删除数据表，FBTS 管理的每一个数据表由一个全局唯一的名称来标示。

```
void CreateTable(string alias)

void RemoveTable(string alias)
```

(2) 向数据表中插入键值对，FBTS 提供了单点插入与批量插入两个接口。如同大多数键值数据库，FBTS 不允许在同一个数据表中包含两个相等的键（由用户提供的键比较函数判定），如果用户尝试插入一个已存在的键，系统会抛出异常。

```
void Put(string alias, Tuple<K, V> item)

void Put(string alias, IEnumerable<Tuple<K, V>> items)
```

(3) FBTS 还提供了一个“插入或更新”的原子操作。用户提供键、新值，以及一个更新函数。如果该键不存在，则新键值对会被插入；如果该键存在，系统会调用用户的更新函数更新该键值对的值。该功能不仅是一个语法糖衣（方便用户书写例如

++count[key]的代码），更使 FBTS 在不提供事务处理机制（transaction）的前提下，具备了处理一部分基本原子逻辑的能力。

需要注意的是，FBTS 仅要求键比较函数提供键空间上的一个全序，即使键比较函数判定两个键相等，并不意味着他们在逐成员比较的意义下相等，所以 FBTS 会在更新函数中向用户提供表中存在的键，以便用户可以利用其成员进行更新。

void PutOrUpdate(string alias, Tuple<K, V> item, Func<K, V, V> modify)

void PutOrUpdate(string alias, IEnumerable<Tuple<K, V>> items, Func<K, V, V> modify)

(4) 通过键获取键值对，FBTS 提供了单点查询和区域查询。在区域查询时，用户还可以提供一个参数约定最多返回多少个键值对，这可以帮助用户程序进行内存控制，防止由于区间中包含过多键值对而内存溢出。

Tuple<K, V> Get(string alias, K key)

IEnumerable<Tuple<K, V>> Get(string alias, K stPos, K edPos)

IEnumerable<Tuple<K, V>> Get(string alias, K stPos, K edPos, int maxNumberOfItems)

(5) 通过键删除键值对，FBTS 支持单点删除和区域删除。

void Delete(string alias, K key)

void Delete(string alias, K stPos, K edPos)

5.4 FBTree 逻辑

FBTree 的逻辑分为树逻辑与节点逻辑。树逻辑实现了检索树的检索逻辑，节点部分则实现了 FBTree 的节点内部逻辑，包括 ASS 节点分裂策略、节点内部检索等。树逻辑部分与传统的检索树、BTree 没有太大区别，也没有太多优化空间，下面着重描述 FBTree 的节点逻辑。

5.4.1 节点的序列化布局和结构

与 BTree 相同，FBTree 有两类节点：非叶节点（内部节点）和叶节点，他们都继承自基类 FBTreeNode<K, V>。基类实现了两类节点的共同逻辑，包括元数据的布局结构、序列化到磁盘以及从磁盘反序列化回内存的逻辑。每个 FBTree 节点都对应了硬盘文件中一段连续的空间，并由其磁盘地址（距离文件头的偏移量）唯一标示。一个节点所能容纳的最大键值长度在该节点被创立时即固定，所以节点的总大小也在创立时确定。

head	Item 1			Item 2			.....	Item k		
head	ptr_1	len_1	key_1	ptr_2	len_2	key_2	.....	ptr_k	len_k	key_k
32	8	4	S	8	4	S		8	4	S

图 5-1 非叶节点的布局结构，第一行是粗略结构，中间一行展示了精细结构，第三行则标出了每个部分的长度

图 5-1 展示了一个分支系数为 k、最大键长为 S 的非叶节点序列化后的布局和结构。在节点首部有一个 32 字节长的元数据区域，该区域的内容由基类 FBTreeNode<K, V>定义。节点的剩余部分则包含了 k 个内部数据（非叶节点数据）储存空间。每个内部数据包含一个 8 字节长、指向其孩子的指针 ptr，4 字节长的键长度信息 len，以及 S 字节的键 key。如果键长 len 不超过 S，则 key 中存储的即是序列化后的键；反之 key 位置只是一个堆指针，真正的键被存储于溢出堆中。

根据搜索树的定义，在以此非叶节点为根的子树中，所有键小于 key_i 但大于等于 key_{i-1} 的键值对都位于 ptr_i 指向的子树中。特殊地，ptr_1 指向的子树中包含了所有小于 key_1 的键值对，而最后一个内部数据（不一定是第 k 个数据，因为该节点不一定是满的）的 key 和 len 是空的。

head	Item 1				.....	Item k			
head	$klen_1$	key_1	$vlen_1$	$value_1$	.....	$klen_k$	key_k	$vlen_k$	$value_k$
32	4	S	4	T		4	S	4	T

图 5-2　叶节点的布局结构，第一行是粗略结构，中间一行展示了精细结构，第三行则标出了每个部分的长度。

类似地，图 5-2 展示了一个分支系数为 k、最大键长为 S、最大值长为 T 的叶节点序列化后的布局和结构。与内部数据相比，叶数据包含了一个完整的键值对及其键、值长度。BTree 的叶节点包含一个指向其后继节点的后继指针，FBTree 也继承了这一结构以加速区域查询。按照 FBTree 的严格定义，后继指针需要占据一个键值对的位置，从而使叶节点只能容纳 k-1 个键值对，但我们在实现时将后继指针存储于元数据区域，使叶节点也能容纳 k 个键值对，这与 4.2 节有细微的区别。

元数据区域包含了节点类型（叶节点或内部节点）、最大键长 S、最大值长 T、前驱指针、后继指针等信息，还包含了该节点实际存储的数据数量。

5.4.2　优化节点的内部逻辑

为了高效的实现插入、删除、检索等逻辑，FBTreeNode <K, V>类（及其两个子类）做了许多优化。

首先，FBTreeNode<K, V>使用链表保存用户数据（或子树指针），这样可以在 O(1)的时间内实现新数据（或子树）的插入、删除，而不需要移动已有数据。但是在长度为 k 的链表中进行检索的时间复杂度为 O(k)，而 FBTree 在插入、删除、获取键值对时都需要在数个节点内部进行检索，因此简单地线性检索会严重影响 FBTree 的性能。为此，FBTS 利用几乎有序的数据访问模式，针对数据插入、数据删除两种场景，分别优化节点内部检索操作，使他们的期望时间复杂度降低为 O(1)。

以数据插入为例，由于我们预期大部分插入操作都是增序的，在 FBTree 上检索插入位置的过程是有规律的：根据 4.2 节末尾提到的 ASS 内涵，在“有序部分”与“乱序部分”已经隔离的层次上，新数据都倾向于沿最右子树向右下走，直至插入到某个叶节点的最后端。类似地，我们也可以期待大部分删除操作的检索过程是沿最左子树向左下走的。鉴于此，对于数据插入的场景，FBTree 的节点总是从后向前在节点内检索插入位置；在执行删除操作时，总是从前向后检索删除位置。这样我们便加快了插入、删除操作的节点内部检索速度，提高了数据库性。

对于几乎有序的读取操作，两个检索方向都无法将节点内部检索的期望时间复杂度降低为 O(1)，除非用户在读取数据后立即将其删除。一个未被 FBTS 采用的、更为激进的优化策略可能对提高性能有所帮助：每个节点都记录最近一次由 FBTree 读取操作触发的节点内部检索的位置，并从这个位置继续未来的内部检索。如果只有一个消费者，并依照几乎有序的模式读取这个数据表，这个优化可以将内部检索的时间复杂度由 O(k)降低为 O(1)；但如果有多个消费者同时读取这个数据表，他们的读取位置会被混淆导致优化失败。由于 FBTS 没有提供显式的事务处理功能，系统无法区分多个消费者的身份。

5.5 内存管理子系统

内存管理子系统的主要功能是为 FBTS 提供节点缓存：利用数据的时空本地性，在内存中缓存部分 FBTree 节点，从而加速对节点的访问、修改速度。所有 FBTree 共享同一个内存管理子系统，因此该系统被设计为全体 FBTree 节点的统一拥有者和统一管理者：FBTree 不直接拥有其下辖节点，也不能直接访问其节点，而是通过表名、节点指针（磁盘位置）向内存管理子系统申请节点的访问权限。内存管理子系统向 FBTree 提供了 3 个主要接口：以特定权限访问已存在的节点，申请创建一个新节点，以及删除某个节点。

同时，FBTS 的并发访问控制机制也主要实现在内存管理子系统中，一旦 FBTree 通过该系统以读/写权限打开某节点，就不需要担心在该节点上的并发访问问题，即：独占的写模式，共享但排斥写访问的读模式。我把内存管理子系统本身的逻辑与并发访问控制的逻辑剥离开来，本节主要展示内存管理的逻辑，所以下面的讨论都不考虑并发访问的情况，5.8 节会详细分析 FBTS 的并发访问控制机制。

5.5.1 虚页、物理页，磁盘与内存的一致性

内存管理子系统以分页的方式管理内存。5.4 节中的 FBTree 节点基类 FBTreeNode<K, V>即虚页，二元组<表名，磁盘位置>即虚地址。类 Block<K, V>则代表内存池中的物理页，FBTS 的内存池共有 *NBlock* 个 Block<K, V>对象，代表了 *NBlock* 个物理页。Block<K, V>中有一个指向 FBTreeNode<K, V>类型的指针 content，即是物理页与虚页的挂载点。当一个 FBTree 节点被换入某物理页时，系统从磁盘读出该节点的数据，反序列化为 FBTreeNode<K, V>的某个子类对象（内部节点或叶节点），并将该对象挂载到物理页的 content 指针上。相反地，当一个物理页上缓存的节点被换出时，该物理页 content 指针上挂载的 FBTreeNode<K, V>对象被序列化、写入硬盘，content 指针指向 null 从而销毁挂载的 FBTree 节点。

当一个虚页被缓存在内存池中后，该页面便有了两个版本：硬盘中的版本以及内存池中的版本，我们必须维护这两个版本的一致性。由于 FBTree 只对内存池中缓存的页面进行修改和更新，内存中的版本总是比硬盘中的版本更新（或相同），所以我们需要在 FBTree 写缓存后，从内存向硬盘同步数据。有两种基本的同步策略：穿透写策略（write-through）和回写策略（write-back）。穿透写策略将每次 FBTree 对缓存页面的更改同步地写回硬盘，保证内存与硬盘时刻一致；而回写策略则只更新缓存于内存池的页面，允许内存与硬盘的暂时不一致，当缓存页面被换出时，再一并将更新写回硬盘。两种策略各有优劣：穿透写策略易于实现，且能使硬盘尽早工作，充分利用其闲置时间，但是硬盘操作频繁、琐碎，增加了寻道概率，最重要的是，每次写内存都要与硬盘的同步，使缓存直接受硬盘性能的影响；回写策略写硬盘的次数少、读写范围连续，不易引起寻道，由于写内存与写硬盘异步进行，缓存受硬盘性能的直接影响较小，降低了缓存操作的延时，但是在不发生换页时硬盘一直处于闲置状态，浪费了其潜在能力。

FBTS 采用了穿透写与回写相结合的方案。该方案以回写策略为蓝本，写内存与写硬盘异步进行，不要求内存与硬盘的时刻一致，降低缓存操作的延时；但并非在换页时才进行同步，而是维护一个后台同步线程，按一定时机扫描内存池，对不一致的页面执行同步操作，这样既能减少写硬盘的频率，又能尽量提前写硬盘的时机，充分利用硬盘的潜力。为此，FBTS 为每个物理页维护一个状态机，包括 Clean, Dirty, Reading, Writing 四个状态，其状态转移如图 5-3 所示。

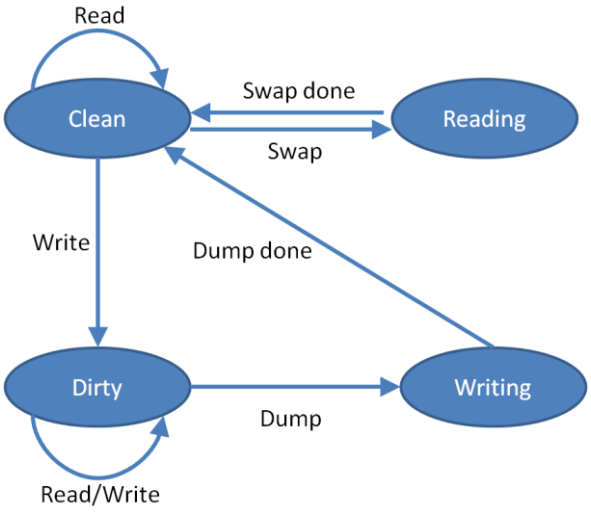

图 5-3　FBTS 的物理页状态机

Clean 状态即表示该物理页所缓存的数据与硬盘一致，Dirty 状态则表示该物理页与硬盘不一致，需要进行同步，Writing 状态则表示该物理页正在进行同步。FBTree 对一致页面进行读操作后，该页面仍然是一致的；但当 FBTree 对一致页面进行写操作后，该页面的内容便不再与硬盘一致了，从而进入 Dirty 状态；对 Dirty 状态的页面进行更多读写操作，它仍然是不一致的。后台的同步进程定期扫描 Dirty 页面并进行同步，一旦同步开始，该页面便进入 Writing 状态，直至同步结束后，该页面便回到了一致的 Clean 状态。图中的 Reading 状态及其与 Clean 状态的转化是换页的范畴，在此处先不讨论。

后台同步线程选择什么时机进行一轮同步，是影响 FBTS 性能的重要因素。如果同步过于频繁，则我们的同步策略会退化成为穿透写策略；如果同步频率太低，则会使大量不一致页面堆积在内存池中，不仅无法充分利用磁盘的空闲时间，还会影响数据库的性能。我们的期望是：每一轮同步都有一定数量的不一致页面，这样可以保证每轮同步都积累了足够数量的更新，避免频繁同步；但这个数量又不能过多，否则就说明同步不及时，可能会影响系统性能。由于单位时间内产生的不一致页面数量取决于数据插入、删除的速度，同步频率不能静态指定，必须动态预测。考虑到该应用场景中数据插入、删除频繁，我们认为每轮同步内存池中 20%的页面是较合理的频率。同步线程根据距上轮同步的时间间隔、新增的不一致页面数，估算当前产生不一致页面的速率，并按总物理页数 20%的参考目标估算下次同步的时间。但同步间隔不应大于 1 秒，否则一旦遇到瞬间爆发的数据流，系统的响应会滞后；另外同步间隔也不能过小（如 10ms），因为这已接近线程休眠的精度极限。如果某轮同步时的不一致页面数量过少（如小于 5%总物理页数），同步线程也会放弃此轮同步，等待 FBTree 积累更多数据更新。

5.5.2　地址翻译与换页算法

上一节展示了内存管理系统的基本设计，以及如何维护内存与硬盘的一致性，本节将展示如何通过地址翻译满足 FBTree 对下辖节点的访问请求，以及换页算法的细节。

分页内存管理的核心是维护一个页表进行虚实地址翻译，在 FBTS 中，页表即一个将<表名，磁盘位置>映射到物理页 Block<K, V>的映射。页表由 c#的词典（Dictionary）维护，该数据结构通过哈希的方法，可以在常数时间内完成插入、删除、查询等操作。当内存管理系统收到 FBTree 对其下辖节点的访问请求时，通过其虚页号在页表中进行查询。如果查询成功，则说明该虚页已被缓存在内存池中，只需通过对应物理页的挂载指针取出该虚页返回给

FBTree 即可。如果查询失败，则说明该虚页尚未被缓存，发生了缺页错误（page fault）。在这种情况下，内存管理系统需要首先将目标页从硬盘中换入内存池，再将其返回给 FBTree 使用，这个过程即换页（page swap）。

换页的整体算法在上节开头已有描述，即：寻找被替换的“受害者”、将受害者换出内存池、从硬盘读出目标页并反序列化成 FBTreeNode<K, V>结构、将目标页挂载到相应的物理页上。其核心在于前两步，即选择谁为受害者，以及如何将受害者换出。

将受害者换出内存池要保证内存与硬盘的一致性，所以“如何换出”与上节提到的同步策略有关。对于穿透写策略，内存与硬盘时刻一致，只需要将受害者从内存抛弃即可；对于回写策略，则需要检查受害者的内存版本是否与硬盘版本一致，并在必要时进行同步，方可将受害者抛弃。而 FBTS 的策略是在后台持续性、周期性地同步，所以我们也不需要在换页时进行同步，只需要在选择受害者时只选择 clean 页面作为受害者，并将其抛弃即可。

如何选择换页的受害者，即换页策略（page replacement algorithm），是一个已在操作系统领域深入研究过的问题[19][20]。常见的换页策略有：随机换页、先入先出、时钟换页、最近最少使用（LRU/NRU）等。LRU 策略虽然效果较好，但计算开销大；大部分分页系统都采取先入先出、时钟换页等较为简单的策略，尤其是时钟换页策略作为“二次机会先入先出”策略的简易实现被广泛应用。

操作系统的内存管理直接与硬件打交道，其“物理页”是硬件概念，而目前的硬件仅为每个物理页提供了 1bit 的使用标记，因此在仅依赖硬件的条件下，换页算法不能确切获得每个物理页的近期使用情况，只能按使用标记为每个物理页提供一次不被换出的机会。而数据库系统不与硬件打交道，其“物理页”是抽象的概念，我们就可以为每个物理页维护详细的近期使用情况，提高换页质量。FBTS 使用了一种改进的时钟换页策略：内存池中的每个物理页维护一个近期使用得分，该得分位于 0-100 之间，页面每次被访问，该得分加 a。内存管理系统维护一个全局的时钟指针，每次换页检查时钟指向的物理页，如该页处于 dirty 状态，表明该页暂时处于不一致状态而不能被换出，时钟指针后移；如该页处于 clean 状态，且得分为 s，则有(100-s)%的概率被选为受害者，如该页没有成为受害者，则其得分减 b，时钟指针后移检查下一个物理页，直至找到一个受害者。由于后台同步线程不断将 dirty 页同步为 clean 页，所以受害者一定能被找到。

5.5.3 页表缓存（TLB）

虽然 FBTS 的页表由 C#的 Dictionary 实现，其检索的平均时间复杂度为 O(1)，但在高吞吐量的情况下，仍会影响数据库的整体效率。假设 FBTree 的高度为 h，则平均一次数据插入、删除或检索都会触发 h 次（甚至更多）虚页访问，每次虚页访问都要查询页表，其开销是十分巨大的。

类似于操作系统，我们也可以引入页表缓存（TLB）来解决这个问题。TLB 是一个很小的缓存，存储最近用到的 10 条页表项。虽然只有 10 条页表项，由于数据访问的时空本地性，TLB 仍可命中大量虚地址翻译请求。在几乎有序的访问模式下，其命中率更为可观，在实验中达到了 99.7%的命中率。只有在 TLB 翻译失败的情况下，我们才会去检索页表，并将该页表项存入 TLB。当一个虚页被换出，或者被 FBTree 删除时，其页表项都会从页表中删除，这时要同步地更新 TLB，以保证 TLB 与页表的一致性。

操作系统的 TLB 是由硬件实现的，在检索 TLB 时，硬件电路可以保证所有缓存项被并发地检索，从而高效地完成虚地址翻译；而在 FBTS 中，我们的 TLB 是软件实现的，只能串行地检索每一条缓存项，但即便如此，其效率仍显著高于 Dictionary。实验表明，运用 TLB 后

FBTS 获得了接近 25%的性能提升。

5.6 堆与变长数据管理

FBTS 允许任意长的键、值，但这对数据库管理系统是个考验。如 5.1 所述，我们不能把所有的键、值都存放在 FBTree 中，这会使 FBTree 节点的大小的不可预期且动态变化，无法存储在硬盘中。一个 FBTree 节点的大小必须在其创立之时就确定，并在其整个生存周期都不改变，所以我们必须规定键、值长度的上限。这便带来了两个问题：（1）如何规定键值长度上限；（2）超长的键值存储在哪里。下面将依次给出这两个问题的解决方案。

5.6.1 键值长度预测

通过调整 FBTree 节点的键值长度的上限，我们可以平衡 FBTree 的空间利用率和数据访问的效率。键值长度的阈值较高时，大部分键值都直接储存在 FBTree 中，访问速度快，但节点较大，页内碎片多，空间利用率低；反之，则 FBTree 的空间利用率较高，但会有更多键值因超长而无法直接存储在 FBTree 中，影响数据访问效率。相对而言，数据访问的性能比空间利用率更重要一些，尤其是键，要尽量使之直接储存在 FBTree 中，否则每次需要该键参与大小比较时，都需要多一次间接访问。所以，一个“好的”阈值应该平衡这两方面，并可以向提升数据访问性能的方向稍作倾斜。

FBTree 并不使用一个全局的静态键值长度上限，而是在每个节点创立之时动态预测键值长度的上限。在更细的粒度上设置键值长度的阈值，可以更好地适应键值长度的局部趋势，有效减少页内碎片、提高空间利用率。

我们通过观察键值长度的“近期趋势”预测将来插入数据库的键值长度，这种对“近期趋势”的观测是每个表单独进行的：每个数据表维护一个大小为 *BranchFactor* 的滑动观察窗，通过窗口观察最近插入该表的 *BranchFactor* 个键值的长度，采用其平均值加 2 个标准差作为预测长度。假设键值的长度分布符合正态分布，则有 95%的数据落在两倍标准差的范围内，这样我们可以保证只有少量数据无法进入 FBTree，同时又不会因为过高的预测产生过多的页内碎片。

为了优先保证数据访问的性能，我们可以将上述预测再乘以一个安全系数（如 1.05），以增加进入 FBTree 的键值数量，保证数据访问的性能。为了控制 FBTS 使用的内存总量，局部的预测长度不能超过全局上限；预测长度也不能过低（小于 8 字节），因为我们需要至少 8 字节为超长键值对维护一个堆指针，见 5.3.2 节。

5.6.2 溢出堆

对于超过长度阈值的键值对，我们将其存放在溢出堆中。溢出堆没有更复杂的内部结构，仅依照数据到来的先后次序将他们存储在硬盘文件中，通过数据在文件中的位置作为获取该数据的指针，并将该指针存放于 FBTree 节点的数据区域中。由于数据在进入 FBTree 之后不会改变（Update 操作是使用新数据替代旧数据，并非直接改变旧数据），所以溢出堆可以将数据紧密堆放，而不需要担心将来可能会发生数据长度变化。

当 FBTree 访问超长键值时，必须多一次堆访问操作，如果仅将堆中的数据存储于硬盘中，则会严重影响数据库的效率。我们可以为堆设立缓存，利用数据访问的时空本地性来提高堆的性能。相比于 5.2 中复杂的 FBTree 节点内存缓冲池，堆缓存要简单许多：

(1) 因为要控制内存总量，我们也只设置一个全局堆缓存，其大小为 *HeapBufferSize*。

(2) 因为堆中数据不会更新，因此没有一致性问题。当数据被插入到堆中时，我们将其

放入堆缓存，并进行唯一一次穿透写操作即可保证一致性；当数据从堆缓存中换出时，无论它是从 FBTree 首次插入的数据还是从硬盘换入的已有数据，都可以直接抛弃。

(3) 由于 5.3.1 中合理设置的键值长度阈值，FBTS 对堆访问的频率很低（见实验 6.5），因此我们只需实现最简单的缓存替换策略（由于堆缓存不是分页管理的，不能称之为换页策略）。FBTS 的堆缓存采取先入先出的替换策略：当一个数据要进入堆缓存时，我们将堆缓存中最古老的数据换出。根据新插入数据的大小与换出数据的大小，可能需要换出多个旧数据，才能为新数据腾出空间。

5.7 硬盘文件管理与外部碎片控制

FBTree 和溢出堆都直接存放于文件中，但是 FBTree 的节点、溢出堆内的数据都会随着数据表的删除操作而被释放。从长期运行的角度考虑，我们不能一直通过增加文件大小的方式申请新空间，必须重新利用被释放的空间。

FBTS 对硬盘空间的申请和释放是以一个 FBTree 节点（或一条堆记录）为单位进行的，但磁盘空间回收和重用不能在这个粒度上进行，因为：（1）每个 FBTree 节点的大小都不相同，以 FBTree 节点的粒度重用空间会产生许多无法分配的外部碎片；（2）在此粒度上维护可重用空间表（free list）代价高昂——因为 FBTree 节点长度可变，我们需要类似区间树的数据结构，每当一个 FBTree 节点被删除，都要进行区间合并操作；（3）在细粒度上重用空间容易导致空间申请不连续，即逻辑上相邻的节点在硬盘上不相邻，这破坏了数据的空间本地性，会引起不必要的硬盘寻道操作。

因此，FBTS 在更大的粒度上进行磁盘空间的回收和重用，采取分区块重用磁盘空间的策略。FBTS 将文件分成长度固定的大区块（如每块 128MB），每个区块都可以存储许多 FBTree 节点，但 FBTree 节点必须完全位于区块内部，而不能跨越两个区块。当一个区块内的空间被完全释放后，这个区块才会被整体重用。

该策略易于实现：每个区块维护一个使用计数器，当一个 FBTree 节点被分配存储在该区块时计数器加 1，当区块内部的某个节点被删除时计数器减 1，计数器为 0 时即说明该区块内的所有 FBTree 节点都已被删除，该区块可以被重用。每个文件还有一个指针记录当前正在使用哪个区块，一旦开始使用某区块，即将该区块内的空间从头至尾逐次分配，直至该区块被写满（或区块内的剩余空间小于某次申请的空间大小）。当前区块被写满后，系统即检索是否有可以重用的区块，如果没有区块可以被重用，则需要在文件末尾建立一个新区块。

5.8 并发访问控制

FBTS 的并发访问控制机制不仅要把每个数据表的生产者与消费者隔离，由于 FBTS 本身的特点，还需要控制用户线程与内存池的后台同步线程之间的并发访问。另外由于所有数据表共享一个内存池，不同表的工作线程、同一个表的多个消费者线程都会引起读写-换页问题和并发换页问题。本节将针对上述四个主要的并发控制问题，展示并发访问控制系统的设计方案。

5.8.1 主要的锁

FBTS 系统主要有三类锁：每个表有一个全表的读写锁，每个物理页有一个页锁，以及一把全局的换页锁，我们用前文提到的四个并发控制问题来解释这三把锁分别的作用。

隔离同一个表的消费者与生产者是最简单的并发控制问题，只需要在每个表上放置一把

全局读写锁即可。

后台同步线程的存在会引起数据竞争和不一致问题。例如当生产者线程修改某叶节点时，如果后台同步线程正在将该节点同步到硬盘，则会发生竞争和数据不一致：用户此次对叶节点的修改可能未被同步到硬盘上，而同步结束后该节点的状态却置为 clean，这导致节点的状态与内存、硬盘的版本关系不一致，一旦该节点随后被选为换页的受害者，则用户此次的修改将丢失。

读写-换页问题与上面的问题类似，例如：当表 A 的一个消费者线程 A1 读取某个 FBTree 节点时，表 A 的另一个消费者线程 A2（或另一个表的任意线程 B1）发生了缺页错误，而换页策略又恰好将 A1 正在读取的节点换出，则 A1 就会出错，甚至读到新换入节点的数据。

而上述两个问题的解决方案是一致的，就是为每个物理页引入一把锁。物理页锁不需要做到完全互斥，用一把读写锁即可满足需要。当 FBTree 以读权限打开某节点时，其所在的物理页上读锁，当以写权限打开时则上写锁。另外，当后台同步线程将某物理页同步到硬盘时，需要读取该页数据，但同时还要修改物理页的状态（dirty 到 clean），所以要上写锁；而当换页策略将某页选为受害者时，将其上写锁。这样就解决了上述两个并发访问问题：对于第一个问题，用户节点的修改是写操作，同步时读操作，双方互斥；而在第二个问题中，换页是写操作，与任何作用在此页的其他操作互斥，只要某线程持有了页面的读锁或写锁，则该页面必然不会被换出。

并发换页也会引起严重的错误，例如：双方可能会选中同一个换页受害者，或引起页表不一致，等等。为了解决并发换页的问题，我们设置一个全局的换页锁，这是一把完全互斥锁，保证任何两个换页操作不会同时发生。

5.8.2 页表与地址翻译的并发访问控制

5.5.2 节已经讨论了利用页表进行地址翻译、满足 FBTree 对下辖页面访问请求的过程，但其中并没有考虑多线程并发访问的问题。事实上，由于页表查询、给物理页加锁不是一个原子过程，再加上换页的存在，如果不对地址翻译和物理页获取的过程进行并发访问控制，将导致严重的错误。

在不考虑 TLB 的情况下，当 FBTree 对某节点提出访问请求后，内存管理子系统实际要进行两步操作：（1）查询页表，获取节点所在的物理页；（2）按照用户提出的访问权限给物理页加锁。这两步并不是原子操作，假设在（1）（2）之间有另一个用户线程切入，并造成了一次换页，则有可能将步骤（1）中已查到的物理页替换，这就会导致内存池返回错误的页面。虽然页锁可以保证页面在上锁后不会被换出，但并不能保证在地址翻译之后、获取页锁之前的空档发生竞争。

解决这个问题的一个简单方法是：在地址翻译之前获取换页锁，给物理页上锁之后再释放换页锁。如果是这样，则“换页锁”就成了“页表锁”。由于节点访问请求是十分频繁的，完全互斥的页表锁就会严重影响系统性能。将页表锁改为读写锁可以在一定程度上缓解这个问题，但获取读锁依旧有一定开销。从根本上讲，物理页 A 换页不应成为阻碍他人访问物理页 B 的理由，因此这种页表锁的设计是不合理的。

FBTS 采取双重检查锁定(double-checked lock)的方法来解决页表的并发访问控制[21]。具体来说，当 FBTree 申请访问某节点时，内存管理系统在不获取换页锁的情况下直接查询页表，如果翻译失败则直接进入缺页逻辑，如果翻译成功则按 FBTree 申请的访问权限给查询到的物理页加锁。当这个物理页被锁定后，他无法被换出，我们便可检查这个物理页内是否挂载了我们需要的节点。如果在前面两步之间没有发生竞争，则这个物理页内必然挂载了我

们所要的节点，将其返回给 FBTree 即可；如果在翻译与加锁之间发生了竞争，则这个物理页内挂载的可能不是我们需要的节点，此时我们将该页页锁释放，进入缺页逻辑。

需要注意的是，无论是翻译失败还是双重检查失败，进入缺页错误逻辑并不代表真正发生了缺页错误，因为在翻译与获取换页锁之间可能有其他线程将目标页面换入了物理内存。我们在获取换页锁后再次查询页表进行双重检查，判定是否真正发生了缺页。由于此时我们已经获得了换页锁，页表给出的查询结果一定是正确的。

该方法看似复杂，却巧妙避免了在地址翻译过程中获取页表锁，使地址翻译与换页操作、以及并发地址翻译可以完全并行。双重检查失败仅在地址翻译与换页操作按特定顺序交错进行时才会发生，属于十分罕见的情况，所以大部分翻译过程都不涉及换页锁，使整个逻辑接近于无页表锁的设计，这正体现了“make common fast, make rare correct”的计算机系统设计理念。

5.9 本章小结

本课题的第二大成果就是设计并实现了基于 FBTree 的键值数据库 FBTS。作为一个完整的键值数据库系统，仅有一个索引结构是远远不够的，还要解决诸如数据缓存、并发控制等一系列问题。本章首先提出了对 FBTree 的一些优化，尤其是通过合理选择检索方向提高节点内部检索的速度。随后，本章解决了系统设计中的四个大问题：内存缓冲设计、变长数据管理、硬盘空间重利用，以及并发访问控制。

FBTS 通过分页的方式管理内存缓冲。在内存-硬盘一致性控制方面，FBTS 没有直接采用传统的 write-through 策略或 write-back 策略，而是采用了两者相结合的后台同步策略，该策略的优势是：既可以防止重复、琐碎的细粒度同步，又能充分利用硬盘的空闲时间。在换页算法上，FBTS 采用了时钟换页的策略，并通过记录更精确的页面使用情况优化换页算法的性能。

为了有效管理变长数据，FBTS 引入了数据长度预测机制和溢出堆。FBTS 保证 FBTree 节点的大小在创立之初便固定下来，同时在创立新节点时动态预测数据长度，尽可能地平衡空间利用率和数据访问效率。

由于生产者-消费者仓库的数据存储具有一过性的特征，FBTS 必须回收利用磁盘空间。FBTS 采取分区块重用磁盘空间的策略，在较大的粒度上重用磁盘空间，这样可以有效降低磁盘管理的开销，同时保证空间重用、空间分配的连续性，提高磁盘访问的性能。

页表的并发访问控制是 FBTS 系统设计的又一亮点。通过 double-checked locking 策略，FBTS 基本实现了无锁的页表查询，避免页表翻译成为影响系统性能的瓶颈。

第六章　实验及分析

6.1　测试环境

以下所有实验都在同一个高性能商业集群中完成。该集群中的每台机器配备 Intel Xeon X3360 处理器，共有 4 核，工作频率 2.83GHz；内存大小为 8GB；操作系统为 64 位 Windows 7；硬盘则是 Dell Virtual Disk SCSI，这是一个由 SCSI 硬盘组成的 RAID 阵列，总容量 2TB。平均读速度 160MB/s，最高读速度 210MB/s，突发传输速率 118MB/s，实测写速度在 100MB/s 以上。

在所有测试中，我们固定 FBTS 的分叉系数为 100，内存池最多缓存 1024 个节点，每个节点的最大键长为 32 字节，最大值长为 5000 字节，得到内存缓冲池大小约为 500MB，而堆缓存大小也设置为 512MB，FBTS 的总内存占用不超过 1G。

键、值都是字节数组，键的序定义为从最高位到最低位依次比较的结果。

6.2　与 Berkeley DB 的性能比较

根据 Oracle 对 Berkeley DB 的说明文档[22]，Berkeley DB 在**无硬盘读写**（数据只在内存中缓存）、无事务处理的模式下，单点读取速度为每秒 1,000,000 次（1,000K/s），单点写入速度为每秒 447,628 次，区域读取速度为每秒 4,565,910 次。但上述惊人的性能是在无硬盘参与的情况下达到的，根据同一文档，Berkeley DB 在涉及硬盘操作、开启事务处理模块并支持 ACID Transaction 的条件下，写速度为仅为 45K/s。这些数据对本文的参考价值不大，因为：（1）上述数据在 06 年测得，与 FBTS 的测试环境差异较大；（2）上述数据是在插入键长 8 字节，值长 32 字节的环境中测出的，场景比较单一，只体现了键值长度较小、硬盘性能不成为瓶颈时的性能，未能体现 Berkeley DB 对硬盘性能的利用率；（3）该文档并没有明确提供 Berkeley DB 在涉及硬盘操作但不涉及 Transaction 的性能，与 FBTS 的可比性不强。因此，我在同一集群中重新测试了 Berkeley DB 在关闭 Transaction 情况下的性能，以便可以与 FBTS 进行公平的比较。

我们分别向 Berkeley DB 与 FBTS 插入 10,000,000 个键长为 8 字节，值长为 32 字节、64 字节、256 字节、1024 字节的键值对，比较两者的插入速度和硬盘速度，结果如表 6-1。试验中 Berkeley DB 的内存缓冲池为 1G，以便于 FBTS 的内存占用情况匹配。

表 6-1　Berkeley DB 与 FBTS 性能比较

Value Len (Byte)	BDB Insert SPD (K/s)	FBTS Insert SPD (K/s)	BDB Disk SPD (MB/s)	FBTS Disk SPD (MB/s)
32	65.97	200.87	2.06	6.28
64	65.44	198.10	4.09	12.38
256	43.38	172.52	10.84	43.13
1024	16.18	108.59	16.18	108.59

结果显示，在几乎有序的数据访问模式下，FBTS 的性能是 Berkeley DB 的 3 倍多，尤其需要注意 Berkeley DB 对硬盘性能的利用能力并不好，当键值长度较大、需要大量读写硬盘时，效率更差。通过这个比较试验，我们可以清楚地认识到（在特定应用场景下）专一型数据库对提升系统性能的重要价值。

6.3 键值长度对 FBTS 性能的影响

该测试反映不同键值长度对 FBTS 性能的影响。我们固定键长度为 8 字节（对应常见的 int64），分别将值长度设置为 32、64、128、192、256、…、2048 字节，按升序向一个数据表中插入 10,000,000 个不同的键值对，分别检测其插入速度（以 K/s 为单位，千次插入每秒）和硬盘速度（以 MB/s 为单位，兆字节每秒）。实验结果见表 6-2、图 6-1，注意图中的横轴为对数坐标。

表 6-2 不同长度键值对的插入速度

Value Len (Byte)	Insert SPD (K/s)	Disk SPD (MB/s)
32	205.53	6.42
64	199.51	12.47
128	189.41	23.68
192	181.55	34.04
256	170.32	42.58
384	153.29	57.48
512	139.76	69.88
640	129.26	80.79
768	120.81	90.61
896	112.31	98.27
1024	101.65	101.65
1536	73.94	110.91
2048	53.40	106.80

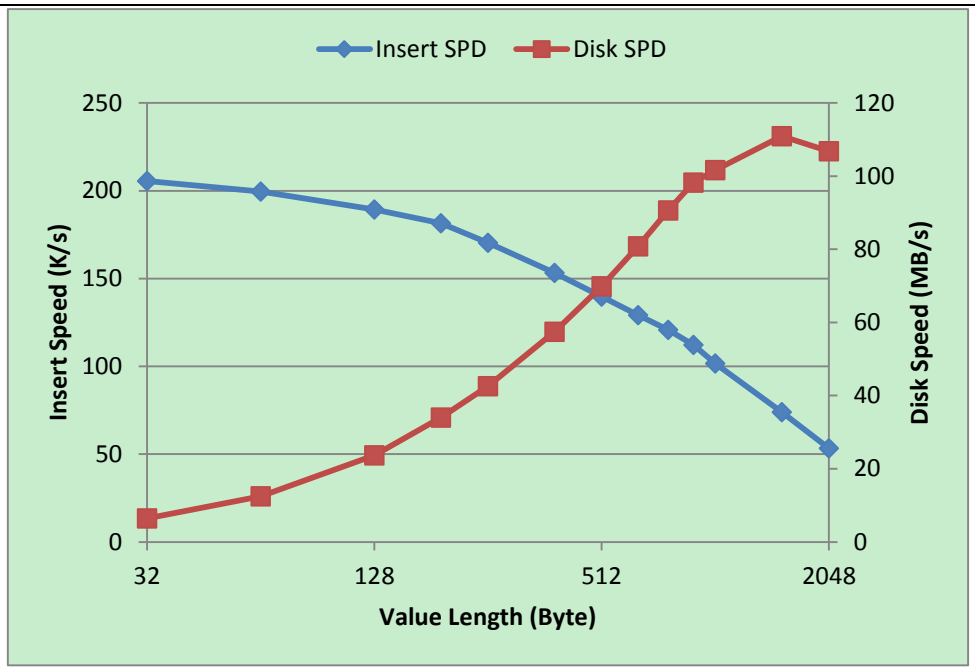

图 6-1　不同长度键值对的插入速度

通过该实验可以发现，有两个潜在瓶颈会制约 FBTS 的性能：一个是 FBTS 进行一次插入/访问/删除操作固有的时间复杂度，另一个则是硬盘的读写性能。当键值长度较小时，硬盘不成为制约系统性能的瓶颈，因此 FBTS 的最大吞吐量主要受 FBTree 操作速度的影响；当用户插入较大的键值对时，FBTS 的性能主要受磁盘读写速度的限制。

需要解释的一点是：该实验的实测数据与理论分析稍有差距。理论上讲，在硬盘速度成为瓶颈前，键值数据库的性能不应受值长的影响，而且第五章的设计也符合这一原则。因此在硬盘成为瓶颈前，增大值长，插入速度应维持不变，硬盘速度应正比于值长。而实测数据中，插入速度在值长增大时稍有下降，这与理论分析有一定差距。我通过更多实验深入分析了该现象背后的原因，有以下几点：

（1）　虽然数据库本身的逻辑与值长无关，但键值对却要实实在在地流经内存池，因此增大值长便增加了内存池的吞吐量，导致数据库的性能必然受到影响。最主要的影响因素是：当键值对被换出内存池、写入硬盘后，系统必须对内存进行垃圾回收，增大值长便加重了垃圾回收的负担，以 256 字节值长为例，此时系统每秒吞吐 17 万个键值对，必须回收 43MB 左右的内存空间，而这些键值对在内存空间中并不是连续的，因此垃圾回收工作的负担很大。

（2）　一个次要原因是，在测试中，测试程序必须实时地创建每个键值对并进行赋值，而这些操作都是被计入系统运行时间的。尽管数据库后台线程也在时刻进行同步工作，但在硬盘成为瓶颈之前，创建键值对的操作仍使测得的运行时间比键值数据库的实际工作时间更长。

6.4　区域查询对 FBTS 性能的提升

该实验测试区域查询（range query）对 FBTS 性能的影响。一个生产者按升序向数据库中插入 10,000,000 个键值对，当生产者完成全部插入操作后，一个消费者再从数据库中获取键值对。之所以要等生产者完成全部插入后再测试消费者的读取速度，是因为：（1）读速度一般大于写速度，两者同时进行则插入速度会成为瓶颈；（2）防止生产者、消费者并发访

问成为瓶颈。本实验中键长为 8 字节，值长采用 64 字节和 1024 字节（由于 1024 字节的实验比较耗时，我们只插入 2,000,000 个键值对），而区域查询的宽度（Bulk Factor）则分别为 1、2、4、8、16、32、64、128、256、1024，检测在不同宽度下的读取速度（以 K/s 为单位），实验结果如图 6-2，注意横轴仍为对数坐标。

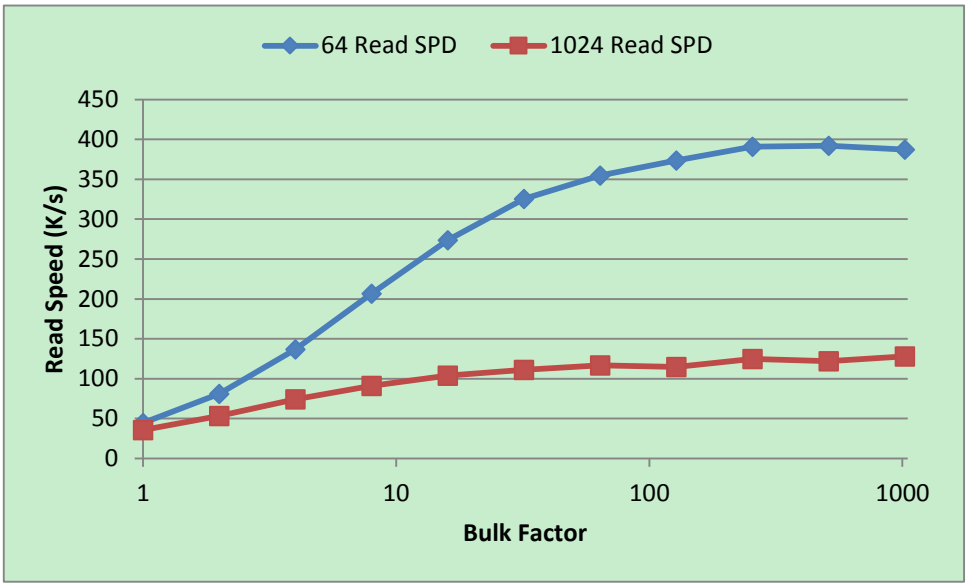

图 6-2　区域查询对性能的提升

从实验结果可以看到，点查询的读取速度很低，以 64 字节的值长为例，每秒仅能进行 45,000 次查询，返回 45,000 个键值对。当区间查询的宽度较小时，增加区间宽度对性能的提升十分明显，例如当宽度增加到 8 时，FBTS 便能在一秒内返回超过 20 万个键值对了；但当区域查询的宽度达到 100 以上时，再增加区间宽度对性能的提升便不明显了，系统稳定地在一秒内返回 35 万-40 万个键值对。这是因为查询操作有两个步骤：（1）在 FBTree 中定位目标键位置；（2）读取并返回键值对。采取区域查询，可以减少操作（1）的次数，从而提高查询效率；但区域查询仍需获取每个键值、访问每个叶节点，因此并不能减少操作（2）的次数，尤其是不能减少叶节点缺页、换页的次数。因此区域查询并不能无限制地提升访问速度。

当键值长度较大时，区域查询的性能会在更小的区间宽度上饱和，这是因为键值长度较大时，硬盘的读写速度会更早地成为瓶颈。例如对于 1024 字节的值长来说，区间长度达到 16 时 FBTS 的性能便已饱和，此时 FBTS 每秒返回约 12 万个键值对，对应硬盘读取速度约 120MB/s。

6.5　变长数据对 FBTS 性能的影响

衡量 FBTS 对变长数据支持能力的主要指标有：（1）FBTS 对键值长度阈值的预测是否合理；（2）变长数据对数据插入速度的影响。根据 5.6.1 节，判断键值长度阈值是否合理，主要看它能否平衡 FBTree 节点的空间利用率和溢出数据的比例。如果键值长度的阈值过高，虽然保证了大部分数据都不会进入溢出堆，但会降低 FBTree 节点的空间利用率，产生页内碎片，浪费磁盘空间；反之，虽然空间利用率得到了提高，却使过多数据被迫进入溢出堆，会影响 FBTS 的效率。综上，我们选取数据溢出率、空间利用率、插入操作速度三个指标来衡量 FBTS 对变长数据的支持能力。

我们选取 64 字节作为数据的平均值长度，数据长度的分布服从正态分布，令其标准差从 0 增加到 25，插入 10,000,000 个键值对，观察 FBTS 对值长阈值的预测（表中第三列），统计溢出数据的总数和比例（第四列、第五列）。FBTS 叶节点的空间利用率可以用平均值长度除以值长阈值得到（第六列）。测试结果如表 6-3 所示。

表 6-3　FBTS 对变长数据的支持情况

Mean Value Len (Byte)	Standard Deviation	Predict Length Threshold (Byte)	Overshoot Values	Overshoot Rate (%)	Space Utilization (%)	Insert SPD (K/s)
64	0	66	101	0.00	96.97	203.56
64	5	76	37873	0.38	84.21	199.19
64	10	87	90524	0.91	73.56	192.44
64	15	97	119721	1.20	65.98	187.08
64	20	107	138383	1.55	59.81	182.45
64	25	117	154955	1.55	54.70	185.02

回忆 5.6.1 节，FBTS 对键值长度阈值的估计采取加两倍标准差的办法，公式为：

*Threshold = (Observed Avg + 2 * (Observed Standard Deviation)) * 1.05*

正态分布在两个标准差内涵盖了约 95%的分布，而阈值作为上界，应该涵盖约 97.8%的分布，即溢出数据的比例应接近 2.2%。由于 FBTS 在两个标准差以外还乘以了安全系数 1.05，因此溢出数据应该更少。表 6-3 的结果完全印证了上述理论分析，我们观察到随着标准差的增大，溢出数据的比例稍有上升，但收敛于 2.2%以下。把阈值估计式改写：

*Threshold = Observed Avg + 2.1 * Observed Standard Deviation + 0.05 * Observed Avg*

因此 FBTS 实际容忍了 2.1 个标准差加一个常数项（0.05 倍平均值）的变长区间，随着标准差的增加，常数项的影响递减，容忍区间向下收敛于 2.1 个标准差，这解释了为什么溢出数据比例会增加但收敛于 2.2%以下。

随着标准差的增加，硬盘的空间利用率在降低，但相对于标准差与数据平均长度的比例，（数据平均长度仅 64 字节，但标准差达到了 25），这个利用率仍然是可以接受的。我们尽量让大部分数据都直接进入 FBTree 节点而不进入溢出堆，因此保证了插入操作的速度基本不受影响，这与 5.6.1 节中提到的“令阈值向提升数据访问性能的方向稍作倾斜”是一致的。

6.6　并发操作对 FBTS 性能的影响

为了测试并发访问对 FBTS 性能的影响，本实验令一个生产者向数据表中插入 10,000,000 个键值对，同时分别有 0 个、1 个、2 个、4 个消费者并发地读取该数据表。根据 6.4 节关于区域查询的实验结果，我们固定消费者读取操作的 Bulk Factor 为 100，实验结果如下，其中读取速度、磁盘速度都是每个消费者单独的速度，而非所有消费者的速度之和。

表 6-4　并发操作对 FBTS 性能的影响

Value Len (Bytes)	#Reader	Read SPD (K/s)	Disk SPD (MB/s)
64	0	184.27	11.25

64	1	132.21	8.07
64	2	121.73	7.43
64	4	112.17	6.85
64	8	106.99	6.53
1024	0	78.86	77.01
1024	1	54.50	53.23
1024	2	52.13	50.90
1024	4	49.81	48.64
1024	8	49.79	48.62

相比于只有单一生产者或消费者的情况，一个生产者、一个消费者并发访问对性能的影响较大，实验结果显示后者的平均性能大约为前者的 70%；然而，增加消费者的数量并不会对系统效率产生明显的影响，这是因为：（1）FBTS 中大部分并发访问控制都是由读写锁实现的，对它来说，是否有消费者会明显影响生产者对共享资源的访问效率，但在存在消费者与生产者竞争的前提下，一个消费者与多个消费者对并发访问效率并没有明显影响；（2）FBTS 的内存缓冲池可以命中大多数消费者的数据访问请求，因此多个消费者并发访问并不会引起更多缺页和磁盘访问。其中第（2）点在硬盘速度上已有体现，因为多个消费者读取硬盘的速度之和已经远超过了硬盘的最大速度。需要注意的是，如果多个消费者请求不同的数据，则（2）未必成立，但在我们的应用场景中，多个消费者的情形主要发生在广播节点上，因此他们倾向于请求相同的数据，而且在同一时刻，数据读取的位置也比较接近。

6.7 乱序插入对 FBTS 性能的影响

FBTS 的稳定性主要体现在对乱序插入的容忍能力。由于“几乎有序”这一模式的界限是模糊的，我们必须通过实测数据说明：当乱序数据的比例不超过多少时，我们可以认为数据插入是“几乎有序”的。这一比例越高，FBTS 的稳定性就越好。

但是“乱序序列”多种多样：有完全随机的乱序序列，也有按照一定的模式生成的有规律的乱序序列。我们要用哪一种“乱序序列”来测试呢？一个完全随机产生的序列当然是“乱序”的，但对大部分数据库，包括 FBTS，插入完全随机序列是一个随机访问的问题，涉及磁盘和内存的随机访问，必然伴随大量的缺页错误和磁盘寻道，效率都很低。事实上，选用哪一种的乱序序列要看实验目的是什么。FBTS 是专用于生产者-消费者仓库的数据库，该实验的目的是测试 FBTS 对其应用场景中可能出现的乱序插入的支持能力，而不是测试 FBTS 对其他访问模式和随机访问的支持能力。因此，我们要选择在生产者-消费者仓库中常见的乱序模式进行实验。

生产者-消费者仓库的应用场景中最常见的一个乱序模式是：某些数据的插入时间晚于他在升序中的位置，即数据的插入位置向后移动。例如”1,2,3,5,6,4,7,8”中的数据 4。这个模式对应的情况是：因为信道丢包、生产者丢失数据等原因，某些数据未能及时进入仓库，然而系统需要一段时间才能探测到这个问题、命令生产者补发或重新产生丢失的数据，在此期间它之后的一段数据已经被插入到了数据库中。另外，由于大多数系统可以及时探测到数据丢失，乱序数据向后移动的距离一般不会太长。

我们针对这个乱序模式进行实验：向 FBTS 的数据表中插入 10,000,000 个键值对，键长 8 字节，值长 64 字节，在第一组实验中（Small Delay），有一定比例的数据随机向后移动 1-20,000 个位置，乱序数据比例从 0%变化到 5%；在第二组试验中（Large Delay），我们将乱

序数据向后移动的范围增大到 1-100,000，以刻画一个对数据丢失反应迟钝的分布式系统。

我们测量三个指标：插入速度、FBTree 总节点数以及缺页次数。测量总节点数是因为 FBTree 在处理乱序插入时可能会退化，通过测量总节点数可以计算综合空间利用率（升序插入时的节点总数除以乱序插入时的节点总数），从而判定 FBTree 的退化程度；测量缺页次数则可帮助我们理解乱序数据对 FBTree 效率产生影响的原因。结果如表 6-5、图 6-3、图 6-4：

表 6-5　乱序插入对 FBTS 性能的而影响

OOO Rate (%)	Small Delay			Large Delay		
	Insert SPD (K/s)	# Nodes	# PG Faults	Insert SPD (K/s)	# Nodes	# PG Faults
0.0	198.20	101011	0	196.46	101011	0
0.5	190.43	141568	85	165.55	141902	21798
1.0	181.48	165726	463	146.39	165783	46216
1.5	174.38	180367	1262	130.17	180513	70117
2.0	169.31	189021	2290	121.04	189050	91689
2.5	165.54	194357	3393	113.15	194265	112301
3.0	160.36	197275	4446	106.58	197379	132825
3.5	158.45	199361	5614	101.02	199335	153721
4.0	155.25	200483	6726	96.57	200578	174184
4.5	153.25	201049	7814	91.46	201233	193970
5.0	150.26	201387	8517	87.00	201513	214855

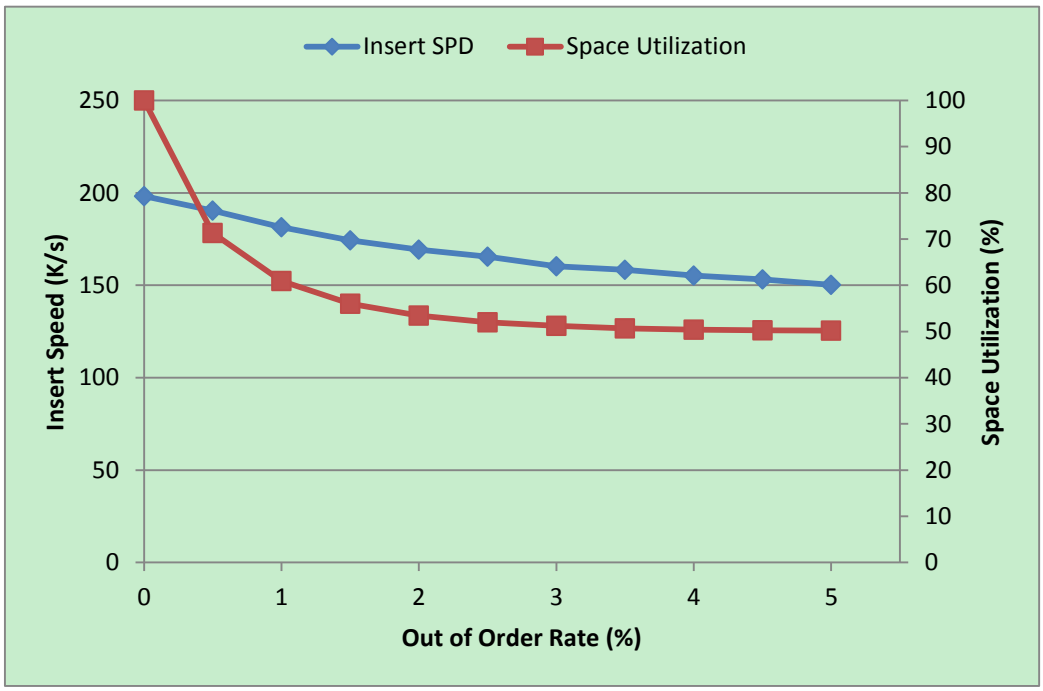

图 6-3　乱序插入对 FBTS 性能的影响，第一组实验

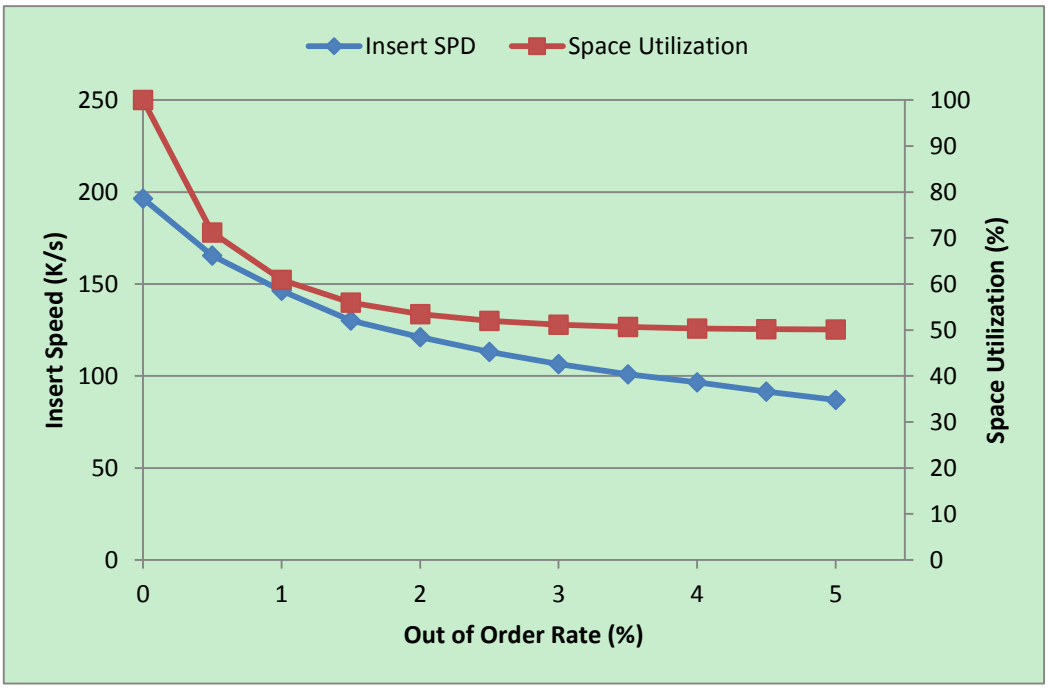

图 6-4　乱序插入对 FBTS 性能的影响，第二组实验

对比两组实验可以看到，如果分布式系统对数据丢失的探测较及时（数据后移距离在 20,000 以内），FBTS 可以很好地容忍因数据丢失造成的乱序数据：FBTS 可以在损失不超过 25%性能的前提下容忍 5%比例的乱序数据，这个比例已经完全可以满足运行于商业集群上分布式系统的需求。然而，如果分布式系统对数据丢失的反应较迟钝，数据后移距离较长，则 FBTS 的性能会明显下降。两组实验在缺页次数上的巨大差异反映了其中的原因：如果乱序数据的后移距离过长，则它所在的 FBTree 节点届时已被从内存池中换出，因此很大比例（实验中约 40%）的乱序插入都会引起缺页错误，处理缺页的高昂代价导致插入速度进一步下降。

另外，FBTree 在该乱序模式下的空间利用率快速收敛于 50%（即使只有 0.5%比例的乱序数据，空间利用率已经低于 70%），这是可以被理论分析预见的。假设乱序数据的比例不大（例如 20%以下），FBTree 在有序插入的部分仍可以生成许多满载叶节点，而在满载叶节点上再进行任何一次乱序插入都会引起叶节点的分裂。由于 FBTree 的分支系数很高，叶节点的数量远小于数据数量，即使仅有 1%-2%比例的数据是乱序插入的，几乎每个叶节点仍能分摊到乱序数据，从而分裂成为两个叶节点；分裂后的两个新节点不满，因此在同一区间上更多的乱序插入不会使他们继续分裂成更多节点。综上，在实验的乱序模式下，FBTree 的空间利用率应收敛于 50%，与观测相符。

6.8　本章小结

本章通过六个实验展示了 FBTS 数据库的性能，并对实验数据做出了合理的分析。

其中，通过 FBTS 与 Berkeley DB 的对比试验，我们可以看到 FBTS 在特定应用场景下的性能可达 Berkeley DB 的三倍，对于小规模数据的插入速度可达每秒 200,000 次。第二个实验揭示了影响 FBTS 性能的两个主要瓶颈：FBTS 内部逻辑的效率以及硬盘的读写速度。第三到第六个实验则证实了 FBTS 系统设计的合理性，以及 FBTree 对少量乱序访问的容忍能力。

第七章　总结和结论

本文系统地研究了如何为生产者-消费者数据缓冲仓库设计高效的专一型键值数据库。该应用场景要求数据库按键建立有序索引，高效地支持插入、查询和删除操作，并能优化批量操作和区域查询的性能。经过深入分析，我们可以发现该应用场景中蕴含了“几乎有序”的数据访问模式，利用这个规律，我们设计了新型索引数据结构 FBTree，以及基于 FBTree 的数据管理系统 FBTS。

FBTree 的每个设计细节和优化策略都与“几乎有序”的数据访问模式紧密相关。适应性分裂策略（ASS）解决了传统 BTree 在升序插入下空间利用率低、节点分裂代价高的问题，而相比于 Oracle 的 90-10 分裂策略，ASS 又可以自动隔离乱序数据，提高 FBTree 对乱序插入的容忍度，防止 FBTree 退化。类似地，我们利用几乎有序访问的特点选择 FBTree 节点内部检索的方向，进一步优化 FBTree 的性能。

我们也能发现 FBTS 系统设计与应用需求、数据访问特征之间的类似联系。例如：（1）由于 “几乎有序”的数据访问模式具有强烈的时空本地性，FBTS 配备了内存缓冲池和 TLB，以利用时空本地性提高数据库的性能；（2）因为数据长度是可变的，所以 FBTS 设立了溢出堆以妥善管理超长数据，并动态预测键值长度，平衡空间利用率和数据访问性能；（3）生产者-消费者仓库中的数据是不断更新的，伴随着新数据的插入，也不断有旧数据被删除，即使每时每刻仓库中的数据都不多，流经仓库的数据总量却可以十分庞大，为此 FBTS 设计了硬盘空间重用机制，避免了数据库大小不断增加、耗尽硬盘空间。

综上，FBTree 和 FBTS 都大量利用了应用场景的特异化性质，从而使 FBTS 成为了一个高度适应该应用场景的专一化数据库系统。实验数据表明 FBTS 在该应用场景下的性能要远高于 Berkeley DB 等通用型键值数据库，并能良好支持变长数据、并发读写等应用需求。更重要的是，由于 FBTS 对应用场景中出现的乱序访问模式也有一定的容忍能力，使 FBTS 系统十分稳定，可以从容应对分布式系统中的各类动态事件。

FBTS 已被应用于 Grape 系统中。在数月的高负荷测试中，FBTS 的性能表现优异，运行稳定，系统资源占用少，为提高 Grape 系统的吞吐量做出了贡献。然而，FBTS 系统本身并不是该课题的全部意义，通过设计 FBTS 系统的整个流程，我们也可以窥斑见豹，对专一型数据库的意义有更深刻的了解。

参考文献

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. "Bigtable: a distributed storage system for structured data"[C]. Operating Systems Design and Implementation (OSDI), 205—218, 2006.
[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. "Dynamo: Amazon's Highly Available Key-value Store"[C]. ACM Symposium on Operating Systems Principles - SOSP, pp. 205-220, 2007.
[3] Michael A. Olson, Keith Bostic, and Margo Seltzer. "Berkeley DB"[C]. USENIX Technical Conference - USENIX, pp. 183-191, 1999.
[4] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. "Scalaris: reliable transactional p2p key/value store"[C]. Erlang Workshop - ERLANG, pp. 41-48, 2008.
[5] Jim Gray, and Andreas Reuter. "Transaction Processing: Concepts and Techniques"[M]. Morgan-Kaufman Publishers, 1993.
[6] Michael Stonebraker. "SQL databases v. NoSQL databases"[J]. Communications of The ACM - CACM , vol. 53, no. 4, pp. 10-11, 2010.
[7] Rabi Prasad Padhy, Manas Ranjan Patra, et al. "RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's"[J]. International Journal of Advanced Engineering Sciences and Technologies, Vol. No.11, Issue No.1, 015 - 030, 2011.
[8] Kai Orend. "Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer"[D]. Master's Thesis in Technical University Munich, 2010.
[9] David K. Gifford. "Weighted voting for replicated data"[C]. ACM Symposium on Operating Systems Principles – SOSP, 1979.
[10] Neal Leavitt. "Will NoSQL Databases Live Up to Their Promise?"[J]. Computer, vol. 43, pp. 12-14, February 2010.
[11] Zhengping Qian, Zheng Zhang, Yong He, et al. "Grape: Large-Scale Real-time Stream Processing in the Cloud"[C]. In submission of OSDI 2012.
[12] ALI, M. H., GEREA, C., RAMAN, B. S., SEZGIN, B., TARNAVSKI, et al. "Microsoft CEP server and online behavioral targeting"[C]. Proc. VLDB Endow, 1558–1561, Aug. 2009.
[13] Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters"[J]. Communications of the ACM, 2008.
[14] Zhengping Qian, Xiuwei Chen, Nanxi Kang, et al. "MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud"[C]. EuroSys, 2012.
[15] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. "Fault-Tolerance in the Borealis Distributed Stream Processing System"[C]. In ACM SIGMOD Conf. 2005.
[16] "Twitter's 2011 Year in review"[R/OL]. http://yearinreview.twitter.com/en/tps.html
[17] Bayer, R., McCreight, E. "Organization and Maintenance of Large Ordered Indexes"[J]. Acta Informatica 1 (3): 173–189, 1972.

[18] Hemant K. Chitale , Index Block Splits : 90-10[R/OL], http://hemantoracledba.blogspot.com/2009/05/index-block-splits-90-10.html.
[19] Aho, Denning and Ullman. “Principles of Optimal Page Replacement”[J]. Journal of the ACM, Vol. 18, Issue 1,January 1971, pp 80–93.
[20] Tanenbaum, Andrew S. “Operating Systems: Design and Implementation (Second Edition)”[M]. New Jersey: Prentice-Hall 1997.
[21] Schmidt, D et al. “Pattern-Oriented Software Architecture Vol 2”[M]. 2000 pp353-363.
[22] “Oracle Berkeley DB: Performance Metrics and Benchmarks”[R/OL]. August, 2006.

谢辞

首先我要感谢与我共同在微软亚洲研究院从事 Grape 系统研究的导师和同事，他们是：Dr. Zheng Zhang, Dr. Zhengping Qian, Yong He 以及 Zhuojie Wu。如果没有 Grape 项目，也就不会有这个课题。在整个 Grape 项目中，他们对我的悉心教导和热心帮助令我获益匪浅。在我研究本课题的过程中，他们也对本课题提出了宝贵的意见和建议。

其次我要感谢本课题的导师：上海交通大学的过敏意教授、沈耀老师。过老师和沈老师在毕业设计的整个过程中，一直对我的科研工作进行指导、并提供必要的帮助，大到课题的研究方向，小到系统的细节问题，无所不谈，无微不至。感谢两位老师对我的指导和鼓励！

我还要感谢上海交通大学计算机系和软件学院的其他教授，尤其是邹恒明老师、梁阿磊老师、朱其立老师，三位老师分别教授操作系统、计算机组成和数据库系统的课程，正是他们为我从事计算机系统方向的研究打下了坚实的基础。

在本科四年期间，我受到了 ACM 班老师和同学们无微不至的关怀和帮助。尤其是俞勇教授，不但教育我如何做学问，更教育我如何做人。俞教授为了给我们争取更好的学习环境和科研机会，不辞辛苦、东奔西走，感谢您和 ACM 班为我打开了计算机科学领域的大门！

最后，我要感谢 Yao Chen，一年以来她对我的科研工作给予了莫大的支持，与我一同分享喜悦、分担挫折。Yao Chen 也参与了该课题的讨论，本课题中的新数据结构的名称——Flat BTree，正是由她提出的。

BUILDING APPLICATION-SPECIFIC KEY-VALUE DATABASES BY EXPLOITING DEMANDS AND USAGE PATTERNS

General-purpose databases, with rich functions, complex structures and wide scope of applications, have dominated the area of DBMS for a long time. However, as we are making progress in parallel computing and distributed systems, general-purpose databases cannot catch up with the increasing amount of data processed by modern computing clusters, and people start to build application-specific databases by exploiting unique characters and patterns in their applications. In this thesis, I'll illustrate how to build an application-specific database system for producer-consumer warehouses in distributed systems with 5 steps: analyzing application demands, exploring data access patterns, designing indexing data structure, designing DBMS, and run tests.

In distributed computing systems such as Grape and MadLinq, large amount of data is transferred from one computing node to another due to the pipelined execution of infinite stream. Key-value databases are deployed between producer-consumer pairs as warehouses to coordinate speed differences and store data for a longer period as upstream backup. The database must fulfill the following functional requirements: (1) it must support bulk operations, especially range queries; (2) In some rare cases such as data loss, producer may generate out-of-order data, so DBMS is required to index and reorder data; (3) Since producer and consumer may access database concurrently, and there may be multiple consumers, we need DBMS to provide proper concurrency control mechanisms. Besides these 3 functional requirements, there are 2 additional performance requirements: (1) DBMS must provide high throughput with very low latency; (2) DBMS must control its memory usage.

For this application, we can find many useful patterns, for example, there is only 1 producer, so DBMS doesn't need to worry about multi-writer concurrency control. However, the most important pattern is that, the insertions are almost ordered. In normal run, vertices output data in an ordered way. Out-of-order insertions might appear only around time when data loss occurs, vertex is abnormal, heavy-loaded, crashed, or in recovery, which are rare cases comparing to normal run. Not only insertion is in ordered way, retrieval and deletion also stick to the "almost-ordered pattern".

With such demands and usage patterns, we can understand why hash table, list, and B tree are not qualified to be the indexing data structure in this application: (1) Hash table cannot provide ordered indexing and range queries, so it doesn't meet the functional requirements; (2) Lists provide best performance if insertions are strictly ordered, but it cannot tolerate even a few out-of-order insertions, because out-of-order insertion may cause large amount of data shifts; (3) When data items are inserted in ordered way, the balanced split strategy (BSS) makes B-tree half-full, which is its worst case. A half-full B-tree wastes large amount of disk spaces. Moreover,

every split operation moves k/2 data items from one node to another, which is costly.

In order to solve the "half-full" problem for BTree, I proposed FBTree. FBTree is quite similar as B-tree. It is a search tree with a pre-defined width k: internal nodes have no more than k children, and leaf nodes contain no more than k-1 data items. In internal nodes, each pointer has a corresponding key range, which indicates range of keys in that sub-tree. For siblings, their ranges are continuous, monotonous, but not intersecting, and child's range is a sub-range of its parent.

Search algorithm in FBTree is totally the same as that in BTree, and they also share same frameworks of insertion and deletion algorithms. However, huge differences exist as following:

(1) In BTree, nodes have at least k/2 children or data items. But this restriction doesn't apply to FBTree.

(2) When a data item is inserted into BTree, causing the width of a node goes beyond k, a balanced split strategy is applied, i.e., a new node is inserted, and old children or items are assigned evenly between two nodes, so that both of them have at least k/2 children or data items. In FBTree, this balanced strategy is replaced by a new one called "**adaptive split strategy**"(ASS). Only Children or items that lie behind the newly inserted one are assigned to the new node. If the newly inserted child or item lies at last, only itself will be assigned to the new node.

(3) In deletion, FBTree makes no attempt of balancing, i.e., FBTree doesn't transfer contents nor fuse siblings when a node has too few children or items.

FBTree is specially designed for the "almost ordered" data access pattern: Although the adaptive split strategy may cause FBTree to degenerate when data insertion is not "almost ordered", (for example, when data is inserted in descending order), FBTree is better than BTree under almost ordered insertions: (1) When data is fully ordered, ASS makes every node in FBTree to be full; (2) When there are a few out-of-order data, ASS can automatically isolate out-of-order parts from ordered ones by splitting them into different FBTree nodes, so that the out-of-order data won't harm the global structure and future insertions.

In implementation of FBTree, I make further optimizations according to the application demands and usage patterns. For example, in-node search (locating target key) of FBTree can be optimized by wisely choosing the search direction. According to the "almost ordered pattern", when we locate position for insertion, they tend to go to right-most child at internal nodes; when we locate position for deletion, they tend to go to left-most one. So choosing proper search directions for different operations can speed up in-node search from O(k) to O(1).

With FBTree as indexing data structure, I designed and implemented FBTS, which is an application-specific key-value database for producer-consumer warehouses. I make further use of application demands and data access patterns to optimize FBTS for this unique application.

One instance of FBTS manages multiple independent tables. Each table is a set of <key, value> pairs. FBTS allows user to use arbitrary classes as types of key and value, as long as a comparer is provided on key type, and serialization/deserialization methods are provided on both

types. Each table in FBTS is directly organized and indexed by FBTree. Most <key, value> pairs are directly stored in leaf nodes of FBTree, except for those extremely long keys/values.

The “almost ordered” data access pattern suggests that data accesses in FBTS have high temporal and spatial locality, so I deployed a memory buffer pool to speed up data accesses. In order to control memory usage, all tables share a same memory buffer pool. FBTS manages this memory pool by paging: we can regard nodes in FBTree as pages, disk as virtual memory, and memory pool as physical memory. Just as computer program accesses pages by their virtual address, FBTree accesses nodes by their disk addresses. Memory manager maintains a map between node’s disk address and its memory pool position which is called page table. If a target node isn’t in memory pool, a page fault occurs. We read the node from disk, swap out a victim node in memory pool and put the target node into memory pool.

In order to speed up address translation, I designed a small TLB for FBTS, which buffers the latest 10 translations. This optimization brings a 25% performance increase.

Memory buffer needs to keep the consistency between memory and disk. FBTS uses a new synchronization policy called “backstage aggressive synchronization”, which combines advantages of both write-back policy and writ-through policy. When FBTree modifies a memory page, FBTS doesn’t synchronously update disk, but put a “dirty” flag on that page, which is similar as that of write-back policy. However, FBTS doesn’t wait until the page is swapped out, there is a backstage thread that periodically polls pages in memory and aggressively synchronizes dirty pages to disk. If we set the polling period properly, we can utilize the disk idle time to dump dirty pages in advance, without causing too many additional dumps (synchronize one page for multiple times) comparing to write-through policy.

FBTS supports variable length data well. FBTS set a length threshold for keys and values on each FBTree nodes. Data with length being smaller than threshold are directly stored in FBTree nodes, otherwise they are stored to an overflow heap. By changing this threshold, FBTS can make trade-offs between space utilization and performance. FBTS dynamically predict this threshold to balance these 2 factors.

Even if the amount of data stored the warehouse might be small all the time, the total amount of data that ever went through the warehouse could be extremely huge. So FBTS cannot keep allocating new disk space. FBTS contains a disk manager that reuse disk space from deleted data. The disk manager divides the disk file into large blocks of 128MB, and recycles disk space in this granularity, so that it doesn’t need to maintain a complicated free list. This strategy also keeps the spatial locality of FBTree nodes.

FBTS uses 3 kinds of locks to control concurrent accesses. They are table locks, physical page locks, and a global page-table lock. FBTS applies the double-checked locking pattern for accessing page table, so that most address translation doesn’t need to acquire the page-table lock. This design enhances the concurrent performance of FBTS.

Experiments show the inserting speed of FBTS under almost ordered pattern is over 200 K/s, which is as 3 times as that of Berkeley DB. With range queries, the retrieving speed of FBTS is over 400 K/s. FBTS supports variable length data very well, and is immune to concurrent readings.

FBTS also tolerates over 5% out-of-order insertions caused by data loss with an acceptable performance reduction of 25%.