Aalto University School of Science and Technology
TKK Research Reports in Computer Science and Engineering A
Espoo 2010 TKK-CSE-A5/10

ACCESSING MULTIVERSION DATA IN DATABASE TRANSACTIONS

Doctoral Dissertation

Tuukka Haapasalo

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium TU1 at the Aalto University School of Science and Technology (Espoo, Finland) on the 22nd of October 2010 at 12 noon.

Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Aalto-yliopiston teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos

Distribution:
Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 15400
FI-00076 Aalto
FINLAND

URL	`http://www.cse.tkk.fi/`
Tel.	+358 9 470 23228
Fax.	+358 9 470 23293
E-mail	tuukka.haapasalo@tkk.fi

© 2010 Tuukka Haapasalo

Layout: Tuukka Haapasalo (except cover)
Cover image: Anu and Tuukka Haapasalo
Set in the Computer Modern typeface family designed by Donald E. Knuth.

ISBN	978-952-60-3359-4
ISBN	978-952-60-3360-0 (PDF)
ISSN	1797-6928
ISSN	1797-6936 (PDF)
URL	`http://lib.tkk.fi/Diss/2010/isbn9789526033600/`

Multiprint Oy
Espoo 2010

ABSTRACT OF DOCTORAL DISSERTATION	AALTO UNIVERSITY SCHOOL OF SCIENCE AND TECHNOLOGY P.O. BOX 11000, FI-00076 AALTO http://www.aalto.fi

Author Tuukka Haapasalo

Name of the dissertation

Accessing Multiversion Data in Database Transactions

Manuscript submitted	15th June 2010	Manuscript revised	21st September 2010
Date of the defence	22nd October 2010		
☒ Monograph		☐ Article dissertation (summary + original articles)	

Faculty	Faculty of Information and Natural Sciences
Department	Department of Computer Science and Engineering
Field of research	Software Systems
Opponent(s)	Prof. Peter Widmayer
Supervisor	Prof. Eljas Soisalon-Soininen
Instructor	Prof. Seppo Sippu

Abstract

Many important database applications need to access previous versions of the data set, thus requiring that the data are stored in a multiversion database and indexed with a multiversion index, such as the multiversion B^+-tree (MVBT) of Becker et al. The MVBT is optimal, so that any version of the database can be accessed as efficiently as with a single-version B^+-tree that is used to index only the data items of that version, but it cannot be used in a full-fledged database system because it follows a single-update model, and the update cannot be rolled back.

We have redesigned the MVBT index so that a single multi-action updating transaction can operate on the index structure concurrently with multiple concurrent read-only transactions. Data items created by the transaction become part of the same version, and the transaction can roll back. We call this structure the *transactional MVBT* (TMVBT). The TMVBT index remains optimal even in the presence of logical key deletions. Even though deletions in a multiversion index must not physically delete the history of the data items, queries and range scans can become more efficient, if the leaf pages of the index structure are merged to retain optimality.

For the general transactional setting with multiple updating transactions, we propose a multiversion database structure called the *concurrent MVBT* (CMVBT), which stores the updates of active transactions in a separate main-memory-resident *versioned B^+-tree* index. A system maintenance transaction is periodically run to apply the updates of committed transactions into the TMVBT index. We show how multiple updating transactions can operate on the CMVBT index concurrently, and our recovery algorithm is based on the standard ARIES recovery algorithm.

We prove that the TMVBT index is asymptotically optimal, and show that the performance of the CMVBT index in general transaction processing is on par with the performance of the time-split B^+-tree (TSB-tree) of Lomet and Salzberg. The TSB-tree does not merge leaf pages and is therefore not optimal if logical data-item deletions are allowed. Our experiments show that the CMVBT outperforms the TSB-tree with range queries in the presence of deletions.

Keywords temporal database, multiversion data, index structure, concurrency control, recovery

ISBN (printed)	978-952-60-3359-4	ISSN (printed)	1797-6928
ISBN (pdf)	978-952-60-3360-0	ISSN (pdf)	1797-6936
Language	English	Number of pages	200 p.
Publisher	Department of Computer Science and Engineering		
Print distribution	Department of Computer Science and Engineering		

☒ The dissertation can be read at http://lib.tkk.fi/Diss/2010/isbn9789526033600/

Aalto-yliopisto

VÄITÖSKIRJAN TIIVISTELMÄ	AALTO-YLIOPISTO TEKNILLINEN KORKEAKOULU PL 11000, 00076 AALTO http://www.aalto.fi

Tekijä Tuukka Haapasalo

Väitöskirjan nimi

Versioidun tiedon haku tietokantatransaktioissa

Käsikirjoituksen päivämäärä	15.6.2010	Korjatun käsikirjoituksen päivämäärä	21.9.2010
Väitöstilaisuuden ajankohta	22.10.2010		
[✓] Monografia		[ ] Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)	

Tiedekunta	Informaatio- ja luonnontieteiden tiedekunta
Laitos	Tietotekniikan laitos
Tutkimusala	Ohjelmistojärjestelmät
Vastaväittäjä(t)	Prof. Peter Widmayer
Työn valvoja	Prof. Eljas Soisalon-Soininen
Työn ohjaaja	Prof. Seppo Sippu

Tiivistelmä

Perinteisesti tietokantasovellusten tietokantaan tekemät päivitykset korvaavat tietokannan tilan uudella, jolloin aiempaa tilaa eli versiota ei enää ole olemassa. Nykyisin tietokantasovelluksilla haetaan kuitenkin myös aiempien versioiden monikoita, mikä tulee ottaa huomioon tietokantojen hakemistorakenteiden suunnittelussa. Becker ja kumppanit ovat kehittäneet *moniversio-B$^+$-puun* (*multiversion B$^+$-tree*, MVBT), joka on eräs optimaalinen moniversiohakemistorakenne. Optimaalisuus tarkoittaa tässä sitä, että kaikki operaatiot ovat aina yhtä tehokkaita kuin vastaavassa yhden version hakemistorakenteessa. MVBT-rakenteen rajoitteena on kuitenkin se, että siinä yksi versio voi sisältää vain yhden päivitysoperaation, kun taas transaktiomallissa yksi transaktio voi tehdä monta päivitystä. Toinen rajoite rakenteessa on se, että päivityksiä ei voi peruuttaa.

Väitöskirjassa laajennetaan MVBT-rakennetta siten, että yksi transaktio voi päivittää useampaa avainta ja transaktiot voidaan peruuttaa. Laajennetussa *transaktionaalisessa MVBT*-rakenteessa (*transactional MVBT*, TMVBT) voidaan yhtä päivitystransaktiota suorittaa rinnakkain usean lukutransaktion kanssa. Hakemistorakenne on optimaalinen kaikissa tilanteissa, myös poisto-operaatioiden jälkeen.

TMVBT-rakennetta ei voi sellaisenaan käyttää tilanteissa, joissa tietokantasovelluksessa täytyy suorittaa rinnakkain useita päivitystransaktiota. Näitä tilanteita varten väitöskirjassa esitellään *rinnakkainen MVBT*-rakenne (*concurrent MVBT*, CMVBT), joka koostuu TMVBT-rakenteesta sekä sen rinnalla toimivasta keskusmuistissa pidettävästä *versioidusta B$^+$-puusta* (VBT), johon aktiivisten transaktioiden muutokset tallennetaan transaktioiden suorituksen ajaksi. Kun transaktiot sitoutuvat, muutokset siirretään versioidusta B$^+$-puusta TMVBT-rakenteeseen. CMVBT toimii yleisten rinnakkaisuudenhallinta-algoritmien kanssa ja sen elvytysalgoritmi perustuu yleisesti käytettyyn ARIES-elvytysalgoritmiin.

Väitöskirjassa todistetaan, että päähakemistorakenteena toimiva TMVBT on optimaalinen. Lisäksi osoitetaan kokeellisesti, että CMVBT-rakenne on yhtä tehokas kuin Lometin ja Salzbergin TSB-puu (time-split B$^+$-tree) yleisissä kyselyissä ja päivityksissä, sekä tehokkaampi avainvälihauissa, kun tietokannasta on poistettu monikoita. Tehokkuusero johtuu siitä, että TSB-puu ei yhdistä tietokantasivuja eikä täten ole optimaalinen.

Asiasanat historiallinen tietokanta, versioitu tieto, hakemistorakenne, rinnakkaisuus, elvytys

ISBN (painettu)	978-952-60-3359-4	ISSN (painettu)	1797-6928
ISBN (pdf)	978-952-60-3360-0	ISSN (pdf)	1797-6936
Kieli	Englanti	Sivumäärä	200 s.
Julkaisija	Tietotekniikan laitos		
Painetun väitöskirjan jakelu	Tietotekniikan laitos		

[✓] Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2010/isbn9789526033600/

Omistettu rakkaalle vaimolleni Anulle,
joka jaksaa ymmärtää
hulluja päähänpistojani.

Preface

First of all, I would like to thank my supervisor, Professor Eljas Soisalon-Soininen, for introducing me to the field of algorithm research, and for providing me with an interesting and challenging research topic. I have often heard that the topic for a dissertation is half the work; in my case, I certainly got off to a swift start thanks to being able to start my research work directly. I would also like to thank my instructor, Professor Seppo Sippu from the University of Helsinki, for taking such an active role in supporting my research work, and for always having time to provide me with exhaustive comments on my contributions.

I would especially like to thank Professor Peter Widmayer (ETH Zürich) for honoring me by agreeing to be the opponent for this dissertation, and I am also grateful to my pre-examiners, Assistant Professor Dr. Alexander Markowetz (Rheinische Friedrich-Wilhelms-Universität Bonn) and Dr. Jan Lindström (IBM Helsinki), for their invaluable comments on the manuscript. Many interesting future research ideas have already emerged from their observations.

My research work was carried out at the Department of Computer Science and Engineering, and I want to thank all my colleagues for their support and company. I thank my team colleague, Dr. Ibrahim Jaluta, for introducing me to the myriad details of database algorithms and for always providing me with timely research papers on the research topics at hand. My work and post-graduate studies have been funded by the Helsinki Graduate School in Computer Science and Engineering, and by the Academy of Finland, for which I am grateful.

I thank my friends and family for their support and encouragement; I would especially like to thank my mother for setting an example I could aspire to. Finally, I thank my darling wife Anu for her love and support, for her patience in having to listen to my rants on database research, and also for her practical help with the dissertation.

Espoo, September 2010

Tuukka Haapasalo

Contents

List of Abbreviations

ACID	Properties that are required of transactions: atomicity, consistency, isolation, and durability	9
CMVBT	Concurrent multiversion B$^+$-tree. A fully concurrent index structure composed of a TMVBT index and a main-memory-resident VBT index	111
CTI	Commit-time-to-identifier table. This table is used in the CMVBT index to map commit-time versions into transaction identifiers	115
DAG	Directed acyclic graph	29
Immortal DB	A multiversion database system built on top of Microsoft SQL Server. Based on the TSB-tree index structure	59
LRU	Least recently used	138
LSN	Log sequence number	27
MBR	Minimum bounding region	44
MVAS	Multiversion access structure. A multiversion index structure designed by Varman and Verma. Similar to the MVBT	69
MVBT	Multiversion B$^+$-tree. An asymptotically optimal multiversion index by Becker et al.	61
PTT	Persistent timestamp table. The PTT is used to map transaction identifiers into commit-time versions in the TSB-tree. It is similar to the CTI table but has a reverse mapping	59

SI	Snapshot isolation, a commonly used multiversion concurrency-control algorithm that can be made serializable	25
SMO	A structure-modification operation is an operation that modifies the structure of the database index by splitting or merging a database page	27
TMVBT	Transactional multiversion B$^+$-tree. Our transactional extension to the MVBT index. Allows a single updating transaction to run concurrently with multiple read-only transactions	75
TSB-tree	Time-split B$^+$-tree. A multiversion index structure by Lomet and Salzberg	55
VBT	Versioned B$^+$-tree. A B$^+$-tree used for storing entries that contain version information in addition to the data item key and value	32
VTT	Volatile timestamp table. Used as a main-memory cache for the PTT in the TSB-tree	59
WAL	Write-ahead logging. A policy that states that the database log file must be flushed onto disk before any pages that contain the corresponding updates described in the log are written onto disk	10
WORM	Write-once, read-many. Describes media that can be written to only once, but read from many times, such as CD-R, DVD-R, BD-R, and tape archives .	41
WMRM	Write-many, read-many. Describes media that can be written to multiple times, such as hard drives. While this definition includes re-writable discs (CD-RW, DVD-RW, BD-RW), it is often assumed that WMRM media is faster than WORM media	58

List of Symbols

v_{commit}	The latest committed version of a multiversion database index	12
v_{active}	The active version of a TMVBT index. Updates to the index receive this version	76
v_{stable}	The most recent version that has been moved to the TMVBT from the VBT in a CMVBT index	113
v_{move}	The version currently being moved from the VBT into the TMVBT in a CMVBT index	115
$\mathsf{id}(T)$	The unique identifier of an active transaction T	17
$\mathsf{snap}(T)$	The snapshot version of transaction T; i.e., the version that T is reading	19
$\mathsf{commit}(T)$	The unique commit-time version assigned to transaction T upon committing	17
B	The logical capacity of a database page, i.e., how many entries fit to a page	30
m_v	The number of entries in the database that are alive at version v	33
$\perp$	A special value used to record key deletions	18
$root^*$	A search structure that stores the page identifiers of the root pages of a multi-root multiversion index, indexed by the root page versions	38
$\mathsf{kr}(p)$	The key-range of page p	62
$\mathsf{vr}(p)$	The version-range, or life span, of page p	62
min-live	The minimum number of live entries allowed on a single page of a multiversion index	57

List of Figures

CHAPTER 1

Introduction

Modern society has become dependent on large software systems that rely on databases for storing important data. These database systems traditionally maintain only a single version of the stored data set. Updates to the database erase the information content of any previous states, and therefore these databases cannot be used efficiently by any application that needs to query the history of the database. There are many applications, however, that do require access to previous database states, such as trend-processing and inventory control applications. Furthermore, any business-critical applications would benefit from the accountability and traceability that is obtained when the past database states are not deleted, so that it is always possible to find out *if* a change to the database state has occurred, and *what* was changed.

In general, we can identify several major application areas that require access to past database states. These are, for example, applications that deal with evolving data, such as engineering design, land register, and scheduling applications, inventory control systems and moving object databases [8, 10, 94]; statistical applications such as decision support systems and online analytical processing (OLAP) applications [10, 42]; and applications that require accountability, such as medical, legal, financial, accounting, banking, and insurance applications [42, 58, 94]. It is theoretically possible to reconstruct past database states based on database logs and system backups, but this is time-consuming and costly, and thus not feasible for everyday statistical queries. The applications listed here need a database structure that is specifically tailored for storing the evolution of the data items alongside with the data items themselves. These kind of databases are generally called *temporal databases* [14].

When new updates are made to a temporal database, the database states are updated to reflect the state of the world at different time instants. In temporal database theory, there are two orthogonal definitions for system-supported time: transaction time and valid time [14]. *Valid time* is used to record the time when a modelled event happens in the real world, and *transaction time* is used to mark the time when an event or a data item has been recorded in the database. Temporal databases that support both of these time dimensions are called *bitemporal databases* [14]. A traditional database that does not incorporate either valid time or transaction time is called a *snapshot database*.

Some transaction-time databases allow data items to evolve along different time lines, so that a new version may be based on any previous version, not just the latest current version. This time model is required by applications such as engineering design software and version control systems, and the databases that follow this model are called *fully persistent databases* [76]. Most of the temporal databases, however, are designed to work with a single linear data history, in which a new version of the data set is always created based on the most recent database version. These database systems are called *partially persistent databases*, and they are used by applications that record events that happen in the real world. Partially persistent transaction-time databases are called *multiversion databases* by many authors [8, 10, 43, 58, 92], and the different transaction-time instants in these databases are called *versions*. The database thus starts with an initial version, and any updates to the database create new versions of the stored data item set. The most recent version of the database is called the *current version* of the database. In this dissertation, we concentrate our research on multiversion databases.

Temporal databases are not a new idea. The need for supporting the time dimension in database systems has been long identified; for example, Bubenko has presented a conceptual framework for incorporating the temporal dimension into database management systems some thirty years ago [18]. Consequently, there now exists a multitude of different temporal database index structures. Most of these are specifically tailored for some operations, and they are either inefficient for other standard database operations, or their performance may deteriorate with some histories of user actions. Because of this, these structures are not suitable for use as general-purpose temporal indexes. For example, multiversion index structures that are based on hashing techniques are very efficient for single-key queries [45], but they cannot be used for efficient key-range queries. Similarly, tree-duplicating indexes can be queried efficiently [78], but updating them is inefficient. To our knowledge, the most efficient multiversion indexes are the time-split B$^+$-tree (TSB-tree) of Lomet and Salzberg [58, 59], the multiversion B$^+$-tree (MVBT) of Becker et al. [7, 8], and the multiversion access method (MVAS) of Varman and Verma [92].

The comparison of different multiversion index structures is naturally dependent on how we define the efficiency of a given index structure. One such ranking is based on the *asymptotic optimality* of the index operations. That is, if we can define what the performance of an optimal multiversion index structure is, we can compare the structures by analyzing how much their performance differs from the optimal performance. In single-version (snapshot) databases, the most commonly used index structure is the B-tree [5, 6, 23]. The B-tree has many variants, of which the B$^+$-tree is the most often used. The B$^+$-tree is regarded to be an asymptotically optimal database index structure for snapshot databases. All single-key operations on the B$^+$-tree (item query, insertion, and deletion) require access to $\Theta(\log_B m)$ pages of the B$^+$-tree index [23], where m is the number of data items indexed by the B$^+$-tree and B is the page capacity; and the range-query operation requires access to $\Theta(\log_B m + r/B)$ B$^+$-tree pages, where r is the number of data items in the queried range. This holds for all histories of user actions and guarantees that the performance of the B$^+$-tree index never deteriorates.

When querying for data items of a given version v, an ideal multiversion index structure should work as efficiently as a single-version index structure that only contains the items that are part of the queried version v. Conceptually, an index structure that creates a new index tree for each database version achieves this optimality for all query actions, assuming that the root page of the queried version is known; but not for update actions. The single-key operations in an asymptotically optimal multiversion index structure may thus require access to at most $\mathcal{O}(\log_B m_v)$ disk pages, where m_v is the number of data items that are part of the queried version v; and the range-query operation may require access to at most $\mathcal{O}(\log_B m_v + r/B)$ pages. With this definition, only the MVBT [7, 8] is optimal. The MVAS [92] guarantees a slightly different upper bound on the number of page accesses per operation (see Section 4.5), and the TSB-tree [58, 59] has no guarantees if the database history contains deletions.

There are still other requirements for a multiversion database index structure that is to be used in a general-purpose database management system. Firstly, it must be possible to perform multiple updates within a single version in the database to accommodate all the updates of a multi-action updating transaction. Both the MVBT and the MVAS unfortunately require that each update creates a new version, so that multiple updates always result in multiple versions of the database. Secondly, it must be possible to use concurrency-control and recovery algorithms with the index structure so that multiple transactions may query and update the index structure concurrently. The MVBT and the MVAS only allow a single updating transaction to make modifications to the structure at a time, and the transaction cannot be rolled back. The TBS-tree [58, 59] does not have these restrictions, but it fails to guarantee optimality for any of its operations if the database history contains deletions.

The purpose of this dissertation is to design and implement an efficient multiversion database index structure for indexing the evolution of the logical data set. The structure should be optimal, or as close to optimal as possible; and the structure must be usable in a multi-user environment with multiple updating and read-only transactions that are running concurrently. We have selected the asymptotically optimal multiversion B$^+$-tree structure of Becker et al. [7, 8] as the basis of our research work. Our index structure is based on the ideas we have presented in two conference articles [35, 37] and one PhD workshop article [36]; I have been the primary author in all of these articles. We have shown how to extend the MVBT structure into the *transactional multiversion B$^+$-tree* (TMVBT) so as to allow a single multi-action transaction to operate on the structure at a time [35], but we have not been able to extend the TMVBT further so that multiple updating transactions could operate on it concurrently, without compromising the optimality of the index structure. We have therefore used a separate main-memory-resident versioned B$^+$-tree (VBT) structure to store the updates of active transactions, so that a system maintenance transaction can apply the updates of committed transactions to the TMVBT successively, one transaction at a time [37]. We call the combined index structure the *concurrent multiversion B$^+$-tree*, or CMVBT.

Logical data consistency in the CMVBT can be maintained by using any standard multiversion concurrency-control algorithm, such as snapshot isolation [11, 19, 28]. Our algorithms are based on the ARIES recovery algorithm [63, 66], and we perform structure-modification operations (SMOs) on all of the index structures atomically, so that each SMO transforms a structurally valid and balanced index structure into another structurally valid and balanced index [39–41].

We show in this dissertation that the combined CMVBT index is efficient: the performance of the CMVBT is comparable to that of the TSB-tree in general transaction processing, but the CMVBT outperforms the TSB-tree in range queries when the database history contains key deletions. The TSB-tree performs worse with key deletions because it does not merge pages. We further strengthen the test results by showing that the performance of the combined CMVBT index organization is comparable to the performance of the optimal TMVBT index. In fact, with longer transactions, the CMVBT performs better, because the main-memory-resident VBT index groups all the updates of a single transaction together, thus enabling them to be inserted in the TMVBT as an efficient batch update. Batch updating is commonly used to increase the performance of index structures [72, 73].

We begin the dissertation with a general introduction to multiversion database theory in Chapter 2. After that, in Chapter 3, we describe the evolution of multiversion index structures and present a survey of some of the existing structures. In Chapter 4, we review three of the most efficient multiversion structures; namely, the TSB-tree of Lomet and Salzberg [58, 59], the MVBT of Becker et al. [7, 8], and the MVAS of Varman and Verma [92]. Chapter 5 describes our transactional extension to the MVBT, and shows how the TMVBT index retains the optimality of the MVBT. In Chapter 6, we present the concurrent multiversion B$^+$-tree, which is a fully concurrent index structure that is composed of the TMVBT index and a main-memory-resident versioned B$^+$-tree index that is used for storing the updates of active transactions. After that, in Chapter 7, we evaluate the performance of the CMVBT index and compare it to the TSB-tree. We have implemented both indexes and present the results of our test runs on them. Finally, in Chapter 8, we give the conclusions of the research work.

CHAPTER 2

Transactions on Multiversion Data

The theory of general transaction processing in traditional databases is well-defined and mature, and the basic principles are well presented in many textbooks [13, 32, 71]. In this chapter, we concentrate on the theory of multiversion transaction processing, and highlight the differences to classical transaction theory. The traditional read-write model assumes that transactions are sequences of reads and writes on data items, without distinguishing item deletions and insertions from updates [13, 71]. The theory of transaction processing in this dissertation is based on the recoverable transaction model presented by Sippu and Soisalon-Soininen [84], which in turn is based on the model proposed by C. Mohan [63, 64]. In this model, data-item insertions and deletions are made explicit, and structure-modification operations are included in the model. We assume the partially persistent transaction-time model, as described in the introduction. We will use the terminology presented in the consensus glossary of temporal database concepts by Böhlen et al. [14], for the relevant parts. The chapter begins with a short review of the fundamental concepts of transaction management in traditional databases, and continues to describe the most important aspects of multiversion database theory.

2.1 Fundamentals of Snapshot Database Theory

We assume our logical database consists of data items of the form (k, w), where k is the key and w is the value of the data item [84]. The logical model thus has no knowledge of any time dimension. Databases that follow this data model are called *snapshot databases*. As Figure 2.1 shows, any change to a snapshot database overwrites the information about previous states. The *schedule* of a transaction, as shown in the figure, is a list of actions issued by the transaction. The format of the schedule is explained in the following paragraphs.

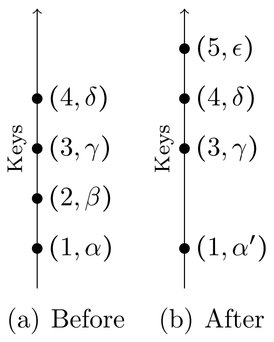


Figure 2.1. Transaction in a snapshot database. The transaction has executed the schedule $bw[1,\alpha,\alpha']d[2]w[5,\epsilon]c$.

In the traditional transaction model, transactions in snapshot databases consist of user actions that either *write* keys to or *read* keys from the database. In addition, a transaction must issue either a *commit* or an *abort action* to indicate that the transaction is finished. After an abort action, the read and write actions are undone and the transaction commits. A read action of a key k by a transaction T_i is denoted by $r_i[k]$, and a write action by $w_i[k]$. The commit and abort actions are denoted by c_i and a_i, respectively. This classical transaction model is inadequate for modelling key-range queries and next-key queries, and it does not differentiate between key insertions, updates, and deletes. Mohan has proposed a more general transaction model alongside his ARIES recovery algorithm [63, 64], which Sippu and Soisalon-Soininen have further refined [84]. In the model of Sippu and Soisalon-Soininen, transactions may *retrieve* the first matching data item (k,w) for which $k\ \theta\ x$, where θ is either $>$ or $\geq$ and x is a given search key; *insert* a new data item (k,w); or *delete* the data item that has the supplied key k. In this model, key-range queries are performed by consecutive key retrievals with θ set to $>$ and $x_{i+1} = k_i$. Retrieval of a data item (k,w) with $k\ \theta\ x$ by transaction T_i is denoted by $r_i[k,\theta x,w]$; insertion of data item (k,w) by $n_i[k,w]$; and deletion of data item (k,w) by $d_i[k,w]$. Transactions must also issue an explicit *begin action* b_i before performing any other operations.

Updating an existing data item is often not defined as a separate action, but rather implemented by first deleting the existing data item and then inserting the new data item to replace the old one. For convenience and simplicity, we define a *write action* to either insert a new data item, if no data item with the same key already exists; or to first logically delete an existing data item and then replace the deleted data item by a new one. A write action of transaction T_i inserting a data item (k,w) is denoted by $w_i[k, w]$, and a write action replacing an existing data item (k, w') by (k, w) is denoted by $w_i[k, w', w]$. We do not equate the term "update" with the write action, because a key deletion is also an update action. An update action can thus refer to any data-item modifying action. We will follow this combined model with read, write, and delete actions in this dissertation. When sufficient, we will use shortened forms $r_i[k, \theta x]$ (or $r_i[k]$ as a shorthand for $r_i[k, =k]$) to denote item retrieval, $w_i[k]$ to denote the write action, and $d_i[k]$ to denote the delete action; possibly omitting the indices if discussing only a single transaction.

Transactions that only consist of data-item retrieval actions are called *read-only transactions*, and transactions that include write and delete actions (as described above) are called *updating transactions*. Updating transactions modify the same data set, in the order the update actions are issued. Concurrency-control and recovery algorithms are used to maintain the ACID properties of transactions: atomicity, consistency, isolation, and durability [32]. If a database management system supports the serializable isolation level, the actions of the transactions will be executed in such an order that the outcome is the same as if the transactions had executed in a serial order, each transaction in its entirety in isolation from the others.

Transaction isolation in databases is achieved by using a concurrency-control algorithm, such as key-value locking (also known as key-range locking [32, 63]). The ANSI/ISO SQL-92 specification [3] defines four isolation levels based on the anomalies that can occur during the execution of concurrent transactions. The isolation levels are named (1) read uncommitted, (2) read committed, (3) repeatable read, and (4) serializable. The initial definition of the phenomena, however, fails to properly classify the different isolation levels. Berenson et al. propose an updated model for the isolation levels [11]. The anomalies, or phenomena, that cannot occur in each of the levels in the updated model are (1) dirty writes, (2) dirty reads, (3) fuzzy reads, and (4) phantoms. The phenomena exclusion is additive, so that transactions operating on the repeatable read level must not encounter dirty writes, dirty reads, or fuzzy reads; but may encounter phantoms. Only the serializable isolation level avoids all the anomalies, and thus guarantees transaction isolation.

The serializable isolation level is implemented by key-value locking, for example. In fact, phantom avoidance cannot be achieved by locking schemes that only lock those key values that have been read or written. The key-value locking approach avoids phantoms by locking both the accessed key and the next key found in the database [63, 64], thus in effect locking the range of keys from the accessed key to the next existing key. Another approach for avoiding phantoms is to use predicate locking, but this is often unpractical as predicate satisfiability in general is a known NP-complete problem and thus the approach can be highly inefficient [32]. A drawback of the serializable concurrency-control algorithms is that they hinder the overall performance of the database system by forcing other transactions to wait for access to locked keys. Database management systems thus allow the user to set some transactions to run on a lower isolation level to enhance the performance of the system.

Recovery is another important aspect of database systems. Applications rely on databases to maintain their data and to make sure that the data is available and consistent even after a system crash. The standard recovery algorithm is the ARIES algorithm (*Algorithm for Recovery and Isolation Exploiting Semantics* [63–66]), which was developed by C. Mohan at IBM. The ARIES algorithm is used in many commercial database systems, such as IBM DB2 and Microsoft SQL Server [65]; and it is taught in database courses in many universities, including ours. The algorithm maintains data integrity even in the presence of system crashes, and guarantees that the data set contains all the updates of all committed transactions, and none of the updates of transactions that were aborted or had not committed before the system crashed. In our discussion, we will assume the standard write-ahead-logging (WAL [66]) and steal-and-no-force page-buffering policy [32]. Under these policies, the buffer manager may steal unfixed dirty pages from the page buffer and flush them to disk, possibly causing updates of uncommitted transactions to be written to disk (*steal*). When a transaction commits, the log is forced to the disk, but the dirty pages are not (*no-force*). Whenever writing a data page to the disk, the log file must first be written so that all the updates in the disk versions of the data pages are also present in the disk version of the log, as dictated by the write-ahead-logging policy. Furthermore, the log file must be flushed to disk whenever committing a transaction, at least up to and including the log entry for the commit action. These requirements are necessary to ensure that database recovery can bring the database to a consistent state after a system crash.

2.2 Different Concepts of Time and History

The concept of time in connection with database theory has multiple definitions. The latest consensus glossary of temporal database concepts [14] presents three different definitions for time: valid time, transaction time, and user-defined time. The first of these, *valid time*, is the time when a recorded fact (data item) is true in the modelled reality. Valid time is usually provided by the user, and it may be changed later if the corresponding real-life fact changes. For example, events in a calendar database have starting times and ending times. These times may naturally be changed if the event is rescheduled to another time, and the time of the scheduled event is thus an example of valid time in a database. It should be noted that the valid time may also be a single time instant, instead of a time range. Any change in a valid-time database overwrites the information about the earlier state of the database, in the same way as in snapshot databases. An example of a transaction operating in a valid-time database is shown in Figure 2.2.

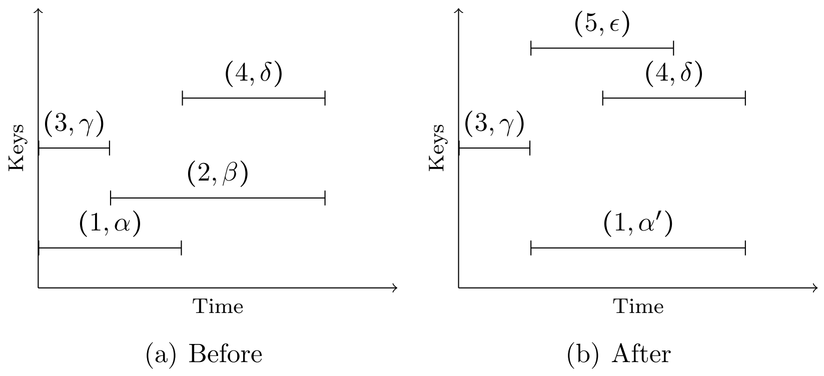


Figure 2.2. Transaction in a valid-time database. The transaction has executed the schedule $bw[1,\alpha,\alpha']d[2]w[5,\epsilon]c$.

The second time concept, *transaction time*, is used to record the time when a data item is current in the database and may be retrieved. The transaction time of a data item is always an interval which begins when the item is stored in the database. Initially, the end of the transaction-time interval is undefined, and the item is retrievable in all transaction-time instants starting from the instant the item was inserted into the database. Transaction time is always provided by the database management system, and it must be an increasing value so that the ordering of the transaction-time instants is consistent. The transaction-time interval of a data item is called a life span, and we give a formal definition below:

Definition 2.1. The *life span* of a data item is a closed-open transaction-time interval $\vec{v} = [v_1, v_2)$. The time instant v_1 is called the *creation time* of the data item, and it specifies the transaction-time instant when the item was inserted to the database. Similarly, v_2 is called the *deletion time* of the data item, and it specifies the transaction-time instant when the item was deleted. If the data item has not been deleted, $v_2 = \infty$. □

In transaction-time databases, all the data items have an associated life span, and we call them multiversion data items to distinguish them from the data items of the logical database. The formal definition of a multiversion data item is given later (in Definition 2.7), but from now on, we will use the term *data item* to mean a multiversion data item. When we wish to refer to the data items of a snapshot database (i.e., data items without associated life spans), we will use the term *snapshot data item.*

In contrast to snapshot and valid-time databases, any change in a transaction-time database creates a new state and thus does not overwrite any previous states. An example of this is shown in Figure 2.3.

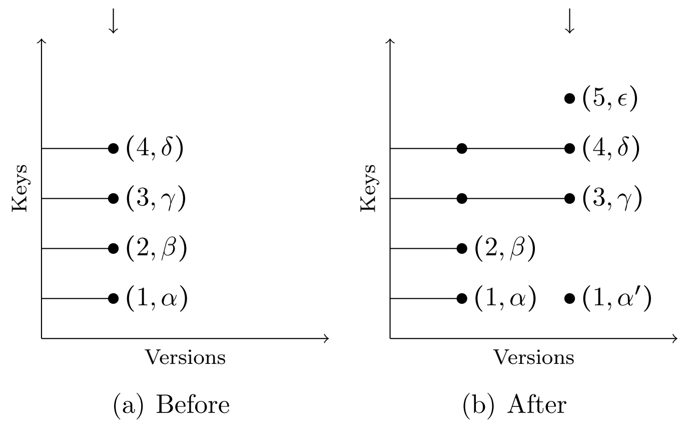


Figure 2.3. Transaction in a transaction-time database. The transaction has executed the schedule $bw[1,\alpha,\alpha']d[2]w[5,\epsilon]c$. The arrow on the top shows the latest committed version v_{commit}.

The transaction-time instants are often called *database versions*, as the contents of the database item set may only change between each transaction-time instant. Because this term is used by many authors, we will use the term *version* from now on to mean transaction-time instants. Moreover, we will use the term *alive* to mean that a data item is part of the current data set. This is defined more formally for different versions below:

Definition 2.2. A data item is *alive at version* v, if the life span of the data item covers v; that is, if $v_1 \leq v < v_2$, where $[v_1, v_2)$ is the life span of the data item. Queries that target a version v only return items that are alive at version v. □

If $v < v_1$ or $v \geq v_2$, the data item is not part of database version v. We call the most recent committed database version the *current version* of the database. This version is denoted by v_{commit}.

Definition 2.3. A data item is *alive* if it is alive at version v_{commit}. □

The data items that are alive at version v form a *database state* s_v. A transaction-time database thus consists of a series of consecutive states $s_0, s_1, \ldots, s_{v_{commit}}$. Data items that belong to database state s_v automatically belong to the next state s_{v+1}, unless they are explicitly deleted at version $v + 1$. The state $s_{v_{commit}}$, identified by the current database version v_{commit}, is called the *current state*, and all the states $s_v : v < v_{commit}$ preceding the current state are called *historical states*.

Efficient queries within the transaction-time domain or the valid-time domain require database index structures that are specifically designed for such queries. In contrast, the third time concept, the *user-defined time*, is an uninterpreted attribute used to refer to any application-specific time value. The user-defined time has no special meaning in the database management system. Database systems are thus categorized by their ability to index valid time and transaction time. Valid-time databases can be used to index events that occur at some time in the modelled reality. Database systems that can be used to access historical states are called transaction-time databases. Database systems that encompass both of these time dimensions are called *bitemporal databases*. This taxonomy of the time dimension was first presented by Snodgrass and Ahn [85]. The terms transaction time, valid time, and user-defined time were introduced to replace the previous, vague terms *physical* and *logical time*. Finally, we will use the term *real time* to mean the time of the real world (often called user time or wall-clock time). In practice, valid time is often the same as real time, but it can also be used to model some other time domain. For example, consider a valid-time database used in a science-fiction game—the time in the modelled virtual universe would be the valid-time domain in this case.

An example of the possible contents of a bitemporal database, shown in Figure 2.4, clarifies the difference between the time concepts. This example has been adapted from the taxonomy of time by Snodgrass and Ahn [85]. Reading from the contents of the example database, Merrie was appointed as an associate professor 1st January 2001, but this fact was entered into the database earlier on, on the 20th December. Merrie was promoted to full professorship 1st May 2003, the fact of which was retroactively inserted into the database on the 5th. Associate professor Tom's employment begun 10th March 2003, which was tentatively recorded to the database on the 7th of March. On the 13th, it was however noted that he had been accidentally recorded as being a full professor, and the fact was then corrected. Mike left the faculty 16th May 2005, which was

Name	**Rank**	**Valid time**		**Transaction time**	
		From	To	From	To
Merrie	Associate	2001-01-01	∞	2000-12-20	2003-05-05
Merrie	Associate	2001-01-01	2003-05-01	2003-05-05	∞
Merrie	Full	2003-05-01	∞	2003-05-05	∞
Tom	Full	2003-03-10	∞	2003-03-07	2003-03-13
Tom	Associate	2003-03-10	∞	2003-03-13	∞
Mike	Assistant	2004-03-22	∞	2004-03-24	2005-05-17
Mike	Assistant	2004-03-22	2005-05-16	2005-05-17	∞

Figure 2.4. Examples of the use of valid time and transaction time in a bitemporal database.

recorded to the database on the 17th. Note that only the bitemporal database model records all these aspects of the evolution of the employment data.

Yet another aspect of temporal databases is data persistence. In contrast to other database types, transaction-time databases and bitemporal databases offer more in terms of traceability and accountability, because the entire history of database states is stored and available for querying. By the terminology of Driscoll et al. [24], transaction-time databases and bitemporal databases are called *persistent databases*, whereas snapshot databases and valid-time databases are called *ephemeral databases*.

Intuitively, when using a transaction-time database to model changes in reality, a linear version history is often appropriate. In this model, a new database version is always based on the current version of the database:

Definition 2.4. A *partially persistent* transaction-time database incorporates a linear history, in which new versions of data items are created based on the most recent version. □

Many engineering applications, such as version control systems and engineering design databases, require that new versions can be based on any earlier version:

Definition 2.5. A *fully persistent* transaction-time database system allows new versions to be created based on any earlier version, thus enabling the creation of branching and diverging histories. □

Note that an implication of this definition is that every fully persistent database system is necessarily also a partially persistent database system.

Most of the temporal database systems are designed on the partially persistent transaction-time model. These database systems are called *multiversion databases* by many authors [8, 10, 43, 58, 92]. From now on, we will be discussing multiversion databases based on this definition. Fully persistent database systems are discussed by other authors [50, 77]. In fully persistent databases, a central challenge is finding out whether two different versions are in the same version branch in the version history. Salzberg and Lomet [77] suggest using sequence numbers to identify data-item versions. Their index structure assumes a small number of different branches. Each data item is assigned a version and a branch identifier, and the branch identifier is used to check whether a data item is an ancestor of another data item. This check requires at most $\mathcal{O}(n)$ operations, where n is the number of branches in the system. Landau et al. [50] use a history tree with arrays that store change information in each node of the history tree. Searches along a history branch first locate the queried version, and then reconstruct the database state by combining the change sets of the located node and its ancestor nodes. Landau et al. note that an index structure such as the snapshot index of Tsotras and Kangelaris [90] (see Section 3.5, p. 43) could be used to make the state reconstruction more efficient.

2.3 Query Types

An important property of temporal databases is the different types of queries that can be run on them. Ideally, the user should be able to define any bounds for the key and the transaction time of the sought data item in a multiversion database. The corresponding ideal model allows the user to specify the key, the transaction time, and the valid time in a bitemporal database. All of these specified properties can be either intervals or points in the corresponding dimension. Most temporal databases, however, restrict the query types that are possible for the database. Tsotras et al. [89] define a notation for specifying the query types that can be run on different kinds of databases. We use the notation for temporal databases that was adapted by Salzberg and Tsotras in their temporal database comparison [78]. The notation is *key/valid/transaction*, where *key* means the key dimension, *valid* means the valid-time dimension, and *transaction* means the transaction-time dimension. Each part of the query type may be either *point*, *range*, $*$, or $-$. Here, *point* means that a single point in the corresponding dimension is queried, *range* means that a range of the appropriate dimension is queried, $*$ means that all values in the dimension are to be included in the result, and – means that the dimension does not apply to this database. Conceptually, $*$ is the same as defining a *range* that covers all the values in the corresponding dimension.

On an ideal multiversion database, the database user should be able to run any queries of the form $x/–/x'$, where x and x' are both either *point*, *range*, or $*$. Most multiversion databases, however, restrict the query type to $x/–/point$ [8, 45, 54, 58]. This means that queries may target data items with different keys, but only within a single version of the database at a time. In the terms of the consensus glossary [14], the multiversion databases only allow queries that apply a *transaction-timeslice operator* with an *instant-type* argument. In our dissertation, we will assume the query type $x/–/point$ as the basis of our multiversion data model. In this model, the user always supplies a single version when querying on the database. For discussion on the $x/–/x'$ query type for the MVBT index structure (see Section 4.4), we direct the reader to the article on query processing techniques by van den Bercken and Seeger [10].

2.4 Representing Multiversion Data

At this point, we can formally define a multiversion database:

Definition 2.6. A *multiversion database* is a transaction-time database that is partially persistent and enables efficient $x/–/point$ queries on data items, where x is either *point*, *range*, or $*$. The transaction-time instants of the data items in a multiversion database are called *database versions*. The versions of the multiversion database are ordered based on the commit-time ordering of the transactions. □

The versions of a multiversion database do not necessarily directly map to any real-time instants. In fact, using consecutive integer values for database versions has the advantage that the next version can always be easily determined. Users of the system, however, often wish to query past states based on real-time instants. There are two intuitive approaches for overcoming this discrepancy. In the first approach, database versions are increasing integer values, and a separate mapping between versions and real time is maintained in the database system. In the other approach, database versions are timestamps that are based on real time. In this approach, given a version v_1 that was used by transaction T_{v_1}, it is not possible to directly determine the next version v_2 used by the next transaction T_{v_2}, unless all the different versions are separately stored. Salzberg and Lomet [77] suggest using the former approach. They use an array that maps the sequence numbers into real-time instants. Converting from database versions to real time is straightforward, as the array can be directly accessed from the right offset. To convert back from real-time instants to database versions, a binary search can be used, as the real-time instants are also sorted in increasing order. We will assume that this approach is used in the algorithms presented in this dissertation.

The ordering of the database versions must be based on the commit-time ordering of transactions to ensure consistency of the data items between different versions. An ordering that is based on the starting time of transactions, for example, guarantees consistency only if the transaction commit order is forced to be the same as the starting order. Because we do not wish to impose such a requirement, the data items created by a transaction must receive the commit-time version of the transaction, which may differ from the starting time of the transaction. Because the commit-time version is not known at the beginning of the transaction, active transactions must use a separate identifier to identify their updates. Following Lomet et al. [55], we assume that a transaction T is assigned a *transaction identifier*, denoted by $\mathsf{id}(T)$, when it is created. When the transaction T commits, a *commit-time version*, denoted by $\mathsf{commit}(T)$, is assigned to the transaction. The commit-time version is the version used by the database to order the data items, and users of the database system use it to query previous database states. In contrast, the transaction identifiers are internal to the database system and not seen by the users.

The discrepancy between the transaction identifiers and the commit-time versions is a common challenge in all multiversion databases. Often the entries that represent the data items need to be initially stored with the transaction identifier, and later on revisited to change the transaction identifier into the commit-time version. A *lazy timestamping* scheme [55] initially uses the transaction identifier to store the versions and lets the transaction commit. Later, when the entries are accessed, the transaction identifiers are changed to the commit-time version of the transaction. In contrast, in an *eager timestamping* scheme [55], the temporary identifiers are changed as soon as the transaction commits.

The data-update model adopted by most earlier proposals for indexing versioned data, including the model adopted for the multiversion B^+-tree (MVBT) of Becker et al. [8] and the multiversion access structure (MVAS) of Varman and Verma [92], assumes that each update action creates a new version of the database, so that the versions are unique across all versions of all data items. This model is not adequate for modern transactional applications, because data items within a single transaction should be assigned the same version. Following Salzberg et al. [76]

and Lomet et al. [55], we assume a multi-action-transaction approach in which all data-item versions created by a transaction T get the same commit-time version $\mathsf{commit}(T)$, unless the transaction updates a data item more than once, in which case only the final data-item version gets the commit-time version and remains in the database.

We will now formalize the notion of data items in a multiversion database. Remember from Section 2.1 that the logical database consists of tuples of the form (k, w), where k is the key of the data item, and w the value associated with the item. Because the data items need an additional version attribute—the life span $\vec{v}$ (Definition 2.1)—the data item model used in snapshot databases is not sufficient for multiversion databases.

Definition 2.7. A *multiversion data item* in a multiversion database is a tuple of the form $(k, \vec{v}, w)$, where k and w correspond to the key and value of the snapshot data items of the logical database, and $\vec{v}$ is the life span of the multiversion data item. The life span $\vec{v}$ is either $[v_1, \infty)$, if the data item has not been deleted; or $[v_1, v_2)$, if the data item has been deleted. The version v_1 is the commit-time version of the transaction that inserted the data item, and version v_2 is the commit-time version of the transaction that deleted the data item. □

Definition 2.8. The updates performed by an active updating transaction T are called *pending updates*, and they are represented by tuples of the form $(k, \mathsf{id}(T), \delta)$, where k is the key, $\mathsf{id}(T)$ is the transaction identifier, and δ tells whether the update is an insertion or a deletion. In the case of an insertion, $\delta = w$, the value of the data item; and in the case of a deletion, $\delta = \bot$, a special marker value used to denote item deletion. □

In the logical database model, the pending updates created by a transaction T are incorporated into the database immediately when T commits, either by inserting new data items or by updating the existing data items. Depending on the implementation of the multiversion database, the entries that are used to store the pending updates may exist for some time after the transaction has committed. In this situation, the data-item entries in the database may differ from the data items of the logical database, and the database management system must be prepared to apply the effects of the pending updates on the existing data items during the execution of database queries.

With this definition, the logical data items can be identified by the key-life-span pair $(k, \vec{v})$, or $(k, [v_1, v_2))$. However, we can also uniquely identify a data item using only the creation time of the data item, that is, the version v_1. In this dissertation, we will use the pair (k, v_1) to uniquely identify a data item $(k, [v_1, v_2), w)$. This is possible, because the life spans of data items with the same key k cannot overlap. We also define all other pairs (k, v), with version v in the life span of the data item, $v_1 < v < v_2$, to refer to the same data item. With this definition, users of the database system may query a data item using any version that is covered by the life span of the data item.

2.5 Read-Only Transactions

A fully concurrent multiversion database management system allows any number of read-only and updating transactions to operate on the database concurrently. As explained in Section 2.3, the read-only transactions must explicitly specify which version they want to read; however, they are only allowed to read versions that were already committed when the read-only transaction began. In our model, database users must supply the queried version at the beginning of the read-only transaction. All the queries of the transaction then target the same version of the database. We call this version, associated with transaction T, the *snapshot time* of T, and denote it by $\mathsf{snap}(T)$.

A *read-only transaction* T operating on a multiversion database may contain the following actions:

- **begin-read-only**(version v): begins a new read-only transaction; this action records the value $\mathsf{snap}(T) \leftarrow v$ for the transaction. At this point, it is also checked if $v \leq v_{commit}$. If $v > v_{commit}$, the transaction is either aborted or blocked until $v \leq v_{commit}$.
- **query**(key k) $\rightarrow \gamma$: retrieves the data item $(k, [v_1, v_2), w)$ that covers version $\mathsf{snap}(T)$, that is, $v_1 \leq \mathsf{snap}(T) < v_2$. Returns $\gamma = w$ if such a data item exists, otherwise returns $\gamma = \emptyset$.
- **range-query**(range $[k_1, k_2)$) $\rightarrow \Gamma$: this action retrieves the set of data items $(k_i, [v_i^1, v_i^2), w_i)$ with $k_1 \leq k_i < k_2$ and $v_i^1 \leq \mathsf{snap}(T) < v_i^2$. Returns the set Γ of snapshot data items (k_i, w_i) representing the multiversion data items alive at the queried version $\mathsf{snap}(T)$.
- **commit-read-only**: commits the transaction T by removing it from the system.

A read-only transaction that is aborted by the system may directly be removed from the database system; no undo actions are necessary. Read-only transactions in general do not involve any logging, because they perform no updates to the database.

An example of a read-only transaction, operating on the example multiversion database shown in Figure 2.5, is given below:

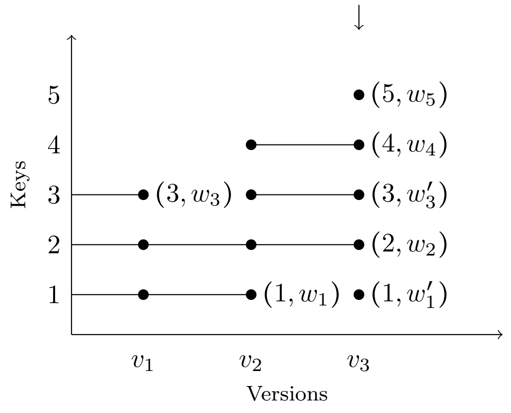


Figure 2.5. An example history of a multiversion database. The figure shows the contents of a database created by three transactions with commit-time versions v_1, v_2, and v_3. The arrow on the top shows the latest committed version $v_{commit} = v_3$.

1. **begin-read-only**(v_2)
2. **query**$(1) \to w_1$
3. **query**$(3) \to w_3'$
4. **query**$(5) \to \emptyset$
5. **range-query**$([0, 10)) \to \{(1, w_1), (2, w_2), (3, w_3'), (4, w_4)\}$
6. **commit-read-only**

2.6 Updating Transactions

All updating transactions operating on multiversion databases must always operate on live data items. Standard multiversion concurrency-control algorithms are used to maintain data consistency, so that concurrent updating transactions do not update the same data items.

As discussed in Section 2.4, data items created by and existing data items deleted by an updating transaction T must eventually receive the commit-time version $\mathsf{commit}(T)$ as the version associated with the data item. For a write action, the new data item must have $\mathsf{commit}(T)$ as its creation time, and for a delete action, the deleted data item must have $\mathsf{commit}(T)$ as its deletion time (see Definitions 2.1 and 2.7).

Because the commit-time version of a transaction T is not known during the execution of T, the updates of T are represented by pending updates (see Definition 2.8) that use the transaction identifier $\mathsf{id}(T)$

to separate the updates of different transaction. When the transaction T commits, the database management system performs a *release-version* action that incorporates the pending updates into the database, using the commit-time version $\mathsf{commit}(T)$. The committed version $\mathsf{commit}(T)$ is visible to other transactions after the release-version operation has completed.

An *updating transaction* T operating on a multiversion database may contain the following actions:

- **begin-update**: begins a new updating transaction; this action records the snapshot version $\mathsf{snap}(T) \leftarrow v_{commit}$, and creates the transaction identifier $\mathsf{id}(T) \leftarrow$ *new identifier*.
- **query**(key k) $\rightarrow \gamma$: if the transaction T has already performed an update action on the key k, so that a pending update $(k, \mathsf{id}(T), \delta)$ exists, this action returns either $\gamma = \delta$ if the pending update was a write action, or $\gamma = \varnothing$ if the pending update was a deletion (i.e., $\delta = \perp$). If such a pending update does not exist, the action retrieves the data item $(k, [v_1, v_2), w)$ such that $v_1 \leq \mathsf{snap}(T) < v_2$. If such an item is found, the action returns $\gamma = w$; otherwise the action returns $\gamma = \varnothing$.
- **range-query**(range $[k_1, k_2)$) $\rightarrow \Gamma$: this action retrieves all the pending updates $(k_i, \mathsf{id}(T), \delta_i)$ and all data items $(k_i, [v_i^1, v_i^2), w_i)$ such that $k_1 \leq k_i < k_2$ and $v_i^1 \leq \mathsf{snap}(T) < v_i^2$. For each key k_i, if a pending update for that key is found, the return set Γ contains the data item (k_i, δ_i), if $\delta_i \neq \perp$. For each key k_i, such that there is no pending update for that key but there is a data item with key k_i, the return set Γ contains the snapshot data item (k_i, w_i). A pending update that is a deletion thus prevents the deleted key from appearing in the result set.
- **write**(key k, data w): a forward-rolling action that either inserts a pending update $(k, \mathsf{id}(T), w)$ into the database, if no earlier pending update of the form $(k, \mathsf{id}(T), \delta)$ exists; or replaces the existing pending update $(k, \mathsf{id}(T), \delta)$ with $(k, \mathsf{id}(T), w)$. This action writes a redo-undo log record that contains sufficient information for redoing or undoing the write action.
- **delete**(key k): this action logically deletes an existing data item. If the multiversion database contains a pending update of the form $(k, \mathsf{id}(T), \delta)$, $\delta \neq \perp$, this action replaces it with $(k, \mathsf{id}(T), \perp)$. If the database does not contain any pending update $(k, \mathsf{id}(T), \delta)$, then this action is legal if the multiversion database contains a live

data item $(k, [v_1, \infty), w)$. In this case, the action inserts a pending update $(k, \mathsf{id}(T), \perp)$. The action finishes by writing a redo-undo log record that contains sufficient information for redoing or undoing the deletion.

- **set-savepoint** $\to p$: creates a savepoint by generating a savepoint identifier p and by returning it to the transaction.
- **rollback-to-savepoint**(savepoint p): rolls the transaction back to a preset savepoint p. This action is followed by the **undo-write** and **undo-delete** actions for the **write** and **delete** actions done after setting savepoint p, executed in the reverse order.
- **commit-update**: commits an active updating transaction. This action generates a commit-time version and assigns it to the transaction by setting $\mathsf{commit}(T) \leftarrow$ *new commit-time version*; updates the current-version counter $v_{commit} \leftarrow \mathsf{commit}(T)$; and invokes the **release-version** method to incorporate the pending updates into the database.
- **release-version**: applies the pending updates performed by the transaction T to the multiversion database. For each write action described by a pending update $(k, \mathsf{id}(T), w)$, the action retrieves the live data item $(k, [v_1, \infty), w')$, if such an item exists. If such a live data item exists, it is replaced by $(k, [v_1, \mathsf{commit}(T)), w')$. Finally, this action inserts a data item $(k, [\mathsf{commit}(T), \infty), w)$ into the multiversion database. For each delete action represented by a pending update $(k, \mathsf{id}(T), \perp)$, the action retrieves the live data item $(k, [v_1, \infty), w')$, if such an item exists. If such an item is found, this action replaces it by $(k, [v_1, \mathsf{commit}(T)), w')$. No such item is necessarily found, if the transaction first inserted a new data item and then deleted it. In this case, no action is taken for this pending update. Finally, this action removes all the pending updates created by T—i.e., pending updates of the form $(k, \mathsf{id}(T), \delta)$—from the database.
- **abort**: labels the updating transaction as aborted and starts the backward-rolling phase. This action is followed by the **undo-write** and **undo-delete** actions for all the not-yet-undone **write** and **delete** actions of the forward-rolling phase, executed in reverse order.
- **undo-write**(log record r): backward-rolling action that undoes the write action logged with the log record r by either removing the pending update $(k, \mathsf{id}(T), w)$; or by replacing it with $(k, \mathsf{id}(T), \delta)$,

if the corresponding write action replaced an earlier update by the same transaction.

- **undo-delete**(log record r): backward-rolling action that undoes the delete action logged with r by either removing the pending update $(k, \mathsf{id}(T), \perp)$; or by replacing it with $(k, \mathsf{id}(T), w)$, if the corresponding delete replaced an earlier update by the same transaction.
- **finish-rollback**: finishes the backward-rolling phase of an aborted updating transaction.

An example of an updating transaction, operating on the example database shown in Figure 2.5 in the previous section, is given below:

1. **begin-update**
2. **query**$(1) \rightarrow w'_1$
3. **range-query**$([0, 10)) \rightarrow \{(1, w'_1), (2, w_2), (3, w'_3), (4, w_4), (5, w_5)\}$
4. **delete**(4)
5. **set-savepoint** $\rightarrow p_1$
6. **write**$(6, w_6)$
7. **range-query**$([0, 10)) \rightarrow \{(1, w'_1), (2, w_2), (3, w'_3), (5, w_5), (6, w_6)\}$
8. **rollback-to-savepoint**(p_1)
9. **undo-write**(r)
10. **range-query**$([0, 10)) \rightarrow \{(1, w'_1), (2, w_2), (3, w'_3), (5, w_5)\}$
11. **commit-update**

The form of transactions in multiversion databases is a generalization of the recoverable snapshot database theory presented by Sippu and Soisalon-Soininen [84], with the addition of setting and restoring savepoints as presented for B-trees in page-server database systems by Jaluta et al. [41]. To abort and rollback an entire transaction in this model, it is possible just to remove all the pending updates of the form $(k, \mathsf{id}(T), \delta)$ from the database. Rolling back to a predefined savepoint, however, requires that the undo actions are properly defined and that logging is used so that it is possible to restore the earlier pending updates by the same transaction that have been overwritten. The logging of the actions is dependent on the implementation. Examples of how the actions can be logged are given in Chapters 5 and 6.

2.7 Concurrency Control

Transaction isolation in multiversion database systems can be maintained by traditional concurrency-control algorithms, but there are multiversion concurrency-control algorithms that are better suited to the task. These algorithms take advantage of the different versions of the data items stored in the database. Each data-item update creates a new version of the item, and the different versions persist in the database system, for at least as long as the transaction that created the data item is active. This is a requirement for using multiversion concurrency-control algorithms in snapshot databases. Read-only transactions may thus run queries without acquiring locks on keys, because writes by other transactions do not modify the versions that are being read. Early multiversion concurrency-control algorithms include multiversion timestamping, multiversion locking and the multiversion mixed method. These are presented, for example, in an article by Bernstein and Goodman [12] and in the textbook of Bernstein et al. [13]. These algorithms generally assign unique timestamps to transactions and store timestamp-identified copies of any modified data items. These copies may be overwritten only after the transaction that created them has committed.

The *multiversion timestamping method*, or *multiversion timestamp ordering*, was originally presented by Reed in his PhD thesis [75]. This concurrency-control algorithm uses only item timestamps to control access to items; the transactions take no locks on keys. Modifications to data items create new versions of the items, and each transaction reads the most recent version that precedes the transaction's own timestamp. Here we assume that the transaction identifiers are based on the starting time of the transaction, and that the timestamp of a transaction T is thus $\mathsf{id}(T)$. The data-item timestamps can be used to ensure that the transactions operate in a serializable isolation level. When a transaction T_i issues a write of value w to key k, the following is considered: if there is a transaction T_j with a larger timestamp $\mathsf{id}(T_j) > \mathsf{id}(T_i)$ that has already read the earlier value w' of key k, then the write of T_i is not allowed, and T_i is aborted, or the write action is simply denied. Additionally, the commit of transaction T_i must be delayed until all transactions T_k that have written data that T_i has read have committed. If any of the transactions T_k aborts, T_i must also be aborted. By denying such writes-after-reads, the multiversion timestamping method can be proven to form serializable histories [12, 13].

The *multiversion locking method* is another approach that uses locking, and not timestamps, to manage concurrent access to the database.

It is a multiversion method, however, because more than one version of data items is stored. The multiversion locking method marks item modifications first as *uncertified*, and later on tries to *certify* all of them. The transaction may commit only if the certification is successful. Bernstein and Goodman [12] discuss a general algorithm for multiversion locking; whereas in their textbook [13] they describe the two-version two-phase locking algorithm (2V2PL). In 2V2PL, transactions acquire read and write locks as in two-phase locking, but the write locks only conflict with each other, not with read locks. For each data item, a single extra version is kept stored in the database (the uncertified update; there can be no more than two uncertified updates at a time because of the write locks). When a transaction commits, it tries to upgrade all of its write locks into *certify locks*. The certify locks are defined to conflict with all other lock types, including read locks. If other transactions thus hold any read locks on any keys that the committing transaction has updated, the commit operation is delayed until the conflicting transaction commits and releases the locks. This can naturally lead to deadlocks, which must then be detected and resolved by aborting one of the transactions. The transactions executing under multiversion locking algorithms also form serializable histories.

The *multiversion mixed method* uses both timestamping and locking to control the actions of transactions. In short, read-only transactions are controlled with multiversion timestamping, and updating transactions with multiversion locking. That is, read-only transactions are assigned a timestamp that is smaller than the timestamp of any active uncertified transaction. The read-only transactions may then directly read the latest version preceding their assigned timestamps, without needing to lock any keys. The updating transactions, on the other hand, lock updated keys like in the multiversion locking approach. This time, however, the data items are timestamped. As noticed in Section 2.4, the items must be timestamped according to their commit-time ordering. In the multiversion mixed method, the timestamps are assigned when the transaction commits (after the updates have been verified). Bernstein et al. [13] also suggest replacing the timestamps by commit lists so that the items do not need to be revisited. However, this approach requires tracking and passing around commit lists that tell which transactions have committed.

A more recent multiversion concurrency-control algorithm is the *snapshot isolation* algorithm, or SI [11, 19, 28], which is an extension of the multiversion mixed method. A transaction T executing under snapshot isolation obtains a timestamp at the beginning of the transaction, or at any time before the first read (or write) action. The transaction can only read data items with timestamps that precede the start timestamp of the transaction. When the transaction T commits, it gets a commit timestamp that is larger than any start timestamp or commit timestamp in the system. The updates made by the transaction must receive this commit timestamp, as noted before. The transaction will successfully commit only if no other transaction T' with $\mathsf{snap}(T) < \mathsf{commit}(T') < \mathsf{commit}(T)$ wrote data that T also wrote. If there is such a transaction T', then T must abort. This principle is called *first-committer-wins.* The original definition by Berenson et al. [11] notes that snapshot isolation precludes all the phenomena defined by ANSI SQL-92 [3], but is still weaker than some of the isolation levels defined by those phenomena. SI allows histories such as

$$b_1 r_1[x] r_1[y] b_2 r_2[x] r_2[y] w_2[x] c_2 w_1[y] c_1.$$

The history is accepted in SI, because the write sets of T_1 and T_2 are disjoint, but it is not serializable, because $r_1[x]$ is an unrepeatable read. This history is an example of a *write skew* which happens when transaction T_1 first reads x and y, which are consistent with a constraint C, and then T_2 reads x and y, writes x, and commits. If T_1 now writes y, the constraint C may be violated. Regardless of the non-serializability of snapshot isolation, it is widely used, and Oracle's implementation of the serializable isolation level is based on snapshot isolation [19, 28, 68]. Oracle thus needs to store the history of recent data item versions in data pages to be able to apply snapshot isolation.

Snapshot isolation can be made serializable, however. Fekete et al. [28] show how to analyze applications and to modify them so that they are equivalent but preclude the write skew anomalies. One such technique is to materialize the conflicts, i.e., to write the constraints into the database relation, thus enforcing write-write conflicts into the transactions. Alomari et al. [2] show that the modified applications may be less efficient, but the performance loss is negligible if the correct method is used. Cahill et al. [19] extend the snapshot isolation algorithm by adding book-keeping code to detect situations where non-serializable executions could occur. If such a situation is found, one of the transactions is aborted. This method guarantees serializability for all transactions. Cahill et al. also present test results that show that the performance of the serializable snapshot isolation algorithm is comparable to snapshot isolation, and always better than strict two-phase locking.

The definition of snapshot isolation [11] does not include any suggestions on how the isolation level should be enforced. Standard locking is inadequate, because a transaction T must abort if it tries to update a data item that has been updated by an overlapping but already committed transaction T'. Alomari et al. [2] describe how the PostgreSQL database implements snapshot isolation by having each transaction take exclusive locks on any modified data items; and by aborting any transaction T that tries to modify a data item for which the latest version $(k, [v_1, v_2), w)$ has been modified after transaction T begun, so that $v_1 > \mathsf{snap}(T)$.

We do not presuppose that snapshot isolation be used in connection with the database structures and algorithms presented in this dissertation. However, we do assume that such a multiversion concurrency-control algorithm is used that allows read-only transactions to operate without blocking updating transactions and vice versa. We will use the snapshot isolation algorithm in our examples.

2.8 Structure-Modification Operations

When a database page becomes filled with data items, it needs to be split into two separate pages using a page-split operation. Similarly, when enough data items in a page have been marked as deleted, the page needs to be merged with a sibling page using a page-merge operation to maintain an appropriate minimum number of live data items in each page. These operations are called *structure-modification operations*, or SMOs. In snapshot database theory, the algorithms of the ARIES family use *nested top actions* [64, 66] when executing structure-modification operations. When a structure-modification operation begins, the log sequence number (LSN) of the last action performed prior to the SMO is recorded. All the individual operations required by the SMO are logged using nested top actions. When the SMO finishes, the Undo-Next-LSN [64] of the last nested top action is set to the LSN value recorded at the beginning of the SMO, so that the action chain determined by the Undo-Next-LSN values skips the operations performed by the SMO, if the SMO has finished. The effect of this technique is that it breaks the standard backward chaining of log records, so that partial and total rollbacks skip the nested top actions. The SMOs are not undone even if the transaction that triggered the SMO aborts and is rolled back.

There are, however, some challenges with nested top actions. They often require tree latches or special structure-modification bits to be set on the database pages so that concurrent transactions can notice that a structure-modification operation is ongoing. They may also be undone if a system crash occurs before all the required nested top actions have finished. Jaluta et al. have proposed a more straightforward method for performing structure-modification operations for database index structures [39–41]. In their approach, each structure-modification operation targets a single page of the index structure, requiring modifications to be made to at most three pages at two adjacent levels (for the B$^+$-tree index): one parent page, and two sibling child pages. This also means that at most three different pages need to be latched at a time (again, for the B$^+$-tree; different index structures may require different number of pages to be latched at a time). If an SMO at a lower level causes an SMO in an upper level, the SMO at the upper level is applied before the SMO at the lower level.

Each structure-modification operation is logged using a single redo-only log record. In contrast to nested top actions, interrupted SMOs are never undone in this approach, and the index structure remains consistent after each SMO, so that each SMO transforms a structurally consistent and balanced index tree into another structurally consistent and balanced index tree. The log records must contain sufficient information of all the entries moved or copied between pages, so that the effect of the operation can be redone on any single page involved in the operation, as is required by the ARIES algorithm [66]. This approach does not require any special structure-modification bits or tree latches for applying the modification. We will use this approach to apply structure-modification operations to the index structures discussed in this dissertation.

CHAPTER 3

Multiversion Index Structures

We have now discussed the theory behind temporal databases, concentrating mostly on multiversion databases. In this chapter, we describe some of the index structures used in multiversion databases. We begin the chapter by defining a few general properties of multiversion index structures in Section 3.1. In Section 3.2, we will demonstrate that a single-version index is not an efficient structure for indexing multiversion data. To be able to properly determine the efficiency of multiversion index structures, Section 3.3 defines what we mean by an optimal multiversion index structure, and Section 3.4 lists common design ideas used in efficient multiversion indexes. In Section 3.5, we describe some of the early multiversion index structures. For a comprehensive presentation and comparison of different multiversion access methods, the reader is referred to Salzberg and Tsotras [78], and Özsoyoğlu and Snodgrass [94]. The rest of this chapter is dedicated to different kinds of structures that have been used to index multiversion data or are otherwise related: spatial indexes (Section 3.6), hashing structures (Section 3.7), version-control systems (Section 3.8), and other structures (Section 3.9).

3.1 Properties of Multiversion Indexes

To begin, let us define what we mean by a multiversion index structure:

Definition 3.1. A *multiversion index structure* is a transaction-time index that is partially persistent and enables efficient $x/\text{–}/point$ queries on the data items, where x is either *point*, *range*, or $*$. The index is a collection of nodes that forms a tree or a directed acyclic graph (DAG). The nodes of the graph are fixed-size database pages. The graph contains one or possibly many *root pages*, which serve as starting points for search operations. Pages that have child pages are called *index pages* or *parent* *pages*, and pages that do not have child pages are called *leaf pages*. Each page contains *entries* that represent either data items (called *data entries*, see Definition 2.7) or routers to child pages (called *index entries*). Page capacity B tells how many entries fit into the page. The capacity is dictated by the entry format and the page size, but for the simplicity of the theoretical discussion, we assume that the page capacity B is the same for all index and leaf pages. The data entries stored in the index may contain either the actual data stored with the key (the row in the relation), in the case of a *primary* or *sparse index*; or a pointer to a separate storage location, in the case of a *secondary* or *dense index*. □

The multiversion index structure defines the way the data items are stored and accessed. Similar to the B$^+$-tree, most often the index pages of a multiversion index contain only index entries, and the leaf pages contain only data entries. Searches in a multiversion index follow the same logic as searches in a single-version index structure: each node has a number of child nodes, and each child page covers a more restricted area of the search space. The search spaces of sibling pages usually do not overlap, but there are exceptions. In a multiversion index, the search space is the key-version space. Each page thus covers a region in key-version space. If the multiversion index contains a single root page, then that root page covers the entire key-version space. A child page's search-space region overlaps with the parent page's region, and often the child page's region is a subset of the parent page's region. A key k that is part of version v (alive at version v) is located at the leaf page whose key-version region covers the key-version coordinate (k, v).

In multiversion index structures, the most important property to optimize is the number of pages that an action needs to read or write to perform an action, because I/O operations on disk storage are still the most significant bottleneck in most database applications [44]. A good index structure requires a minimal number of page accesses for its actions. If a search operation on a database index requires access to m pages to locate key k, then m is normally logarithmic in the number of data items indexed by the structure, if the index is a tree structure. For analyzing the performance of index structures, we define the cost of an action:

Definition 3.2. The *cost* of an action or an operation is the number of index-structure pages the action needs to access (read and/or write). In the case of a sparse index, this includes all the pages the actions need to access. In the case of a dense index, this includes only the pages of the index structure itself, and not the data pages that may need to be accessed additionally for each data item. □

3.2 Versioned B$^+$-tree

The B-tree [5, 6, 23] is a widely used search tree structure that is optimized for use as a database index structure. A single node of the B-tree is stored in a single database page. Each database page is designed to fit in a single disk block (or a fixed number of consecutive blocks), so that reading and writing database pages is as efficient as possible with the underlying storage medium. Because disk block sizes range from 4 KiB to 64 KiB [81], the nodes of the B-tree have a huge number of children, ranging from hundreds to even thousands. For example, if a 4 KiB index page contains index entries that consist of a four-byte key separator and a four-byte child page identifier, the page can contain in the excess of 500 entries; an 8 KiB page can contain a thousand index entries, and so on. The fan-out of a B-tree is therefore high, and the trees tend to be very low. There are typically only three to five pages on a path from the root to a leaf page in even a very large database system.

The most widely used variant of the B-tree is the B$^+$-tree, which stores data entries only in leaf nodes. The index nodes of a B$^+$-tree thus contain only index entries. The leaf pages of a B$^+$-tree index are at level one, and index pages are at consecutively higher levels. The height of the B$^+$-tree is the level of its root page[1]. With this convention, an empty B$^+$-tree has a height of zero (no pages allocated[2]); a single leaf-page root page forms a B$^+$-tree of height one; and a B$^+$-tree with n levels of index pages and a single level of leaf pages forms a B$^+$-tree of height $n + 1$. A standard B$^+$-tree index stores entries of the form (k, w) in its leaf pages and entries of the form (k, p) in its index pages, where k is the data item key, w is the data value, and p is the page identifier of a child page that resides at the next lower level. The key k used in the index entries is also called a *router* that directs the search to the correct child page. The data value w is either the value itself, in the case of a sparse index; or a pointer to where the data is stored (a record identifier of the data), in the case of a dense index. The entries are ordered by the key k. An update on a data item in a standard B$^+$-tree is performed by physically deleting the old entry from the index, and by inserting a new entry to replace the old one. No version history is recorded in a B$^+$-tree. However, with a slight modification to the entry structure we can record the history of data-item changes in the B$^+$-tree index.

[1]There are overlapping definitions for tree height and page levels; we adopt the convention used by Bayer [5] because it seems most natural.

[2]For practical efficiency reasons, an empty root page might need be kept allocated for an empty B$^+$-tree index.

As discussed in Section 2.4, multiversion data items are tuples of the form $(k, \vec{v}, w)$, where $\vec{v}$ is the life span of the data item. Furthermore, the data items can be uniquely identified by the key-version pair (k, v_1), when $\vec{v} = [v_1, v_2)$. An update to a database index is logically either a key insertion or update (a write action), or a key deletion (a delete action); recall the update model from Section 2.6. The multiversion history can thus be stored in a B$^+$-tree index if we simply change the data entry format to (k, v_1, w) for a write action and $(k, v_2, \perp)$ for an item deletion. The version v_1 represents the commit-time version of the transaction that inserted the new value w, and v_2 represents the commit-time version of the transaction that performed the deletion. In this convention, each deleted multiversion data item (i.e., a multiversion data item with a life span $\vec{v} = [v_1, v_2) : v_2 \neq \infty$) is represented by two entries: one marking the insertion of the multiversion data item, and one the deletion or updation of the item. We call this extended B$^+$-tree structure the *versioned B$^+$-tree*, or VBT for short. The entries in a VBT are ordered first by the keys, and then by the versions, so that $(k, v, w) < (k', v', w')$ if either $k < k'$ or $k = k' \wedge v < v'$. This defines a total ordering, because entries are uniquely identified by the pair (k, v), so that no two updates can have the same key and version.

Because the entries are now ordered by the key-version pairs (k, v), the index entries in index pages need to reflect this. The index entries of a VBT are therefore of the form (k, v, p), where p is the identifier of a child page. As with a B$^+$-tree, the (k, v) pairs in index entries are separator values that are used to direct searches. If an index page contains n entries (k_i, v_i, p_i), with $i \in \{1, 2, \ldots, n\}$, and $(k_j, v_j) \leq (k, v) < (k_{j+1}, v_{j+1})$, then the page identifier p_j is the identifier of the child page whose key-version range contains the entry (k, v). We define $(k, v) \leq (k', v')$ if either $(k, v) < (k', v')$ or $k = k' \wedge v = v'$. The key-version pair (k_i, v_i) in an index entry thus defines a lower limit for key-version pairs in the child page pointed to by the index entry, and the key-version pair (k_{i+1}, v_{i+1}) in the next entry defines the upper limit. This could as well be defined the other way around, so that the key-version separator value stored with an index entry defines the upper limit for the page pointed to by that entry; and the separator value stored in the previous entry defines the lower limit.

Searching for a single key k at version v in the VBT is performed by locating the entry (k, v', δ) with the largest v' such that $v' \leq v$. If no such entry is found, or if $\delta = \perp$, then the query should return $\varnothing$ to indicate that no entry for the given key at the given version exists in the database. Otherwise, the query returns the value δ. Because all the versions of the data items are stored in the same index tree, the cost of a single-key retrieval, write, or delete action is $\Theta(\log_B m)$ pages, where m is the total number of entries stored in the VBT. This holds regardless of the version that is queried. Let us denote by m_v the number of data items that are alive at version v. Note that m_v might be significantly smaller than m, if the database history contains many deletions. Early versions may also contain very few live data items. However, because even large B$^+$-trees tend to be low in height, this access cost is usually acceptable.

The problematic operation in the VBT is the range query operation. An efficient range query in a single-version B$^+$-tree index needs to process $\Theta(\log_B m + r/B)$ B$^+$-tree index pages, where m is the number of entries stored in the index, and r is the number of data items in the queried range. This is because data item entries that are next to each other in the key dimension are stored next to each other in the leaf pages and each leaf page contains $\Theta(B)$ entries. B$^+$-trees can generally guarantee a minimum number of entries per page, such as $^B/_3$, for example. Most of the B$^+$-tree implementations have the leaf-page level siblings linked to enhance range queries, so the range can be scanned without backtracking to index pages once the other end of the range is located. However, even backtracking the search through index pages to locate all the leaf pages requires access to only $\Theta(r/B)$ pages, since the number of leaf pages is asymptotically much higher than the number of index pages required to index them (see proof below).

Theorem 3.1. Locating the r entries in a queried range $[k_1, k_2)$ in a B$^+$-tree index requires access to $\Theta(\log_B m + r/B)$ B$^+$-tree pages, where r is the number of entries in the queried range $[k_1, k_2)$, and m is the number of entries stored in the index structure.

Proof. Because each index page in a B$^+$-tree has a fan-out of $\Theta(B)$, and all the root-to-leaf paths are of the same length, the height of the B$^+$-tree index is $\Theta(\log_B m)$. This explains the logarithmic part $\log_B m$ of the costs, as the search tree must be traversed from the root to the correct leaf node to locate key k_1. The r entries returned by the query require $\Theta(r/B)$ pages to store them, which gives the second part of the cost. If the index has direct links between sibling pages (like the B$^{\text{link}}$-tree), then the pages can be directly traversed, and the proof is complete. If there are no sibling links, the leaf pages must be located by backtracking through the index pages, resulting in additional page accesses. The number of index pages required to locate the leaf pages is however asymptotically smaller than the number of leaf pages indexed by the index pages; with the possible exception of an extra root-to-leaf traversal of the entire index. This result is proven by, for example, Brown and Tarjan [17] in a more general form. We formulate a short proof for this result here. In each index page at level $l + 1$ (except for the leftmost and rightmost index pages), the search will locate and use at least $\Theta(B)$ pointers to pages at level l. Thus, to locate all $n = \Theta(r/B)$ leaf pages, at most $\Theta(\log_B n)$ pages at level two need to be processed. Similarly, only $\Theta(\log_B \log_B n)$ pages at level three need to be processed. In general, exponentially fewer pages are required at each higher level, and the sum of all the required pages is bounded from the above by the power series of $1/2$ times n, which in turn converges to $2n = \Theta(n) = \Theta(r/B)$. Furthermore, while the leftmost index pages at each level might contain fewer than $\Theta(B)$ entries that are relevant to the query (and similarly for the rightmost index pages), if we reserve two extra pages for each level, these are accounted for. Because the initial traversal from the root page to the leftmost leaf page has already added an asymptotic cost of one page for each level, these extra pages reserved for each level do not add to the asymptotic cost of the range query. □

Corollary 3.2. In the VBT, the entries are not clustered next to each other by the key values—the different versions of these entries are in the way. The only guaranteed cost limit for a key-range query of the range $[k_1, k_2)$ in a VBT index is of the form $\Theta(\log_B m + (n \times m_k)/B)$, where m is the number of entries in the entire index, n is the number of versions in the database history, and m_k is the maximum possible amount of discrete keys in the queried range (for databases that store integer keys, $m_k = k_2 - k_1$). □

Corollary 3.2 implies that in the worst case, there are n different versions of each data item in the range, and they must all be scanned to find the relevant entries. Note that this does not have anything to do with the size of the result set of the query—none of the records in the range need to be alive at the queried version, resulting in an empty answer set to the query. Thus, the VBT is not sufficient for use as a multipurpose multiversion database index structure.

3.3 Asymptotic Optimality

As we saw in the previous section, the B$^+$-tree index is not efficient when used as a multiversion index structure. To properly categorize the efficiency of index structures, let us now define what the action costs of an optimal index structure are. When designing a dynamic index structure where item search is based on the comparison of key values, and the items are ordered, the minimum number of key comparisons required for the search is $\Theta(\log_2 m)$, when the structure contains m entries. This can be achieved, for example, with a binary search tree. Similarly, the minimum number of key comparisons for locating a range of entries is $\Theta(\log_2 m + r)$, if we do not assume that the r entries of the queried range are stored in consecutive storage locations so that it would suffice to locate only the storage locations of the range endpoints.

As we have shown in the previous section, the B$^+$-tree is an optimal snapshot index structure that achieves these bounds, although in the case of database tree structures, we are calculating the number of page accesses instead of item value comparisons. If we wish to consider the number of key value comparisons in a B$^+$-tree, note that a single-key search in a B$^+$-tree index requires access to $\Theta(\log_B m)$ pages, as shown in Section 3.2, where B is the page capacity and m is the number of entries stored in the index. Because each page contains at most B entries, at most $\Theta(\log_2 B)$ key value comparisons need to be performed for each page if the entries of the page are ordered. The total number of key comparisons is thus $\Theta(\log_2 B \log_B m)$. Once the page capacity B is fixed, the term $\log_2 B$ becomes a constant, and can be omitted from the asymptotic analysis.

Based on the discussion above, we formally define the requirement for optimality in a multiversion database index structure:

Definition 3.3. A general-purpose multiversion database index structure is *optimal*, if the action costs are logarithmically dependent in the number of data items alive at the queried version. More specifically, the corresponding action costs must be at most $c_1 = \mathcal{O}(\log_B m_v)$ pages of the index structure for single-key actions and $c_2 = \mathcal{O}(\log_B m_v + r/B)$ pages of the index structure for the range-query action in the worst case, where m_v is the number of data items that are alive at the queried version v. □

An optimal multiversion index structure must be as efficient as an optimal single-version index structure that only indexes the data items alive at the queried version. The definition presented here is the same as the definition of Becker et al. [7, 8], and stricter than the one assumed by Varman and Verma [92], in which the logarithms are taken from the total number of updates performed to the index structure, instead of the total number of entries that are alive at the queried version.

In practice, the index structure must also be able to index multiple data items that receive the same version, corresponding to multiple data items inserted or deleted by the same transaction. We naturally also require that the structural consistency of the index must be preserved in the presence of multiple updating transactions. The logical key-level consistency of the index, and of the set of data items themselves, is assumed to be preserved by using an appropriate multiversion concurrency-control algorithm, such as snapshot isolation [11] (see Section 2.7). These are practical requirements that are necessary for the index to be useful in a modern multi-user database environment.

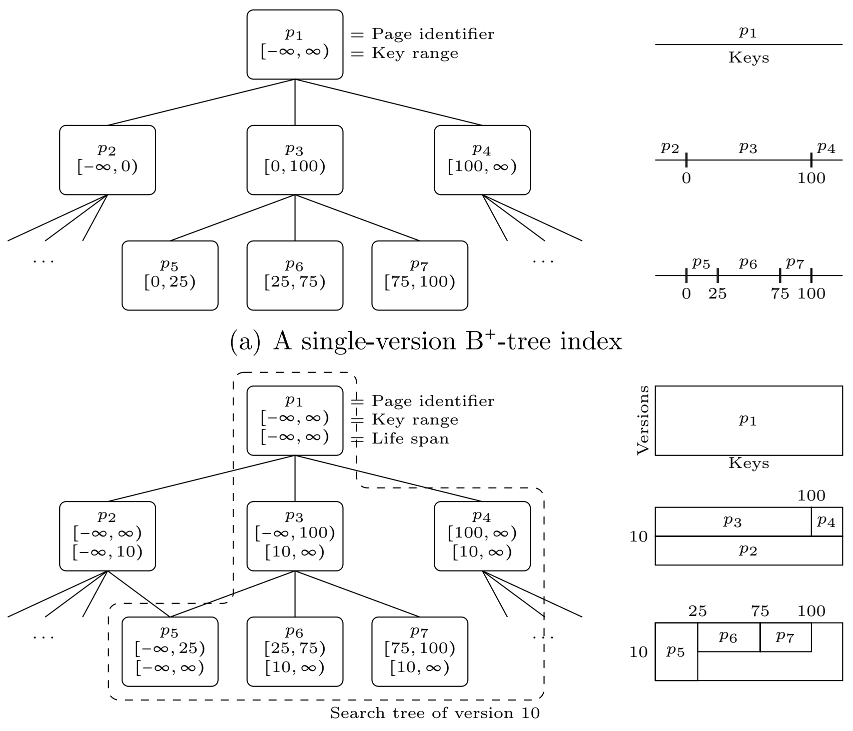
3.4 Common Multiversion Index Design

It is apparent that a specifically tailored structure is required for efficiently storing the history of data items. A common design in newer multiversion index structures is that each database page covers a region in key-version space, and that these regions at any level of the index do not overlap. The root page covers the entire key-version space, and each page lower in the index structure covers a smaller region. These multiversion index structures are often directed acyclic graphs, rather than trees, and child pages may have more than one parent page.

An example of the difference between a single-version B$^+$-tree index and a common multiversion index structure design is shown in Figure 3.1. The figure also shows the key ranges (or regions of key-version space) covered by each page, and an example of the search tree of a single version is shown for the multiversion index structure. In this multiversion index structure, there is a unique *search tree* for each version of the database. This concept is defined formally in Definition 4.2 in page 54. The search trees of different versions of the database may share pages, as shown by the leaf page p_5 in the figure. The page p_5 is shared by all the search trees, as indicated by its life span $[-\infty, \infty)$, and therefore it also has multiple parents so that it can be reached from each search tree.

Because the pages of a multiversion index structure cover regions in key-version space instead of just key ranges, they can be split either by keys or by versions. These operations are generally called *key split* and *version split*, respectively. The main challenge in multiversion index structures is to design these operations so that the data is distributed in such a way that the operations remain efficient in the presence of varying histories of user actions.

It is also important to merge pages to preserve data locality after key deletions. It is true that entries are generally not physically removed from a multiversion index, because the history information must be preserved. However, once a page p is version-split into pages p' and p'' using a version v as the separating version, so that the entries in p' have creation times (Definition 2.1) that precede the creation times in p'', the entries that have been logically deleted before v need not be present in page p'', because the items represented by those entries are not alive at the ver-



(a) A single-version B⁺-tree index

(b) A common design in multiversion indexes

Figure 3.1. Comparison of single-version and multiversion indexes.

sions following v. For an example, suppose that a page in a multiversion database holds an entry created by a transaction with commit-time version v_1, and that this entry is later deleted by a transaction with version v_2. Page p is now version-split into a historical page p', and a current page p''.[3] Now, page p' must hold the information of the insertion at v_1 and the deletion at v_2, but the new current page p'' does not need to contain any traces of the key k, because k is not alive at any version $v \geq v_3$ unless explicitly re-inserted into the database. Key deletions may thus physically remove entries from new copies of old pages, and in this way the number of entries in the new pages may fall below the acceptable minimum so that a page merge is required.

It may seem that the requirement for merging pages is critical only in the asymptotic sense, and not in practice, as the pages can still be version-split and reused for storing new data items. The fact is, however, that unless pages are merged, the key ranges they cover can only shrink,

[3]Depending on the index structure, the physical page p will be reused as either p' or p''.

and never expand. Because the search tree of the current version must cover the entire key dimension, this means that the search tree of the current version cannot shrink unless pages are merged.

Imagine now a warehouse inventory application for keeping track of the goods stored in the warehouse. The application creates a data item for each product stored in the warehouse. Each item is assigned an increasing integer identifier and stored in a multiversion database, indexed by the identifier. As the database fills, pages are key-split and thus the leaf pages cover key-ranges that are close to the lower end of the integer key space. When the products are taken out of the warehouse, the data items are deleted. New items have increasing identifiers and are thus inserted to the leaf pages in the higher end of the key space. Eventually, all the pages with lower key ranges contain only deleted entries, and all the live entries are clustered in the leaf pages with higher key ranges. Suppose that a reporting transaction performs a range query for the entire key range of the database. The range query must scan through all the pages at the lower end of the key range that contain only deleted entries before it reaches the pages where the live entries are stored. The performance of the reporting process will only get worse as the database accumulates more historical entries, even if the database contains the same number of live entries. As shown here, it is important to guarantee that all pages that are part of the search tree of a version v contain enough entries that are alive at version v so that range queries that target version v are efficient.

Merging pages can lead to a tree height decrease in a single-version index structure. In multiversion structures, the height of the entire index cannot decrease, because the heights of the version trees of historical versions must remain as they are. The search trees of different versions can, however, be of varying heights. The MVBT [7, 8], for example, has a separate structure called the *root** (see Definition 3.4 below) that stores the page identifiers of the root pages of different versions. This is illustrated in Figure 3.2. The TSB-tree [58], in contrast, has a single root and thus all the search trees of different versions are of the same height.

Definition 3.4. A *root** is an auxiliary structure that can be used to efficiently retrieve the page identifiers of the root pages of different versions. □

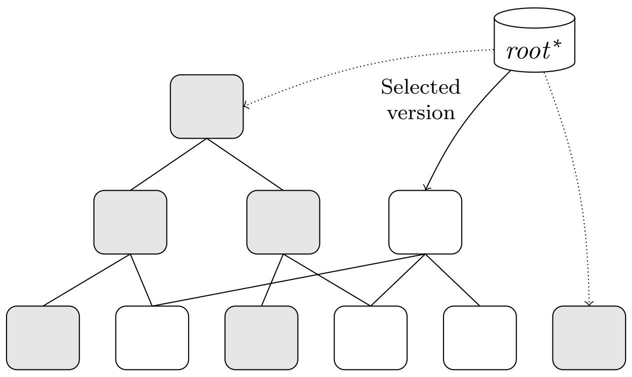


Figure 3.2. Search trees of different heights. Search trees of different versions in a multiversion index can have different heights.

3.5 Early Multiversion Index Structures

The idea of storing historical versions in the database is not new. Overmars discussed general methods for making data structures persistent in 1981 [69, 70], and the classification of the time concepts ranges back to the mid-1980s [62, 85, 86]. The earliest specifications used the term *rollback database* when talking of multiversion databases (that is, transaction-time databases); *historical database* when discussing valid-time databases; and *temporal databases* when talking of bitemporal databases (see Section 2.2 for current definitions of the time concepts). This section reviews some of the early index structures and the design ideas behind them.

One of the earliest approaches to data persistence is *reverse chaining* that is used to chain the history of the data entries together. Lum et al. [62] have described an index structure that uses a current version tree to index the current version, and a historical tree that is used to index historical versions. Both trees contain pointers to a reverse-chained linked list of entry values, ordered by the update times, so that the latest version is at the start of the list. While both of these index structures may be efficient (optimal) B-trees, the number of updates on a data item directly affects the length of the version-history chain, and thus the query actions for previous versions on this index structure can have very high costs, especially for key-range queries.

Another technique used in early multiversion index structures is *path copying* [80, 87]. This method achieves persistence by creating copies of changed nodes, so that the old nodes are left untouched. Because the new nodes need to be attached to parent nodes, and the original parent nodes cannot be changed, the entire path from the changed node up to the root node needs to be copied. This method therefore effectively creates a new path that can be used for current-version queries and retains the old path for historical queries. Sarnak and Tarjan [80] describe how to make red-black trees [4, 33] persistent by creating a new copy of the path along which a change has occurred. Search trees of different versions share common subtrees, and differ only on the single copied path. As the entire path is copied, a new root node is created for each update. Soisalon-Soininen and Widmayer also use path copying with AVL-trees to make the tree structure recoverable [87].

Because there are multiple roots, all the roots of different versions must be stored in some structure. Sarnak and Tarjan [80] use an array of page identifiers, ordered by the creation time of the corresponding pages. This array is essentially a *root** structure of Definition 3.4. While this method is efficient for binary search trees, with a logarithmic cost of $\Theta(\log_2 m_v)$ pages for both updates and query operations, where m_v denotes the number of entries that are alive at version v, the method is not feasible for disk storage, because only a single entry is stored per node and each update requires $\Theta(\log_2 m_v)$ space for the new copied path. Sarnak and Tarjan further enhance the space consumption of their method by allowing nodes to grow fat. A *fat node* is a node that can contain an arbitrary number of entries; in this case, an arbitrary number of left and right pointers in the binary search tree, corresponding to paths in different versions. The fat node can be implemented, for example, by chaining together a list of database pages. Path copying for each update can thus be avoided. However, the enhanced structure still stores only a single data-item entry in each node, so the approach remains unsuitable as a disk-based access method.

Shoshani and Kawagoe [82] have presented a more general framework for indexing data with different types of *time sequences*. A time sequence is the collection of changes associated with a data item. In our multiversion data item model, a data item has a life span during which it is alive, and any change to the data item creates a new data item (see Section 2.4). In contrast, a time sequence records all the updates that target a single key. A data item in Shoshani's and Kawagoe's framework is represented by a tuple (k, S), where $S = \{(v_i, w_i) : i \in \{0, 1, \ldots, n-1\}\}$ is the time sequence of the data item. An individual tuple (v_i, w_i) of the time sequence denotes that the data item with the key k was assigned the value w_i at version v_i.

Shoshani's and Kawagoe's framework furthermore identifies time sequences with different update patterns. Item costs in a grocery store database, for example, have continuous, irregularly and step-wisely changing values, while the number of items sold is a discrete value that is measured at regular intervals. The dynamic index structures that Shoshani and Kawagoe propose assigns cells or database pages to data keys (surrogates). As the pages fill up, new pages are linked to form a chain of pages. These pages are further indexed into an ordered list of pointers so that the entire structure is doubly indexed—first by the key attribute to locate the secondary index and then by the version to locate the correct page. Finally, the data page itself must be searched for the correct version. Single-key operations in these index structures have a logarithmic cost, but the constant overhead is high as there are multiple indexes that have to be traversed. Furthermore, the data items are not clustered on the key attribute, and key-range queries are thus inefficient.

Easton's *write-once balanced tree* (WOBT, [25]) is a multiversion index structure that stores multiple versions of data items on indelible storage. The structure can therefore be used with WORM (write once, read many) media. This index structure is based on the B$^+$-tree, but rather than overwriting old data, new versions are written next to the old ones. When pages fill up with different versions, a new copy is created and possibly split into two separate pages. The old one, however, remains as it was. Root pages of the WOBT are forward-chained, so that the most recent root page can be located by starting a search from the first root (first page on the database index) when the structure is loaded from the disk during database startup. The latest root, and possibly the other roots as well, are then cached in main memory for fast access. The WOBT is not very space-efficient, because no data can be overwritten. It is, however, an important structure, because the more recent TSB-tree (described in Section 4.2) is based on it, and the TSB-tree in turn is used in the multiversion database engine that Microsoft is developing on top of the SQL Server (see Section 4.3).

Driscoll et al. have discussed a more general way of making main-memory data structures either partially or fully persistent [24]. Their solution to partial persistence is based on the ideas presented by Overmars [69, 70]. They suggest using either fat nodes or node copying to make binary search trees persistent. In the *fat node* method, the binary tree nodes can grow arbitrarily large. In the *node copying* method, the nodes may contain a fixed number of left and right pointers to child nodes. The pointers have a version attached to them that is used to select the correct pointer to traverse. When a node fills, a new copy of it is created. Only the most recent pointers are copied to the new node, and a pointer to the new node is attached to the parent. The old node is thus left in place and can be used for historical queries. While Driscoll et al. designed the algorithms for in-memory structures, the node copying approach has been adapted for use with disk-based temporal index structures. The more recent index structures, such as the TBS-tree, the MVBT, and the MVAS (see Chapter 4), all employ page-copying methods such as these to organize the database pages. For discussion on the fully persistent index structures, refer to the article by Driscoll et al. [24].

The *time index* is yet another early multiversion index structure that was proposed by Elmasri et al. [26, 27]. The authors had noted that the other index structures chained the versions of data items separately, and thus did not cluster the data items of a given version next to each other, and the time index was designed to correct this. Elmasri et al. talk about valid time [26, 27], but they make the assumption that changes occur in an increasing time order, and that changes to previous times do not happen, so the time index is more properly classified as a transaction-time index structure (recall the definition from Section 2.2, p. 11).

The time index is based on the B$^+$-tree, but it is organized by the data-item versions (transaction-time instants), instead of data-item keys. Each leaf page holds a range of database versions. The leaf-page entries are tuples of the form (v_i, b_i), where v_i is the version of the entry and b_i a pointer to a bucket (i.e., database page or a collection of pages) containing information about the entries of that version. To save space, only the bucket identified by b_0 contains a full snapshot of entries; the rest of the buckets (identified by b_i, $i \neq 0$) store updates. Elmasri et al. enhance this index by various techniques (such as separating incremental and decremental buckets [26]), and Kouramajian et al. further develop the structure into the *time index*$^+$ [48]. The enhanced structure uses a compression technique that shares the live entries of two sibling leaf pages in a shared bucket. This technique can also be used by a larger set of leaf pages, so that entries shared by all of the leaf pages are placed in a shared bucket. The compression technique greatly enhances the space usage, but the bucket updating is costly, and the situation at any given version needs to be reconstructed based on multiple sources.

Lanka and Mays have applied the persistence methods of Driscoll et al. discussed above to make a fully persistent B$^+$-tree [51]. They propose various different schemes for converting a standard B$^+$-tree into a persistent one; namely, the *fat node* method, the *fat field* method, and the *pure version* method. All the methods are built on top of a standard, ephemeral B$^+$-tree. A *fat node* in the persistent B$^+$-tree of Lanka and Mays is the extension of a B$^+$-tree node. It is a logical collection of B$^+$-tree nodes, with an added version block that is used to locate the B$^+$-tree node corresponding to a given version. Different versions may point to the same

B$^+$-tree node, if the contents of the subtree rooted at that node are identical for both versions. The version block is attached to the beginning of the fat node. In the fat node method, the logical index thus contains alternating index blocks and version blocks, so that a search traversing from a root page first selects the correct pointer based on the searched key value, then selects the correct pointer based on the version attribute, and so on. The problem with the fat node method is space usage—each update creates a new disk block. In the *fat field* method, some of the versioning information is attached to the B$^+$-tree pages, but the version blocks are still used in some situations. The fat field method is in fact quite close to the structure of the more recent indexes. Because the tree contents and the pointers in index pages change between versions, the root of the index also changes between different versions. The fully persistent B$^+$-tree thus uses a *root** structure to store pointers to the roots of different versions. The *pure version* method, on the other hand, is a straightforward extension of a B$^+$-tree, where the index pages are left as they are, and the versioning information is attached to the entries at leaf pages. This approach is more space efficient, but it does not cluster different keys of a single version close to each other, and range queries are therefore not efficient.

The fully persistent B$^+$-tree was designed to be fully persistent, so that new versions can be based on any previous version (see Definition 2.5). This is achieved by maintaining an auxiliary structure that describes the version history. The history is modelled as a directed acyclic graph, so that different version branches can be merged. In effect, the auxiliary structure is used to fetch the ancestor set A_v for each version v. For convenience, A_v is defined to include the version v. When a query for a version v is performed on the fully persistent B$^+$-tree, only versions that are present in A_v are considered. For partial persistence, we can omit the auxiliary structure and simply consider all versions.

Tsotras and Kangelaris have discussed I/O optimality and propose an I/O optimal (according to their definition) index structure called the *snapshot index* [90]. The snapshot index stores new data items sequentially in a doubly-linked list of pages. The underlying structure is therefore basically a log file of item insertions. At all times, one of the pages is an *accessor* page that receives all new records that represent item insertions or updates. Access to current entries is made efficient by keeping only pages that are *useful* on the list. Useful pages have at least a minimum amount of entries that are alive. Pages that are no longer useful are moved away from the main item list but they are kept linked to the pages in the list. The index also maintains an auxiliary dynamic hash structure for locating the most recent update to any given data item, hashed by the item key. The hashing function is used to make updates efficient: any item that is to be deleted can be located via the hashing function in expected constant time, and any new item to be inserted can be placed at the single accessor page in constant time. In practice, the snapshot index has an expected constant update cost. Queries, however, need to search through all the entries that are alive at the queried version, because the entries are not ordered on the key attribute. The snapshot index is therefore efficient for constructing the entire (unordered) set of data items belonging to any given version, but not for querying a range of keys or a single key of a given version. More formally, the snapshot index is I/O optimal for **/–/point* queries, but not for *range/–/point* or *point/–/point* queries.

3.6 Multidimensional Index Structures

Multidimensional index structures have also been used for indexing multiversion data. Recall from Section 2.4 that multiversion data items are tuples of the form $(k, \overrightarrow{v}, w)$, where k is the data key and $\overrightarrow{v}$ is life span of the data item; that is, the range of versions (transaction-time instants) during which the item is alive. The key and version dimensions are orthogonal, and the data item may thus be uniquely identified by a two-dimensional line segment in key-version space that is parallel to the version axis (see Figure 3.3). When considered in this way, the data items can then be indexed by a multidimensional index structure. In this case, the pages of the multidimensional index structure cover regions of key-version space. These regions are generally known as *minimum bounding regions*, or MBRs. Figure 3.3(b) shows one possible way to cover a set of multiversion data items with suitable MBRs.

Perhaps the most widely used multidimensional index structure is the R-tree of Guttman [34] and its variants, such as the R*-tree of Beckmann et al. [9]. R-trees do not, in general, guarantee logarithmic access times in all situations. Even an exact-match query in the standard R-tree may require traversing multiple paths, because the key-version regions of sibling pages may overlap. However, when discussing multiversion data, it is worth noticing that the key-version ranges of data items stored in the leaf pages cannot overlap. This means that the overlap in index pages may be reduced when storing multiversion data. On the other hand, standard multidimensional index structures are designed to index data items with static spatial dimensions. Multiversion data items have a life span $\overrightarrow{v}$ that

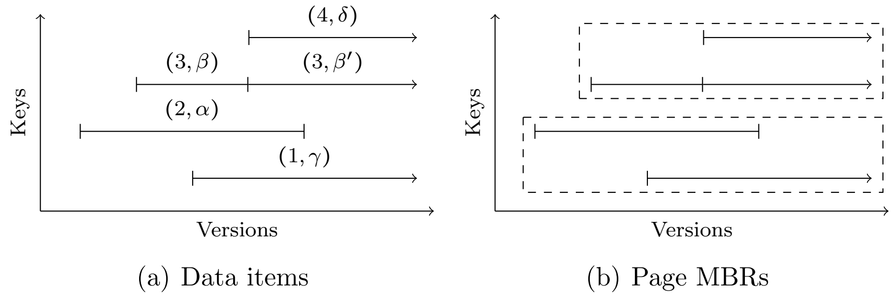


Figure 3.3. Multiversion data indexed in a multidimensional index. Reading from the left, first items with keys 2, 3 and 1 were consecutively inserted into the database at different versions. After that, data item with key 3 was updated and 4 was inserted at the same version. Finally, the item with key 2 was deleted.

is initially an infinite range $[v_1, \infty)$ starting from v_1, as the item is not yet deleted. When the multiversion data item is deleted, its life span is cropped to a finite range $[v_1, v_2)$. The life span is thus not static, and the data items are not perfectly suited for multidimensional indexing. The initially infinite ranges of the data items also cause problems for indexing.

Once a multiversion data item has been deleted, it can no longer be modified, and its life span and key-version range become static. The PostgreSQL database system, initially introduced in an article by Michael Stonebraker [88], takes advantage of this fact. Initial versions of the PostgreSQL database clustered the data item entries in a standard snapshot index (i.e., a B$^+$-tree) based on the key attribute. Different versions of the data item were linked in a chain of *delta records* that described the item updates. A *vacuum cleaner* process was run periodically to move the earlier versions of data items into an R-tree index. At this point, the moved data items were static, and could be properly indexed in an R-tree. The R-tree was thus used as a storage for historical entries. Kolovson and Stonebraker [46] have also designed different variants of this design that utilize magnetic disks or combinations of magnetic and optical disks. Current versions of the PostgreSQL [74] include GiST indexes (generalized search trees [38]) that can be used to implement R-trees.

There are still problems even when storing only static multiversion data items (i.e., multiversion data items that have already been deleted) in an R-tree, because the data items have highly varying lengths. A single multiversion data item with a long life span causes the entire leaf page where it resides in to have a wide bounding region. A wide leaf page causes the entire path from the leaf page up to the root page to have wide bounding regions, because the MBRs of the parent pages must cover the MBRs of their child pages. This in turn leads to high overlap between pages, which causes search performance to degrade. Kolovson and Stonebraker have designed the *segment R-tree*, or SR-tree to alleviate these issues [47]. The SR-tree allows data items with long life spans to be stored higher in the tree, thus reducing the overlap on lower levels of the tree. Because the pages on the higher levels should also have space for enough index entries (i.e., pointers to child pages), the SR-tree allows pages that are higher on the tree to be larger. The index structure can be further enhanced if some assumption of the distribution of the data items can be made. Kolovson and Stonebraker also introduce *skeleton SR-trees* that make this initial assumption and organize the structure of the tree based on it. The index is dynamic, however, so it will adapt to the actual distribution of the data items. According to their tests, the skeleton SR-tree outperforms both the SR-tree and the R-tree when used as a multiversion index.

Regardless of the enhanced performance of the skeleton SR-tree, the fact remains that R-trees and other methods based on them do not have logarithmic cost guarantees, and cannot therefore be considered optimal for indexing multiversion data. Another problematic issue with R-trees is concurrency control and recovery. The MBRs of the R-tree pages need to be consistent with the MBRs of the child pages, so they need to be updated whenever entries are inserted or deleted. In standard R-trees, insertion causes MBR enlarging on the path from root to a leaf page. This can normally be processed during the search for a proper page to accommodate the new entry. Item deletion, however, causes MBRs to shrink, and the shrinking must be done bottom-up. This is challenging for concurrency control, because large parts of the tree need to be kept locked during the deletion operation to ensure correct operation and to avoid deadlocks.

3.7 Hashing Structures

Hashing is generally used for very efficient, constant-time membership queries, when a suitable hashing function for the indexed values exists. A hashing function $h : X \to \mathbb{I}$ maps all values x in the source domain X into a finite set of integers. The size of the target set is normally much smaller than the size of the source set. Some source values are therefore necessarily hashed into the same integer value in the target set, thus causing collisions. Each hashed value $h(x)$ identifies a *bucket* that conceptually holds all the objects that have the same hash value. The range of hashed integers is thus $0, 1, \ldots, n-1$, where n is the number of buckets allocated for the index. Each bucket can be implemented, for example, as a linked list of database pages.

With traditional hashing methods, the hashing function is static, and the size of the target set (n) is therefore predefined. If too small a target set was initially selected, hashing becomes inefficient because collisions occur more and more frequently, and the object chains become longer. In these situations, a complete reorganization (a *re-hashing*) of the hash table is required. Litwin has presented a linear hashing scheme that dynamically changes the hashing function so that the size of the target set can grow [53]. Periodically, a single bucket b_i is *split* into two buckets b_i and b_j by changing the hashing function so that instead of mapping the entries $x \in b_i$ into b_i, some of them are mapped to b_i and some into b_j. This can be achieved by organizing the bucket splits correctly. In practice, if a hashing function h_i used at round i maps objects into a range $0, 1, \ldots, n_i - 1$, then the hashing function h_{i+1} used for the next round $i+1$ maps objects into the range $0, 1, \ldots, 2n_i - 1$, thus doubling the range size $n_{i+1} = 2n_i$. For all the objects x either $h_{i+1}(x) = h_i(x)$ or $h_{i+1}(x) = h_i x + n$. This internal reorganizing happens one bucket at a time, so there is no extensive one-time reorganization. At most two different hashing functions need to be used to access any object, and so the expected performance of the hashing remains the same, even if the set of stored objects grows.

Kollios and Tsotras have expanded the linear hashing method for hashing multiversion data items [45]. In their partially persistent linear hashing method, each data bucket stores multiversion data items similar to our definition (see Definition 2.7). When a bucket split occurs, the hashing function changes and some entries need to be moved to a new bucket. In this situation, the entries are moved *logically*; that is, a live multiversion item $(k, [v_1, \infty), w)$ that is moved during a split at version v_2 from bucket b_i to bucket b_j remains in b_i, but the entry representing it is changed to the historical entry $(k, [v_1, v_2), w)$, and a new live entry $(k, [v_2, \infty), w)$ is inserted into the new bucket b_2. The historical query operation then reduces to finding the correct hashing function used at the historical version, and then finding the historical entry from the correct page. For the first part of the query, Kollios and Tsotras maintain an array that stores information about the hashing functions used for different versions. Each array slot represents a version during which the hashing function has changed, and locating the correct function thus requires a binary search on the array. This search adds a logarithmic cost on the historical query, which is otherwise an expected amortized constant-time operation. Kollios and Tsotras note, however, that the array should be small enough to remain in main memory and thus the effect of this search on the overall query performance should be negligible.

Hashing functions are efficient for solving the *point/–/point* query, with (almost) an expected $\Theta(1)$ constant query and update costs. The trade-off is that other query types cannot be processed efficiently. Key-range queries are especially inefficient, because the keys are not clustered near each other. In fact, finding the next key in a hashing structure requires either traversing through the entire structure or, even worse, trying out all possible next keys. Hashing structures are therefore not suitable as a general multiversion index structure.

3.8 Version-Control Systems

Software engineers use version-control systems (VCS) to share the software code between the developers and to maintain the history of all the previous versions of the software. A developer *checks out* the code from the central version-control system, creates his modifications, and finally *commits* them to the VCS, thus creating a new version. In this sense, version-control systems are in fact transaction-time databases. Most version-control systems allow software engineers to create branches on the version history, so that different teams can work with different features without disturbing each other. Version-control systems are therefore fully persistent. Examples of version-control systems are the concurrent versions system (CVS) [20], Subversion [22], and Git [30].

The main difference between version-control systems and traditional databases is in their usage. Multiversion database indexes are used to locate data item tuples based on a single key or a key range, by either fixing the version to a single point or by specifying a range of versions (recall the different query types from Section 2.3). Version-control systems, on the other hand, are used for much more specific queries, such as "find all the differences between versions v_1 and v_2" (with v_1 and v_2 possibly on totally different branches), "find all the changes to the data item identified by k", or "find the version during which the line l was changed in the text file identified by k". These are queries that could not be effectively answered using a traditional database index. A multiversion database system could, however, conceivably use an existing version-control system for indexing the multiversion data items, and it is therefore worth it to examine these systems a little further to determine whether they would be efficient for this purpose.

Fendt has evaluated the performance of some of the more recent version-control systems in an online article at the Linux Foundation [29]. He shows that the Git version-control system is the fastest one, alongside with another VCS called Mercurial. We will therefore concentrate on Git.

The Git version-control system is based on indexing objects by the SHA-1[4] hash values of their contents [30, 93]. The objects are stored in the file system, named with the hexadecimal representation of the SHA-1 hash value. The file system therefore works as the primary index for the stored objects. Four types of objects can be stored in the Git database: blobs, trees, commits, and tags. The *blob* type is used to represent any object that is stored in the Git VCS, and so every file that is added to Git is stored in a blob object. *Tree* objects are used to maintain directory hierarchies, *commit* tags represent commits (i.e., database versions), and *tags* are used to assign cryptographic tags for other objects (e.g., for verifying their contents).

Each tree object logically represents a single node and all its children in a directed acyclic graph. The tree object is basically a collection of pointers to objects stored under that node. A pointer that points to an object o is stored in a tuple of the form (m, t, s, n), where m is the file system mode of the file that represents o, t is the type, s is the SHA-1 digest of the object o, and n is the name of the object. Subdirectories are maintained by storing pointers to other tree nodes. To proceed from a tree node x to its child node, we find the correct child pointer (m, t, s, n), and use the SHA-1 hash value s to retrieve the child object from the index (i.e., by loading the file named s from the file system). Different versions are created by possibly creating new tree objects for that version. Any child object (e.g, a text file) that has changed has an updated SHA-1 hash value, because the hash is based on the contents of the stored objects. This in turn changes the tree object, because the tree object contains the SHA-1 values of its children in the pointer records. The changed tree object again needs to be indexed with a different SHA-1 value, and thus a change propagates all the way to the root of the directory structure. Unchanged nodes can however be efficiently reused between different versions. The Git VCS therefore relies on a form of *path copying* (see Section 3.5, p. 39).

A straightforward way of using the Git version-control system as a multiversion index structure would be to store each key in a single tree object. A binary search could then be used to quickly locate the searched key among the pointers stored in the tree object. The single tree object

[4]SHA-1, the Secure Hash Algorithm, is a cryptographic hash function that computes a 160-bit hash value (called *digest*) from its input.

would then work as a dense index for the stored objects. While querying would be efficient, note that updating any key necessitates duplicating the entire tree object, thus duplicating the entire indexing portion of the multiversion database. The update performance and the space usage of this structure would be suboptimal. Also note that this kind of a structure could not be used as a sparse (primary) index structure, because all the child objects (data items stored in the database) must be stored in separate files. As a conclusion, the version-control systems are designed for maintaining versions of files and file directories, and they are not really suited for usage as general-purpose database indexes.

3.9 Other Structures

In addition to the index structure already reviewed in this chapter, there are still a few structures worth noticing. For example, the search engine conglomerate Google has built a customized database system for organizing the petabytes of data their search engines need to search through. Their database system is called Bigtable [21], and it is designed to be a highly scalable database system that can be used to index vast amounts of data in a distributed database environment. What is interesting from our point of view is that Bigtable also offers some basic versioning functionality: data items are stored with versions, and they can be used for querying. There is a catch, however, as Bigtable was not designed to be primarily a multiversion database index. The data items are ordered on the item keys, and all the versions of a data item are stored in the same page (or *cell*, by the terminology of Bigtable). Cells can have multiple versions which are chained together by placing the most recent version of the cell in the front of the chain. The Bigtable thus has problems similar to those of the versioned B^+-tree described in Section 3.2 and of indexes that use reverse chaining (described in Section 3.5, p. 39): early versions cannot be directly accessed (because of the version chain), and key-range queries are not efficient. The problem for key-range queries of the most recent version is alleviated somewhat, because not all of the versions of a given data item need to be scanned. Key-range queries of the earlier versions, however, must still traverse the version-chains to locate the correct version. From this we can arrive to the conclusion that the Bigtable is not optimal for indexing multiversion data.

Jouini and Jomier [43] have also recently published an article comparing three different approaches for indexing fully persistent transaction-time data (i.e., data with a possibly branched evolution). Their structures are called the B+V-tree, the OB+tree, and the BT-tree. From these, the B+V-tree resembles the versioned B$^+$-tree introduced in Section 3.2, but with support for branched history evolution. Like the versioned B$^+$-tree, the B+V-tree is not efficient for range queries because the entries are clustered primarily by their keys, and only then by their versions. The second structure, the OB+tree, builds multiple B$^+$-trees that are allowed to share unchanged branches. This approach is an example of path copying (see Section 3.5, p. 39), and any update performed to a leaf page at database version v to create a new version v^+ therefore necessitates the creation of a new root-to-leaf path, thus requiring $\Omega(\log m_v)$ space for each update, where m_v is the number of entries alive at version v. The final index structure, the BT-tree, indexes entries based on both the key and the version attributes, thus making this structure closer to the more efficient approaches described in the next section. As the details of the index structure are not discussed, we cannot really determine the characteristics of this index structure. The other two structures are suboptimal for indexing partially persistent data because of the design choices they are based on, as explained above.

CHAPTER 4

Time-Split and Multiversion B$^+$-trees

The previous chapters have discussed the theory behind multiversion databases (that is, partially persistent transaction-time databases), and reviewed some approached for indexing such data. So far, none of the structures introduced have been optimal, according to our definition (Definition 3.3). In this chapter, we review three of the more recent multiversion index structures, one of which are optimal. In addition, we shortly discuss a multiversion database system that uses one of these structures and is built on top of the commercial Microsoft SQL Server.

We begin this chapter in Section 4.1 by listing some of the common design ideas shared by all the efficient structures reviewed in this chapter. Then, in Section 4.2, we review the first of these structures, the time-split B$^+$-tree of Lomet and Salzberg (TSB-tree [58, 59]). After that, Section 4.3 describes Immortal DB [54–57], which is based on the TSB-tree. Immortal DB is a multiversion database management system that Microsoft is researching and developing on top of the Microsoft SQL Server. In Section 4.4, we review the multiversion B$^+$-tree of Becker et al. (MVBT, [7, 8]), the first optimal multiversion index structure. Finally, Section 4.5 describes the multiversion access method of Varman and Verma (MVAS, [92]), which was developed at about the same time as the MVBT, and shares many characteristics of the MVBT. Varman and Verma use a slightly more relaxed definition of optimality for multiversion indexes, and the MVAS is not optimal according to our definition.

4.1 Common Design Bases

All of the index structures presented in this chapter have a similar structure: the database pages cover regions in key-version space, the index structures form a directed acyclic graph of database pages (although these structures may still be called *trees*), and for each version, there exists a search tree S_v (see definition below) that is used to locate the entries belonging to that version. Let us call these multiversion structures region-based multiversion index structures:

Definition 4.1. A *region-based multiversion index* is a multiversion database index structure in which all the pages cover regions in key-version space. The structure of the pages forms a directed acyclic graph. Let $\mathsf{kvr}(p)$ denote the key-version region of a page p. Each parent page p at level l contains links to child pages Q at level $l-1$ so that $q \in Q \Leftrightarrow \mathsf{kvr}(p) \cap \mathsf{kvr}(q) \neq \emptyset$. The key-version regions of pages on the same level of the graph do not overlap. □

Definition 4.2. For each version v in a region-based multiversion index, there is a *search tree* S_v that is a subgraph of the entire multiversion index graph. The subgraph S_v is a tree and all the entries of the data items that are alive at version v are located in the pages of S_v. An example of a search tree S_{10} is shown in Figure 3.1(b) on page 37. For each search tree S_v and all levels l of S_v, the pages that belong to S_v at the same level l partition the entire key-space into disjoint regions. Each search tree thus covers the entire key space at each level of the search tree. □

Definition 4.3. A region-based multiversion index structure is said to be *structurally consistent*, if all the index-specific invariants of the structure hold; and *balanced*, if (1) it is structurally consistent; (2) all the pages of the search tree S_v contain at least a minimum number of entries that are alive at version v, for each version v; and (3) for any search tree S_v, all the root-to-leaf paths of S_v are of the same length. □

Note that our definition of a balanced index structure requires that the lengths of all the search paths within any one version are of the same length. In practice, this is guaranteed by designing the structure-modification operations so that the index never becomes unbalanced.

Figure 3.1(b) in the previous chapter shows the general structure of these multiversion index structures, and Figure 4.1 shows the structure of a balanced region-based multiversion index that has an auxiliary *root** structure for locating the roots of different search trees. Note that search trees S_{v_1} and S_{v_2} rooted at pages p_6 and p_9 have a different height. Let us also define what we mean by live entries and live pages in the multiversion indexes:

Definition 4.4. For all versions v, an entry that represents a data item is *alive* at version v if the data item is alive at version v; and a database page is *alive* at version v if it is part of the search tree S_v. □

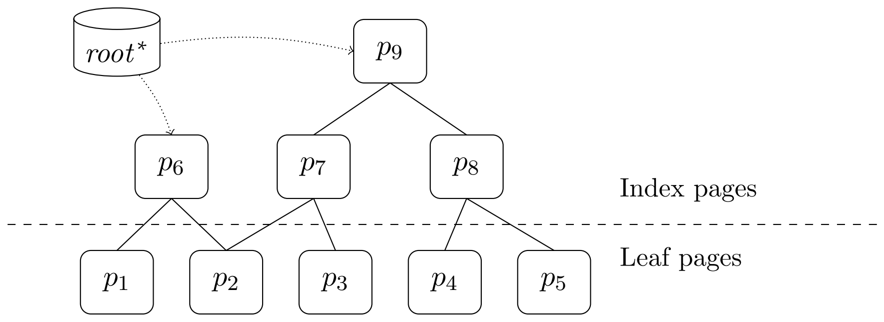


Figure 4.1. Structure of a balanced multiversion index.

Salzberg et al. have recently published a general framework for indexing fully persistent transaction-time data [76] (recall Definition 2.5). Although designed to include fully persistent indexes, the framework encompasses many of the features present in the efficient indexes reviewed in this chapter. Such features are, for example, maintaining a minimum number of live entries (Definition 2.2) in each page and consolidating (merging) those pages where the number of live entries falls below the acceptable limit. Because the framework is designed for full persistence, some of its features are however unnecessarily complicated for partial persistence. More specifically, the framework describes a version-tree structure that is used to determine which data item entries are along the same version branch. This adds a significant overhead to finding the correct version of a data item from a database page. Furthermore, some of the pages in the index structure described in the framework can contain *ghost pages*, which are pages that only contain *null markers* that are used to specify endpoints of the life spans of data items. As is shown with the multiversion B$^+$-tree (described in Section 4.4), these pages can be avoided if the structure is only partially persistent.

4.2 Time-split B$^+$-tree

The *time-split B$^+$-tree* (TSB-tree) of Lomet and Salzberg [58, 59] is a multiversion index structure that is based on Easton's write-once B$^+$-tree (see Section 3.5, p. 41). The TSB-tree structure forms a directed acyclic graph of database pages. There is a single root page that is shared by all versions, and the height of the search tree S_v is therefore the same for all versions v. The root page of the TSB-tree covers the entire key-version space. The graph structure is formed by splitting pages that have become full. Each page can be either *key-split* or *time-split*. Splitting a page p

creates a new page p' and the entries are distributed between p and p' based on the new dimensions of the pages. If the key-version range of a child page (or the life span of a leaf-page entry) extends past the split boundary, the child page identifier (or leaf-page entry) is duplicated to both of the pages p and p'.

There are two types of pages in the TSB-tree: *current pages* and *historical pages*. The life spans of all current pages cover the current version; that is, the current pages are alive. The time-split operation creates a new historical page p'. When time-splitting page p, entries with deletion times smaller than the split boundary are moved to the historical page p', entries with life spans covering the split boundary are copied to the historical page, and the rest of the entries are left in the current page p. The page p thus remains in use for current-version queries, and the historical page p' can be moved to a tertiary storage for archival. This is possible, because the TSB-tree maintains the following invariant:

Invariant 4.5. In the TSB-tree, current pages have at most a single parent. Historical pages can have multiple parents.

If such an invariant were not enforced, there might be an unbounded number of parent pages that need to be updated when a page is split. However, this invariant imposes a restriction on the time-split operation: none of the live entries on page p must be copied to the historical page p'. A page p therefore cannot be time-split into a historical page p' based on a version v that is covered by the life span of a live entry; that is, if p contains an entry with a life span $[v_1, \infty)$ such that $v_1 < v$. By adhering to these rules, the following invariant can be maintained, and the historical pages can be directly written to a write-once media during a time-split.

Invariant 4.6. In the TSB-tree, historical pages are never modified.

When fine-tuning the TSB-tree, it is possible to make a choice on how often key splits and time splits are used. If the goal is to minimize the index size, key splits should be used more often. If the performance of the current-version queries is important, version splits should be more frequently applied. This is a natural way for adjusting the behaviour of the index for varying database loads. The original TSB-tree [58] was designed mainly for scenarios where data is never deleted, and pages therefore do not need to be merged. This limitation still exists, and it means that the key ranges of pages can only shrink, and never expand.

The TSB-tree that was adapted for the Immortal DB (discussed in the next section) manages data deletion by inserting new entries that mark item deletion. While this approach works, it means that key splits and time splits alone cannot guarantee that pages have a minimum number of live entries for all the versions that are stored in the pages. For optimality, the access costs of the operations must be logarithmic in the number of entries that are alive at the queried version. This is guaranteed if the index structure is balanced (see Definition 4.3), so that each page p with a life span $[v_1, v_2)$ contains at least min-live entries that are alive at version v, for each $v \in [v_1, v_2)$. Here, min-live is an integer configuration variable that must be larger than one so that the fan-out at each level of the search tree is greater than one, thus guaranteeing a logarithmic height for the search tree and maintaining the performance of range queries. The variable min-live must furthermore be linearly dependent on the page capacity B, so that the cost differs only by a constant when the base of the logarithm is B.

Lomet and Salzberg [58, 59] do not give asymptotic bounds for the costs of the user actions, but rather derive exact formulas for calculating the size of the index structure and for the amount of redundancy in the index. A general space complexity class for multiversion indexes is $\mathcal{O}(n/B)$ database pages, where n is the number of updates performed in the history of the database; the TSB-tree belongs to this complexity class [78]. Regarding time complexity, we formulate some general bounds for the costs of the actions in the TSB-tree here:

Theorem 4.1. All single-key actions in the TSB-tree targeting any version v have a cost of $\Theta(\log_B m)$ pages, where m is the number of updates performed on the TSB-tree during its history. The worst-case cost of a key-range query action that queries the range $[k_1, k_2)$ of version v is $\Theta(\log_B m + m_k/B)$, where m_k is the maximum possible amount of discrete keys in the queried range (for databases that store integer keys, $m_k = k_2 - k_1$).

Proof. Because the TSB-tree has a single root page, the height of the index structure is determined by the total number of entries indexed by the entire TSB-tree, which in turn is dependent on the number of updates performed on the index. The cost of the single-key operations derives directly from the traversal from the root page to the correct leaf page. For the range query, the number of page accesses is not bounded by the size of the retrieved item set, because it is possible that there are leaf pages that contain only deleted entries that are not relevant to the query but still need to be processed. In the worst case, the index contains m_k live entries in the queried range at version v_1; that is, the index contains a live entry for each possible key in the queried range. Assuming that the history contains no deletions, then $m = m_{v_1}$ (when version v_1 is the latest version), and the range query for version v_1 is optimal (as per Definition 3.3), because $m_k = r$, the size of the retrieved item set. If all the entries are deleted at version v_2, the entries are marked deleted but the leaf pages are not merged. The range-query operation targeting version v_2 must process as many pages as the query that targets version v_1, even though the pages do not contain any entries that are relevant to the query. □

Lomet and Salzberg have evaluated the performance of several different splitting policies [59]. According to their experiments, the policy called Isolated-Key-Split (IKS) is a good choice if rewritable media (write-many, read-many, or WMRM media) is available and inexpensive[1]. With this policy, pages are key-split whenever more than two thirds of the entries in the page are alive; and time-split otherwise. When time-splitting a page, the split is always performed based on the latest committed version. This policy optimizes the size of the index structure, and was designed at a time when WMRM media was not as inexpensive as it is currently. With the Immortal DB more recent split policies have been introduced. These are reviewed in the next section.

From the beginning, it has been assumed that lazy timestamping (see Section 2.4, p. 17) is used with the TSB-tree. The initial TSB-tree article [58] does not explain how and when the records should be timestamped with the correct commit-time versions, but rather suggests that Stonebraker's approach for the PostgreSQL database system [88] could be used. In this approach, each relation may be assigned to one of the following three levels: no archive, light archive, and heavy archive. From these, *no archive* practically means that the relation is a single-version relation, and no access to past states is possible. For the other settings, the entries are first stamped with the transaction identifier, and the mapping from commit-time versions to transaction identifiers is inserted in a special relation when the transaction commits. With the *light archive* setting, each time a historical entry is requested, the transaction identifier is mapped to the commit-time version by reading the proper value from the special relation. With the *heavy archive* setting, the mapping is performed on the first access, and the historical entry is updated with the commit-time version so that subsequent accesses do not need to load the mapping from the special relation. The Immortal DB (discussed in the next section) more clearly defines that a timestamping scheme similar to the heavy archive option is used to lazily timestamp the entries.

[1]Their exact wording was that the cost of WORM storage should be less than a factor of ten cheaper than WMRM storage cost [59].

In conclusion, the TBS-tree is a practical index structure that can be easily fine-tuned for different data sets, but it does not guarantee optimal access costs for updates or queries of any version.

4.3 Immortal DB

Lomet et al. have chosen the time-split B$^+$-tree (TSB-tree) as the basis when implementing multiversion support to the *Immortal DB* multiversion database system [54–57]. The multiversion database is built on top of existing Microsoft SQL Server code. An important aspect of the implementation was that the existing program code must be compatible with the new multiversion code. The multiversion functionality has been introduced gradually, first by chaining versions together in the original SQL Server B$^+$-tree index, and then by replacing the B$^+$-tree based index with the TSB-tree [55, 56]. Immortal DB therefore serves as an example that multiversion functionality can be gradually added to an existing database system.

The Immortal DB also introduced changes to the TSB-tree index structure. From our point of view, the important ones are the more specific explanation of lazy timestamping and the new splitting policies designed to cluster the data items that are alive at the current version more efficiently. The articles furthermore specify how the entries are stored in the data pages [55, 56] and how the pages can be compressed [56].

The entries in Immortal DB are initially timestamped with the temporary transaction identifier. When the transaction commits, a mapping between the transaction identifier and the commit-time version of the transaction is inserted into a *persistent timestamp table* (PTT). This table is stored in a B$^+$-tree index so that the contents are made persistent and can be recovered in the event of a system crash. Because all entries need to be timestamped during the first access, there will be many queries to the PTT. The Immortal DB therefore also maintains a main-memory-based *volatile timestamp table* (VTT) that serves as a cache for the PTT. In addition to storing these mappings, the VTT also contains a reference counter for determining how many entries there are that have not yet been timestamped. When new data items are inserted to the database, the reference count is incremented. When entries in the database pages are timestamped after the transaction that inserted them has committed, the reference counter is decremented. When the reference counter reaches zero, the mapping is removed from both the VTT and the PTT. The PTT is thus used only to provide recoverability: if the system fails, the VTT can be rebuilt from the PTT contents. The reference counter, however, is not present in the PTT, and is thus lost if the system fails. This can cause some entries to remain indefinitely in the PTT and VTT, because it is not known if there still exists entries that need to be timestamped with the commit-time version.

The Immortal DB articles also specify two new split policies for the TSB-tree, namely the WOB-tree split policy and the deferred split policy. The *WOB-tree split policy* [56] updates the IKS split policy (see Section 4.2, p. 58) so that whenever a page contains more than two thirds of live entries, it is first time-split and afterwards key-split. Otherwise the page is time-split, as with the IKS split policy. The split threshold (two thirds of the entries in this example) is now a configuration variable, but Lomet et al. use two thirds when there is no compression in the pages [56]. The *deferred split policy* [57] defers the key-split that is performed after the time-split in the WOB-tree split policy. That is, if a page that needs to be split contains enough live entries according to the threshold, then the page is time-split and a key-split is deferred by marking the page. When the marked page next needs to be split, it is key-split without first time-splitting it. Unmarked pages that have fewer live entries than the threshold are simply time-split, as in the WOB-tree and IKS split policies.

In addition to the deferred key-split, the deferred split policy introduces batch updates to the PTT table. Normally, whenever a transaction commits, the mapping from transaction identifiers to commit-time versions is inserted to the PTT, thus adding extra I/O operations to each commit operation. With the deferred split policy, the mappings are only inserted into the VTT during a commit operation. The PTT is then updated later on with a larger batch of mappings. This reduces the number of I/O operations as the PTT entries are clustered next to each other and also because some of the entries may already have been removed from the VTT and are therefore never inserted into the PTT at all. This happens if all the entries a transaction T inserted have been accessed and timestamped before the next batch of updates is applied into the PTT. In this situation, the transaction entry has already been removed from the VTT and is never inserted into the PTT.

With Immortal DB, the focus on optimizing the TBS-tree has moved from optimizing space usage into optimizing both space usage and query performance [56, 57]. The original TBS-tree (see previous section) was designed for data that is never deleted [58]. When data deletion is allowed, maintaining the number of live entries for each page is more challenging. Even though the new Immortal DB splitting policies are better suited for transaction histories that also contain deletions, the fact remains that pages are not merged, and thus key ranges cannot expand. If a current page of the TSB-tree covers the key range $[k_1, k_2)$, all the items in this range are deleted, and no new items are inserted that fall into that range, then the current page will contain no live entries. It will still be part of the current-version database, and key-range queries will process it (also recall the discussion in Section 3.3). The TSB-tree and the Immortal DB are therefore suboptimal multiversion index structures, and range-query performance may degrade when deletions are present.

4.4 Multiversion B$^+$-tree

The first optimal multiversion index structure presented was the multiversion B$^+$-tree (MVBT) of Becker et al. [7, 8]. The MVBT follows a single-update model, in which each update to the index creates a new version of the index. The update cannot be rolled back, so the MVBT structure is not transactional.

We can model a single multi-action updating transaction operating on the MVBT, if we accept that the transaction cannot roll back. The updates performed by the updating transaction T obtain versions that form a contiguous range $[v_1, v_2]$. This means that the intermediate states $v : v_1 < v < v_2$ also persist, even though they are never queried as they are internal to the transaction. For example, suppose that a transaction T first inserts a data item with key k, and then updates it again and again. If the last action of the transaction with commit-time version v on the data item with key k is an update with value w, then an optimal multiversion index only records the data item (k, v, w). In contrast, the MVBT stores information of uncommitted versions of the data item that never will be committed.

There are some benefits to this model, however. Concurrency control is straightforward, because only one updating transaction is allowed to operate on the index at a time. As we will show in Section 5.1, multiple read-only transactions can be allowed to operate on the index concurrently with the single updating transaction. Because there is only a single updating transaction, and every update operation creates a new version of the index, the most recent committed version can be maintained in a single variable. The variable v_{commit} denotes the current version of the MVBT index. At the beginning of each action, v_{commit} is incremented, and each operation on the index is then tagged with the incremented value. For correct operation, the read-only transactions may only query the latest version after the active updating transaction has committed. Therefore the implementation should maintain a separate variable that records the latest committed version that precedes the version of the active updating transaction. This variable is then read by the read-only transactions, and updated when the active updating transaction commits. For simplicity of the explanations, we will assume that v_{commit} denotes the latest committed version as seen by the updating transaction, and that it is incremented at the beginning of the updating transaction.

As in the TSB-tree, pages of the MVBT cover axis-aligned rectangles of key-version space. The key-version rectangle of a page p is unambiguously defined by its components: the key range $\mathsf{kr}(p)$, and the version range, or life span $\mathsf{vr}(p)$. The pages form a directed acyclic graph with leaf pages at level one, and index pages on consecutively higher levels[2]. In the MVBT, life spans are explicitly stored in each entry in the index structure. Leaf-page entries are thus tuples of the form $(k, \overrightarrow{v}, w)$, where k is the key, $\overrightarrow{v} = [v_1, v_2)$ is the life span, and w is the associated data stored in the data item. When a leaf-page entry is first inserted into the index, its life span is set to $\overrightarrow{v} \leftarrow [v_{commit}, \infty)$, indicating that it has been inserted at the current version and that it has not yet been deleted. If an entry with a life span $\overrightarrow{v} = [v_1, \infty)$ is deleted, the life span is replaced with $\overrightarrow{v} \leftarrow [v_1, v_{commit})$, thus marking that the entry was deleted at the current version. The life span of a leaf-page entry is therefore not static, but it changes when the item is deleted. After an item has been deleted (or updated by replacing it with another item with a different associated value), the life span of the historical entry becomes static.

For each level l in the MVBT index, the pages of level l partition the key-version space into disjoint rectangles. The pages of level l cover the entire key-version space, except for those ranges of versions v for which the height of the search tree S_v was lower than l. An example of this is shown in Figure 4.2. In the example, the search tree S_0 of version v_0 contains only the single leaf-level root page p_1, and therefore the MVBT index does not contain any pages at the second level before version v_1. At version v_1, the root page is split into pages p_2 and p_3, and therefore a new root page p_9 at the next level is created.

The index pages contain pointers, or *routers*, to child pages [7, 8]. A pointer from a parent page p to a child page p' is represented by a tuple of the form $(\overrightarrow{k}, \overrightarrow{v}, p')$, where p' is the page identifier of the child page p'. The key range $\overrightarrow{k}$ and life span $\overrightarrow{v}$ form the router $(\overrightarrow{k}, \overrightarrow{v})$ that

[2]Becker et al. say that leaf pages are at level zero, but as this is just a matter of convention we have modified the definition here to better suit the convention used in this dissertation.

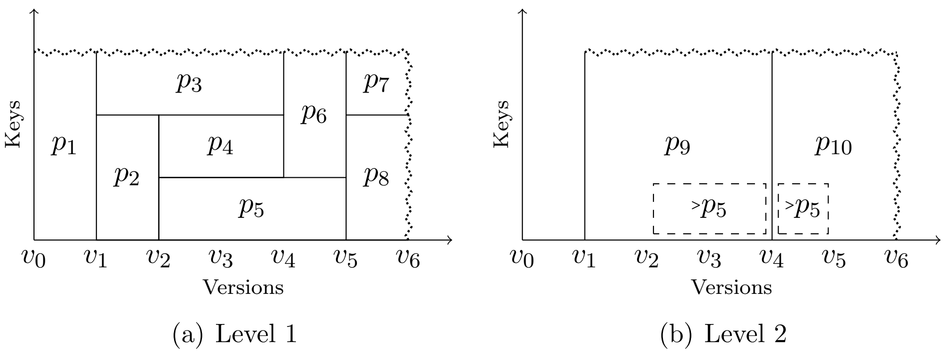


Figure 4.2. Partitioning of the key-version space in the MVBT. In the image, $v_{commit} = v_6$, which is indicated by the open ends of the life spans of pages p_7, p_8, and p_{10}. The level 2 image shows routers to page p_5 in pages p_9 and p_{10}.

guides the search to the child page. The router is set to the intersection of the key ranges and life spans of the parent page and the child page. This is illustrated in Figure 4.2(b), where the routers to the child page p_5 are shown in both parent pages p_9 and p_{10}. For illustration, the page identifiers stored in index pages are prefixed with a marker $>$. A parent page p contains routers to each child page p' such that $\mathsf{kr}(p) \cap \mathsf{kr}(p') \neq \emptyset$ and $\mathsf{vr}(p) \cap \mathsf{vr}(p') \neq \emptyset$, and the format of the MVBT index therefore follows the general convention shown in Figure 3.1(b) on page 37.

The structure-modification operations on the MVBT are based on the *version-split* operation, in which a live page p is *killed*; that is, p is split at the current version v_{commit}, and a new live copy p' is created. The life span $[v, \infty)$ of the old page p is cropped to $[v, v_{commit})$, and the life span of the new page p' is set to $[v_{commit}, \infty)$. All the live entries in p are copied to p'. Page p is now considered a *dead page* and is only used for historical queries, and the new page p' is used for current-version queries. This operation is the same for index pages and leaf pages. All structure-modification operations target the new page p', and the killed page p becomes static. Dead pages are never modified again, although they might be deleted later to save space (see the discussion of purging old versions in the MVBT article by Becker et al. [7, 8]). This is stated more formally below.

Invariant 4.7. Dead pages in the MVBT can only be modified by a purging process that is run to remove old versions. If an old version is removed, it can no longer be queried. From the viewpoint of a user transaction querying for any version, all encountered dead pages are static.

There are two basic structure-modification operations in the MVBT: page split and page merge. A page split is triggered when an insertion is attempted into a page that is full and cannot accommodate the new entry, and a page merge is triggered when the number of live entries in a page falls below min-live. To avoid thrashing between splitting and merging the same page, all pages in the MVBT must contain between min-split = min-live + s and max-split = $B - s$ entries immediately after any structure-modification operation, where s is a split-tolerance variable. The variable s determines how many updates must at least be performed on the page before a new structure-modification operation is required.

When a page p is *split*, it is first version-split into a new live page p'. As described above, page p' contains the live entries copied from page p. If the number of live entries in page p' is now between min-split and max-split, the operation terminates, because the page can accommodate at least s more updates before it needs to be either split or merged. If the number of live entries is above max-split, page p' is further key-split into two pages. The key-split operation is identical to a standard B$^+$-tree key split. This is possible also for index pages, because at this point the new page p' contains only live entries, and thus there are no entries with overlapping key ranges. If the number of live entries on page p' is below min-split, the page must be merged with a sibling page. In this case, a sibling page p'' is located, version-split, and merged with the page p'. If the merged page contains more than max-split entries, it is further key-split into two pages, similar to the key-split situation above.

The *page merge* operation, which is triggered when the number of live entries on a page p falls below min-live, is identical to the merge operation that takes place after a version-split. That is, page p is first version-split, then a sibling page p' is located, version-split, and the resulting live pages are merged. The merged page might again have more than max-split entries, in which case it is further key-split into two pages. In this situation, the merge operation resembles the key-redistribution operation of the B$^+$-tree.

Because no historical data is removed from the killed page p in all the structure-modification operations, and all the live entries are copied to the new page p', we can deduce an even stronger invariant for the MVBT:

Invariant 4.8. All entries in the MVBT pages remain in place. They are never moved to another page. Only the deletion time of an entry may be changed, always from ∞ to the current version v_{commit}.

When a non-leaf page is version-split, the copying of entries creates a new parent, in addition to the old one(s), for the child pages pointed to by the copied entries. As noted before, the MVBT is not a tree but a directed acyclic graph, which may have several roots. When a root page is split, a new root page is created, but the old root page is still used as a starting point when searching for historical entries. To accomplish this, a separate structure called *root** (see Definition 3.4) is used to store all the different roots of the MVBT. The *root** structure can be implemented, for example, as a B$^+$-tree that contains the page identifiers of the different root pages, indexed by their creation versions. It is assumed that the number of different roots is small, and that the *root** structure can fully reside in main memory.

The optimality of the MVBT is based on maintaining the following invariant:

Invariant 4.9. Assuming that each version of the MVBT index consists of only a single update, then for all versions v and all pages p, either (1) page p contains at least min-live entries that are alive at version v, where min-live is a configuration variable that is linearly dependent on the page capacity B; or (2) page p is a root page of a search tree S_v and p contains at least two entries that are alive at version v; or (3) page p is the single page of a search tree S_v that has a height of one; or (4) page p is not part of the search tree S_v and therefore contains no entries that are alive at version v.

The first case of this invariant states that each page either contains enough entries of the queried version so that all queries (including the key-range query) are efficient, or the page is not part of the search tree of the queried version and is therefore not processed at all. The two other cases are special cases listed for completeness: a root page may contain as few as two live entries, if there are no more pages at the next lower level. Similarly, if all the n live entries of a version v fit on a single page, the search tree S_v contains only a single page that has exactly n live entries. Because this invariant is maintained, each search tree S_v is asymptotically equivalent to a single-version B$^+$-tree index, and all queries in the MVBT index are optimal (see Definition 3.3), if the root page of the search tree of the queried version is known.

Like the TSB-tree, the space complexity of the MVBT index is also $\mathcal{O}(n/B)$ database pages, where n is the number of updates in the database history [8, 78]. The time complexity of the MVBT actions has been shown by Becker et al. [7, 8], and we reproduce the results here:

Theorem 4.2. Assume that every transaction consists of only one update, the transactions all run in a serial order and they all commit. The cost (see Definition 3.2) of the current-version single-key operations (key query, key insertion, and key deletion) in the MVBT index, when the roots of the queried search trees are known, is $\Theta(\log_B m_{v_{commit}})$ pages, the cost of the single-key query operation for version v is $\Theta(\log_B m_v)$ pages, and the cost of the key-range query operation for version v is $\Theta(\log_B m_v + r/B)$ pages, where m_v denotes the number of data items that are alive at version v, r is the number of live entries in the queried range and B is the page capacity.

Proof. By Invariant 4.9, each page of the search tree S_v of version v in the MVBT contains at least min-live entries of version v, except possibly the root page of S_v. The MVBT is always balanced, as shown by Becker et al. [7, 8]. The height of the search tree S_v is thus $\Theta(\log_B m_v)$, because there are m_v entries that are alive at version v, and min-live is a linear function of the page capacity B. This explains the logarithmic part $\log_B m_v$ of the costs, as the search tree must be traversed from the root to the correct leaf node. The cost of the range query operation is $\Theta(\log_B m_v + r/B)$ by Theorem 3.1, because each page of the search tree S_v contains at least min-live entries that are alive at version v. □

In the discussion above, we have omitted the page accesses required for locating the root page of version v, when performing a query that targets a version v. As mentioned earlier, the MVBT uses a separate *root** structure to store the root page identifiers of different versions. In practice, the *root** is either a table or a single-version B+-tree, indexed by the version during which the root of the search tree has changed. For example, suppose that the *root** contains the following entries: $\{(v_0, p_1), (v_3, p_5), (v_{20}, p_{14})\}$. Reading from this, versions $[v_0, v_3)$ have page p_1 as their root page, versions $[v_3, v_{20})$ use page p_5, and the rest of the versions $[v_{20}, \infty)$ use page p_{14}.

Let us now discuss the cost of locating the correct root page for a user action. If the *root** is stored in a B+-tree index, locating the current page requires access to $\Theta(\log_B n)$ pages, where n is the number of page identifiers stored in the *root**. If the *root** is stored as a table, the correct root page identifier can be located by a binary search that accesses $\Theta(\log_2 n)$ table slots, requiring access to $\Theta((\log_2 n)/B)$ pages. The number of page identifiers in the *root** is determined by how often the root page of the current version search tree has changed. As noted by Becker et al. [8], if the MVBT index has been created by a sequence of insertions only, the leftmost path of the current version search tree $S_{v_{commit}}$ contains all the root pages of the MVBT index, and the number of different roots is therefore $n = \log_B m = \log_B m_{v_{commit}}$, where m is the number of data items stored in the entire index structure. We therefore expect that the number of different roots will be small in practice. It is possible, however, that the number of roots is much larger.

The worst-case scenario would be to alternatively insert an entry in one version into an originally empty database, and delete it in the next. If the index implementation explicitly stores null root page identifiers $(v, \perp)$ to the *root** to denote that the search tree of version v is empty, then the number of roots is directly dependent on the number of data items stored in the index ($n = \Theta(m)$), because there must then exist a page identifier for each version in the *root**. If the implementation reuses the root page identifier of the previous version leaf-level root page, then the page identifier in the *root** does not change for each update. The root page must however change after $\Theta(B)$ updates, because the single root page can only accommodate that many entries. The number of root pages is therefore at least $n = \Theta(m/B)$, which is still linearly dependent on the number of versions, assuming that the page capacity is a constant. In this pathological worst case, locating the correct version from the *root** can have an asymptotic cost that is higher than the cost of the operation itself. However, the root page does not need to be separately located for each query operation.

For updating transactions, and when querying for the most recent version, the root page of version v_{commit} is required to begin the search tree traversal. For efficiency, the MVBT index implementation should cache the page identifier of the current-version root page. Because there is only one updating transaction, and read-only transactions may not target the version that is being updated, the page identifier can be stored in a single variable without using locks to protect its value. Therefore, there is no additional cost for performing any update action, or any query action that targets the latest version. Once the updating transaction completes, creating version v, the root page of version v remains static, and thus all entries representing historical versions in the *root** can be cached. Each read-only transaction that targets a version $v < v_{commit}$ needs to locate the page identifier of the root page of the search tree S_v only once. The correct page identifier can also be provided by the context. We therefore feel justified in stating that locating the root page of the queried version from the *root** has a negligible effect on overall query performance.

We stated earlier on that the MVBT follows a single-update model, so that only one update may receive the same version. Let us now consider what happens if we try to index more than one update with the same version in the MVBT index. Consider, for example, simply inserting entries with consecutive keys $1, 2, 3, \ldots, n$ to the MVBT. The scenario is shown in Figure 4.3, with an illustrative page capacity of three entries per page. The first three entries are inserted to page p_1, therefore filling the page. The insertion of the fourth item leads to an overflow which triggers a page split. The page split begins with a version-split operation, which in turn causes the life span of the old page and its entries to degenerate into an empty interval $[1, 1)$. This page does not hold any relevant information as it is no longer a part of any version of the database. The pages created earlier by the same transaction are the cause of this problem. As it turns out, this problem can be remedied by redesigning the algorithms as we will show in Section 5.5. In this problem scenario, the page could be key-split directly, without first version-splitting it.

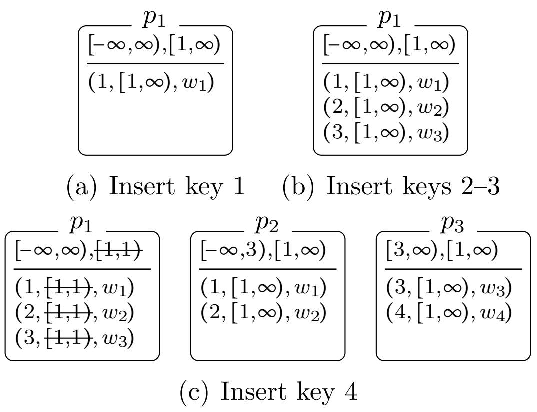


Figure 4.3. MVBT problem scenario. Insertion of key 4 causes an invalid split. The format of the page header is key range, life span; and the format of the entries is (key, life span, data).

Although we have concentrated on queries with a fixed version (that is, *x/–/point* queries, see Section 2.3), the MVBT can also be modified so that version-range queries (*x/–/range* queries) can be performed efficiently on the index structure. The modifications are explained by van den Bercken and Seeger [10]. In practice, page identifiers of temporal predecessor pages are added to each page. A page p' is a said to be the *temporal predecessor* of page p if the live entries of p' were copied to p during a version-split operation; that is, if the key ranges of p and p' overlap and the deletion time of p' is the creation time of p.

When the page identifiers of temporal predecessor are maintained, queries that target ranges of versions can be processed as follows. First, the pages that cover the last version of the queried version range are located. The preceding versions on the range can then be located by using the page identifiers of temporal predecessors. It is sufficient to allocate space for at most two page identifiers in each page, because each page of the MVBT index can have at most two temporal predecessors. This can be verified by enumerating all the possible structure-modification operations. For more details, refer to the article by van den Bercken and Seeger [10].

In conclusion, the MVBT structure is an optimal multiversion index structure, if the root page of the searched version is assumed to be known (and in practice it can be located with negligible cost), but it has three shortcomings: the updates created by multi-action transactions cannot be assigned the same version number, only a single updating transaction can operate on the index at a time, and the transaction cannot be rolled back. In the next chapter, we present a redesign of the MVBT algorithms that allows multiple updates to be performed within the same transaction, so that each update is assigned the same version, and the transaction can be rolled back.

4.5 Multiversion Access Structure

Varman and Verma have also described a multiversion index structure that is optimal according to their definition. Their structure is called the *multiversion access structure* (MVAS [92]), and its structure is similar to the structure of the MVBT. The optimality definition of Varman and Verma is less restrictive than our definition (Definition 3.3), and therefore we cannot consider the MVAS to be optimal. To distinguish the optimality concepts, let us define the optimality of the MVAS separately. According to Varman and Verma, an optimal multiversion structure has a cost of $\mathcal{O}(\log_B m)$ MVAS pages to locate a queried key k in a given version v, where m is the total number of updates in the database history (or, equivalently, the total number of unique data item entries in the index). We will call this *m-optimality*; that is, we say that the MVAS is *m-optimal.* This is in contrast to our required cost of $\mathcal{O}(\log_B m_v)$ index structure pages, where m_v denotes the number of data items alive at version v.

With this altered definition, the MVAS is m-optimal for the query types we require of multiversion indexes (that is, queries of the type *x/–/point*; see Sections 2.3 and 2.4), and also for single-key and snapshot version-range queries (i.e., *x/–/range* queries, where x is either *point* or $*$). Version-range queries for key-ranges (e.g., queries of the type "retrieve all data items with keys in the range $[k_1, k_2)$ that were alive between the versions v_1 and v_2") are however not m-optimal.

The structure of the MVAS is close to that of the MVBT, with the following notable exceptions: (1) the key-range of a page can be altered for later versions, (2) there is no *root** structure, (3) leaf pages are linked together in page creation order, and (4) the index incorporates an access list structure to facilitate version-range queries. These differences will be discussed in more detail below. In other respects, the MVAS shares the shortcomings of the MVBT: the version of the index structure must change after each update, and the structure cannot by updated concurrently by multiple transactions.

As described in the previous section, when a structure-modification operation is triggered in the MVBT on page p, all the involved pages are version-split before the entries are distributed into new pages. If necessary, a sibling page p' is located, version-split, and the entries are redistributed between the two new pages. The sibling page p' might still have usable space left, however. Instead of always creating a copy of the sibling page p', the MVAS reuses the physical page p', and simply inserts a new router to the parent. The MVAS is thus a multigraph with possibly more than one edge between its nodes. The searches for previous versions use the historical router in the index page, and the searches for newer versions use the new router. This means that the key ranges of pages can change, and the regions of key-version space that the MVAS pages cover are not rectangles. While this method does use the space more efficiently, the asymptotic space cost remains unchanged [92], and the index structure is a bit more complicated. Nevertheless, a space-optimization scheme such as this may give some practical benefits, and it could be applied to the MVBT also, if the structural invariants were updated.

The MVAS does not have a separate *root** structure to track the roots of different versions, but rather uses a single root page like the TSB-tree. The search tree height is thus the same for each version, and the root-to-leaf path length cannot become shorter even if entries are deleted from the current version. The MVAS does track the single root for the current version separately, so that updates and queries that target the current version have the asymptotically lower cost of $\Theta(\log_B m_v)$ pages.

To facilitate snapshot queries of different versions (i.e., queries that fetch all the entries that are alive at a given version), the MVAS has sibling links between leaf pages. The pages are linked to each other in the order of their creation time, so that a leaf page p has a link to the leaf page p' if p' is the next leaf page that is created after p was created. Varman and Verma do not explicitly define how these links are maintained, but we assume that the page identifier of the latest leaf page that has been created is maintained, and used to locate the page p that needs to be updated when a new page p' is created. This adds some complexity to concurrency control and deadlock avoidance when a latching protocol is used to protect the integrity of database pages.

The leaf-page links are used to efficiently locate all entries that are alive between any two specified versions v_1 and v_2 such that $v_1 < v_2$. The snapshot version-range query ($*/-/range$) is composed of two subqueries: first, the entire set of entries alive at version v_1 is queried. This query is an m-optimal range query, similar to the key-range query in the MVBT index. After that, the leaf-page sibling links are followed, starting from the most recently created leaf page that was encountered during the first subquery. Because the leaf pages are linked in increasing creation order, all entries that have been created between versions v_1 and v_2 must be located in the leaf pages that were created before v_2; or in the pages that were alive at version v_1, and thus were processed during the first subquery. This method is thus an efficient way of locating all the entries that were alive between the two given versions.

Finally, the MVAS contains a separate index structure called *access list* that is used to efficiently locate all the data items with key k that were alive between two versions v_1 and v_2. The access list is a separate index structure that resembles the versioned B$^+$-tree of Section 3.2; that is, the entries are ordered first by the keys, and then by their versions, in reverse version order. The different versions of each key are thus clustered close to each other. Because there is now a total ordering between the entries in the access list, the leaf pages of the structure can be linked together, as is often done in B$^+$-trees. The problem of finding all the data items with key k that were alive between two versions now reduces to finding the correct version of the entry with key k from the access list and then traversing the sibling links to locate all the versions between the queried range. If the correct starting point can be found, this operation is m-optimal. For this purpose, the entries of the MVAS and the entries of the access list are linked together with two-way links.

When querying for the history of a key k between versions v_1 and v_2, the MVAS index is used to locate the entry that is alive at version v_2. The search then follows the link in this entry to the access list, and traverses the access list by the sibling links to locate all the versions of the entry between v_1 and v_2. However, an entry that was alive during or after version v_1 might have been deleted before version v_2, so that the MVAS search will not find any entry and thus does not find a pointer to the access list. This entry is still alive at a version $v : v_1 \leq v < v_2$, so it should be included in the result set of the query operation. In this case, the B+-tree index of the access list is used to locate the first entry in the queried history. The two-way links between the MVAS and the access list are needed because leaf pages in both structures may be split and the entries may thus be moved around. The links then need to be followed to update the references in the other structure. There is also an extra level of indirection in the MVAS, because an entry may be copied to multiple pages due to a version-split operation that creates copies of entries with life spans that cross the split boundary. Only the first entry in the MVAS contains a link to the access list, and the successive copies of the entry contain links to the first entry. When the address of an access list entry changes, it is then sufficient only to change the address in the first MVAS entry.

There is, however, yet another complication due to the access list. When entries are added to the access list, some nearby pointer from the MVAS is followed to locate the correct page of the access list to insert the corresponding entry there. This is done so that it is not required to traverse the B+-tree index to locate the correct page. Recall that the MVAS separately tracks the root page of the current version, so that update actions have the optimal cost of $\Theta(\log_B m_v)$ pages. The access list size is, however, dependent on the total number of updates performed in the history, m, and traversing the B+-tree index of the access list thus requires $\Theta(\log_B m)$ pages. This additional cost would increase the cost of the update operations above the asymptotically optimal cost. The pages of the access list therefore have two-way links between parent pages and child pages, so that leaf pages can be split without having first searched the path from the root to the leaf page. This means that whenever an index page of the access list is split, each child page whose pointer is moved to the new page needs to be updated. In addition to being costly because there are $\Theta(B)$ pages to update, the approach is also problematic for concurrency control, because the pages all need to be latched at the same time. Varman and Verma show that the amortized complexity of this operation is constant, because it can only occur after enough other operations have been performed [92], and the approach thus retains the amortized asymptotic optimality of the update actions. Like the TSB-tree and MVBT, the space complexity of the MVAS (including the access list structure) is $\mathcal{O}(n/B)$ database pages, where n is the number of updates in the database history [92].

As a conclusion, the MVAS is a structure that is closely related to the MVBT index. Similarly to the MVBT, transaction rollback and recovery cannot be optimally performed on the MVAS. There are several differences in the structures, but all the features of the MVAS could be implemented on the MVBT as well. The access list structure, while preserving the asymptotic behaviour of the algorithms, incurs a constant overhead to the update operations and an occasional high cost when an index page is split and half its child pages need to be updated. In addition to taking much time, the index-page-split operation has to latch many pages at the same time, thus reducing the concurrency of the structure.

CHAPTER 5

Transactions on the MVBT

As shown in the previous chapter, there are efficient multiversion index structures available, but there is no single structure that is both optimal and that can be used in a concurrent transactional environment. We reviewed the result by Becker et al. [7, 8] that showed that the multiversion B$^+$-tree (MVBT) is an optimal multiversion index structure, but it follows a single-update model, and the update cannot be rolled back. In this chapter, we present our redesigned MVBT, called the *transactional multiversion B$^+$-tree* (TMVBT). The TMVBT adds transactions to the MVBT by redesigning the structure-modification operations (SMOs) so that multiple data-item updates can be performed within a single transaction, and the updates can be rolled back. The TMVBT structure was first introduced in our previous article [35]. In the discussion here, we explain the algorithms in more detail, and provide detailed proofs for the properties of the structure.

We begin the chapter by describing the implementation of the transaction model of Sections 2.5 and 2.6 for the TMVBT in Section 5.1. After that, Section 5.2 defines the concept of active entries, which is needed for maintaining the optimality in the presence of multi-action transactions, and Section 5.3 describes the structure of the TMVBT. In Section 5.4, we show how the user actions are performed, and in Section 5.5, we describe the structure-modification operations triggered by the user actions. Finally, in Section 5.6, we illustrate why we cannot allow multiple updating transactions to operate on the index concurrently, and in Section 5.7, we summarize the discussion on the TMVBT index.

5.1 Multi-Action Transactions

We allow two kinds of transactions to operate on the TMVBT concurrently: any number of read-only transactions (as defined in Section 2.5) and at most one updating transaction (as defined in Section 2.6) at a time. For reasons explained in Section 5.6, we cannot allow more than one updating transaction to operate on the TMVBT at a time. Concurrent updating transactions are discussed in the next chapter.

In contrast to the MVBT (Section 4.4), each updating transaction operating on the TMVBT can perform any number of updates, and the updates all receive the same version. Because only one updating transaction can operate on the TMVBT at a time, the commit order of the transactions is known during the execution of the transactions, and each data-item update can be directly performed with the correct transaction-time version. In the context of the TMVBT, we use the term *version* to denote these transaction-time instants. This does mean, however, that the version assigned to an updating transaction cannot be based on the real time instant of the commit action, because that is not known at the beginning of the transaction. We thus assume that the versions used in the TMVBT are increasing integer numbers that are assigned at the beginning of the updating transaction. The versions can be based on an increasing counter value or they can be based on the real-time instant of the begin action of the transaction, as long as they are increasing values. For simplicity, in the discussions in this chapter, we assume that the active version variable (defined below) is based on an integer value that is incremented by one each time a new updating transaction begins.

Like the MVBT, the TMVBT also maintains a commit version variable v_{commit} that records the version of the latest committed transaction. The TMVBT also maintains an active version variable v_{active} that holds the version of the current updating transaction. If there is no active updating transaction operating on the index, $v_{active} = v_{commit}$. When an updating transaction starts, the active version variable is incremented, $v_{active} \leftarrow v_{active} + 1$. When the single updating transaction commits, the commit version is incremented to match the active version variable $v_{commit} \leftarrow v_{active}$. These version variables therefore tell whether there is an active updating transaction running on the TMVBT index: if the active version variable is larger than the commit version variable, then there is an updating transaction running, and no other updating transaction can begin. Read-only transactions can always target any version that is less than or equal to the commit version v_{commit}, unless purging of old versions is implemented (see discussion in Section 4.4 and in the MVBT articles [7, 8]). In that case, the minimum version that can be accessed must also be maintained in a separate variable.

The transaction model used for the TMVBT is the transaction model explained in Sections 2.5 and 2.6. The log records written by a transaction T for the actions presented here must also contain the transaction identifier $\mathsf{id}(T)$. Because the common form of log records $\langle T$, **action**, $\ldots\rangle$ already contains the identifier T, we use T to mean that the transaction identifier $\mathsf{id}(T)$ is written in the log records. The control actions of the transaction model are implemented as shown below; the query and update actions are discussed in Section 5.4.

- **begin-read-only**(version v): begins a new read-only transaction; this action takes a short-duration read lock on the v_{commit} variable, checks if $v \leq v_{commit}$, and records the value $\mathsf{snap}(T) \leftarrow v$ for the transaction. If $v > v_{commit}$, the transaction is aborted.
- **begin-update**: begins a new updating transaction T; this action takes a commit-duration write lock on the active version variable v_{active}, increments the variable $v_{active} \leftarrow v_{active} + 1$, and assigns it to the transaction: $\mathsf{id}(T) \leftarrow v_{active}$. A redo-undo log record $\langle T$, **begin**, v, $v_{active}\rangle$ is written, with v denoting the previous value of the variable v_{active}, but the log is not forced to disk.
- **commit-update**: commits the active updating transaction T by (1) taking a commit-duration write lock on the committed version variable v_{commit}, (2) updating the variable $v_{commit} \leftarrow v_{active}$; (3) writing a log record $\langle T$, **commit**, $v_{commit}\rangle$; (4) forcing the log onto disk; and (5) calling the release-version action.
- **release-version**: this action does nothing, because the updating transaction has already assigned correct versions to each updated data item entry. If the versions of the TMVBT index should be timestamps that are based on the time of the commit action (which is not known at the beginning of the updating transaction), then this action can perform the necessary changes to update the entries.
- **abort**: labels the updating transaction as aborted and starts the backward-rolling phase. This action writes the log record $\langle T$, **abort**$\rangle$.
- **finish-rollback**: finishes the rollback of an aborting transaction by decrementing the active version variable $v_{active} \leftarrow v_{active} - 1$, writing a log record $\langle T$, **finish-rollback**, $v_{active}\rangle$, and forcing the log to disk.

All the update actions of an updating transaction are logged using the write-ahead logging protocol as in ARIES [66]. In addition to the log records described above, redo-undo log records are written for an **insert** action and a **delete** action, while redo-only log records are written for an **undo-insert** action and an **undo-delete** action. These log records are described in Section 5.4. A read-only transaction does not create any log records; it only stores transient control information in the active-transactions table when it begins, and removes that information when it commits.

5.2 Active Entries

Recall from Figure 4.3 on page 68 the problem of inserting multiple data-item entries into the MVBT index with the same version. When performing a version-split operation on page p, a new copy p' of the page is created and the life span of the original page p is truncated to the current version and the page is left as it is for use in historical queries. If the page p was created by the same transaction that triggers the version split, the life span of the page will degenerate into an empty range, and the page will thus not be part of any search tree in the database. In these situations, the key split can be performed directly on the page p, without applying the version-split operation first.

Let us now define the concepts of active entries and active pages to classify the situations where a version-split operation is not required and in fact must not be performed. Remember that the single updating transaction always has the version v_{active} as its identifier and uses that version to stamp the data-item updates and structure-modification operations.

Definition 5.1. An *active entry* (or *active page*, respectively) in the TMVBT index is an entry (page) that has a life span of $[v_{active}, \infty)$. An active entry (page) has been created earlier on by the same updating transaction. Entries (pages) that are not active are called *inactive entries* (*inactive pages*). □

As stated in the previous chapter, read-only transactions may only read versions that have a commit-time version of at most v_{commit}. This leads to the following observation:

Invariant 5.2. Read-only transactions in the TMVBT index only read inactive entries and pages. Active entries and pages are only seen by the single active updating transaction.

If the active updating transaction is deleting an active entry, the entry can be physically removed from the index, instead of changing its life span. This does not invalidate partial persistence, because the active entry was created by the same transaction, and thus did not exist before the updating transaction first inserted it. Updates that are internal to the transaction are not visible outside the transaction and must not consume space in the index.

Invariant 5.3. When a single updating transaction T deletes an active entry (created by T), the entry is physically removed from the TMVBT index. Similarly, if T updates an active entry it physically removes the old entry and creates a new active entry to replace the old one.

When performing a version-split operation on a page p at version v in the original MVBT index, Becker et al. suggested that the entries that are left in page p may be left unmodified [8], so that the life spans of the entries e_i that were alive at version v remain unbounded on the above; that is, of the form $[v_i, \infty)$, where $v_i < v$. This does not affect any queries, because only historical queries targeting versions $v' < v$ will ever end up in the historical page p; thus, even if an entry e_i is deleted by a transaction with a version $v'' > v$, the queries targeting those newer versions will never encounter the now-outdated entry e_i on the historical page p. However, if a transaction T stores a previously used path as a saved path (see p. 91) and reuses the path later on, it is possible that the pages in the saved path are no longer valid. The transaction T cannot ascertain the validity of the pages unless the consistency of the life spans of all the entries and pages is maintained, so that the deletion times of entries in historical pages are set to the deletion time of the historical page. In the TMVBT index, we explicitly require that the life spans of entries that are left on a historical page are cropped so that they end at the version during which the page was split:

Invariant 5.4. When a page p in the TMVBT index is version-split into a new page p' at version v_{active}, all live entries $(k, [v, \infty), w)$ such that $v < v_{active}$ are processed as follows: a live copy $(k, [v_{active}, \infty), w)$ is created and inserted into the new live page p', and the live entry at page p is changed to the historical entry $(k, [v, v_{active}), w)$. All active live entries of the form $(k, [v_{active}, \infty), w)$ are physically moved to the new page p'.

This invariant is required in order that the key-version regions of all the entries of a given level of the TMVBT index do not overlap, as shown in Figure 4.2 on page 63. By adhering to these rules, we can also obtain the following lemmata:

Lemma 5.1. Active pages in the TMVBT only contain active entries.

Proof. An active page is a page that was created by the active updating transaction. When the transaction commits, the page immediately becomes inactive. When an active page p was created, all the live entries that were copied to it were changed so that their life spans start at the split boundary (i.e., at version v_{active}), thus making the copies of the entries active. If the active entries are changed in any way by the same transaction, they will be physically deleted or replaced by new active entries, as per Invariant 5.3. $\square$

Lemma 5.2. Active pages have at most one parent.

Proof. Multiple parents in a multiversion index are caused by multiple routers to the same page p in the index pages above the page p. When an active page p is created, a new index entry i_p is inserted to the parent page p'. Note that the index entry i_p is also active, and will remain active until the current active transaction commits. When the current transaction commits, both the entry i_p and the page p will immediately become inactive. By Invariant 5.4, active entries are physically moved during a version-split operation. If the parent page p' is version-split before the active transaction commits, the index entry i_p is physically moved to the new page, thereby preventing the creation of new copies of i_p. Because there can be only a single index entry i_p pointing to an active page p, active pages can only have a single parent. $\square$

All the entries of active TMVBT pages have the same life span of $[v_{active}, \infty)$. This holds for both leaf pages and index pages, and is illustrated in Figure 5.1. Because of this fact, we can in fact disregard the life spans of entries when performing an operation on active pages: in effect, we can treat active pages as if they were pages in a non-versioned B$^+$-tree index. The extended TMVBT algorithms are based on this observation. The algorithms themselves are explained in detail in Section 5.5.

For an example, let us review the problem scenario in MVBT as depicted in Figure 4.3 on page 68. In the TMVBT, the page p_1 is active, and thus it can be key-split directly without version-splitting it first. The operation of the same transaction, executing on the TMVBT index, is shown in Figure 5.2.

5.3 Transactional Multiversion B$^+$-tree

As we explained in the previous chapter, only the MVBT index [7, 8] can be considered optimal when updating transactions follow a single-update model, although the MVAS of Varman and Verma has access cost guarantees that are close to optimal (m-optimal, see Section 4.5). We have chosen the MVBT as the basis of our work, instead of the MVAS, be-

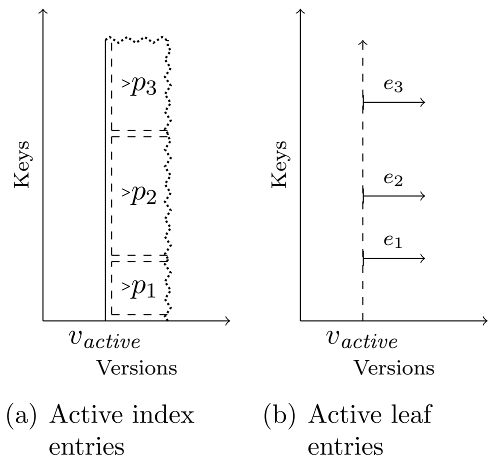


Figure 5.1. Active entries in the TMVBT index. The index page contains three index entries with routers to pages p_1, p_2, and p_3.

cause (1) the page reusing rules of the MVAS make the structure of the pages more complicated without improving the space complexity bounds, (2) the lack of a separate *root** structure makes history queries less efficient, and (3) the access list incurs a high maintenance cost. Nevertheless, the improvements presented in this chapter could also be implemented on the MVAS index structure.

The *transactional multiversion B⁺-tree* (TMVBT) index, which was first introduced in our previous article [35], is a directed acyclic graph with multiple root pages that is based on the multiversion B⁺-tree of Becker et al. [7, 8]. The original MVBT structure was reviewed in Section 4.4. The different roots of the TMVBT index are stored in a *root** structure, exactly as in the MVBT index. The page format in the TMVBT is identical to that of the MVBT, with the addition of recovery information required for our ARIES-based recovery algorithm, such as a Page-LSN field that stores the log sequence number (LSN) of the log record of the latest update on the page. We assume that each page p explicitly stores the life span $\mathsf{vr}(p)$, the key range $\mathsf{kr}(p)$, and also the height of the page. The height of a page is one for all leaf pages, and greater for index pages.

As discussed in Section 4.4, the MVBT has three variables that determine how many live entries there are in each page and how often pages are split or merged. These variables are used in the TMVBT in the same meaning. The variable min-live determines the minimum number of live entries that must be present in each live page (see Invariant 4.9 on page 65), and variables min-split and max-split control how many live en-

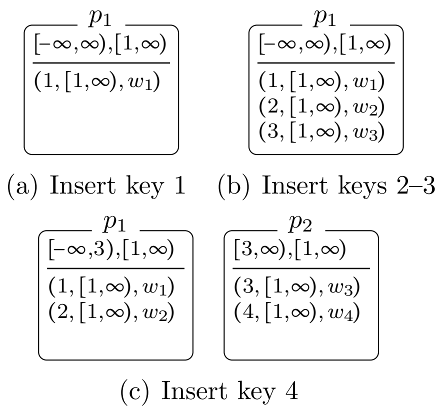


Figure 5.2. Key split without version split in the TMVBT. A key-split is triggered by the insertion of key 4. The leaf page contains three entries, namely e_1, e_2, and e_3. The format of the page header is key range, life span; and the format of the entries is (key, life span, data).

tries must be present in each live page created by a structure-modification operation. Becker et al. use the term *weak version condition* to refer to the first requirement, and the term *strong version condition* to refer to the second [7, 8].

The variables min-split and max-split are defined as min-split = min-live+ s, and max-split = $B - s$, where s is a *split tolerance variable* that determines how many actions must at least be performed on the page before a new structure-modification operation is required. If the strong version condition holds, then at least s entries can be deleted from the page before the number of live entries falls below min-live, and similarly at least s entries can be inserted to the page before the page becomes full. In effect, s is used to prevent thrashing. When a page has more than max-split entries immediately after a version-split, it will be key-split into two pages. We thus require that max-split $\geq 2 \times$ min-split so that the two new pages will have at least min-split entries each.

Invariant 5.5. All the live pages at level l that are involved in a structure-modification operation that targets a page p at level l must contain from min-split to max-split live entries immediately after the SMO.

Note that these requirements do not need to hold for the parent page q at level $l+1$, because only a small constant number of updates is applied to it during any SMO. The router entries in a parent page are only updated by the SMOs at a lower level, so the updates performed on the parent page q correspond to inserting or deleting entries from a leaf page.

The values chosen to the variables affect the size of the index structure and the frequency of structure-modification operations. It is theoretically possible to set min-live as high as $B/2$, if $s = 0$, but this means that thrashing is not prevented. The upper limit of the value of s is $B/3$, but with this setting min-live = 0 and thus the optimality constraints are lost. For the discussion in this dissertation, we assume that the following values are used: min-live = $1/5\,B$, $s = 1/5\,B$, min-split = min-live + s = $2/5\,B$, and max-split = $B - s$ = $4/5\,B$.

Although the representations of the variables differ from the definition used by Becker et al., we can show that the variables are the same as in the MVBT article [8]. Becker et al. require that min-live = $d = B/k$, min-split = $(1+\epsilon) \times d$ and max-split = $(k-\epsilon) \times d$, where k and ϵ are variables that can be selected. If we assign $s = \epsilon d$, we obtain min-split = $d + \epsilon d$ = min-live + ϵd = min-live + s, and max-split = $kd - \epsilon d = B - s$, which are the definitions used here.

For optimality of the index structure, we wish to keep the structure of the TMVBT index as close to the MVBT index as possible. Most importantly, we wish to maintain Invariant 4.9, so that all pages of each search tree S_v contain at least min-live entries that are alive at version v, for all versions v. Let us first restate Invariant 4.8 for the TMVBT:

Invariant 5.6. All inactive entries in the TMVBT pages remain in place. They are never moved to another page. Only the deletion time of an inactive entry may be changed, always to the current active version v_{active}. Active entries in the TMVBT pages may be physically deleted, updated, or moved to another page (see Invariants 5.3 and 5.4).

This means, in practice, that the structure of the search tree S_v of a version v can only change when $v = v_{active}$. After T commits, version v becomes inactive, and the structure of the search tree S_v becomes static. By this we mean that the set of pages that forms the search tree S_v can no longer change, and the entries that are alive at version v are never physically deleted or moved to another page. If we design the algorithms in such a way that the search tree of the active version is balanced (Definition 4.3) in all situations, this implies that search trees of all versions are balanced, and thus optimal. This follows from the fact that the search tree S_v of the active version is balanced immediately before the active transaction commits and thus also at the moment version v becomes inactive. Furthermore, because inactive search trees are static, S_v will always remain balanced. We will show in the next sections that the TMVBT algorithms maintain Invariant 4.9 for the TMVBT. This, together with the observation that all the root-to-leaf paths in the search tree S_v are always of the same length, implies that the balance conditions of the active-version search tree are also maintained.

Invariant 5.7. Invariant 4.9 holds for the TMVBT index. That is, for all versions v and all pages p, page p contains at least min-live entries that are alive at version v; or p is a root page of S_v, in which case it contains at least 2 entries that are alive at version v; or p is the only page of S_v and contains at least one entry that is alive at version v; or p is not part of the search tree S_v and therefore contains no entries that are alive at version v.

Figures 5.3–5.5 show an example of the TMVBT page operations. In this illustrative example, the index is structurally consistent and balanced, with suboptimal settings of min-live = 1 and $s = 1$ for a page capacity of $B = 5$. All the following examples have been generated by our visualization software TreeLib (see Chapter 7). Pages p_1, p_2, and p_4 are not shown in the figures, because p_1 and p_2 are used as database information pages, and page p_4 is the root page of the *root** index.

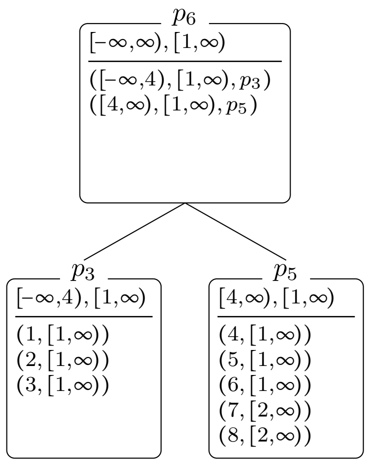


Figure 5.3. Example of a TMVBT index after insertions. The page header shows the page identifier followed by the key range and version of the page; the format of index-page entries is (key range, life span, page identifier); and the format of leaf-page entries is (key, life span, data), but the associated data has been left out for clarity. This TMVBT has been created by transaction T_1 inserting keys 1–6 and transaction T_2 inserting keys 7 and 8.

In Figure 5.3, the index contains six inactive live entries inserted by transaction T_1 (entries with keys 1–6), and two active entries inserted by transaction T_2 (entries with keys 7 and 8). During the execution of T_1, the leaf page p_3 was key-split into pages p_3 and p_5, and a new root page p_6 was created, thus incrementing the height of the search tree S_1 by one.

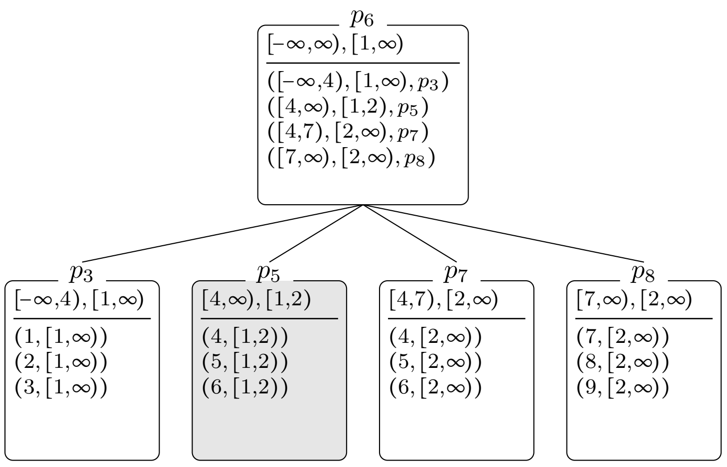


Figure 5.4. TMVBT after inserting a data item with key 9. The format of the figure is the same as in Figure 5.3. White rectangles denote live pages, and gray rectangles denote dead pages. Transaction T_2 has caused a version-split on p_5 by inserting key 9.

Figure 5.4 shows the result of a version split after transaction T_2 tried to insert key 9 to the full page p_5. The page p_5 was version-split into pages p_7 and p_8. The historical entries are left stored in the dead page p_5, and active copies of the entries have been created into pages p_7 and p_8. Note that all the active entries have been physically moved away from page p_5.

Figure 5.5 shows the status of the database after transaction T_2 has deleted entries 4–9. Deleting the active entries has caused the number of live entries in pages p_7 and p_8 to fall below min-live, so the pages have been consolidated by merging them. In more detail, first p_7 was merged with p_8 by moving the active entries of p_7 to p_8, which caused p_7 to be deallocated. Note that also the active router to p_7 was deleted from the parent page p_6. When the rest of the entries in p_8 were deleted, page p_8 was further merged with p_3 by killing the page p_3 and by creating active copies of the live entries in p_3 into p_8. As we will show in Section 5.5, the algorithms actually created a new live copy of p_3 when killing it (call it p_9), and the active live copy p_9 was then merged with p_8, causing p_9 to be deallocated. At this point p_8 was the only live page at level 1, so the height of the current-version search tree S_2 was decremented by making

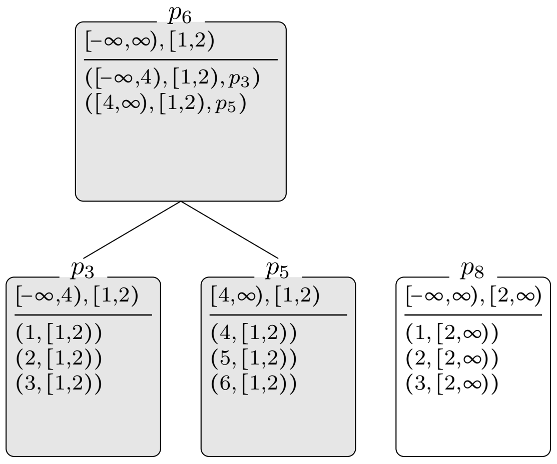


Figure 5.5. TMVBT after deleting most of the entries. Transaction T_2 deleted keys 4–9, thus shrinking the current-version search tree to a single page.

p_8 the root page of version 2. The auxiliary structure *root** now contains page identifiers of root pages p_6 (for version 1) and p_8 (for version 2).

A more diverse example of a TMVBT index is shown in Figures 5.6–5.8, with the same settings used as in the previous examples. This example has been generated by our visualization software with the action sequence given below:

- Transaction T_1: insert data items with keys 1–9 (Figure 5.6).
- Transaction T_2: delete data items with keys 7–9 (Figure 5.7); insert data items with keys 10–15 (Figure 5.8).

The transactions on this TMVBT index have induced the following structure-modification operations:

- The first six insertions by transaction T_1 have triggered a key-split, splitting page p_3 to p_3 and p_5. At this point, the root page p_6 was created to hold the routers to these pages, and *root** was updated by replacing the page identifier stored for version 1 from p_3 to p_6.
- The further three insertions by T_1 have triggered another key-split on p_5, creating the new leaf page p_7. The situation after these SMOs is depicted in Figure 5.6.
- After T_2 has deleted the entries with entries 7–9, a page-merge operation was triggered on p_7 to merge the page with p_5. Because

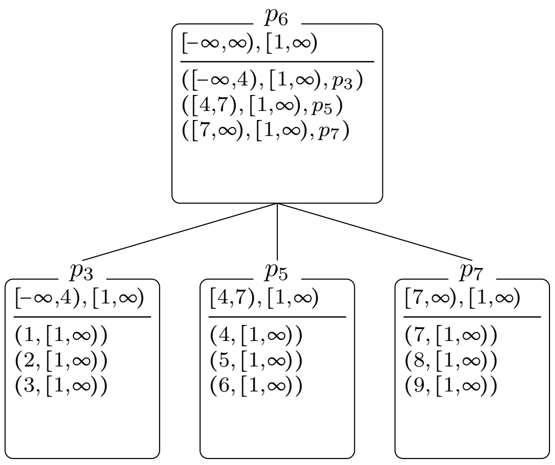


Figure 5.6. Example of a TMVBT index after insertions. In this figure, transaction T_1 has inserted keys 1–9.

both of these pages were inactive, they were first killed, creating two new active pages. These were then merged into the active leaf page p_8. The situation after this SMO is shown in Figure 5.7.

- The insertions by T_2 further induced two page splits; first on the active page p_8, creating the active page p_9; and then on p_9, thus creating page p_{11}.
- Insertion of the router to p_{11} to the parent page p_6 caused a split operation on the parent page p_6. Because p_6 was inactive, it was first version-split into p_{10}. At this point p_{10} had enough space to hold the router to p_{11}, so p_{10} was not further key-split into two pages. The page identifier p_{10} was inserted to the *root** to mark that the root page of version 2 differs from the root page of version 1. The situation after these SMOs is shown in Figure 5.8.

Figure 5.8 shows that the page p_3 containing entries with keys 1–3 is shared by both roots of the TMVBT index. Note that page p_3 is alive but not active, because $v_{active} = 2$ (assuming that transaction T_2 has not yet committed), and p_3 has a life span other than $[2, \infty)$. It is thus possible for this page to have more than one parent. The pages p_8 to p_{11} are active and only contain entries of the most recent version. Also note that the index page p_{10} is active even though it contains a router to the inactive page p_3, because the router itself is active.

In the previous chapter, we briefly discussed efficient version-range queries (i.e., *x/–/range* queries) on the MVBT index structure. These

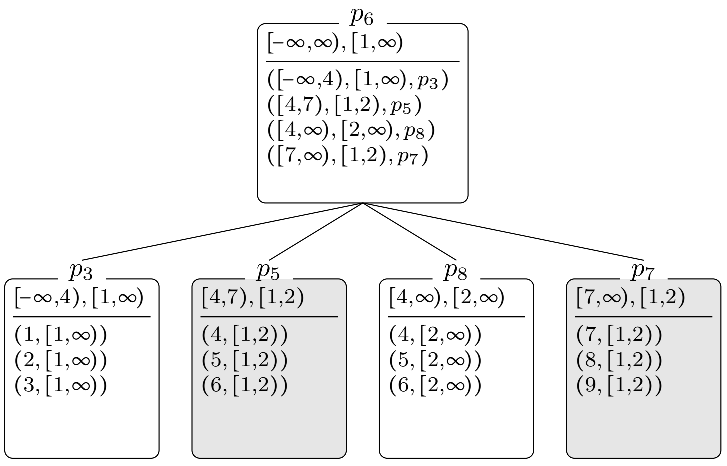


Figure 5.7. Example of a TMVBT index after deletions. In this figure, transaction T_2 has deleted keys 7–9.

were introduced by van den Bercken and Seeger [10]. Even though the TMVBT is based on the MVBT index, the most efficient Link_{Ref} technique cannot be used with the TMVBT index. This is because the technique relies on storing links to historical pages that temporally precede a page p. In the MVBT, each page can have at most two temporal predecessors (see the discussion in the end of Section 4.4), and the links to those pages can therefore be tracked. In the TMVBT, pages can have an unlimited number of temporal predecessors, because merging active pages combines the temporal predecessors of the merged pages.

5.4 User Actions

Having defined the transactions and the structure of the TMVBT index, we will now describe the implementation of the user actions in this section. As a general rule, we assume that the physical consistency of the database during normal processing is maintained by short-duration latching [66] of pages, so that the server process or thread that executes a transaction keeps a page p read-latched for the time a read action is performed on p, and write-latched for the time an update action is performed. We also assume that the buffer manager applies the standard steal-and-no-force buffering policy [32]. These assumptions are in accordance with the ARIES recovery algorithm [66]. No logical key-level locking is required for the TMVBT, because (1) for read-only transactions, the historical versions that the read-only transactions read are never deleted from the

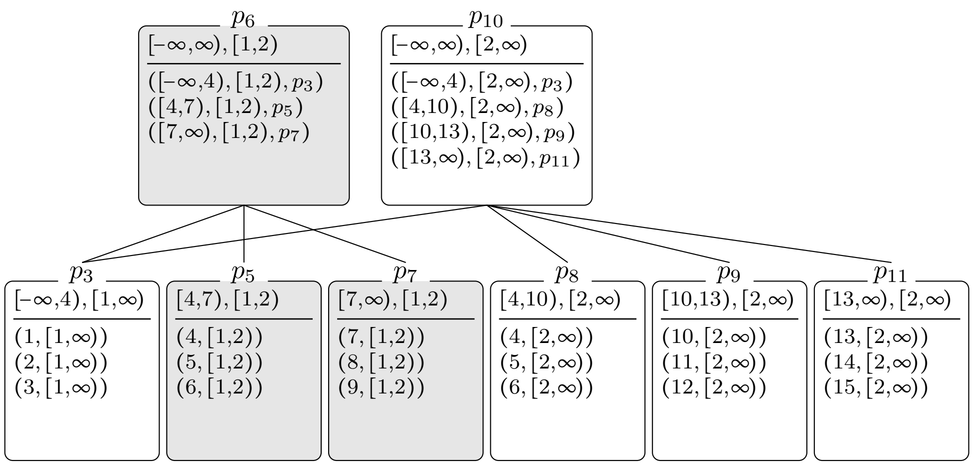


Figure 5.8. Example of a TMVBT index after more insertions. In this figure, transaction T_2 has inserted keys 10–15.

index; and (2) for updating transactions, there can be only one updating transaction operating on the index at a time.

The global version variables v_{commit} and v_{active} are maintained in the permanent database and their reading and writing is protected by locking. A **begin-read-only** action acquires a short-duration read lock on v_{commit} for reading its value, and a **commit-update** action acquires a commit-duration write lock on it for incrementing its value. A **begin-update** action acquires a commit-duration write lock on v_{active}, thus guaranteeing that at most one updating transaction is active at a time. The decrement of v_{active} in a **finish-rollback** action is performed under the protection of that lock. The **begin-read-only** and **commit-read-only** actions do not write any log records, because read-only transactions do not involve any logging.

In a fully dynamic index structure in which any inserted data can be physically deleted at any time, *latch-coupling* (called *crabbing* by Gray and Reuter [32]) is the standard way to guarantee the validity of traversed search paths in all circumstances. In a general situation, the validity of the traversed path can be ascertained by releasing the latch on the parent page only after a latch on a child page has been acquired. Latch-coupling is deadlock-free if the latches are acquired in a predefined order, such as first top-down, then left-to-right. However, in the case of the TMVBT index the fact that inactive data always remains in place (Invariant 5.6), together with our assumption that a read-only transaction only reads inactive data (Invariant 5.2), implies that the **query** and **range-query** actions of read-only transactions do not need to perform latch-coupling, and a parent page may be unlatched during tree traversal before acquiring a latch on the child page.

Accordingly, an action **query**(k) in a read-only transaction that is reading the version $\mathsf{snap}(T)$ can be implemented as follows. First, the root page for version $\mathsf{snap}(T)$ is located from *root** and read-latched. Then the TMVBT is traversed using read latches without latch-coupling until the leaf page p is found that covers key k and version $\mathsf{snap}(T)$; that is, $k \in \mathsf{kr}(p)$ and $\mathsf{snap}(T) \in \mathsf{vr}(p)$. At each index page p' on the traversed path, the next page on the path is the child page p'' of p' with $k \in \mathsf{kr}(p'')$ and $\mathsf{snap}(T) \in \mathsf{vr}(p'')$. Once the identifier of the child page p'' has been determined, the read latch on the parent page p' is released and the child page p'' is read-latched. When the correct leaf page p has been found, the proper entry $(k, [v_1, v_2), w)$ with $v_1 \leq \mathsf{snap}(T) < v_2$ is located, and page p is unlatched.

An action **range-query**$([k_1, k_2))$ is implemented similarly, except that for each index page p' in the search path we need to traverse all subtrees rooted at each child page p'' such that $[k_1, k_2) \cap \mathsf{kr}(p'') \neq \emptyset$ and $\mathsf{snap}(T) \in \mathsf{vr}(p'')$. If there are more than one such child page p'', then the page identifiers of all but the first child page are pushed into a stack, and the traversal proceeds to the subtree rooted at the first child. When a subtree has been searched, a page identifier (if any) is popped from the stack, the corresponding page is read-latched, and the search is continued at the subtree rooted at that page. Because the inactive entries and pages are static (Invariant 5.6), the pages do not need to be latched while the page identifiers are queued in the stack. Latching is used only to prevent inconsistent reads if the updating transaction needs to modify a page at the same time the read-only transaction is reading it.

The following theorem follows directly from the definitions of the query actions of the read-only transactions and from the fact that only one updating transaction can be active at a time:

Theorem 5.3. The TMVBT algorithms produce a snapshot-isolated schedule [11] for the transactions.

Proof. Firstly, because there can only be a single active updating transaction that operates on the TMVBT at a time, the updating transactions are processed in a fully serialized manner, thus fulfilling the requirements for snapshot-isolated transactions. Secondly, read-only transaction only read committed data that is never deleted, so they also form snapshot-isolated schedules. □

An updating transaction begins with the **begin-update** action and ends with the **commit-update** action, as described in Section 5.1, unless the transaction is aborted and rolled back. The **query** and **range-query** actions are the same as for read-only transactions, except that they now target the version v_{active}, and the actions may read active entries and pages. As with read-only transactions, these actions in an updating transaction do not write log records, because they do not create changes to the database that would have to be redone or undone during restart recovery.

For efficiency, we assume that the TMVBT index records the page identifier of the root page of version v_{active} separately so that the queries in updating transactions do not need to use the *root** structure to find it. Similarly, because read-only transactions reading the most recent committed version always target the version v_{commit}, the page identifier of the root page of that version is also maintained separately.

Theorem 5.4. When the root of the search tree of version v is known, the cost of a single-key query action in the TMVBT targeting version v is $\Theta(\log_B m_v)$ pages, and the cost of the key-range query action for version v is $\Theta(\log_B m_v + r/B)$ pages of the TMVBT structure, where m_v denotes the number of data items that are alive at version v, r is the number of entries returned by the range query and B is the page capacity.

Proof. Assuming that Invariant 5.7 holds, each page of the TMVBT that is part of the search tree S_v has at least min-live entries that are alive at version v. The proof is therefore the same as the proof of Theorem 4.2. We will show later on in Lemmas 5.5, 5.6, and 5.7 that all the SMOs maintain Invariant 5.7, thereby confirming this result. □

We assume that all TMVBT page traversals maintain a *saved path* [60, 61]; that is, an array *path* local to the server process or thread in question and indexed by the height of pages. An entry *path*$[i]$ holds the page identifier, key range, life span, and Page-LSN of the page that was located at level i when traversing the root-to-leaf path. The saved-path concept can be used to accelerate the user actions by starting the traversal at the lowest-level page in the saved path that, according to the saved information, covers the queried search space. This page is known to be the correct page to start the tree traversal, because (1) for read-only transactions, the inactive data is never moved away from the pages; and (2) for updating transactions, there can be no other updating transaction that would invalidate the data in the saved path of the current updating transaction. This holds regardless of whether a concurrent purging process is allowed, because the purging process only deletes pages that are part of historical versions that are no longer queried.

For the **write**(k, w) and **delete**(k) actions of the updating transaction, the TMVBT is traversed using read latches without latch-coupling as for the **query**(k) action of the updating transaction, except that the target leaf page p is write-latched. If the target leaf page p can accommodate the update, then the update is applied on page p directly; otherwise a structure-modification operation is performed before the action can proceed. After the update has been applied, a redo-undo log record for the action is generated, its LSN is stamped in the Page-LSN field of p, and the write latch on p is released.

In the **write**(k, w) action, if the index contains a live entry of the form $(k, [v, \infty), w')$, then that entry is logically deleted by either replacing it with a new entry $(k, [v, v_{active}), w')$, if $v \neq v_{active}$; or by physically removing the old entry, if $v = v_{active}$. After the existing entry has been deleted, a new entry $(k, [v_{active}, \infty), w)$ is inserted into the page p. The page p can accommodate this update action, if the operations explained above can be carried out without the page overflowing. The redo-undo log entry written for this action contains the version and data of the replaced entry, in addition to the version and data of the inserted entry. The log entry written by an updating transaction T is thus $\langle T$, **write**, p, k, v_{active}, w, v, w', $n\rangle$, where n is the log sequence number of the previous not-yet-undone action of T, and v and w' are null if the index contained no live entry with the key k.

In the case of the **delete**(k) action, page p can accommodate the update if replacing the entry $(k, [v, \infty), w)$ by $(k, [v, v_{active}), w)$ (in the case $v \neq v_{active}$), or physically removing the entry $(k, [v_{active}, \infty), w)$ (otherwise) does not decrease the number of live entries in the page below the required minimum number of live entries, min-live. An updating transaction T writes a redo-undo log record $\langle T$, **delete**, p, k, v_{active}, v, w, $n\rangle$ for this action.

When the target leaf page p cannot accommodate the update, structure modifications are needed. These operations are explained in Section 5.5. For writes, the operation *split-page* is called before the write action can proceed. For deletes, the page p needs to be consolidated by the operation *merge-page* before the entry can be deleted from the page. When the structure-modification operations are initiated, the page p, recorded on the saved path, is left write-latched. After the operations, the saved path contains the correct write-latched leaf page p' whose key range covers the key k. As with the earlier situation, the update is now performed on page p', a redo-undo log record is generated, the LSN is stamped on p' and page p' is unlatched.

As will be explained in the next section, the structure modifications (page splits or merges) are applied in a top-down, level-by-level manner, logging the structure modification done at each level using a single redo-only log record. Each of these structure modifications involves a maximum of five pages at two adjacent levels. The sequence of structure modifications results in a target leaf page that can accommodate the **insert** or **delete** action in question.

An undo action, **undo-write**(r) or **undo-delete**(r), is performed as a physical undo if possible and as a logical undo otherwise [66]. For a physical undo, the page mentioned in the log record r is write-latched and the Page-LSN field is examined. If the Page-LSN field still contains the LSN of r, or if the page contents show that the page is the correct target for the undo action and the page can accommodate the undo action, then the undo action is performed on the page, a redo-only log record is generated, its LSN is stamped in the Page-LSN field of the page, and the page is unlatched. If the page mentioned in the log record r cannot be seen to be the correct target for the undo action or if the page cannot accommodate the undo action, a *logical undo* is performed, starting with a search for the key mentioned in r and performing any structure modifications that may be necessary to make the target page accommodate the undo action. The undo actions write redo-only compensation log records, as dictated by the ARIES algorithm. If the page cannot accommodate the undo action, the page is first split or merged after write-latching it, as with the forward-rolling write and delete actions.

The **undo-write** action reads the log record $\langle T$, **write**, p, k, v_{active}, w, v, w', $n\rangle$ created by the **write** action that is to be undone. Let page p' be the current correct leaf page that covers key k at version v_{active}. As discussed above, p' is either p or it has been located with a root-to-leaf traversal of the TMVBT index structure. This action locates the active entry $(k, [v_{active}, \infty), w)$ from page p' and physically removes it. If v and w' are not null, then this action furthermore restores the previous entry. If $v < v_{active}$, the currently dead inactive entry $(k, [v, v_{active}), w')$ is located (if it is still present on p) and restored to life by replacing it with $(k, [v, \infty), w')$. If $v = v_{active}$, or if p' has been version-split so that the historical entry $(k, [v, v_{active}), w')$ is not present on p', then the active entry $(k, [v_{active}, \infty), w')$ is inserted to page p' to undo the write action. The leaf page p' can accommodate the undo action if the operations described can be carried out without the page overflowing, and without the number of live entries on the page decreasing below min-live. The action finishes by writing a redo-only compensation log record $\langle T$, **undo-write**, p', k, v_{active}, v, w', $n\rangle$.

The **undo-delete** action examines the log record $\langle T$, **delete**, p, k, v_{active}, v, w, $n\rangle$ created by the delete action, and locates the correct leaf page p'. If $v < v_{active}$, then this action locates the logically deleted entry

$(k, [v, v_{active}), w)$, and replaces it with $(k, [v, \infty), w)$. If $v = v_{active}$, or if the logically deleted entry $(k, [v, v_{active}), w)$ is no longer present on p', then this action inserts a new active entry $(k, [v_{active}, \infty), w)$ into page p'. The leaf page p' can accommodate the undo action if the operation described above can be performed on p' without it overflowing. The action finishes by writing a redo-only compensation log record $\langle T$, **undo-delete**, p', k, v_{active}, v, w, $n\rangle$.

5.5 Structure-Modification Operations

In this section, we will describe the structure-modification operations (SMOs) used with the TMVBT index. The two main operations described here, *split-page* and *merge-page*, are triggered by the user actions described in the previous section. For convenience, we will use the same notation for entries in both index and leaf pages. The format of an index-page entry is $(\mathsf{kr}(p), \mathsf{vr}(p), p)$; that is $([k_1, k_2), [v_1, v_2), p)$, where $\mathsf{kr}(p) = [k_1, k_2)$ is the key range and $\mathsf{vr}(p) = [v_1, v_2)$ is the life span of the page p. A leaf-page entry $(k, \vec{v}, w)$, for key k and data-item value w, that is alive at versions in the range $\vec{v} = [v_1, v_2)$, is represented by the tuple $([k, k^+), [v_1, v_2), w)$, where k^+ is the key immediately following key k in the key space (for databases that store integer keys, $k^+ = k + 1$). In this way we can use the same algorithms for manipulating both the leaf pages and the index pages.

The convention in the TMVBT structure-modification operations is that each SMO transforms a balanced (Definition 4.3) TMVBT index into another balanced TMVBT. Each operation is logged with a single redo-only log record, so that structure modifications are never undone when a transaction aborts or system fails [40, 41]. The structure-modification operations are performed top-down, starting from the highest page on the search path that requires splitting or merging.

The actual implementation of the operations traverses the search path bottom-up in order to determine which kind of a structure modification is needed at each level, yet without performing any modification. When a parent page which does not need any modification is encountered, the search path is traversed top-down, and the structure modifications are performed level-by-level, logging each operation with a single redo-only log record. The search path state is guaranteed to remain valid throughout the operations because only one updating transaction can be active at a time. The information about the traversed search path stored in the saved path can thus be trusted. For clarity, the algorithms presented in this section only describe the structure-modification operations applied at a single level.

Both of the SMOs *split-page* and *merge-page* are based on the page-killing operation *kill-page*. Page killing is not a separate SMO, but it is used in both of the actual SMOs. This operation creates a live copy of a page p, and marks the original page p as killed. The operation is described in Algorithm 5.1. The page p and its parent q (located from the saved path) are both write-latched before the operation is performed. As discussed above, we assume that all required SMOs have been applied to the parent page q, so that q can accommodate the entry inserted by the algorithm.

```
KILL-PAGE($p, q$):
1 $p' \leftarrow$ allocate, write-latch, and format a new page
2 move all active live entries of $p$ to $p'$
3 create active copies of all inactive live entries of $p$ to $p'$
4 kill all live entries of $p$
5 $r \leftarrow$ find the router $([k_1, k_2), [v, \infty), p)$ to $p$ from $q$
6 replace the retrieved router $r$ in $q$ with $([k_1, k_2), [v, v_{active}), p)$
7 insert a new router $([k_1, k_2), [v_{active}, \infty), p')$ to $q$
8 **return** $p'$
```

Algorithm 5.1. Page-killing algorithm. The algorithm assumes an inactive page p and its parent page q, kills the page p, creates an active live copy p' of p and updates the routers in the parent page q.

The page-killing operation begins by allocating a new page p', write-latching it and formatting it as a TMVBT page. All the live entries of page p are now either copied or moved to page p' in such a way that all active entries are physically moved to p', and all inactive live entries are copied to p'. The life spans of the copied entries are split at version v_{active}, so that a life span $[v, \infty)$ is changed to $[v, v_{active})$ in the entry that remains in page p and to $[v_{active}, \infty)$ in the new entry created to page p'. The logical state of the database at all versions prior to v_{active} is thus maintained in page p, but the page p is no longer part of the search trees of versions $v \geq v_{active}$. The router to page p in the parent page q must be updated by setting its end version to v_{active}, and the router to the new page must also be inserted into the parent. After this, the old page p is replaced with the active page p' in the saved path. All pages modified by this operation are kept write-latched, because the acquired page latches can be released only after the associated log record has been written.

Lemma 5.5. The *kill-page* operation maintains Invariant 5.7.

Proof. Because the TMVBT index is balanced at the point when the SMO that uses the page-killing operation is triggered, page p must contain at least min-live entries that are alive at version v', for each version $v' \in [v, v_{active}]$, where v is the starting point of the life span of p. There are thus at least min-live entries in page p that are alive at version v_{active}. These entries are either copied or moved to the new page p', which thus has at least min-live live entries after the page-killing operation. Because the life spans of the copied entries were cropped to start from version v_{active}, page p' contains no entries that are alive at any earlier version $v' < v_{active}$. All of the entries that are alive at version v_{active} in page p are deleted from p—either physically, if they were active entries; or logically, if they were inactive entries. After the page-killing operation, page p thus does not have any entries that are alive at version v_{active}. The number of entries in page p that were alive at an earlier version $v' < v_{active}$ remains unchanged. Clearly, the invariant holds for pages p and p'. Furthermore, a single router that is alive at version v_{active} is logically deleted from the parent page q, and a new active router is inserted to the parent to replace it. The number of live entries in the parent page is therefore not affected for any version. Because all the pages retain a valid number of live entries for all versions, we conclude that the page-killing operation maintains Invariant 5.7. It is clear that Invariant 5.6 is also maintained. □

The *split-page* operation is a structure-modification operation that splits a page that has become full. The operation is similar to the MVBT version-split operation, with the exception that active pages are directly key-split without first killing the page. This operation is triggered by a **write** action when a data page has become full, by both **undo-write** and **undo-delete** actions if the leaf page cannot accommodate the undo action, and the operation is also used to split index pages along the search path. At the beginning, the page p to be split is retrieved from the saved path along with its parent page q, and both pages are write-latched for modification; unless the operation was triggered by a user action, in which case page p is already write-latched. As explained in the beginning of this section, we expect that parent pages in the saved path have already been split so that the parent page q can accommodate the routers to the new pages created by the split operation.

An overview of the actual *split-page* operation is simple: if page p is active, it will be key-split, and if it is inactive, it will be version-split. Key-splitting and version-splitting will be defined in more detail below. An overview of the page-split operation is shown in Algorithm 5.2. In the

```
SPLIT-PAGE($p, q$):
if $p$ is active then // Figure 5.9, key split
   $p'$ ← allocate, write-latch, and format a new page
   distribute entries of $p$ between $p$ and $p'$
else // Figures 5.10(a)–5.10(f), version split
   $a_p$ ← count the number of live entries in $p$
   $p'$ ← KILL-PAGE($p$)
   if $a_p$ > max-split then // Figure 5.10(b)
      $p''$ ← allocate, write-latch, and format a new page
      distribute entries of $p'$ between $p'$ and $p''$
   else if $a_p$ < min-split then // Figures 5.10(c)–5.10(f)
      $s$ ← find a live sibling page of $p'$ from $q$ and write-latch it
      $a_s$ ← count the number of live entries in $s$
      $p''$ ← $s$ if $s$ is active, KILL-PAGE($s$) otherwise
      if $a_p + a_s$ > max-split then // Figures 5.10(d),5.10(f)
         redistribute entries of $p'$ and $p''$
      else // Figures 5.10(c),5.10(e)
         move all entries of $p''$ to $p'$
         deallocate $p''$
      end if
   else // Figure 5.10(a)
      // No further action required
   end if
end if
```

Algorithm 5.2. The page-splitting algorithm. Splits a page p, possibly redistributing the live entries with a sibling page s. Updates the routers in the parent page q.

algorithms presented in this section, we denote by a_p the number of live entries in page p. Note that the bottom-up checking phase needs to do the same checks that are described in Algorithm 5.2 to determine which kind of a split needs to be done. When performing the actual structure modification, the same checks must be performed again (as described here), or results saved during the checking phase can be used.

The first part of the split algorithm, the *key-split* operation, is similar to the page-split operation in a standard B$^+$-tree. This operation begins by allocating, write-latching, and formatting a new page p'. After that, the entries of the old page p are distributed between p and p' by moving half of the entries from p to p'. Note that all the entries of page p are alive and active at the beginning of the operation because p is active, and thus the entries may be physically distributed between the pages. The operation finishes by adjusting the the router to page p in the parent page q, and by inserting a router to the new page p' into q. The key-split operation is illustrated in Figure 5.9.

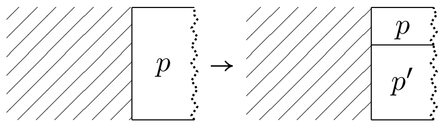


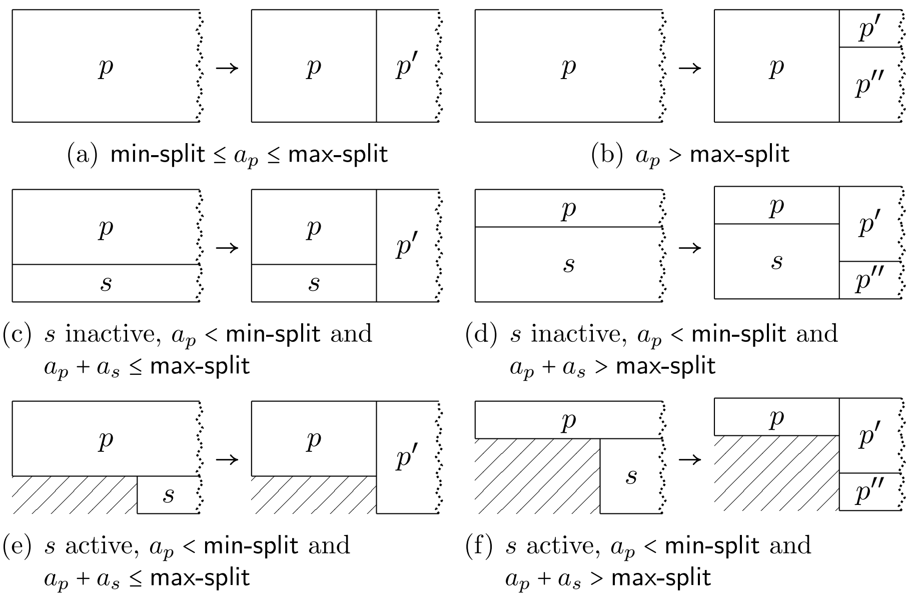
Figure 5.9. Key-splitting an active page p. The horizontal axis represents life spans, and the vertical axis key ranges.

An exception to the key-split algorithm is the situation when page p has no parent page, and the saved path only contains the page p. This happens when page p is a root page. In this situation a new parent page r is allocated, write-latched and formatted; and routers to pages p and p' are inserted to it. The new root is attached to the TMVBT by inserting the tuple (v_{active}, r) into *root**, and by updating the cached root page identifier for version v_{active}. This will replace the existing tuple (v_{active}, p) in the *root**, which pointed to the previous root page p. The rest of the key-split operation is otherwise similar to the normal situation. Note that this operation increases the height of the search tree $S_{v_{active}}$ by one.

The *version-split* operation begins by killing the inactive page p with the page-killing operation defined earlier. The new active copy of page p is denoted by p'. At this point, page p' may contain too few or too many entries to satisfy Invariant 5.5. If the number of entries in p' (i.e., a_p, the number of live entries in p before the page-killing operation) is less than min-split, the page will be merged with a sibling page by consolidating it in the same way as pages are merged in the *merge-page* operation that will be shortly described. If the number of entries in p' is more than max-split, page p' will be key-split in the same way as active pages are split.

As with the key-split algorithm, it is possible that the inactive page p has no parent, if it is the root page of the current-version search tree. In this case, the page-killing operation cannot update the parent page, but instead a new tuple (v_{active}, p') is inserted to *root** to indicate that the new active page p' is the current root page. If page p' contains more than max-split entries after the page-killing operation, it is further key-split, a new parent page r is created, and the tuple (v_{active}, p') in *root** is replaced by (v_{active}, r). The cached root page identifier must also be updated.

The entire split operation, consisting of either a key split or a version split (possibly followed by a key split or consolidation), and including the possible tree-height increase or root-page update, is logged with a single redo-only log record containing the page identifiers of all the pages involved in the SMO; that is, a subset of the pages p, p', p'', s, q, and r. The redo-only log record must also contain information of all the entries moved or copied between pages, as explained in Section 2.8. For example, the log record of a version split followed by a key split (shown in Figure 5.10(b)) is $\langle T$, **version-split-key-split**, p, p', p'', q, $E_{p'}$, $E_{p''}\rangle$, where E_x denotes the set of entries that were written on page x. This operation can be redone on p by deleting all the live entries from p; and on pages p' and p'' by clearing the page and inserting the entries from the corresponding set $E_{p'}$ and $E_{p''}$. As usual, all the pages are kept latched until the log record has been generated and its LSN stamped in the Page-LSN fields of the pages. The split operation finishes by replacing p in the saved path with the active page whose key range covers k (either p' or p''), and by unlatching all the other pages involved in the operation.



p
p
p'
(a) min-split $\le a_p \le$ max-split
p
p
p'
p''
(b) $a_p >$ max-split
p
s
p
s
p'
(c) s inactive, $a_p <$ min-split and $a_p + a_s \le$ max-split
p
s
p
s
p'
p''
(d) s inactive, $a_p <$ min-split and $a_p + a_s >$ max-split
p
s
p
p'
(e) s active, $a_p <$ min-split and $a_p + a_s \le$ max-split
p
s
p
p'
p''
(f) s active, $a_p <$ min-split and $a_p + a_s >$ max-split

Figure 5.10. Version-splitting an inactive page p. The horizontal axis represents life spans, and the vertical axis key ranges. Case (a) represents a version split, (b) a version split followed by a key split, (c) a version split followed by a merge with an inactive sibling, (d) a version split followed by a redistribution of live entries with an inactive sibling, (e) a version split followed by a merge with an active sibling, and (f) a version split followed by a redistribution of live entries with an active sibling.

All the possible different page-split scenarios for inactive pages are shown in Figure 5.10. In the figure, the horizontal axis represents life spans, and the vertical axis represents key ranges. In the presented scenarios, page p is split. Page s is the sibling page that is located from the parent of p found in the saved path. Pages p' and p'' are new pages allocated by the operation. As can be seen from the figures, all the scenarios preserve the initial key-version extents of p and s. That is, the new pages cover exactly the same region in key-version space as the old pages did. The version-split operation therefore preserves the combined spatial extents of the pages that are involved in the split operation, and thus can neither cause pages to overlap nor create gaps in the key-version space.

A more detailed example of a version split followed by a merge with an inactive sibling (as depicted in Figure 5.10(c)) is shown in Figure 5.11. This example shows how the live entries of a leaf page p are merged with the live entries of a sibling page s into a newly allocated page p'. As seen from the figure, the entries labeled 1 and 2 are killed, and active copies of them are created into the new page p'. The third entry, entry 3, is already active, and it is therefore physically moved from page s into the new page p', thus reducing the number of entries stored in s by one. After the split operation, pages p and s are dead and only contain dead entries.

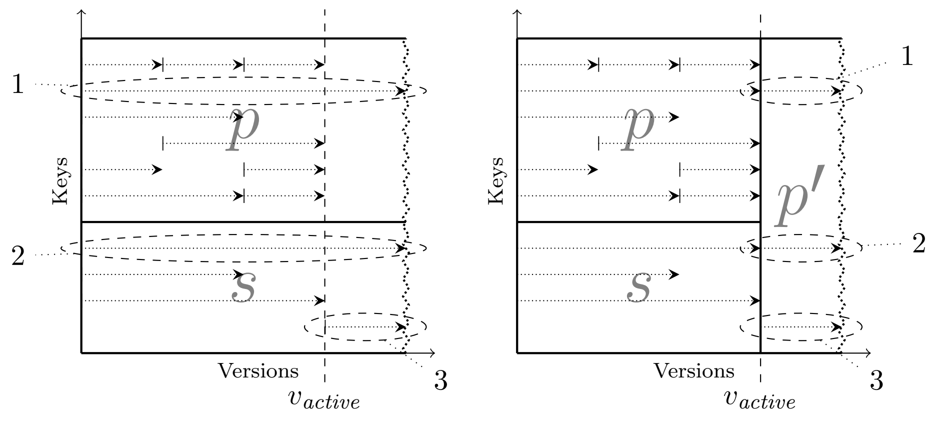


Figure 5.11. Example of a page split. This figure shows a version split followed by a merge with an inactive sibling. Entries labeled 1 and 2 are killed and new copies of them are created in page p', while the entry 3 is physically moved from page s into page p'.

Lemma 5.6. The *split-page* operation maintains Invariant 5.7.

Proof. First of all, if the page p that is to be split is an active page, then the split operation is triggered because the page became full and thus contains $a_p = B$ live entries. In this case, the entries will be split between the page p and a newly allocated page p', resulting in $B/2$ entries in each page, satisfying min-split $\leq B/2 \leq$ max-split. Secondly, if the page p is inactive, it is first killed, and the resulting page p' contains a_p entries, where min-live $\leq a_p \leq B$. This is because the *split-page* operation is only invoked when the page p has become full, and by Invariant 5.7 it must contain at least min-live live entries. If $a_p <$ min-split, the page will be merged with another page to avoid thrashing. Merging pages maintains the invariant, which is proven in the proof of Lemma 5.7 for the *merge-page* operation. If $a_p >$ max-split, the page will be key-split into two pages, both of which will contain more than min-split entries. □

```
MERGE-PAGE($p, q$):
1  $a_p \leftarrow$ count the number of live entries in $p$
2  $p' \leftarrow p$ if $p$ is active, KILL-PAGE($p$) otherwise
3  $s \leftarrow$ find a live sibling page of $p$ from $q$ and write-latch it
4  $a_s \leftarrow$ count the number of live entries in $s$
5  $p'' \leftarrow s$ if $s$ is active, KILL-PAGE($s$) otherwise
6  if $a_p + a_s \leq$ max-split then // Figures 5.12(a),5.12(c),5.13(a),5.13(c)
7      move all entries of $p''$ to $p'$
8      deallocate $p''$
9  else // Figures 5.12(b),5.12(d),5.13(b),5.13(d)
10     redistribute entries between $p'$ and $p''$
11 end if
```

Algorithm 5.3. The page-merging algorithm. Merges a page p with a sibling page s located from the parent page q, killing the pages if they are inactive. Updates the routers in the parent page q accordingly.

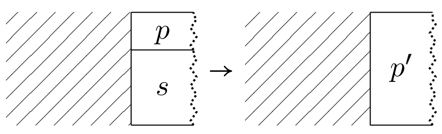
The *merge-page* operation is a structure-modification operation that merges a page with a sibling page when the number of live entries in the page is about to decrease below min-live. This operation is triggered by the **delete** or **undo-write** action, if an entry deletion is about to decrease the number of live entries below min-live; and by SMOs at a lower level. An overview of the operation is shown in Algorithm 5.3. The operation begins by retrieving the page p to be merged from the saved path. The page p is assumed to be write-latched at this point. The parent page q also needs to be modified, so it is retrieved from the path, and write-latched for modification. As with the *split-page* operation, it is assumed that the parent page q can accommodate the insertions or deletions possibly triggered by this operation (insertion of up to two new routers; or deletion of a single router).

The merge operation continues by finding a live adjacent sibling page s from the parent page q, and by write-latching it. Such a page is guaranteed to be found, because by Invariant 5.7, the parent page must contain at least two live entries, and live entries are adjacent to each other. If either page p or the sibling page s is inactive, it is killed with the page-killing operation defined earlier; otherwise the merge operation will be performed on it directly. We denote the active pages by p' (from page p) and p'' (from page s). The operation now merges the active pages p' and p''.

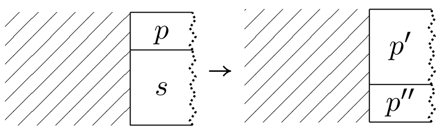
The merging is similar to the standard B$^+$-tree merge operation. If the number of combined entries of pages p' and p'' (equivalently, number of live entries in pages p and s before the page-killing operations) is larger than max-split, the entries will be redistributed between the two pages. If the number of entries is less than or equal to max-split, the entries will be moved to page p', and page p'' will be deallocated by removing the router to it from the parent and by deallocating the page from the corresponding space-map page. The router to page p' (and to page p'' in the former case) in the parent q must be updated to match the new key ranges. Updating the routers in the parent page q is sufficient, because the pages are active, and therefore have only one parent by Lemma 5.2.

An exception to the normal operation of *merge-page* is when the page p is a root page. First of all, by Invariant 5.7, a root page of version v must contain at least two entries that are alive at version v, unless the root page is the only page in the search tree S_v. If p is the only page in S_v, then p is left as it is until a delete operation deletes the last live entry from p. At this point, it is possible to insert a null marker $(v_{active}, \varnothing)$ to $root^*$ to signify that the current version search tree is empty. However, for optimizing the space usage of the index, it is also possible to allow the searches to locate the root page p. Because p contains no live entries, the queries will immediately notice this and stop traversing the search tree. If the null marker is inserted to $root^*$ and p is active, then the insertion of $(v_{active}, \varnothing)$ replaces an existing tuple (v_{active}, p), and p can be deallocated.

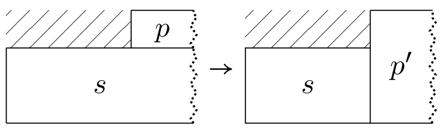
If p is a root page and the height of p is greater than one, then the page-merge operation is triggered by an SMO at a lower level, and we must check whether the operation would cause the number of live entries in p to decrease to one. If this is the case, then the height of the search tree S_v will be decremented by making the only remaining live child page of p the new root of S_v. This cannot be done before the SMO at the lower



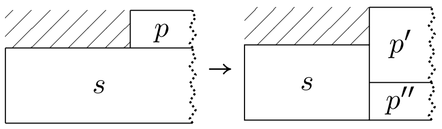
(a) p active, s active, $a_p + a_s \leq$ max-split



(b) p active, s active, $a_p + a_s >$ max-split



(c) p active, s inactive, $a_p + a_s \leq$ max-split



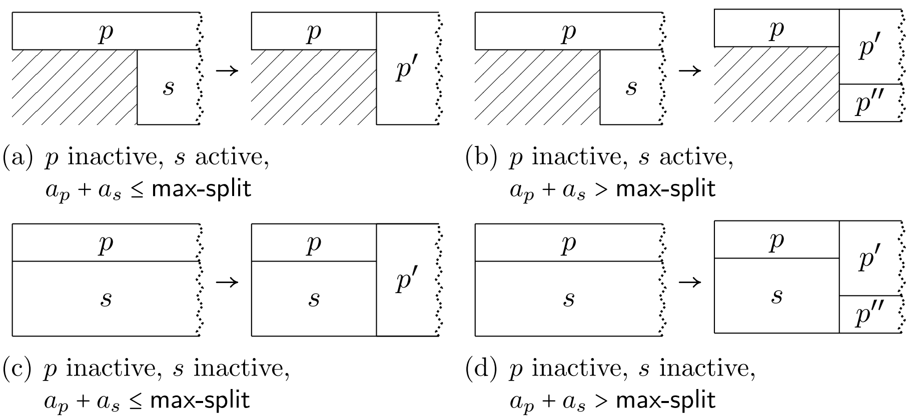
(d) p active, s inactive, $a_p + a_s >$ max-split

Figure 5.12. Merging an active page p. The horizontal axis represents life spans, and the vertical axis key ranges. Case (a) represents a merge, (b) a redistribution of live entries, (c) a merge with an inactive sibling, (d) a redistribution of live entries with an inactive sibling.

lever, however, so the tree-height-decrease operation must be performed as a part of the SMO at the lower level. Therefore, after the SMO at the lower level, if the number of entries in the root page p has decreased to one, the only remaining child page r of p is located from p, and a tuple (v_{active}, r) is inserted to *root**. If p is active, this will replace the existing tuple (v_{active}, p), and page p can be deallocated. The cached root page identifier of $S_{v_{active}}$ must also be updated.

The entire *merge-page* operation, including the possible root-page-update operation, is logged with a single redo-only log record containing the page identifiers of all related pages—this means pages p, s, p', p'', q, and r. The log record must contain sufficient information of all the moved entries, like the log record of the split-page operation. For example, the log record for a *merge-page* operation that merges an active page with an inactive sibling, resulting in a single active page (Figure 5.12(c)), is $\langle T$, **merge-active-inactive**, p, s, q, E_p, $E_s\rangle$, where E_x denotes the set of entries present in page x after the operation. Note that in this situation, the active page p is reused as p', and thus p' is not present in the log record. After the log record has been written, the saved path is returned to a proper state, so that the active page whose key range covers k (either p' or p'') is placed in the saved path to replace the merged page. The operation finishes by releasing the write latches on the pages.

The possible page-merge scenarios for page p are shown in Figures 5.12 and 5.13. In the figures, page p is merged with a live sibling page s.



(a) p inactive, s active, $a_p + a_s \le \textsf{max-split}$

(b) p inactive, s active, $a_p + a_s > \textsf{max-split}$

(c) p inactive, s inactive, $a_p + a_s \le \textsf{max-split}$

(d) p inactive, s inactive, $a_p + a_s > \textsf{max-split}$

Figure 5.13. Merging an inactive page p. The horizontal axis represents life spans, and the vertical axis key ranges. Case (a) a represents a version split followed by a merge, (b) a version split followed by a redistribution of live entries, (c) a version split followed by a merge with an inactive sibling, and (d) a version split followed by a redistribution of live entries with an inactive sibling.

For consistency with Figures 5.9 and 5.10, the resulting active pages are denoted p' and p'', even though $p' = p$ when the page p is active, and $p'' = s$ when s is active. For clarity of presentation, the algorithms now create a new page p' in the situation depicted in Figure 5.13(a), and deallocate the existing active sibling page s, instead of reusing s as the resulting merged page. An actual implementation may of course correct this and reuse page s instead of creating p'.

Lemma 5.7. The *merge-page* operation maintains Invariant 5.7.

Proof. The page p that is to be merged must have exactly $\textsf{min-live}$ live entries, while the sibling page s can have between $\textsf{min-live}$ and B live entries. The combined number of live entries $e : 2 \times \textsf{min-live} \le e \le \textsf{min-live} + B$ can always be distributed among the two pages so that the number of entries in both pages is between $\textsf{min-split}$ and $\textsf{max-split}$. In the minimum case, $2 \times \textsf{min-live} \ge \textsf{min-split}$ entries are moved to page p', and page p'' is deleted. Note also that $2 \times \textsf{min-live} < \textsf{max-split}$, and Invariant 5.5 thus holds for page p'. In the maximum case, $\textsf{min-live} + B$ entries are distributed between p' and p'', resulting in more than $\textsf{min-split}$ and less than $\textsf{max-split}$ entries per page. The former holds trivially (as we are distributing more than $\textsf{max-split} \ge 2 \times \textsf{min-split}$ entries between two pages), and the latter holds because $\textsf{min-live} + B = (\textsf{min-split} - s) + (\textsf{max-split} + s) = \textsf{min-split} + \textsf{max-split} > 2 \times \textsf{min-split}$. □

Lemma 5.8. Any structure-modification operation needed in the implementation of the **insert**, **delete**, **undo-insert** and **undo-delete** actions keeps at most five TMVBT pages latched simultaneously and transforms a balanced (Definition 4.3) TMVBT index into another balanced one. For any of the actions, at most $h + 1$ structure modifications are needed, where h is the height of the search tree $S_{v_{active}}$ of the active database version v_{active}.

Proof. As seen from the Figures 5.9–5.13, when an SMO is triggered on page p at level l, at most four pages at level l are involved in the operation. In addition to these four, the parent page q is also latched during the SMO. If a root-page update occurs (either a tree-height-increase or a tree-height-decrease), then the root page identifier is also present in the log records. However, the root page does not increment the number of pages latched at a time, because (1) during a tree-height-increase, there is no parent page q, so the maximum number of latches required is still five; and (2) during a tree-height-decrease, the new root r is the page p', because just before the root-page update, p' is the only remaining child page of q.

When considering the balance condition, Lemmas 5.5, 5.6 and 5.7 show that each SMO maintains Invariant 5.7, which proves that if the TMVBT is balanced before the SMO, then it is also balanced immediately after the SMO. □

The following theorem states that the update actions also maintain the asymptotic bounds of the MVBT:

Theorem 5.9. Assuming that the root page of the database version v is known, each of the actions **insert**, **delete**, **undo-insert**, and **undo-delete** is performed in time $\Theta(\log_B m_{v_{active}})$, where $m_{v_{active}}$ is the number of data items in the active database version v_{active}.

Proof. Searching for the correct leaf page to perform the update has a cost of $\Theta(\log_B m_{v_{active}})$ pages, because the search tree S_v is balanced. By Lemma 5.8, at most $h + 1$ structure-modification operations are needed to perform the update, where $h = \Theta(\log_B m_{v_{active}})$. At each level, at most five pages need to be accessed to perform the structure-modification operations. The structure-modification operations thus have a maximum cost of $\Theta(\log_B m_{v_{active}})$, which is the same as the initial tree traversal. □

As discussed in Section 4.4, the MVBT index has a space complexity of $\mathcal{O}(n/B)$ database pages, where n is the number of updates performed during the history of the database, and B is the page capacity. This has been proven by Becker et al. [7, 8]. Their proof relies on the fact that, for leaf pages, (1) each structure-modification operation that targets a page p creates at most a constant number of new pages (at most two); and (2) at least a constant minimum number of update operations (ϵd in the discussion of Becker et al. [8], s for the TMVBT) must have been performed on a page p for it to require a structure-modification operation. Therefore, the amortized cost of the update actions is at most $\mathcal{O}(2B/s) = \mathcal{O}(1)$ new entries, if s is linearly dependant on the page capacity B. This directly translates to the reported $\mathcal{O}(n/B)$ leaf pages for a history of n update actions, because each leaf page holds at least min-live = $\mathcal{O}(B)$ entries. Similarly, for the index pages of the MVBT, each SMO requires space for at most two new index entries at the first index page level (at height two, directly above the leaf pages). Becker et al. [8] show that these insertions at the index level do not increase the amortized space complexity of the update actions, if $d \geq {}^2\!/\epsilon$. This translates to $s \geq 2$, which is a reasonable requirement for the split tolerance variable in any case.

When considering the space complexity of the TMVBT, we note that the premises of the MVBT space complexity proof remain true: each SMO creates at most a constant number of new pages (at most two), at most a constant number of new entries (again, two) are inserted into the parent page, and at least a minimum number of update actions must target page p before any SMO is required on it (s actions). We can thus conclude that the space complexity of the TMVBT is $\mathcal{O}(n/B)$ pages, if s is linearly dependent on the page capacity B, and $s \geq 2$.

In the discussion of space complexity for any of the efficient multiversion index structures, the complexity has been given in terms of the total number of update actions performed on the index structure. However, a transaction may insert many entries which it then later on deletes; the index structure should not consume space for the actions that are undone. For the TMVBT we can state the space complexity in terms of the number of entries that are alive after transactions have committed; in these terms, the space complexity is $\mathcal{O}((\sum_{v \in V} m_v)/B)$ database pages, where V is the set of all versions, and m_v is the number of entries that are alive at version v.

Theorem 5.10. The space complexity of the TMVBT index is

$$\mathcal{O}(\min\{n, \sum_{v \in V} m_v\}/B).$$

Proof. The proof for the first part of the result, $\mathcal{O}(n/B)$, is the same as the proof for MVBT [7, 8] (see the discussion above). For the next part, consider any fixed version v. Invariant 5.7 states that each page of the search tree S_v contains at least min-live entries that are alive at version v. Consider now the leaf pages of the search tree S_v. We can see from the SMOs that entries are duplicated only when creating live copies of them; that is, duplicates differ in their life spans, not in their keys or key ranges. This means that within a search tree of any version v, there are no duplicates of any of the entries that are alive at version v, and thus there are at most $\mathcal{O}(m_v/\text{min-live}) = \mathcal{O}(m_v/B)$ leaf pages in S_v, because we require that min-live is linearly dependent on the page capacity B. Furthermore, by a proof that is similar to the proof of Theorem 3.1, there are at most $\mathcal{O}(m_v/B)$ index pages in the search tree S_v. From this we arrive to our result by summation over the set of all versions V. □

In restart recovery from a system crash, an ARIES-based [63, 66] recovery algorithm is used.

Theorem 5.11. In the event of a system crash, the redo pass of restart recovery produces a balanced TMVBT on which the undo actions by a backward-rolling updating transaction (if any) can be performed logically if a physical undo is impossible.

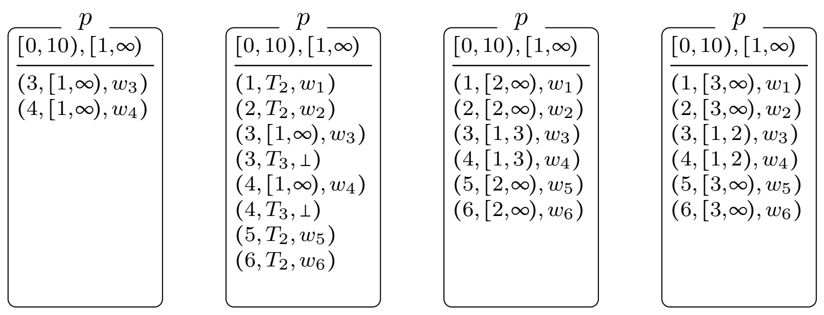
Proof. Each structure-modification operation is performed as a single atomic operation that transforms a balanced TMVBT index into another balanced TMVBT index. After the redo pass has finished, the TMVBT index is therefore balanced. Thus, each of the undo actions is performed logically, if a physical undo is not possible. □

5.6 Challenges With Concurrent Updates

The structure-modification operations presented in the previous section are based on the fact that there can be only a single updating transaction operating on the index structure at a time. We believe that it is not directly possible to further generalize the index structure so that multiple updating transactions could apply their updates directly on the leaf pages of the TMVBT index. We now present an example of the problematic situations that can arise should multiple updating transactions be allowed to operate on the index concurrently.

Assume that a transaction T_1 has created a leaf page p that contains two entries inserted by the transaction. In this illustrative example, we have set min-live = 2, so that Invariant 5.7 holds for the index structure. The situation after T_1 has committed with a commit-time version $\text{commit}(T_1) = 1$ is depicted in Figure 5.14(a).

Assume now that two active updating transactions T_2 and T_3 perform updates so that (1) T_2 inserts entries with keys preceding and succeeding the keys of the entries inserted by T_1; and (2) T_3 deletes the entries inserted by T_1. If we wish to be able to store the updates of multiple active updating transactions in the leaf pages of the TMVBT index, we must use entries such as the pending updates defined in Chapter 2 (see Definition 2.8). One possible way of modeling this situation is shown in Figure 5.14(b).



(a) T_1 has committed (b) Updates by T_2 and T_3 (c) T_2 commits first (d) T_3 commits first

Figure 5.14. TMVBT invariant fails with concurrent updates. If transaction T_3 commits before T_2, the number entries that are alive at version 2 is zero.

If T_2 commits first with $\mathsf{commit}(T_2) = 2$, and T_3 afterwards with $\mathsf{commit}(T_3) = 3$, the logical state of the database after both commit operations is shown in Figure 5.14(c). We assume here that a lazy timestamping scheme is employed so that the actual entries stored in the database may be the same entries as shown in Figure 5.14(b), but when the page is later accessed the pending updates on the page will be converted into the entries shown in Figure 5.14(c). In this situation, Invariant 5.7 holds for the page p, because $m_1 = 2$, $m_2 = 6$, and $m_3 = 4$; where m_v denotes the number of entries that are alive at version v.

Suppose now that transactions T_2 and T_3 commit in reverse order, so that T_3 commits first with $\mathsf{commit}(T_3) = 2$, and T_2 commits later with $\mathsf{commit}(T_2) = 3$. The situation after these commits is shown in Figure 5.14(d). Now, $m_2 = 0$, and Invariant 5.7 does not hold. Clearly, situations such as these should be prevented if the optimality of the TMVBT index is to be preserved.

To prevent such situations we would have to be able to (1) identify such situations when the updating transactions are active, and (2) remedy the situation by applying proper structure-modification operations. The situations must be identified while the transactions are active, because we cannot assume that the invariant could be checked and enforced at transaction commit, because this would require that each page the transaction has modified be checked during transaction commit, which is too costly to be practical. Secondly, even if we assume that such situations could be identified when the updating transactions are active, it may be impossible to remedy the situation. To show this, consider the example situation shown in Figure 5.14(b). As Figure 5.14(d) shows, this situation may lead to the invalidation of Invariant 5.7. Normally, when the number of live entries in a page falls below min-live, it is possible to merge the page with a sibling page. In this situation, however, there is no more room on the page for it to be merged with any sibling page, and the page cannot be version-split because the ordering of the entries created by the active transactions is not known. We conclude that we cannot allow multiple updating transactions to apply updates on the leaf pages of the TMVBT index directly, if the optimality of the index is to be maintained.

5.7 Summary

The TMVBT index is an optimal multiversion index for key-range queries (i.e., *x/–/point* queries, see Section 2.3), provided that the root of the queried version is known. However, only a single updating transaction can operate on it at a time. The algorithms presented in this chapter are efficient, because no key-level locking is required, and no latch-coupling is needed. As such, the TMVBT index is an optimal multiversion index structure for any application where there is only a single source of updates, and many clients performing queries. Our goal is, however, to design an efficient general-purpose multiversion index structure. The next chapter reviews the concurrent multiversion B$^+$-tree (CMVBT), which is a general-purpose multiversion index that is based on the optimal TMVBT index. Multiple updating transaction can operate on the CMVBT index concurrently.

CHAPTER 6

Concurrent Updating of the MVBT

The optimal TMVBT index that was presented in the previous chapter can only be used by at most one updating transaction at a time, but the transaction may operate concurrently with many read-only transactions. We now describe the concurrent multiversion B$^+$-tree (CMVBT) index, which uses a TMVBT index for storing the updates of committed transactions, and a separate main-memory-resident versioned B$^+$-tree (VBT) index for storing the updates of active (and recently committed) transactions. With this organization, multiple updating transactions can operate on the structure concurrently. A system maintenance transaction is run periodically to apply the updates of committed transactions into the TMVBT index, thus keeping the VBT index small. The discussion in this chapter is based on the design ideas presented in our previous article [37]. The chapter begins with an overview of the concurrent index organization in Section 6.1. After that, Section 6.2 describes the general principles on how concurrency control and recovery are managed on the index. Section 6.3 shows how the user actions are implemented, and Section 6.4 describes the maintenance transaction. Finally, in Section 6.5, we present a summary of the CMVBT structure.

6.1 Concurrent Index Organization

As discussed in the previous chapter, we believe that the TMVBT index cannot be further extended so as to be updatable by multiple concurrent updating transactions, without compromising the optimality of the structure (specifically Invariant 5.7). The concurrent index structure we propose is therefore based on the idea of collecting all the updates of active transactions and applying them to the TMVBT index as a batch update, after the transactions have committed. Batch updates have been previously used to enhance the performance of B$^+$-tree indexes [72, 73] and also to create indexes on large existing data sets while simultaneously allowing concurrent updating transactions to modify the data set [67]. In both of these cases, a separate structure is used to record the updates of active transactions. The updates are then applied to the main index structure later, after the transactions have committed (for the former case) or after the index has been created (for the latter case).

Our *concurrent multiversion B$^+$-tree*, or CMVBT, is a multiversion index structure that is composed of two parts: a main-memory-resident versioned B$^+$-tree index (VBT, as defined in Section 3.2) that is used as a temporary storage for the pending updates created by active transactions, and a transactional multiversion B$^+$-tree index (TMVBT, as described in the previous chapter) that is used as a final, stable storage in which all the committed multiversion data items are eventually stored. We assume that the VBT index has sibling pointers at each level pointing to the next page at the same level, like in a B$^{\text{link}}$-tree [52]. These pointers will be used to accelerate key-range scans in the VBT index. The transactions operating on the CMVBT index follow the transaction model presented in Sections 2.5 (for read-only transactions) and 2.6 (for updating transactions). For user transactions, the TMVBT index is a read-only structure. The TMVBT is only updated by a system *maintenance transaction*, which is run periodically to apply the updates of committed transactions from the VBT index into the main TMVBT index, and to delete the pending updates from the VBT. This setup is illustrated in Figure 6.1. We assume that only a single instance of the maintenance transaction is running at a time; for discussion on multithreading the maintenance transaction, see the discussion at the end of Section 6.4.

We expect that the VBT index will remain small under typical database workloads, as the maintenance transaction is run periodically to apply the updates to the TMVBT and delete them from the VBT. For efficiency, we thus reserve a portion of the page buffer specifically for the VBT index. This way the entire VBT index can be kept in main memory during normal transaction processing. In all of our tests, the VBT size was between 0 and 20 pages, with a typical size of just a few pages (see Section 7.3, page 149).

Like the TMVBT, the CMVBT maintains a variable v_{commit} that records the version of the latest committed transaction. In the CMVBT index, the updates are performed by inserting pending updates (see Definition 2.8) to the VBT index. The pending updates are then later on applied to the TMVBT index and deleted from the VBT. If there is a large number of concurrent updating transactions that commit at around the same time, the TMVBT index will "lag behind" until the maintenance

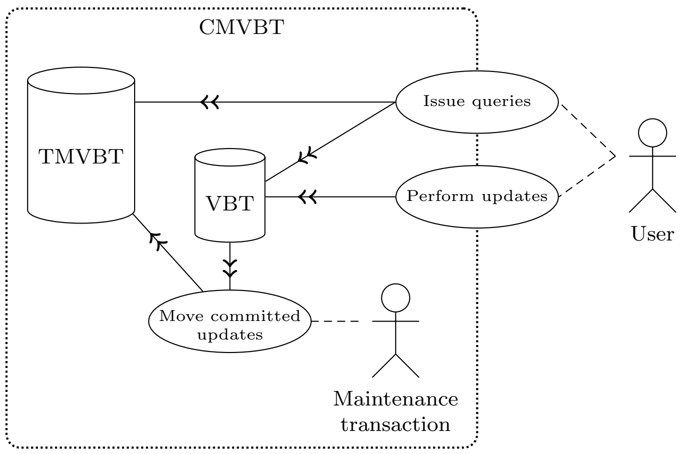


Figure 6.1. The CMVBT index structure organization. User transactions issue queries both to the VBT and the TMVBT indexes, but updates are performed only on the VBT. A system maintenance transaction is run periodically to apply the pending updates of committed transactions into the TMVBT index and to delete them from the VBT.

transaction has had time to apply the pending updates into the TMVBT. Therefore, in the context of the CMVBT index, the v_{commit} variable does not tell the maximum committed version of the TMVBT index. Accordingly, the CMVBT maintains a separate variable v_{stable} that tells which commit-time versions are already reflected in the TMVBT index. In other words, for each version $v \leq v_{stable}$, all the updates of a transaction T with $\mathsf{commit}(T) = v$ have been applied to the TMVBT index.

Definition 6.1. In the CMVBT index, the variable v_{commit} tells the version of the latest committed transaction. A separate variable $v_{stable} \leq v_{commit}$ tells the commit-time version of the latest transaction whose updates have been applied to the TMVBT index. The commit-time versions $v \leq v_{stable}$ are called *stable versions*, and the transactions that created the versions are called *stable transactions*. All the updates of stable transactions have been applied to the TMVBT index by the maintenance transaction. The commit-time versions $v > v_{stable}$ are called *transient versions*, and the corresponding transactions *transient transactions*. All the updates of transient transactions are located in the VBT index. □

By this definition, an active updating transaction is always transient, while a committed transaction can be either transient or stable. An example of the commit-time versions and transaction identifiers used in a CMVBT database is given in Figure 6.2. The example shows a database with five committed transactions (versions 1–5), and two active, uncommitted transactions (with temporary identifiers 102 and 104). Three out of the five committed transactions are stable, namely the transactions with commit-time versions 1–3, and the transaction identifiers 103 and 101 correspond to transient committed versions 4 and 5, respectively. The CTI table present in the figure is explained in the following paragraphs.

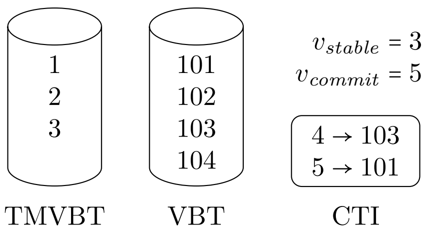


Figure 6.2. Example of the logical contents of a CMVBT index. The database contains the updates of five committed versions, versions 1–5. The TMVBT index contains all the updates of stable versions 1–3, and the updates of committed transient versions 4–5 are still located in the VBT index, identified with transaction identifiers 103 and 101. The VBT additionally contains updates by two active updating transactions that have transaction identifiers 102 and 104.

As explained in the transaction model for updating transactions in Section 2.6, we assign a transaction identifier called $\mathsf{id}(T)$ for each new transaction T. The updates of the transaction T are stored into the VBT index using transaction identifiers in the VBT entries (identifiers 101–104 in the example situation of Figure 6.2). The commit-time versions $\mathsf{commit}(T)$ of committed transactions define the ordering of the transactions (versions 1–5 in the example). When the updates of a committed transaction T are applied to the TMVBT index, the commit-time version is known, and will be used in the TMVBT index. The TMVBT thus stores commit-time versions exclusively, and the VBT index stores only transaction identifiers. The transaction identifiers are internal to the database system, and not visible to the users. The users only see the commit-time versions when issuing historical queries to the database. Both of these versions can be based either on the real time, or on an increasing counter value, as long as they are unique and increasing. In the following discussion, we assume that the versions are based on an increasing counter value.

Because the transaction identifiers and the actual, commit-time versions of the transactions may differ, a mapping from commit-time versions to the transaction identifiers is maintained in an in-memory hash-table called the *commit-time-to-identifier* table, or CTI table. In the example of Figure 6.2, the updates of committed transactions with commit-time versions 4 and 5 are still located in the VBT index with transaction identifiers 103 and 101. A mapping $v_c \rightarrow v_i$, for a committed transaction T with $\mathsf{commit}(T) = v_c$ and $\mathsf{id}(T) = v_i$, is removed from the CTI table once the maintenance transaction has applied all the committed updates of T into the TMVBT, deleted the pending updates from the VBT, and committed. The CTI table does not need to be backed onto disk. If the system fails, the CTI table can be reconstructed from the log file contents during the analysis pass of the ARIES restart recovery [66].

Definition 6.2. When the maintenance transaction is running, it applies the updates of a single version called the *move version* v_{move} into the TMVBT. The move version is always the earliest transient committed version. □

Invariant 6.3. In our numbering convention, $v_{move} = v_{stable} + 1$ whenever the maintenance transaction is running. The updates of the transaction with $\mathsf{commit}(T) = v_{move}$ can be located in both the VBT and the TMVBT indexes during the execution of the maintenance transaction. If the maintenance transaction is not running, $v_{move} = v_{stable}$.

When the maintenance transaction is running, the user transactions still use the stable version variable v_{stable} to direct the search for the correct versions of data items. In the example of Figure 6.2, the maintenance transaction could be moving the updates of transaction T with $\mathsf{commit}(T) = 4$ from the VBT to the TMVBT.

When multiple active transactions perform updates on the same data item, all the different updates are stored in the VBT, ordered by the transaction identifiers of the transactions that created them. We assume that the convention presented in Section 3.2 is used; that is, the result of a write action by a transaction T with $\mathsf{id}(T) = v_i$ is represented by a tuple (k, v_i, w) in the VBT index, and the result of a delete action is represented by a tuple $(k, v_i, \perp)$. When updating transactions commit, the pending updates remain in the VBT until the maintenance transaction deletes them after applying the updates to the TMVBT index.

Because the ordering of the committed transactions may differ from the ordering of the transaction identifiers of the transactions, the entry values in the VBT might not be in the correct commit-time order. Therefore, reading transactions that wish to read a version $v > v_{stable}$ need to find the transaction identifiers of the transactions with commit-time versions $v_c : v_{stable} < v_c \leq v$, search the VBT for all these versions, and rearrange them according to the commit-time ordering so that the most recent update is found. In the example database of Figure 6.2, a user querying for keys that are present in the commit-time version 5 must find all keys in the VBT stored with transaction identifiers 103 and 101 (corresponding to commit-time versions 4 and 5). This is relevant for situations where the queried key has not been updated by the latest preceding transaction (in the example, the transaction with commit-time version 5), so that the data item that is alive at the queried version has been created by an earlier transaction.

More formally, we define C_v to be the set of transient committed versions that are relevant when querying for the version v: $C_v = \{v_c : v_{stable} < v_c \leq v\}$. In the example of Figure 6.2, $C_5 = \{4, 5\}$. Furthermore, we define a mapping $\hat{C}_v$ from the transaction identifiers of the transactions that created the committed versions into the commit-time versions: $\hat{C}_v[\text{CTI}[v_c]] = v_c$ for all $v_c \in C_v$. In the example, $\hat{C}_5 = \{101 \to 5, 103 \to 4\}$. Now, when looking for the most recent update of key k, the VBT must be queried searching for the updates that have been created by a transaction with an identifier v_i such that there is a mapping $v_i \to v_c$ in $\hat{C}_v$. After all these updates for any given key have been found, they must be rearranged according to the ordering of the corresponding commit-time versions, so that updates with a transaction identifier v_i are sorted by $\hat{C}_v[v_i]$. After this, the most recent update is known to be the last update in the ordering. If no pending updates on key k are found in the VBT, the most recent update is queried from the TMVBT index.

The “wrong” ordering of entries in the VBT is not specific to the CMVBT structure, but is in fact present in all multiversion database systems that allow concurrent updating transactions to commit in an order different from their starting order. Lomet et al. [55] have solved this problem by using lazy timestamping, as described in Section 4.2. This technique also requires a lookup table (the persistent timestamp table [55, 57]) for converting transaction identifiers into commit-time versions. In our technique, the versions of entries are corrected when the committed pending updates are applied to the TMVBT for permanent storage, and stable versions can be queried without having to convert any versions or rearrange any updates.

6.2 Concurrency Control and Recovery

The CMVBT allows us to use various approaches for concurrency control and recovery. In this section, we describe the general idea of our concurrency-control and recovery algorithms. Our approach for database recovery follows the ARIES algorithm [63, 64, 66] with physiological logging and standard steal-and-no-force page buffering policy. Each structure-modification operation on both the VBT and the TMVBT is logged using a single physiological redo-only log record, so that interrupted tree-structure modifications are never rolled back (undone) when a transaction aborts or system fails. This approach has been described for B$^+$-trees and B$^{\text{link}}$-trees by Jaluta et al. [40, 41].

The actions of user transactions on the VBT index are logged with standard redo-undo log records, and the corresponding undo actions with redo-only log records, as described in more detail in Section 6.3. These log records are required for total and partial rollbacks of active transactions. A total rollback for a transaction T could be performed by performing a leaf-level scan of the VBT, and by deleting all the pending updates of the form $(k, \mathsf{id}(T), \delta)$. However, if a single transaction updates the same key multiple times, the previous values stored with the key are overwritten, and need to be restored when rolling back the transaction to a preset savepoint. Thus we write standard physiological redo-undo log records to log forward-rolling update actions, and redo-only log records to log undo actions, of user transactions. The structure-modification operations are logged with redo-only log records, so that they are never undone. These log records are used to bring the VBT up-to-date after a single transaction has crashed. If the entire database system crashes, it is possible to bring the VBT up-to-date either by using ARIES-based recovery [64, 66], or by reconstructing the VBT logically based on the log contents, including only the entries of committed transactions. All update actions on the TMVBT are performed by the maintenance transaction, which is never aborted or undone. If a system crash occurs during the execution of the maintenance transaction, the transaction is resumed after the system has recovered. Redo-only log records are thus sufficient for logging the actions performed on the TMVBT index.

Concurrency control on the key level is provided by the snapshot isolation (SI) algorithms (Section 2.7). In snapshot isolation, each transaction reads data items that were alive when the transaction started; that is, data items that are alive at version $\mathsf{snap}(T)$ (see Section 2.5). The version that transaction T is reading is called the *snapshot of T*. As with the TMVBT index described in the previous chapter, read-only transactions always read data from their own snapshot, and thus do not require locks to protect the keys against concurrent modifications. Updates to the database are performed by adding a new version of the updated data item to the snapshot of the updating transaction (with updates stored in the VBT index), allowing updating transactions to read their own modifications directly from the snapshot. Snapshot isolation is an obvious choice for multiversion structures, because the entire history of the database is preserved, and thus the snapshot state is available for each transaction.

Logical consistency for updating transactions is guaranteed by checking that overlapping transactions do not make updates to the same data items. Transactions T and T' are *overlapping*, if $\mathsf{snap}(T) < \mathsf{commit}(T') < \mathsf{commit}(T)$, or vice versa. We assume the same approach for enforcing snapshot isolation as is used in PostgreSQL [2], as described in Section 2.7. Each updating transaction thus takes a commit-duration lock on the key k of any data item it modifies. Furthermore, an updating transaction T must check that the data item with key k has not been updated by an overlapping committed transaction T'. We do not assume any specific method for implementing this check, but leave the details open. One possible method is to search the VBT and TMVBT to see if any updates have been made by such a committed transaction. Another possibility is for the committed transaction T' to leave persistent "residual locks" on the lock manager that conflict only with transactions that have started before T' committed. If a conflicting update is found, the active transaction T must be aborted to maintain snapshot isolation. Note that the write locks taken by updating transactions in this approach only block other updating transactions, because read-only transactions take no locks.

The global version variables v_{commit}, v_{stable} and v_{move} are maintained in the persistent database, and the reading and writing of v_{commit} and v_{move} are protected with locks, as in the TMVBT (Section 5.4). A **begin-read-only** action acquires a short-duration read lock on v_{commit} for reading its value, and a **commit-update** action acquires a commit-duration write lock on it for incrementing its value. The maintenance transaction acquires a commit-duration write lock on v_{move} at the beginning, thus guaranteeing that at most one maintenance transaction is active at a time. The stable version variable v_{stable} is read often, and thus the reading and writing of it is protected by latching. This approach is sufficient, because v_{stable} is only updated by the maintenance transaction, which is never undone. The updating of v_{stable} is protected by a write latch taken on the database page on which the variable is stored, and the reading of the variable by query actions is protected by a read latch taken on the same page.

Structural consistency of the VBT index is maintained by standard page-latching operations, with latch-coupling applied to ensure child-link consistency during tree traversals. We define the latching order to be top-down, left-to-right for all transactions, and we disallow upgrading read latches to write latches. Furthermore, all page latches must be released whenever a write lock cannot be acquired immediately, so that no page latches are held when the transaction is waiting for a lock. These are necessary (and sufficient) restrictions to avoid deadlocks that involve page latching [32]. For the TMVBT index, pages need to be latched, but latch-coupling is not always required, as explained in Section 5.4. This is because the only transaction that is allowed to perform updates on the TMVBT index is the system maintenance transaction, and there can be only one such transaction running at a time. The maintenance transaction therefore does not need to do latch-coupling. Read-only transactions that read inactive data do not need latch-coupling either, as noted in the previous chapter. This means that read-only transactions that are reading stable versions (from the TMVBT) do not need latch-coupling. Further details of concurrency control and recovery are embedded in the explanations of the algorithms that are described in the next sections, alongside with more detailed explanations of the latching policy.

6.3 User Actions

We allow two kinds of user transactions to operate concurrently on the CMVBT: read-only transactions and updating transactions. There are no restrictions on how many transactions of either type can be running at the same time. Read-only transactions follow the transaction model described in Section 2.5, and updating transactions follow the model described in Section 2.6. As in Section 5.1, we write T in the log records of user actions to mean that the transaction identifier $\mathsf{id}(T)$ is written in the log record.

For the user actions of read-only and updating transactions that operate on the CMVBT index, we need to define the following algorithms:

1. *update-item* to perform an update action (insert or delete).
2. *query-stable* to query for a single key of a stable version.
3. *query-transient* to query for a single key of a transient version.
4. *next-key-stable* to query for the next key of a stable version.
5. *next-key-transient* to query for the next key of a transient version.

These algorithms are presented later on.

With these algorithms, the actions of a read-only transaction are performed as follows. Key-level locking is not required for read-only transactions.

- **begin-read-only**(version v): begins a new read-only transaction; this action takes a short-duration read lock on v_{commit}, reads the value of the variable, checks that $v \leq v_{commit}$, and records the value $\mathsf{snap}(T) \leftarrow v$ for the transaction. If the version check fails, the transaction is not allowed to begin.
- **query**(key k): this action reads the value of the stable-version variable v_{stable}. If $\mathsf{snap}(T) \leq v_{stable}$, then *query-stable* is run to find the correct data item from the TMVBT index; otherwise the *query-transient* algorithm is run to either find the latest relevant pending update from the VBT index or to find the latest data-item entry from the TMVBT index. The reading of v_{stable} is protected by read-latching the database page p that contains the variable. The value can be cached for the transaction, so that subsequent queries do not need to read the page p.
- **range-query**(range $[k_1, k_2)$): like the **query** action, this action reads the value of the stable version variable v_{stable}. If $\mathsf{snap}(T) \leq v_{stable}$, then *next-key-stable* is run to find the set of data items from the TMVBT index; otherwise *next-key-transient* is run to retrieve the set of data items from both the VBT and the TMVBT indexes. The set of data items is retrieved by first finding the first entry from the range with the query $(k, w) \leftarrow \textsc{n}(k_1, \mathsf{snap}(T))$, and $\textsc{n}$ is either *next-key-stable* or *next-key-transient*, as explained above. If $k < k_2$, the snapshot data item (k, w) is added to the result set, and the next data item is fetched with $(k', w') \leftarrow \textsc{n}(k, \mathsf{snap}(T))$. This iteration is continued until a data item that is outside the queried range $[k_1, k_2)$ is found, or until the algorithm $\textsc{n}$ returns no more results. To be more precise, the first entry is located with a slightly different query algorithm because the first entry in the TMVBT index may have the key k_1, and thus the query actually find the first entry (k, w) with $k \geq k_1$, instead of $k > k_1$. When querying for the rest of the keys, the next key must be greater than the previous key. We have omitted the description of the first-key-query algorithm because it is almost identical to the next-key-query algorithm.
- **commit-read-only**: commits the transaction by removing it from the system. No further actions are required.

For updating transactions, the actions are implemented as follows. Key-level locking is only required where specifically stated.

- **begin-update**: begins a new updating transaction T; this action creates an identifier for the transaction ($\mathsf{id}(T) \leftarrow$ *new identifier*), acquires a short-duration read lock on the variable v_{commit}, and records the snapshot version $\mathsf{snap}(T) \leftarrow v_{commit}$. The action finishes by writing the log record $\langle T$, **begin**, $\mathsf{snap}(T)\rangle$, but the log is not forced to disk.
- **query**(key k): this action queries the VBT and the TMVBT indexes to find the data item that is alive at $\mathsf{snap}(T)$. Although $\mathsf{snap}(T)$ might be stable, this action must always use the *query-transient* algorithm to query both the VBT and the TMVBT indexes, because the transaction might have itself updated the key k, and pending updates are located only in the VBT index.
- **range-query**(range $[k_1, k_2)$): like the **query** action, this action always uses the *next-key-transient* algorithm to query both VBT and TMVBT indexes to locate the data items that are alive at $\mathsf{snap}(T)$ in the range $[k_1, k_2)$.
- **write**(key k, data w): this action takes a commit-duration write lock on the key k, and invokes the *update-item* algorithm to insert a pending update into the VBT. This action writes a redo-undo log record $\langle T$, **write**, p, k, w, δ', $n\rangle$, where (1) p is the page identifier of the VBT page on which the pending update was inserted; (2) δ' is the value of an overwritten pending update, if the action replaced an earlier update created by T; or $\varnothing$, otherwise; and (3) n is the Undo-Next-LSN of the log record; that is, the LSN of the last (not-yet-undone) action performed by T before this action.
- **delete**(key k): the operation of this action is identical to the **write** action, except that in the *update-item* algorithm a deletion is performed instead of data-item insertion, and the redo-undo log record that is written is $\langle T$, **delete**, p, k, w', $n\rangle$. We denote the possibly replaced value of a pending update here by w' instead of δ', because in this case, the replaced value cannot be $\perp$.
- **set-savepoint**: sets a savepoint and returns the savepoint identifier s to the transaction. The action first writes a redo-undo log record $\langle T$, **savepoint**, $n\rangle$. The savepoint identifier s is the log sequence number of the savepoint log record.
- **rollback-to-savepoint**(LSN s): rolls the transaction back to a preset savepoint identified by the log sequence number s. This

action is followed by the **undo-write** and **undo-delete** actions for the **write** and **delete** actions performed after setting savepoint s, executed in the reverse order. The **write** and **delete** actions are located with a reverse scan of the log records, starting from the last action performed by T, and by locating the previous records via the Undo-Next-LSN pointers in the log records that point to the log record of the previous action, until the log record for the **set-savepoint** action is found for the savepoint s.

- **commit-update**: commits the updating transaction; this action (1) acquires a commit-duration write lock on v_{commit}, (2) increments the variable $v_{commit} \leftarrow v_{commit} + 1$, (3) assigns a commit-time version to the transaction $\mathsf{commit}(T) \leftarrow v_{commit}$, (4) adds the mapping $\text{CTI}[\mathsf{commit}(T)] \leftarrow \mathsf{id}(T)$, (5) writes a log record $\langle T$, **commit**, $v_{commit}\rangle$, (6) forces the log onto disk, (7) releases the lock on v_{commit}, and (7) removes the transaction T from the system. The release-version action is performed by the system maintenance transaction some time after the commit operation has finished. The commit-time version $\mathsf{commit}(T)$ can be queried by other transactions immediately after this action has completed, before the maintenance transaction has run.
- **release-version**: incorporates the updates performed by the earliest transient committed transaction into the TMVBT index, and removes the pending updates from the VBT index. This action is implemented by the maintenance transaction that is explained in more detain in Section 6.4.
- **abort**: labels the transaction as aborting and starts the backward-rolling phase. This action writes the log record $\langle T$, **abort**, $n\rangle$. In the backward-rolling phase, all the actions of the transaction are undone, in reverse order, by following the previous-record pointers (i.e., the Undo-Next-LSN values) in the log records. This action is thus similar to the **rollback-to-savepoint** action, except that the log is rolled back until the **begin** log record is encountered, at which point the **finish-rollback** action is executed.
- **undo-write**(log record r): this action undoes a write action by reading the log record $r = \langle T$, **write**, p, k, w, $\delta', n\rangle$ and by physically removing the entry $(k, \mathsf{id}(T), w)$ from the index. If $\delta' \neq \emptyset$, then the write action replaced an earlier pending update created by transaction T, and the undo action inserts the pending update $(k, \mathsf{id}(T), \delta')$ into the VBT index to restore the overwritten entry.

Note that δ' may also be $\perp$, if the transaction had deleted the key k earlier. The undo action finishes by writing the redo-only compensation log record $\langle T$, **undo-write**, p', k, w, δ', $n\rangle$, where p' is the page identifier of the VBT page on which the update was performed. If a structure-modification operation has occurred on page p after the **write** actions, it is possible that $p' \neq p$.

- **undo-delete**(log record r): this action undoes a delete action by reading the log record $r = \langle T$, **delete**, p, k, w', $n\rangle$ and by physically removing the entry $(k, \mathsf{id}(T), \perp)$ from the VBT index. If $w' \neq \varnothing$, then the delete action replaced an earlier pending update created by transaction T, and the undo action inserts the pending update $(k, \mathsf{id}(T), w')$ into the VBT index to restore the overwritten pending update. The undo action finishes by writing the redo-only compensation log record $\langle T$, **undo-delete**, p', k, w', $n\rangle$.
- **finish-rollback**: finishes the rollback of a backward-rolling transaction by writing a log record $\langle T$, **finish-rollback**$\rangle$, forcing the log onto disk, and by removing the transaction T from the system.

Let us now consider the algorithms required to implement the user actions. The algorithm *update-item* which performs an update action on the CMVBT index is reasonably straightforward: all that needs to be done is to record the action to the VBT, using top-down, left-to-right latch-coupling for tree traversal. An insertion of a data item with key k by a transaction T with $\mathsf{id}(T) = v$ is recorded by adding the entry (k, v, w), where w is the entry value; and deletion of a data item with key k by transaction T is recorded by adding the entry $(k, v, \perp)$. If the VBT index already contains an entry (k, v, δ') with the same key and version, the old entry will be replaced by the new entry. The possible structure-modification operations on the VBT needed to accommodate the update are performed level-by-level as separate atomic actions that are never undone, before the installation of the update, similarly to the SMOs for the TMVBT index (see Section 5.5).

Querying for stable versions with the *query-stable* algorithm is also straightforward. When a read-only transaction queries a stable version, the transaction can directly query for the version from the TMVBT index, as explained in Section 5.4, by using the TMVBT query action. In this situation, the TMVBT index can be traversed without latch-coupling, and saved paths can be used for key-range and next-key queries without any need to check for page validity when relatching the pages on the path. This is possible because all pages traversed and entries read are inactive, and thus guaranteed to remain where they are located, by Invariant 5.6.

The read-only transaction can determine whether a version is stable by reading the stable version variable v_{stable}. The reading of the variable is protected by read-latching the page on which it is stored. The latch can be released immediately after the value of the variable has been determined. It is also possible to read the variable value once, at the beginning of the transaction, and then cache the result for the transaction.

When a read-only transaction is querying for a transient version, and when an updating transaction is performing any key query, the *query-transient* algorithm must be used. Both read-only and updating transactions use the same *query-transient* algorithm for querying the index structures, and we will thus use the term *reader* to refer to a read-only transaction or an updating transaction that is performing a query action. When the queried version is transient, the reader cannot know beforehand which index structure contains the most recent update that precedes the queried version. The relevant version might be the last update in the TMVBT, or there may be an intermediate update in the VBT. For illustration of such a situation, an example of the possible contents of a CMVBT index is given in Figure 6.3. This example represents the same logical situation as the example shown in Figure 6.2, but now the actual entries stored in the different indexes are shown. Furthermore, when an updating transaction has itself performed an update action, the pending update is only located in the VBT, and it is thus not sufficient to restrict the search to the TMVBT, even if the snapshot version snap(T) of the updating transaction is stable.

In the example of Figure 6.3, when performing queries that target the transient commit-time version 5, the relevant version for key 3 is the last update in the TMVBT index (created by transaction T_3), while the relevant version for key 2 is the update in the VBT by transaction T_5 with transaction identifier 101. Both structures may need to be checked whenever querying for a transient version. For single-key queries (with the key given), it is sufficient to restrict the search to the VBT index if an update on the key is found from there; only if no such update is found from the VBT do we need to consult the TMVBT to find the latest update on the key. However, when performing next-key queries, both structures always need to be checked, because it is impossible to determine which structure contains the nearest key otherwise.

Because the maintenance transaction may be moving the updates of the move version v_{move} to the TMVBT during the execution of the query action, the reader must be prepared to miss some of the updates of version v_{move} in the VBT, and to possibly re-encounter some already encountered updates in the TMVBT. The situation is amended by organizing

TMVBT	**VBT**	**CTI**
$(1, [1, 2), w_1)$	$(1, 102, w'_1)$	$4 \to 103$
$(2, [1, \infty), w_2)$	$(2, 101, w'_2)$	$5 \to 101$
$(3, [2, 3), w_3)$	$(4, 103, \perp)$	
$(3, [3, \infty), w'_3)$	$(4, 104, w'_4)$	
$(4, [3, \infty), w_4)$	$(6, 101, w_6)$	
	$(7, 103, w_7)$	

History of transactions

T_1, $\mathsf{commit}(T_1) = 1$: Insert $(1, w_1)$ and $(2, w_2)$
T_2, $\mathsf{commit}(T_2) = 2$: Insert $(3, w_3)$ and delete item with key 1
T_3, $\mathsf{commit}(T_3) = 3$: Insert $(3, w'_3)$ and $(4, w_4)$
T_4, $\mathsf{commit}(T_4) = 4$: Insert $(7, w_7)$ and delete item with key 4
T_5, $\mathsf{commit}(T_5) = 5$: Insert $(2, w'_2)$ and $(6, w_6)$
T_6, $\mathsf{id}(T_6) = 102$: Insert $(1, w'_1)$
T_7, $\mathsf{id}(T_7) = 104$: Insert $(4, w'_4)$

Figure 6.3. Example of data entries stored in a CMVBT index. This example represents the same situation as Figure 6.2. The format of entries in the TMVBT is (key, life span, data), and the format of entries in the VBT is (key, transaction identifier, update). In addition to the committed transactions, the VBT also contains the updates of two active transactions with transaction identifiers 102 and 104.

the read actions in such a way that the VBT is always consulted first, and the TMVBT only afterwards. This guarantees that no update is missed.

The general algorithm for *query-transient* is given in Algorithm 6.1. Querying for a single key from the TMVBT index with the procedure *query-stable* works exactly like the single-key query action of the TMVBT, explained in Section 5.4, except that the third parameter is used to indicate that the reader may need to read active data and must therefore use latch-coupling when traversing the paths of the TMVBT index. Note that the version v may be greater than the most recent version of the TMVBT (i.e., v_{stable} or v_{move}). The searches in the TMVBT still work, because the live pages (and entries) in the TMVBT have a life span of the form $[v', \infty)$, which covers v.

When reading the pages of the TMVBT index, the reader can determine whether a page has been invalidated by a concurrent structure-modification operation (for example, when relatching pages on a previously released saved path) by internally caching the page-LSN before

```
QUERY-TRANSIENT$(k, v, T)$:
1 $\delta \leftarrow$ KEY-QUERY-VBT$(k, v, T)$
2 if $\delta = \varnothing$ then // If no updates found from VBT
3     return QUERY-STABLE$(k, v, \textbf{true})$
4 else if $\delta = \perp$ then // If latest update in VBT is a deletion
5     return $\varnothing$
6 else // Latest update in VBT is an insertion
7     return $\delta$
8 end if
```

Algorithm 6.1. Key-query algorithm for transient versions. This algorithm is used by read-only transactions when querying for transient versions, and by updating transactions when performing any key queries.

releasing latches and by then comparing the cached value to the value in the page after it has been relatched. If the new page-LSN is greater than the cached one, then the page has been modified by the maintenance transaction, and it is necessary to either re-traverse the entire path from the root of the TMVBT index or to backtrack up the saved path until an unmodified page is found.

Querying for a single pending update from the VBT is described in Algorithm 6.2. The algorithm generates a reverse identifier-to-commit-time (ITC) mapping for all relevant commit-time versions, mapping the possible updates created by the transaction itself to infinity so that they always take precedence over other updates. A transaction identifier list I is also generated. This list contains the transaction identifiers for which there is a mapping in the ITC. The *query-all-VBT*(k, I) operation finds all pending updates (k, v', δ') from the VBT with key k and transaction identifier $v' \in I$. This function is implemented using a standard VBT range-query operation with latch-coupling applied when traversing in the index. After finding all the possibly relevant updates, the *key-query-VBT* function orders the updates in transaction commit order, and returns the value δ' of the latest update. If there are no updates on the key k in the VBT, the function returns the null marker $\varnothing$. If the last update is a deletion, the returned value is $\delta' = \perp$. These values must be separate so that the single-key-query algorithm knows whether to continue searching from the TMVBT or not.

In the example database of Figure 6.3, the reverse ITC mapping when querying for commit-time version 5 with a read-only transaction T_r is $\{101 \rightarrow 5, 103 \rightarrow 4\}$. If the querying transaction is an updating transaction T_u with transaction identifier $\mathsf{id}(T_u) = 104$, the ITC mapping is

$\{101 \to 5,\ 103 \to 4,\ 104 \to \infty\}$. For example, if T_u queries for key 4 (at transient commit-time version 5), the ITC is generated as explained above, and the transaction identifier list would be $I = \{101, 103, 104\}$. The *query-all-VBT* operation finds the entries $(4, 103, \bot)$ and $(4, 104, w'_4)$, which are then converted to commit-time entries and placed in the ordered result map $K_c = \{4 \to \bot, \infty \to w'_4\}$. The query returns the value with the highest key from this map, w'_4, as the result.

```
KEY-QUERY-VBT($k, v, T$):
ITC ← ∅
$I \leftarrow \emptyset$
**for** each $v_c \in \{v_{stable} + 1, \dots, v\}$ **do**
    $v_i \leftarrow \mathrm{CTI}[v_c]$ // *Find the transaction identifier*
    $\mathrm{ITC}[v_i] \leftarrow v_c$
    $I \leftarrow I \cup \{v_i\}$
**end for**
**if** $T$ is an updating transaction **then**
    $\mathrm{ITC}[\mathsf{id}(T)] \leftarrow \infty$ // *Updates by T have precedence*
    $I \leftarrow I \cup \{\mathsf{id}(T)\}$
**end if**
$K_i \leftarrow$ QUERY-ALL-VBT$(k, I)$
$K_c \leftarrow \emptyset$
**for each** $(k, v', \delta') \in K_i$ **do**
    $v_c \leftarrow \mathrm{ITC}[v']$
    $K_c[v_c] \leftarrow \delta'$
**end for**
**return** entry $K_c[v_c]$ with highest version $v_c$; or $\emptyset$, if $K_c$ is empty
```

Algorithm 6.2. Algorithm for querying for a single key from the VBT index.

When a user transaction T is querying for a single key k of a transient version $v > v_{stable}$, and the maintenance transaction T_m is ongoing, one of the following situations may occur, regarding an update of version v_{move} when that update is the latest update on the key k:

1. T finds the pending update in the VBT. The update has not yet been applied to the TMVBT index. This is the normal situation, and no special processing is required.
2. T finds the pending update in the VBT. The update has been applied to the TMVBT by the maintenance transaction T_m. Be-

cause the pending update was found in the VBT, the TMVBT is not searched, and thus this situation is similar to the first one.

3. T does not find the pending update of version v_{move} in the VBT, because the maintenance transaction T_m has already deleted it. Because no update was found in the VBT, the TMVBT is searched to find the latest update. At this point, the maintenance transaction T_m has already applied the update to the TMVBT, so the update is found in there.

If there is a more recent update with commit-time version v_c such that $v_{move} < v_c \leq \mathsf{snap}(T)$, then this update will be found in the VBT in all of the above situations, and it will be returned directly without consulting the TMVBT index.

For next-key queries, the general algorithm follows the same structure as the single-key query: if a read-only transaction is querying for a stable version, the algorithm *next-key-stable* is used to query the TMVBT index directly. If the queried version is transient, or if an updating transaction is performing the range-query, the algorithm *next-key-transient* (Algorithm 6.3) is run. In this case, both the VBT and the TMVBT must always be searched to locate the data item with the next key.

When searching for a transient version, the next keys from both structures need to be retrieved alternatingly. In Algorithm 6.3, starting from line 1, the current key is initialized to the previously found key. After this, at the beginning of the infinite loop, both of the index structures are searched to find the next key, and the algorithm checks which key is nearest to the previously found key. If the next nearest key is found in the TMVBT (line 5), we can return the key and the corresponding value directly. If, on the other hand, the nearest key is in a pending update found in the VBT (line 7), we must check whether the pending update is an insertion (and not a deletion) before returning the key. Similarly, if the keys fetched from both structures are the same (line 11), we need to check that the pending update on the key in the VBT is not a deletion. If the pending update is a deletion, we need to skip this key and scan forward to find the next key. Whenever a pending deletion is found, and the indexes need to be scanned forward, both of the structures need to be queried again. This means that the tuple (k_2, w_2) found from the TMVBT cannot be reused, because an ongoing maintenance transaction might have changed the nearest key in the TMVBT index by inserting a new entry (k_2', w_2') with $k_2' < k_2$ into the TMVBT.

Like the *key-query-VBT* function, the *next-key-VBT* function needs to find the transaction identifiers of all the commit-time versions between

```
NEXT-KEY-TRANSIENT($k, v, T$):
$k_c \leftarrow k$ // Initialize the current key
loop
  $(k_1, \delta_1) \leftarrow$ NEXT-KEY-VBT($k_c, v, T$)
  $(k_2, w_2) \leftarrow$ NEXT-KEY-STABLE($k_c, v,$ true)
  if $k_1 > k_2$ then // Data item with next key is in TMVBT
    return $(k_2, w_2)$
  else if $k_1 < k_2$ then // Pending update for the next key is in VBT
    if $\delta_1 \neq \perp$ then // Is $\delta_1$ an insertion or update
      return $(k_1, \delta_1)$
    end if
  else // Same key returned from both structures
    if $k_1 = \varnothing$ then // No next keys in either index
      return $\varnothing$
    else if $\delta_1 \neq \perp$ then // Is $\delta_1$ an insertion?
      return $(k_1, \delta_1)$
    end if
  end if
  $k_c \leftarrow k_1$ // Scan forward in both indexes
end loop
```

Algorithm 6.3. Algorithm for finding the next live data-item starting from a given key.

v_{stable} and v, find pending updates with these transaction identifiers, and order the updates based on their corresponding commit-time versions. The actual implementation of the next-key query should use saved paths to accelerate the next-key queries from the VBT and TMVBT indexes. With saved paths, most of the consecutive next-key calls to the VBT and CMVBT indexes will fall to the same leaf page, and they will reuse the existing saved path without requiring any additional I/O operations. To further enhance the operation, page fixes and latches in the VBT and TMVBT index structures need not be released between the next-key-query sub-operations in the main loop of Algorithm 6.3.

As an example of a next-key query, suppose a read-only transaction T_r is querying for the key next to key 3 at version 4, in the example database of Figure 6.3. At the beginning of the next-key-query function, the next entries are fetched from both structures: $(k_1, \delta_1) = (4, \perp)$ (from the VBT) and $(k_2, w_2) = (4, w_4)$ (from the TMVBT). Because the keys of both of the entries are the same, the one retrieved from VBT takes precedence. However, because it is a deletion marker, we need to continue the search to find the next entries. The algorithm thus continues by finding the entries next to key 4 from both of the structures: $(k_1, \delta_1) = (7, w_7)$ (from the VBT) and $(k_2, w_2) = \varnothing$ (from the TMVBT). Because the TMVBT has no more entries, the VBT entry is the next key, and the key-value pair $(7, w_7)$ is returned to the user.

The following theorems state the correctness and complexity of the user actions, expressed in terms of how many pages need to be visited:

Theorem 6.1. The update action correctly records a key update (key insertion or deletion) in the VBT. A single update action has a cost of $\Theta(\log_B n_V)$ page accesses, where n_V is the number of entries in the VBT.

Proof. The update action must traverse the VBT once to insert the update marker, so the complexity $\Theta(\log_B n_V)$ comes from the tree traversal. As with standard B$^+$-trees, any required structure-modification operations do not affect the asymptotic complexity of the action. □

Theorem 6.2. The key-query action for key k and version v correctly returns the most recent committed version v' of the queried key, relative to version v so that $v' \leq v$. A single key-query action for a stable version v has a cost of $\Theta(\log_B m_v)$ page accesses, where m_v is the number of entries in the TMVBT index that are alive at version v, and B is the page capacity. A single key-query action for a transient version v requires at most $\Theta(\log_B n_V + (n_a + n_t)/B + \log_B m_{v_{stable}})$ page accesses, where n_V is the number of entries in the VBT, $m_{v_{stable}}$ is the number of entries in the TMVBT that are alive at the stable version v_{stable}, n_a is the number of active updating transactions, and n_t is the number of transient versions.

Proof. The key-query action for a stable version performs a single-key query action on the TMVBT, and thus has a cost of $\Theta(\log_B m_v)$ page accesses (Theorem 5.4). For a transient version, the VBT search requires $\Theta(\log_B n_V)$ page accesses for the initial tree traversal, and at most an additional $\Theta((n_a + n_t)/B)$ leaf page accesses to locate all pending updates that might be relevant to the key query. The latter part of the cost is derived from the fact that there can be at most $n_a + n_t$ pending updates that have the key k in the VBT index, and all of these may have to be scanned. Because the entries with the same key are clustered together, it suffices to scan $(n_a + n_t)/B$ VBT pages. Finally, the query for a transient version may need to further query the stable version from the TMVBT, thus adding the $\Theta(\log_B m_{v_{stable}})$ term to the complexity of the action. □

Theorem 6.3. A range-query action for the key range $[k_1, k_2)$ targeting a stable version v has a cost of $\Theta(\log_B m_v + r/B)$ page accesses, where m_v is the number of entries in the TMVBT index that are alive at version v, r is the number of entries returned by the query, and B is the page capacity. A range-query action targeting a transient version v has a cost of at most $\Theta(m_k \log_B n_V + \log_B m_{v_{stable}} + r/B)$ pages, where m_k is the maximum number of discrete keys at the queried interval (for databases that store integer keys, $k = k_2 - k_1$), and n_V is the number of entries in the VBT.

Proof. The range-query action of a stable version queries the TMVBT index directly, and the proof is thus the same as the proofs of Theorems 5.4 and 4.2. When querying for a transient version, it is possible that there are transient pending deletions in the VBT recorded for every possible discrete key value in the queried range, and the algorithm must process them all. In such a situation, the *next-key-transient* algorithm needs to perform next-key queries for each possible discrete key value in the queried range. Each of these queries requires at most a single root-to-leaf traversal of both the VBT and the TMVBT. In the VBT, it is possible that the saved path cannot be efficiently reused to obtain the next key directly, because there may be other pending updates with the same key but different versions in the way. In the TMVBT, however, the saved path will be reused and the same number of pages need to be accessed in total as when querying for the stable version. □

6.4 Maintenance Transaction

The maintenance transaction is a system-generated transaction that is run periodically to apply the pending updates of committed transactions, one transaction at a time, into the TMVBT index, and to delete the pending updates from the VBT. To ascertain correct operation with concurrent user transactions, the updates must be applied in such a way that the system transaction does not cause user transactions to miss any of them. This is accomplished by first applying the pending updates into the TMVBT, then increasing the stable version variable v_{stable}, and only then removing the pending updates from the VBT index. A consequence of this approach is that the user transactions must be prepared to possibly encounter the same update twice when scanning the index structures.

The maintenance transaction performs the following steps:

1. Acquire a commit-duration write lock on the global variable v_{move}, update the variable $v_{move} \leftarrow v_{stable} + 1$, and find out the corresponding transaction identifier $v_i \leftarrow \text{CTI}[v_{move}]$. Reading of the stable version variable v_{stable} is protected by the read latch taken on the database page on which the variable is stored.

2. Scan through the VBT index to find the pending updates (k, v_i, δ) created by the transaction T with $\mathsf{id}(T) = v_i$. Apply the updates into the TMVBT index, changing the version from v_i to v_{move}.
3. Update the stable version to $v_{stable} \leftarrow v_{move}$. Updating the variable is protected by taking a write latch on the database page on which the variable is stored.
4. Scan through the VBT index a second time, and physically delete all the entries of the form (k, v_i, δ).
5. Remove the mapping $v_{move} \rightarrow v_i$ from the CTI table.

The actions performed by the maintenance transaction are logged using redo-only log records. If the system crashes during the execution of the maintenance transaction T_m, the redo pass of restart recovery will redo all actions performed by T_m to bring the database pages into a consistent state and restart the maintenance transaction. All the steps of the maintenance transaction are idempotent, meaning that performing them multiple times has the same effect as performing them once. That is, $f(f(x)) = f(x)$ for all actions f of the maintenance transaction T_m and for all possible states x of the combined CMVBT (TMVBT + VBT) index structure. When the maintenance transaction is restarted after a system crash, it will automatically skip those actions that already have been performed.

At the beginning, in step 1, the move version v_{move} is incremented to record that the maintenance transaction is active. Using the actions described for the TMVBT index in Section 5.4, the maintenance transaction invokes the **begin-update** action of the TMVBT to update the active version variable v_{active} of the TMVBT to v_{move}. In practice, however, the active version variable should be the same as the move-version variable, when TMVBT is used as a part of the CMVBT index.

The database management system must guarantee that there is only a single maintenance transaction running at any time. Thus, at the very beginning, the maintenance transaction takes a commit-duration write lock on v_{move}. This action is logged by a redo-only log record $\langle T_m$, **begin-maintenance**, v_{move}, $v_i\rangle$, where v_i is the transaction identifier of the transaction that has the commit-time version v_{move}.

At the next step, step 2, the pending updates recorded in the VBT are applied to the TMVBT. The maintenance transaction scans through both structures at the same time, using a saved path for the TMVBT and latch-coupling for the VBT. Because the system transaction is the only transaction updating the TMVBT, no latch-coupling on the TMVBT is required, and only one page needs to be latched at a time, except during structure-modification operations. Because the TMVBT does not have sibling links, the saved path needs to be backtracked to locate the next leaf page. Pages on the saved path can be safely relatched, because there is no other transaction that can update the TMVBT, but pages lower on in the path must still be unlatched before latching pages on a higher level to maintain top-down, left-to-right ordering on page latching. In the VBT, the sibling links assumed in Section 6.1 are used to traverse through the leaf pages efficiently. Because T_m only reads the updates from the VBT at this point, it is sufficient to perform a leaf-level scan, and thus no saved path is required.

A pending update (k, v_i, δ) is applied on the TMVBT by using the actions defined in Section 5.4. If $\delta \neq \bot$, the action **write**(k, δ) is performed; otherwise the action **delete**(k) is invoked. Each update that is performed to the TMVBT is logged using redo-undo log records. The log records defined in Section 5.4 could also be used, but they contain undo information which is not needed, because the actions of the maintenance transaction are never undone. The redo-only log record written for a write action, recorded by the pending update (k, v_i, w), $w \neq \bot$, is thus $\langle T_m$, **apply-write**, p, k, v_{move}, $w\rangle$, where p is the page identifier of the TMVBT page on which the update was performed. Note that the version v_{move} is used by the TMVBT algorithms when applying the update to the TMVBT index, because we require that $v_{active} = v_{move}$. The redo-only log record written for a delete action, recorded by the pending update $(k, v_i, \bot)$, is $\langle T_m$, **apply-delete**, p, k, $v_{move}\rangle$.

Step 3 updates the stable version variable $v_{stable} \leftarrow v_{move}$. The variable is protected by write-latching the page on which the variable is stored. At this point we know that all the updates of transaction T with $\mathsf{commit}(T) = v_{move}$ have been applied to the TMVBT. New read-only transactions can therefore perform queries that target the version v_{move} directly on the TMVBT index. To speed up recovery, this action is also logged with a single redo-only log record $\langle T_m$, **increment-stable**, $v_{stable}\rangle$, and the log is forced onto the disk at this point. If this log record is found after a system crash, steps 1–3 of the maintenance transaction are skipped entirely, and the maintenance transaction continues at step 4 to finish the transaction.

Next, at step 4, the pending updates are deleted from the VBT index. It is safe to delete these updates, because although deleting the entries may cause concurrent user transactions to miss an update they expected to find in the VBT, the transactions will find the missed update later on from the TMVBT, where it has already been applied to at this point. To correctly maintain the structural consistency of the VBT, the tree must be traversed from the root, using a saved path, so that structure-modification operations can be performed if pages contain too few entries after entry deletions. During the search, latch-coupling will be applied, first top-down, then left-to-right, as explained earlier. The delete action for a pending update (k, v_i, δ) is logged with a redo-only log record $\langle T_m$, **delete-update**, p, k, $v_i\rangle$, where p is the page identifier of the VBT page on which the pending update was located.

Finally, in step 5, the temporary transaction identifier mapping is removed from the CTI table. The maintenance transaction commits by writing a redo-only log record $\langle T_m$, **commit-maintenance**$\rangle$. The log must be forced onto disk at this point.

The following theorem states the correctness and complexity of the maintenance transaction:

Theorem 6.4. Let n denote the number of updates applied by the earliest transient committed transaction with version v_{move}, n_V the number of pending updates in the VBT and $n_T = m_{v_{move}-1}$ the number of entries in the TMVBT that are alive at version $v_{move} - 1$. The maintenance transaction correctly transforms the transient version v_{move} into a stable version by applying the updates of the version v_{move} into the TMVBT, and by removing the pending updates from the VBT. The maintenance transaction requires access to at most $\mathcal{O}(n_V/B + \min\{n \log_B(n_T + n), (n_T + n)/B\})$ pages, where B is the page capacity.

Proof. The complexity of the maintenance transaction is composed of the VBT and TMVBT index scans of steps 2 and 4. The complexity of both scans of the VBT is the same, $\Theta(n_V/B)$, because a full leaf-level scan of the VBT is required for both of the steps. Maintaining the entire saved path from root to leaf for the VBT in the deletion phase does not add to the asymptotic complexity of the scan (see the proof of Theorem 3.1). Applying the n updates into the TMVBT requires $\mathcal{O}(n \log_B n_T)$ page accesses in the worst case, because a single update operation in the TMVBT requires $\Theta(\log_B(n_T + n))$ page accesses (see Section 5.4). However, the leaf pages of the search tree $S_{v_{move}-1}$ of the latest version of the TMVBT are accessed at most once, so the operation is also bound by $\mathcal{O}((n_T + n)/B)$; that is, a full leaf-level page scan of the latest version of the TMVBT plus the new pages created by insertions. $\square$

Throughout this chapter, we have assumed that a single thread or process is running a single maintenance transaction to apply the updates of a single committed transaction from the VBT into the TMVBT at a time. If the maintenance transaction becomes a bottleneck for the system, it should be possible to extend the algorithms presented here so that multiple threads or processes run multiple maintenance transactions concurrently. The multiple maintenance transactions still have to apply all the updates of a single committed transaction at a time, before starting to move the updates of another committed transaction, because only the updates of a single version can be applied to the TMVBT at a time (see Section 5.6). In this approach, the maintenance transactions are set up to scan different portions of the VBT index each, and to apply the updates to the TMVBT concurrently. Because there are now more then one transaction updating the TMVBT index, all the updating transactions have to perform latch-coupling while traversing the TMVBT index, because the active pages may be modified by a concurrent instance of the maintenance transaction.

Another approach for multithreading the maintenance transaction is to allow step 4 of the maintenance transaction for a version v_1 to run concurrently with step 2 for the next version v_2. This means that the updates of version v_2 can be applied into the TMVBT at the same time as the pending updates of version v_1 are deleted from the VBT by another thread, or group of threads.

6.5 Summary

We have now described the concurrent MVBT (CMVBT), a general-purpose multiversion database structure that can be used concurrently by multiple updating transactions. The CMVBT is optimal when querying for stable versions, provided that the correct root page of the TMVBT index is known (see Section 5.7). When querying for transient versions, the separate VBT index must also be searched. However, because the VBT is expected to remain in main memory during normal transaction processing, these queries should not require any additional disk I/O operations.

The update actions of user transactions are efficient, because the pending updates are inserted into the small VBT index. This improves the latency of transactions, but does not increase the overall system throughput, because the maintenance transaction still needs to apply the pending updates into the main TMVBT index. As we will show in the next chapter, the overall performance of the CMVBT is on par with the TSB-tree of Lomet and Salzberg [54–59], indicating that the main-memory-resident VBT index does not affect the performance of CMVBT significantly.

CHAPTER 7

Evaluation of CMVBT and TSB-tree

We have now described our multiversion index structure, the concurrent multiversion B$^+$-tree (CMVBT), in the previous chapter. In this chapter we show that the CMVBT performs well under general transaction processing. For this purpose, we have implemented a software library for running tests on database index structures. The relevant implementation details are described in Section 7.1. We have analyzed the performance of the CMVBT experimentally and compared it to the time-split B$^+$-tree (TSB-tree) of Lomet and Salzberg [58, 59], which is used as the basis of Immortal DB [55–57]. The tests are described in Section 7.2, and the results of the tests are given in Sections 7.3 and 7.4. We have also evaluated the performance of the TMVBT in itself, so as to measure the effect of having a separate VBT index for storing the updates of active transactions. In addition, we have run the tests on the VBT index on its own to demonstrate that, when a single-version index structure is used to store multiversion data, the range queries are not efficient. This was discussed in Section 3.2. The test results for these indexes are shown in Section 7.5. We summarize the findings from the tests in Section 7.6.

7.1 Implementation of the Index Structures

Our TreeLib database index software library is implemented using the Java language. We have implemented several common database index structures and a test runner that can be used to generate and execute query workloads that are indexed in the implemented index structures. The pages of the index structures are stored onto disk in a single file using standard Java file I/O routines, which are sufficient for evaluating the performance differences of the various index structures. Page identifiers mark the position (address) of the database page inside the file, so that a page p is located at byte positions $[pB_b, (p+1)B_b)$, where B_b is the size of database pages, in bytes. This is a common design that is presented in database textbooks [32]. The first page of each block of $8B_b$ pages is a page allocation bitmap that tells which pages are currently in use. Each page-allocation bitmap b contains $8B_b$ bits which are set so that the ith bit in b is one if and only if the page $b + i$ is currently in use.

The software includes a working page buffer with a least-recently-used (LRU) page-replacement policy. The page buffer is based on a hash table that is interleaved with free/used page lists, and has an expected $\mathcal{O}(1)$ access time for querying, inserting, or removing a page in the buffer. Because the structure is based on a hash table, the theoretical worst case complexity of the operations is $\mathcal{O}(n)$. All the index structure algorithms in TreeLib fix pages to the page buffer, but page latching is not implemented. The database software can therefore be used by only one client at a time. Similarly, we have not yet implemented any key-level locking operations for transactional concurrency control, nor does the software perform any logging, which means that recovery is not possible. None of these omissions should affect the accuracy of our experiments, however, because we are not evaluating the performance of the concurrency-control or recovery algorithms.

We have not tried to optimize the absolute performance of the various components of the TreeLib software; rather, we have focused on implementing the structures correctly and on measuring the number of page accesses required. The Immortal DB articles [54, 55], for example, explain how the entries should be compressed in the leaf pages. Graefe and Larson [31] suggest different techniques for optimizing the cache sensitivity of the index structure, including using microscopic B-tree structures inside single database pages. Techniques such as these should indeed be applied when building a commercial database system. We have omitted these optimizations from our software, because the index structures we are comparing are very similar in structure, and any optimization that can be applied to one could most probably be applied to the others as well. We have measured how many pages each operation needs to fix to the page buffer (buffer fixes), how many pages are actually read from disk (page reads), and how many pages need to be written back onto disk (page writes). In addition, we have measured the real time taken by the different operations. Because all the index structures to be compared have been implemented in the same software library, and all without final optimizations, the comparisons should be fair.

For our tests, we have implemented the VBT (Section 3.2), TMVBT (Section 5), CMVBT (Section 6), and TSB-tree (Section 4.2) index structures. When the VBT is used as a multiversion index structure on its own, we call it the *multiversion versioned B⁺-tree*, or MV-VBT. In all these structures, the data associated with each entry is a four-byte integer assumed to be a pointer to the data indexed by the structure. The indexes in our tests are therefore dense. The TMVBT, TSB-tree, and the TMVBT index used in the CMVBT all have identical page formats for index pages. The MV-VBT and the VBT index used in CMVBT both have a more compact index-page entry format, because they are B^+-trees.

The MVBT article [8] suggests that data items be stored as tuples of the form (k, v_1, v_2, w), where v_1 denotes the creation time of the data item, and v_2 the deletion time (see Definition 2.1 on page 11). The version range $[v_1, v_2)$ thus gives the life span of the data item. This format is however not well suited for the TSB-tree index, because the TSB-tree also needs to store entries with temporary transaction identifiers. Instead, we use the format suggested in Section 3.2 (tuples of the form (k, v_1, w) for insertion and $(k, v_2, \perp)$ for deletion) in the TSB-tree. For fairness of comparison, we use this leaf-page entry format in all the index structures for storing the leaf-page entries.

A summary of the page formats in the different index structures is given in Table 7.1. The TSB-tree can store slightly fewer entries in its leaf pages because it has to store additional information to separate the entries that been lazily timestamped from the entries that still have transaction identifiers instead of commit-time versions.

	MV-VBT	**TMVBT**	**TSB-tree**
Page size	4096 B	4096 B	4096 B
Index entry size	12 B	20 B	20 B
Index page capacity (entries)	338	203	203
Leaf entry size	12 B	12 B	12 B
Leaf page capacity (entries)	338	338	335

Table 7.1. Page formats for different index structures. The TMVBT and VBT indexes used in the CMVBT database have page formats that are identical to the TMVBT and MV-VBT indexes described in this table, respectively.

With the CMVBT structure, we store the TMVBT index on disk storage. As described in Section 6.1, the VBT is designed to be a main-memory-based structure, and thus does not need to be backed onto disk. In our implementation, we have dedicated a part of the page buffer for the VBT, big enough to guarantee that the VBT resides entirely in the buffer in all our tests. For fairness of comparison, the combined size of the VBT page buffer and the TMVBT page buffer is the same as the buffer size used for the other indexes. In practice, we have used a total buffer size of 200 pages for all the database indexes. The CMVBT splits the capacity into 32 pages for the VBT, and 168 pages for the TMVBT index. This buffer size is large enough so that it is not totally unrealistic, but small enough so that pages will have to be flushed back onto disk during transaction processing. Kollios and Tsotras have used a page buffer size of 10 pages [45], and van den Bercken and Seeger have varied the buffer size from 0 to 200 pages in their experiments [10]. Additionally, we have implemented the *root** search structure as a B$^+$-tree, and stored it onto the disk. The *root** is used to locate the roots of historical versions in the TMVBT (see Definition 3.4).

In the CMVBT tests, all our page buffer measurements (buffer fixes, page reads and writes) include operations performed on the TMVBT index, the VBT index and the *root** search structure. The operation cost of the maintenance transaction is also included in our tests. In practice, if the maintenance transaction is run immediately after transaction T commits, we include the cost of the maintenance transaction into the cost of the last action performed by transaction T. At each invocation, the maintenance transaction is run as many times as is required to move the updates of all committed transactions into to TMVBT index. This is fair when discussing the average action costs, but it also means that the standard deviations of the actions are strongly biased, because the test data now contains actions that seem to take hundreds or even thousands of buffer fixes and page reads. We have left the standard deviations out as they do not convey any meaningful information for the CMVBT index.

The version condition variables of the TMVBT [8, 35] are set to the values given in Section 5.3, so that the absolute minimum number of live entries in the pages (both leaf and index pages) is min-live = $1/5B$, and the split tolerance is min-live = $1/5B$, where B is the page capacity. This means that after any page-split operation, all pages involved in the SMO, except the parent page, must contain from $2/5B$ to $4/5B$ live entries. At other times, the pages can contain from $1/5B$ to B live entries; otherwise a structure-modification operation is triggered. The details of the structure-modification operations are described in Section 5.5.

The frequency of the maintenance transaction directly affects the CMVBT performance, because the VBT index is not an optimal structure for storing multiversion data. It is thus important to keep the VBT as small as possible by running the maintenance transaction often, preferably whenever an updating transaction has committed. To measure the effects of the maintenance transaction, we have run our tests with different settings for the frequency of the maintenance transaction.

For the TSB-tree index, we have implemented the algorithms based on published literature [54–59]. The implemented structure is based on the structure described in the initial TSB-tree articles [58, 59] and the updates described in the Immortal DB articles [55–57]. We have implemented three variants of the TSB-tree: one with the splitting rules based on the deferred split policy [57] (denoted TSB-D), one with the WOB-tree split policy [56] (denoted TSB-W), and a third one with the isolated-key-split (IKS) policy from the TSB-tree performance evaluation article [59] (denoted TSB-I).

We use a data-page key-splitting threshold of 0.67 for the deferred and WOB-tree split policies of the TSB-tree (TSB-D and TSB-W), because our implementation does not do any version compression (see the Immortal DB article [56] for details). For index pages in all variants, we find a split time at which historical index terms can migrate to a historical node without any current index terms ending up there as well. These policies are explained in more detail in the aforementioned articles. We have also implemented the persistent timestamp table and lazy timestamping [54, 55]. These are required in order to change the temporary transaction identifiers to commit-time versions after a transaction has committed. The persistent timestamp table (PTT, see Section 4.3) is stored onto disk as a standard B^+-tree, and the page operations required for its maintenance are included in the TSB-tree measurements. The TSB-D implementation includes the batch updating of the PTT [57]. In practice, the PTT is only updated when the database is shut down; that is, once for each test run.

Finally, the CMVBT structure contains the commit-time-to-identifier (CTI) table, and the TSB-tree has a volatile timestamp table (VTT) which is used as a cache for the entries of the PTT. Both of these structures are temporary, and do not need to be backed onto disk or logged, because they can be recreated after a system crash. The temporary structures are implemented as main-memory Java collections, and operations on these structures cause no disk or page buffer accesses.

7.2 Test Workloads

We have run two types of tests to analyze the performance of the index structures: single-key query-update tests and range-query tests. The query-update tests contain both reads and updates, and the range-query tests consist of range-queries targeting different versions of the database. For the query-update tests, we have generated an *initial* database state that contains a million live entries, created in such a way that the database history also contains some deletions. For the range-query tests, we have created new states by logically deleting the live data items of the initial state, so that each deleted state has successively fewer live entries left. At the last state, all the data items have been deleted. The states are explained in more detail below.

The tests themselves consist of pregenerated workloads that are run sequentially. For the query-update tests, we have created two sets of workloads, one with shorter transactions and another one with longer transactions. The workloads for shorter transactions consist of 2000 transactions with five actions each, and the workloads for longer transactions consist of 100 transactions with 100 actions each. All workloads therefore contain 10 000 actions. For each workload, we set a probability for any transaction to be an updating transaction. The workload with an update probability of 0 % thus contains only read-only transactions; and the workload with update probability of 100 % contains only pure updating transactions without queries. These tests were run on the *initial* state of the database, and the database state was restored after each test.

Other authors have used varying transaction lengths in their database experiments; for example, di Sanzo et al. use transactions that consist of an expected 20 actions each [79], Kumar et al. have used a single action per transaction [49], Tzouramanis et al. have used 300 updates per transaction [91], Silva and Nascimento have used from 100 to 5000 updates [83], whereas Lomet et al. have used up to 32 000 updates in a single transaction [55]. Our experiments are designed to show the differences of the index structures under normal transaction processing. For extremely long-running transactions, it may be better to stop the maintenance transaction, and to apply a single long-running transaction directly into the TMVBT.

Because the total number of transactions in a single workload is not very high, we did not select the transaction types purely at random, but rather forced the number of updating transactions in the workload to exactly match the selected updating transaction probability. The workload with 20 % updating transactions therefore contains exactly 20 % updating transactions and 80 % read-only transactions. The read-only transactions consist of single-key queries, and the pure updating transactions are either inserting transactions or deleting transactions, with 50 % percentage each (this selection was purely random).

All keys in the workloads have been selected with a uniformly random distribution from a predetermined range of $[0, 2\,000\,000\,000)$. For key queries and deletions, we have first selected a random key and located the nearest-matching key actually present in the database at the queried version during the generation of the workload. The key located in this manner has then been stored in the workload, and used in the actual tests. According to Ailamaki et al. [1], results from simple tests such as these have been found out to be substantially similar to results obtained from full-blown TPC-D workloads. We are thus confident that tests generated in this way can be used to find out interesting properties of the efficiency of the database. Binder et al. [15, 16] also use randomized workloads that are processed sequentially.

The *initial* state was created by an updating workload that consists of 100 000 transactions. Each of these transactions contains either 20 inserts or 20 deletes, which leads to a total of 2 000 000 actions for the entire index creation workload. The probability of an inserting transaction is 75 % for the initial database creation workload. The generated database thus contains 1 000 000 live entries, the database history has 100 000 versions, and there are some deletions present in the history. This size should be enough to observe the differences in the performance of the various index structures. Van den Bercken and Seeger have used a database created with 100 000 single-action transactions for their experiments [10], while Binder et al., di Sanzo et al., and Tzouramanis et al. have used an initial database that stores 10 000 entries [15, 79, 91].

The deleted states *del-i* were created by deleting the live data items. The number i of a state *del-i* denotes the total percentage of live data items deleted at that state. We deleted the data in ten steps, creating a state from each of the steps. Each deletion workload consists of 10 000 transactions that delete ten random data items each. The *del*-0 state is thus identical to the *initial* state, the *del*-50 state has 50 % of the live entries of the *initial* state still alive, and the logical database at the *del*-100 state contains no live entries.

The initial, the half deleted, and the fully deleted states are summarized in Table 7.2. The table shows the number of entries in the index structure and the number of pages used. As can be seen, the MV-VBT index is clearly the most compact structure, because it never creates duplicate copies of entries to other pages. The TMVBT index requires about the same number of pages regardless of whether it is used on its own, or as a part of the CMVBT index, which is not surprising. Because of the more frequent page splits, the CMVBT requires from 10 % to 60 % more database pages than the TSB-tree. The TSB-tree variant that uses the

del-0	**Live entries**	**Total entries**	**All pages**	**Leaf pages**	**H**	**C.time (min)**
CMVBT	1 000 000	4 407 606	16 089	15 870	3	31.4
TMVBT	1 000 000	4 420 773	16 170	15 961	3	33.4
TSB-D	1 000 000	3 396 196	13 104	12 793	4	42.6
TSB-W	1 000 000	3 404 358	14 176	13 894	4	41.2
TSB-I	1 000 000	2 502 629	10 609	10 858	4	40.7
MV-VBT	1 000 000	2 000 000	8407	8372	3	29.0

del-50	**Live entries**	**Total entries**	**All pages**	**Leaf pages**	**H**	**C.time (min)**
CMVBT	500 000	5 077 804	19 724	19 463	3	13.6
TMVBT	500 000	5 074 473	19 714	19 458	3	13.6
TSB-D	500 000	3 746 547	16 438	16 070	4	15.3
TSB-W	500 000	3 715 423	16 334	16 005	4	14.8
TSB-I	500 000	2 834 108	13 387	13 072	4	14.2
MV-VBT	500 000	2 500 000	10 485	10 441	3	10.6

del-100	**Live entries**	**Total entries**	**All pages**	**Leaf pages**	**H**	**C.time (min)**
CMVBT	0	6 021 509	25 026	24 689	0	9.5
TMVBT	0	6 034 161	25 076	24 740	0	9.7
TSB-D	0	3 844 617	18 169	18 578	4	13.3
TSB-W	0	3 820 102	18 587	18 195	4	13.2
TSB-I	0	2 958 763	16 207	15 827	4	12.5
MV-VBT	0	3 000 000	13 280	13 213	3	10.6

Table 7.2. Summary of database states. The column H tells the height of the current-version search tree $S_{v_{commit}}$, and the last column tells the creation time of the database states, in minutes, as measured from the previous reported state. TSB-D denotes the TSB-tree variant with the deferred split policy, TSB-W denotes the variant with WOB-tree split policy, and TSB-I the variant with IKS split policy.

IKS split policy [59] is clearly the smallest, requiring only about 25 % more pages than the MV-VBT. Remember from Sections 4.2 and 5.5 that the asymptotic space complexity of all the index structures is the same; that is, $\mathcal{O}(n/B)$, where n is the number of updates performed on the index, and B is the page capacity. We have not presented a direct proof for the space complexity of the VBT (MV-VBT) index, but because the VBT is a B^+-tree that always stores each update as a single entry in its leaf pages, it has the same space complexity as the MVBT [7, 8], and thus also of the TMVBT and TSB-tree.

Table 7.2 also shows the creation times of the database states, measured in minutes. The creation time of the state *del*-50 includes all the changes applied starting from the initial state *del*-0, and the creation time of the state *del*-100 includes all the further changes applied starting from the state *del*-50. Remember that no bulk loading was applied, and all the database states were created by repeated single-key update actions; that is, insertions and deletions. Unsurprisingly, the initial state of the MV-VBT was fastest to create, because the MV-VBT is a B^+-tree. Note however that it was faster to create the last state *del*-100 of the CMVBT index than the last state of the MV-VBT, because the current-version search tree of the CMVBT had shrunk in size. It seems that overall the creation of the CMVBT index is slightly faster than the creation of the TSB-tree variants, if the CMVBT maintenance transaction is allowed to run often. In the creation of the states, the maintenance transaction was run after each transaction had committed.

The computer system on which the tests were run is described in Table 7.3. The details of the computer system naturally do not affect the buffer fixes and file I/O operation counts, only the real time taken by the tests.

7.3 General Transaction Processing

We begin our experiments by comparing the CMVBT index with the TSB-tree variants in general transaction processing. Summaries of the query-update tests are shown in Figures 7.1, 7.2, and 7.3. The figures show two graphs each, one run with the short-transaction workloads and another with the long-transaction workloads. The graphs show buffer fixes, page reads and the real time required for each test. The x-axis in the graphs represents the percentage of updating transactions in the workload. The percentages were set to 0 %, 10 %, 20 %, …, 100 %. For actual numeric results, refer to Tables A.2 and A.3 in Appendix A.

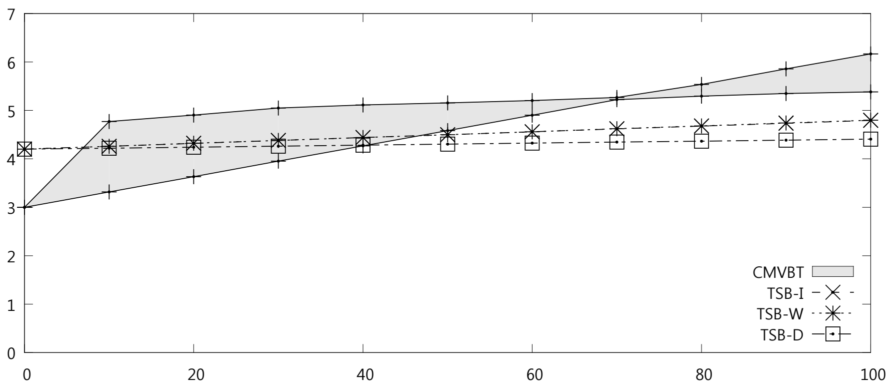
Hardware	
Processor	Intel Core 2 Quad Q6600, 2.40 GHz
Main Memory	4 GB DIMM 667 MHz
Hard Disk	Hitachi HDS721616PLA380
Software	
Operating System	Linux, Fedora Core 8
	Kernel 2.6.26.8
Java Version	1.6.0_04

Table 7.3. Test computer system.

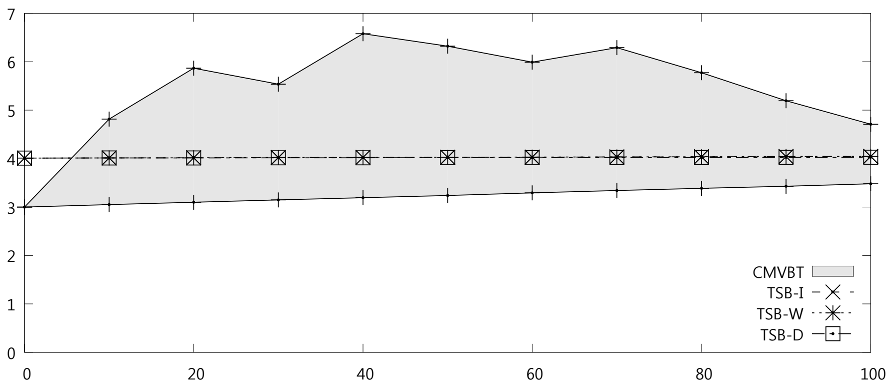
As explained in Section 7.1, we believe that the interval of the maintenance transaction affects the the performance of the CMVBT structure. We have therefore run the tests with varying settings for the maintenance interval; namely, the value $m \in \{1, 5, 10, 50\}$. A value m indicates that we allow m transactions to commit before running the maintenance transaction. When the maintenance transaction is run, it is run m times to apply the updates of all the committed transactions into the TMVBT index. In the figures presented here, for clarity, we show the range of the values obtained by running the CMVBT tests with different maintenance intervals. The absolute values for the different settings of m are shown in Tables A.2 and A.3, and in a summary figure (Figure 7.8).

As can be seen from the graphs of Figure 7.1, with longer transactions all variants of the TSB-tree access an almost constant number of four pages per action regardless of the amount of updates. This number comes almost directly from the height of the TSB-tree index, with occasional structure-modification operations and timestamp maintenance bringing the average a bit above four. With shorter transactions, increasing the percentage of updating transactions increases the number of buffer fixes required per action. This trend is caused by a global database-information page which needs to be updated in both structures when an updating transaction begins and commits. Because this happens only once per transaction, the effect is not seen with longer transactions.

There is greater variance in the number of buffer fixes for the CMVBT index, depending on the frequency of the maintenance transaction, both with shorter and longer transactions. When transactions are long, the CMVBT performs best when the maintenance transaction is run immediately after each transaction commits. When transactions are short and over 70 % of transactions are updating transactions, the CMVBT is more



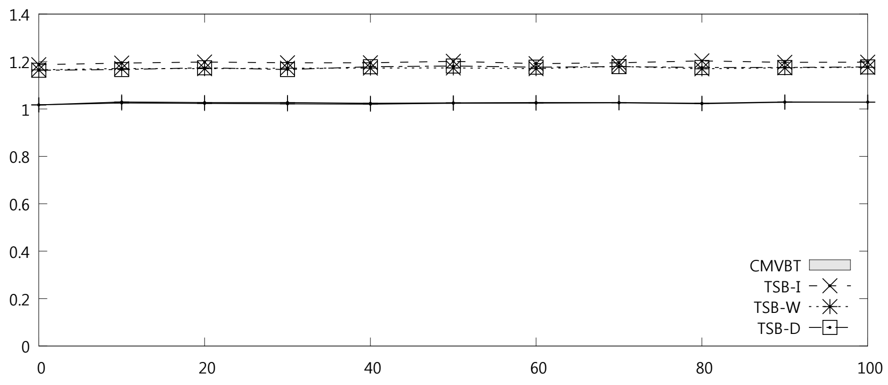
(a) Five actions per transaction



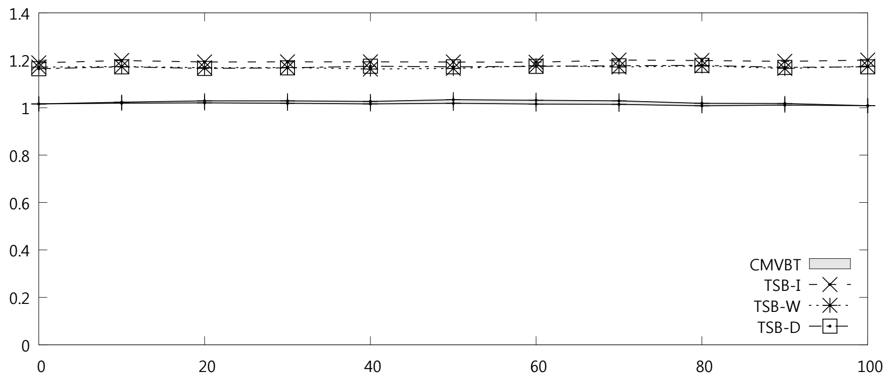
(b) 100 actions per transaction

Figure 7.1. Number of buffer fixes for queries and updates. The x-axis shows the percentage of updating transactions in the workload. Note that the results for all TSB-tree variants are almost identical, and thus the lines are drawn on top of each other.

efficient when the maintenance transaction is run after a few transactions have committed. In this situation the structure benefits from batch updating, because the VBT clusters the updates together. The same approach is used to increase performance in the differential indexes of Pollari-Malmi et al. [72]. However, the CMVBT only benefits from less frequently run maintenance transactions in this specific case. Our general recommendation is to run the maintenance transaction as often as possible, with best results achieved in the majority of different transactional workloads if the maintenance transaction is run immediately after an updating transaction has committed.



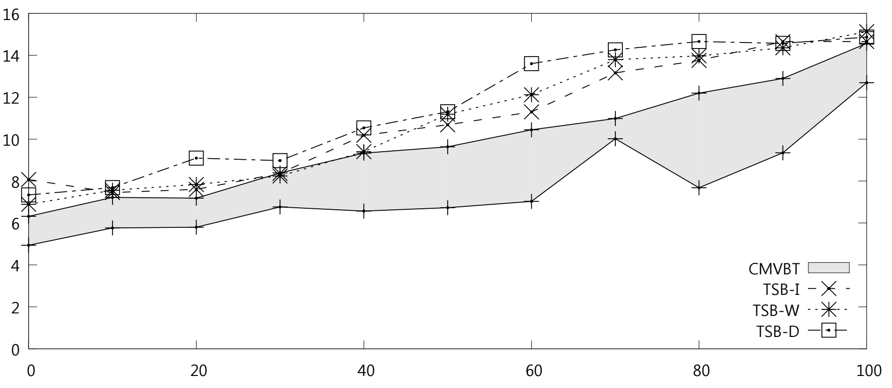
(a) Five actions per transaction



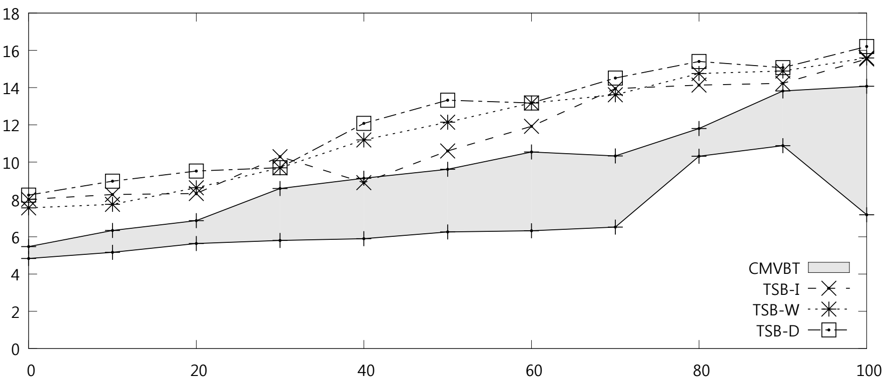
(b) 100 actions per transaction

Figure 7.2. Number of page reads for queries and updates.

In database performance evaluation, the disk I/O operations are the critical operations that should be minimized, as discussed in Section 3.1. Our definition of the cost of an action (Definition 3.2) is based on the number of page accesses, because this is a natural way of determining how large portions of the database the operation must access. In a database system, the database pages are accessed through a page buffer that caches the page data. If a page is used often, it is kept in the page buffer, and consecutive page accesses do not cause disk I/O operations to occur. The number of page accesses (that is, buffer fixes) may thus be significantly different from the number of actual page read operations required for an action. We have measured the number of times a page was read from the disk to the page buffer (page reads), and the results are shown in Figure 7.2.



(a) Five actions per transaction



(b) 100 actions per transaction

Figure 7.3. Real time taken by queries and updates. The x-axis shows the percentage of updating transactions, and the time is shown in seconds.

As expected, the TSB-tree again requires an almost constant number of page reads per operation. Somewhat surprisingly, the interval of the maintenance transaction does not seem to affect the number of pages read per action for the CMVBT, with this number constantly being lower than the corresponding one for the TSB-tree. This can be explained by the fact that a large enough portion of the page buffer is reserved for the VBT index, and the VBT pages do not need to be re-read from disk. In our tests, the VBT required at most 20 pages when all the transactions were long updating transactions (100 updates) and the maintenance interval was very long (run every 50 transactions). In this situation, up to 5000 updates were stored in the VBT. On the average, the size of the VBT was only a few pages. Another interesting point seen from these figures is that the frequency of updating transactions does not seem to affect the number of pages read, even for the short-transaction workload. This means that the updating of the global database-information page does not affect the number of pages read, even though it clearly has an effect on the number of buffer fixes required. However, this only shows that the information page is kept in the buffer because it is used often and thus does not need to be re-read into the page buffer.

Finally, the real time taken by the tests is shown in Figure 7.3. The figures are shown here for completeness—we should not infer too much from them, because there are many implementation details that can affect the run-time of the tree structure algorithms. On a general level, we can see that none of the compared index structure variants is clearly better than the others.

The purpose of the query-update tests was to show that the CMVBT performs on par with the TSB-tree in general transaction processing. We have also ran the query-update tests on the *del*-50 database state (where half the indexed data items have been logically deleted), and the results were exactly as expected—the buffer fixes for each operation were practically identical, but all the tested indexes required slightly more page fixes because the indexes had accumulated more historical entries. All the graph shapes were the same, and thus the second test only confirms the findings from the first test. The numerical values obtained from these tests are shown in Tables A.4 and A.5 in Appendix A.

7.4 Range Queries

Our next tests show that the CMVBT structure benefits from the fact that the underlying TMVBT structure merges pages and thus causes the search trees of later versions to shrink. To verify this, we have run range-query tests at the various *del-i* states, querying for the most recent version v_{commit} at each state. Summaries from these tests are shown in Figures 7.4, 7.5, and 7.6; and the actual numerical results are shown in Table A.6 in Appendix A.

The figures show the buffer fixes, page reads and the real time taken by range queries of the most recent version v_{commit} of the database, performed at the different database states. The reported values are averages for 1000 transactions, each consisting of a single range query. Van den Bercken and Seeger have also used transaction workloads consisting of 1000 range queries in their experiments [10]. In our range-query tests, the starting point of each range was randomly selected from the entire

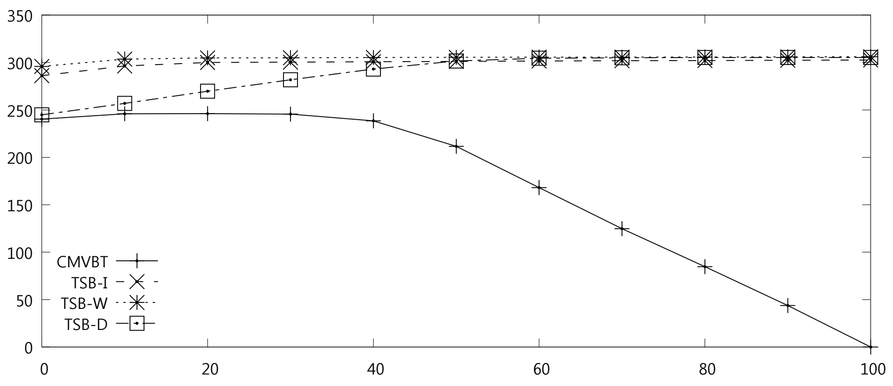


Figure 7.4. Buffer fixes for current-version key-range queries. The x-axis represents the percentage of entries that have been deleted from the initial state. Queried range size is 5 % of the entire key-space size.

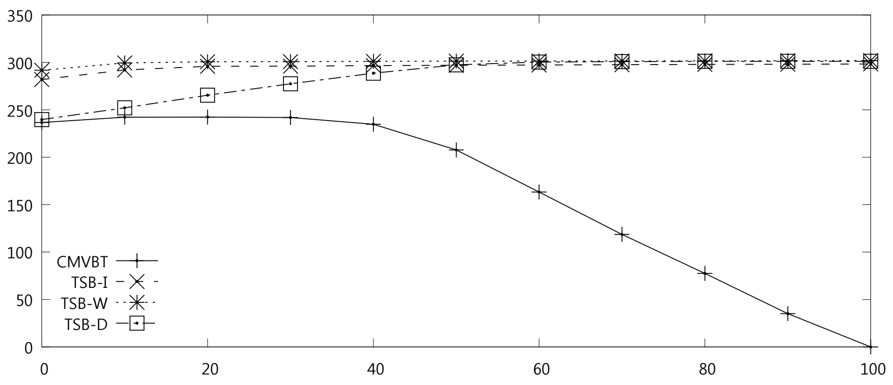


Figure 7.5. Page reads for current-version key-range queries.

key space that was used to populate the index, and the range size was set to 5 % of the key-space size. In this test, the number of page reads per operation is close to the number of buffer fixes, because the ranges fill up a large portion of the page buffer and there is very little page reuse between different range queries. The *del*-0 state (that is, the *initial* state) in the figures shows a baseline value for the range query efficiency. Based on the baseline value and on an overview of the graphs, the CMVBT is slightly more efficient with range queries in general, except for the TSB-D variant. This can be explained by the different splitting policies used in the TSB-tree and the TMVBT.

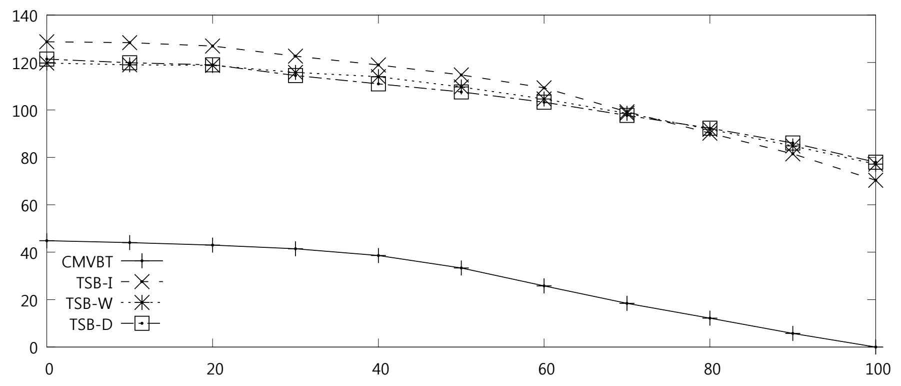


Figure 7.6. Real time taken by current-version key-range queries. The time is measured in seconds.

The optimality of the TMVBT index can be seen when more items are deleted. Because the TMVBT index merges pages, the current-version search tree contains fewer pages and thus the searches become faster. When all the data items have been deleted, the current-version search tree is empty, and the TMVBT does not have to access any pages (the page identifier of the root of the current-version search tree is cached in our TMVBT implementation). None of the TSB-tree variants benefit from deletions, because pages are not merged and page key ranges never grow. This trend is seen clearly from both the buffer fixes and the page reads.

Even though the number of pages the TSB-tree needs to process does not decrease, the run-time of the TSB-tree does decrease in the *del-i* states, as seen from Figure 7.6. This is partly an implementation issue, and partly it may be caused by the lazy timestamping of the TSB-tree. Whenever processing a page in the TSB-tree, we have to check whether there are any committed entries that still have temporary identifiers, and to change those that are found. The number of page writes (shown in Table A.6 in Appendix A) verifies that the TSB-tree needs to write back some pages, because some entries in the pages have been lazily timestamped during the range queries.

7.5 MV-VBT and TMVBT

When running the query-update tests, we also ran the tests by using the versioned B$^+$-tree as a multiversion index (MV-VBT) on its own. The results for the MV-VBT, shown in Tables A.2–A.6, seem to suggest that the MV-VBT index is surprisingly efficient in general transaction processing. The explanation to this is that the query-update tests all target single keys, which are ordered and optimally indexed by the MV-VBT. The MV-VBT index is more compact and the structure-modification operations are simpler, targeting at most three pages at a time. The problematic actions with MV-VBT are range queries, as demonstrated by Figure 7.7. The figure shows the number of buffer fixes required for processing the range queries at the various states of the database. It is clear from the figure that the MV-VBT index is not efficient for processing range queries.

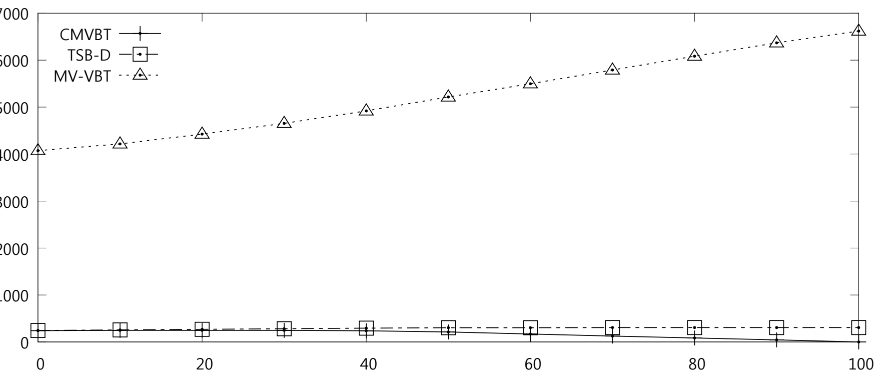
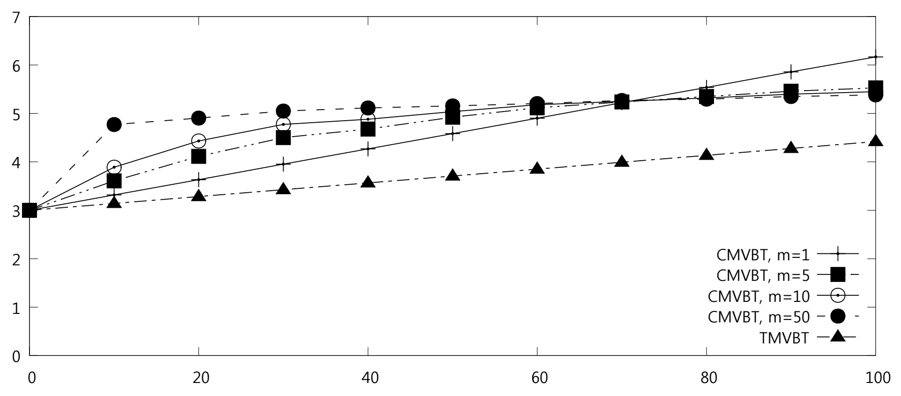


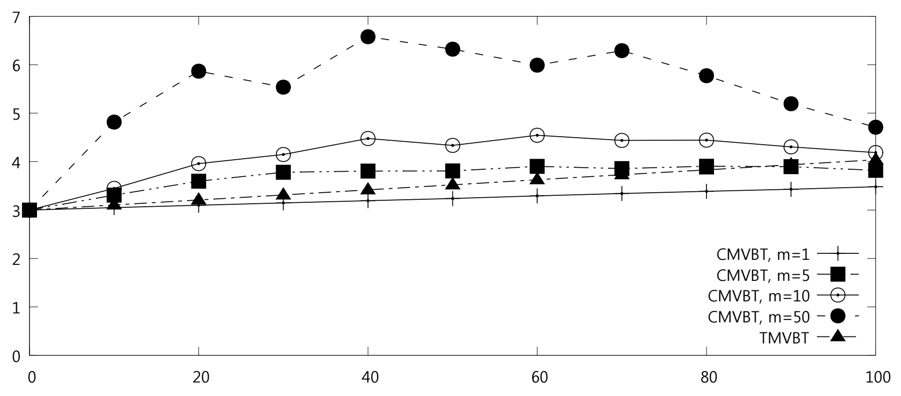
Figure 7.7. Number of buffer fixes for range queries. The x-axis represents the percentage of entries that have been deleted from the initial state. Queried range size is 5 % of the entire key-space size.

Our CMVBT index is composed of a TMVBT index and a VBT index, as described in Chapter 6. We have shown that the CMVBT benefits from the separate VBT index with long transactions, because the updates are clustered in the memory-resident VBT pages and applied as a batch operation to the TMVBT. We now compare the TMVBT index in itself with the combined CMVBT index. We show the number of buffer fixes required for the query-update tests by the TMVBT index in Figure 7.8, alongside with the results for the CMVBT. In the tests, the interval of the CMVBT maintenance transaction was again varied.

Figure 7.8 shows that the TMVBT clearly requires fewer buffer fixes than the CMVBT with shorter transactions, regardless of the interval of the CMVBT maintenance transaction. However, the page reads and the real time taken by the tests are practically the same, as can be verified from the result tables in Appendix A. This means that the actual disk I/O



(a) Five actions per transaction



(b) 100 actions per transaction

Figure 7.8. Number of buffer fixes for queries and updates. The x-axis represents the percentage of updating transactions in the workload. In the legend, m denotes the maintenance interval (maintenance transaction run after m transactions).

operations required by the TMVBT and the CMVBT are almost the same, and thus the separate VBT index does not incur any serious performance loss. In fact, as shown by the longer transactions, the CMVBT requires fewer page fixes than the TMVBT with long updating transactions, if the maintenance transaction is run frequently.

7.6 Summary

We have shown in this chapter that the CMVBT index performs on par with the TSB-tree index in general transaction processing, and outperforms the TBS-tree in range queries in the presence of logical data-item deletions. We have further confirmed that the single-version B$^+$-tree index in itself is not suitable for general multiversion transaction processing, because the range-query operation is not efficient, and we have shown that the separate VBT tree used in the combined CMVBT index does not degrade the overall performance of the CMVBT significantly in any of the tested situations. The main TMVBT index structure remains optimal in the presence of any user transactions. We thus recommend the CMVBT structure for general transaction processing, especially when deletions are frequent, and when the current-version range query performance is critical.

The optimality of the TMVBT index does mean that the index size is larger than the size of the other index structures. As shown in Table 7.2, the TMVBT index in the CMVBT may take up to 10 % to 60 % more space when compared to the TSB-tree. If space usage is the most critical concern, the TSB-tree is a better option. On the other hand, if range queries are never needed, the versioned B$^+$-tree packs the data even tighter, and performs well with single-key queries.

We have not run any tests to analyze the performance of aborting transactions. We can, however, show that aborting transactions are more efficient in the CMVBT than in the TSB-tree. This is because in the CMVBT the pending updates that have to be removed when a transaction aborts are clustered in the small VBT index, whereas in the TSB-tree we have to search through the main index to undo the actions of the aborted transaction. More specifically, suppose that a committed transaction in the CMVBT index needs to access c_c pages, and the same transaction in the TSB-tree needs to access c_t pages. As our tests have shown, $c_c \sim c_t$ in most cases. Now, let us denote by a_c and a_t the number of page accesses required when the transaction aborts and rolls back, instead of committing. In the TSB-tree, $a_t > c_t$, because the updates of the transaction have already been applied to the index structure, and have to be physically undone. In fact, a_t can be almost twice as large as c_t if all the updates of the transaction have been applied to different leaf pages. In contrast, in the CMVBT, $a_c < c_c$, because the number of page accesses for a committing transaction include the deletion of the updates of the transaction from the VBT index by the maintenance transaction. In practice, for an aborting transaction, the maintenance transaction does not have to perform the first scan of the VBT, and the TMVBT does not need to be accessed at all. We can thus conclude that aborting transactions are by nature more efficient when the pending updates created by active transactions are stored in a separate, small, main-memory-resident index structure.

The test data in our workloads has been generated by randomly selecting keys with a uniform distribution. We have also tested whether a different distribution would cause the results to differ by running separate tests on a data set that was generated by selecting random numbers with a Gaussian distribution that is clustered around the middle of the key space. The data set thus contains many entries near the middle key, and almost none near the endpoints. We used uniform distribution to create the range-query and query-update workloads, however. The results from these tests are shown in Appendix B. As the tables show, the relative performance of the indexes was very close to the relative results of the test with uniform distribution (Appendix A). The most notable difference between the different random distributions is that the range query tests were less efficient overall (requiring up to 50 % more time) with the Gaussian distribution, and the single-key queries and updates required about 40 % fewer page reads. The efficiency of queries and updates can be explained by the queries that targeted keys near the endpoints of the key range; because there are few entries near the endpoints, there is more page reuse between actions. The range queries took more time because some queries near the middle of the key range had to process significant portions of the database (at most about 20 % of the live entries instead of 5 %). Most importantly, however, the relative performances of the compared database indexes remained the same, and the Gaussian tests thus confirmed the results of the original tests.

CHAPTER 8

Conclusions

A current trend in database systems is that the history of the database contents must be accessible, in addition to the current state of the data set. The traditional single-version B$^+$-tree index can be straightforwardly extended so that multiversion data may be indexed by it, but the versioned extension is not efficient, as was shown in Chapter 3. The problem with this approach is the range-query action, which is inefficient because the data items with the same key but with different versions are clustered close to each other, while the data items with the same version but with different keys are not. Efficient indexing of the data set evolution therefore requires a multiversion index structure.

When querying for any fixed version v, an optimal multiversion index structure should be as efficient as a single-version index structure that only indexes the data items that are alive at version v. We have defined this as the requirement for optimality of multiversion indexes (Definition 3.3 in Section 3.3). This guarantees that range queries remain efficient even if the database accumulates a long history of updates. When data items are logically deleted, the range queries that target the latest committed version should become more efficient as fewer data items are alive. Range queries are an important operation in a general-purpose database system because index scans and joins are based on them.

We have reviewed three of the most efficient multiversion index structures in Chapter 4. These are the TSB-tree of Lomet and Salzberg [58, 59], the multiversion B$^+$-tree (MVBT) of Becker et al. [7, 8], and the multiversion access structure (MVAS) of Varman and Verma [92]. From these structures, only the MVBT is considered optimal by our definition of optimality. The problem with MVBT and MVAS is that they follow a single-update model, in which the update cannot be rolled back, and therefore these indexes cannot be used as general database indexes in a transactional multi-user environment. On the other hand, the TSB-tree does not have this restriction, but it does not guarantee any optimal bounds for the range-query performance, either. In particular, in the presence of logical deletions, the performance of the TSB-tree degrades because the leaf pages of the index structure are not merged.

As an initial step, we have introduced transactions to the MVBT by redesigning it, as described in Chapter 5. The redesigned transactional MVBT (TMVBT) index retains the optimal bounds of the MVBT and allows one updating multi-action transaction to operate concurrently with multiple read-only transactions. The TMVBT index is an efficient index structure that is usable on its own in situations where there is only a single source of updates, such as in data stream management systems.

In Chapter 6 we presented the design of our concurrent multiversion B$^+$-tree (CMVBT) index which uses a separate main-memory-resident versioned B$^+$-tree (VBT) index to store the pending updates created by active transactions, and a TMVBT index for indexing the data items inserted by committed transactions. Once an active transaction T has committed, a system maintenance transaction is run to apply the updates of T from the VBT index into the main TMVBT index. We say that a version v is stable, if all the updates of the transaction T that created the version v have been applied to the TMVBT index. The CMVBT index is thus optimal when querying for the data items of stable versions, and guarantees that the performance of the queries never degrades, even in the presence of deletions. The separate VBT index is kept small by constantly moving the updates of committed transactions into the TMVBT index. The VBT can be kept entirely in main memory during general transaction processing, and it does not incur any additional I/O operations.

Our CMVBT algorithms are designed to work in a multi-user environment with multiple concurrent updating transactions. We allow transactions to roll back; either entirely, or up to a preset savepoint. Standard concurrency-control algorithms can be used to maintain logical data consistency. The snapshot isolation algorithms [11] are especially well suited for use with our multiversion index structure, and they guarantee snapshot isolation for all transactions. Our algorithms are made recoverable by the ARIES recovery algorithm [64, 66], and we apply structure-modification operations on a level-by-level basis, performing each SMO as an atomic action that transforms a balanced index into another balanced index [39–41].

Because the commit-time version of a data item is not known when the transaction that created it is active, the data-item updates must initially be tagged with transaction identifiers, which are later changed to commit-time versions. The CMVBT index organization allows the data-item versions to be efficiently changed from transaction identifiers into commit-time versions when the maintenance transaction moves the updates from the VBT into the TMVBT index. This is a non-trivial issue that often requires special book-keeping arrangements in other multiversion index structures that support commit-time versioning.

We have experimentally analyzed the performance of the CMVBT index in Chapter 7 and compared it to the performance of the TSB-tree. The results we obtained from our experiments agree with what we expected from our analytical results. The CMVBT index structure performs on par with the TSB-tree index in standard transaction processing, but is more efficient for key-range queries. The efficiency of the key-range queries is especially apparent if the history of the database contains deletions. We have furthermore compared the combined CMVBT index to the TMVBT, and conclude that the separate VBT index does not affect the overall performance significantly. For completeness, we have also demonstrated that the performance of range queries degrades rapidly if the multiversion data items are indexed by a single-version B^+-tree index.

There is a downside to the optimal performance of the TMVBT index; namely, that the index structure requires more space than the TSB-tree. While the asymptotic space complexity of all the compared index structures is the same, the size of the TMVBT (in itself, and as part of the CMVBT index) was 10 % to 60 % greater than the size of the TSB-tree in our tests, depending on the TSB-tree splitting policy and on the number of deletions in the database history. Our conclusion is thus that the CMVBT structure is a good choice for a general-purpose multiversion index when performance is more important than storage space, especially so when it is expected that the history will also contain key deletions. The TSB-tree is a better choice when storage space is limited, particularly if historical data is to be stored on a tertiary storage, because the TSB-tree allows historical pages to be moved during time-splits.

Bibliography

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, 1999.

[2] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 24th International Conference on Data Engineering*, pages 576–585, 2008.

[3] American National Standard for Information Systems. Database Language—SQL, 1992. ANSI X3. 135-1992.

[4] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[5] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. On optimal multiversion access structures. In *Proceedings of the 3rd International Symposium on Advances in Spatial Databases*, pages 123–141, 1993.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[9] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[10] J. van den Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 168–179, 1996.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[12] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[14] M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. The consensus glossary of temporal database concepts: February 1998 version. *Temporal Databases: Research and Practice*, pages 367–405, 1998.

[15] W. Binder, S. Spycher, I. Constantinescu, and B. Faltings. An evaluation of multiversion concurrency control for web service directories. In *Proceedings of the 2007 IEEE International Conference on Web Services*, pages 35–42, 2007.

[16] W. Binder, S. Spycher, I. Constantinescu, and B. Faltings. Multiversion concurrency control for multidimensional index structures. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 172–181, 2007.

[17] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[18] J. A. Bubenko. The temporal dimension in information modelling. In *Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 93–118, 1977.

[19] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 729–738, 2008.

[20] P. Cederqvist et al. Version management with CVS. `http://ximbiot.com/cvs/manual/`, 2008. Referenced 10th February 2010.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[22] B. Collins-Sussman, C. M. Pilato, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2008. `http://svnbook.red-bean.com/`. Referenced 10th February 2010.

[23] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[24] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[25] M. C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30(3):230–241, 1986.

[26] R. Elmasri, Y. J. Kim, and G. T. J. Wuu. Efficient implementation techniques for the time index. In *Proceedings of the 7th International Conference on Data Engineering*, pages 102–111, 1991.

[27] R. Elmasri, G. T. J. Wuu, and Y. J. Kim. The time index: an access structure for temporal data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 1–12, 1990.

[28] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[29] R. Fendt. DVCS round-up: One system to rule them all? Part 3. The Linux Foundation, `http://ldn.linuxfoundation.org/article/dvcs-round-one-system-rule-them-all-part-3`, 2009. Referenced 10th February 2010.

[30] Git community. Git user's manual. `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`, 2010. Referenced 10th February 2010.

[31] G. Graefe and P.-Å. Larson. B-tree indexes and CPU caches. In *Proceedings of the 17th International Conference on Data Engineering*, pages 349–358, 2001.

[32] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[33] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[34] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[35] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen. Transactions on the multiversion B$^+$-tree. In *Proceedings of the 12th International Conference on Extending Database Technology*, pages 1064–1075, 2009.

[36] T. Haapasalo, I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for multiversion database structures. In *Proceedings of the 2nd PhD Workshop on Information and Knowledge Management*, pages 73–80, 2008.

[37] T. Haapasalo, I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrent updating transactions on versioned data. In *Proceedings of the 2009 International Database Engineering and Applications Symposium*, pages 77–87, 2009.

[38] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[39] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Recoverable B$^+$-trees in centralized database management systems. *International Journal of Applied Science and Computations*, 10:160–181, 2003.

[40] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees. *The VLDB Journal*, 14(2):257–277, 2005.

[41] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. B-tree concurrency control and recovery in page-server database systems. *ACM Transactions on Database Systems*, 31(1):82–132, 2006.

[42] C. Jensen and D. Lomet. Transaction timestamping in (temporal) databases. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 441–450, 2001.

[43] K. Jouini and G. Jomier. Indexing multiversion databases. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 915–918, 2007.

[44] P. Kanellakis, S. Ramaswamy, and D. Vengroff. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996.

[45] G. Kollios and V. J. Tsotras. Hashing methods for temporal data. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):902–919, 2002.

[46] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *Proceedings of the 5th International Conference on Data Engineering*, pages 127–137, 1989.

[47] C. P. Kolovson and M. Stonebraker. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. *ACM SIGMOD Record*, 20(2):138–147, 1991.

[48] V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed. The time index$^+$: an incremental access structure for temporal databases. In *Proceedings of the 3rd ACM Conference on Information and Knowledge Management*, pages 296–303, 1994.

[49] A. Kumar, V. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1, 1998.

[50] G. M. Landau, J. P. Schmidt, and V. J. Tsotras. Historical queries along multiple lines of time evolution. *The VLDB Journal*, 4(4):703–726, 1995.

[51] S. Lanka and E. Mays. Fully persistent B$^+$-trees. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1991.

[52] P. L. Lehman and B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[53] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 212–223, 1980.

[54] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 939–941, 2005.

[55] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *Proceedings of the 22nd International Conference on Data Engineering*, page 35, 2006.

[56] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. *Proceedings of the VLDB Endowment*, 1(1):870–881, 2008.

[57] D. Lomet and F. Li. Improving transaction-time DBMS performance and functionality. In *Proceedings of the 25th International Conference on Data Engineering*, pages 581–591, 2009.

[58] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 315–324, 1989.

[59] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 353–363, 1990.

[60] D. Lomet and B. Salzberg. Access method concurrency with recovery. *ACM SIGMOD Record*, 21(2):351–360, 1992.

[61] D. Lomet and B. Salzberg. Concurrency and recovery for index trees. *The VLDB Journal*, 6(3):224–240, 1997.

[62] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill. Designing DBMS support for the temporal dimension. *ACM SIGMOD Record*, 14(2):115–130, 1984.

[63] C. Mohan. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 392–405, 1990.

[64] C. Mohan. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 371–380, 1992.

[65] C. Mohan. Repeating history beyond ARIES. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 1–17, 1999.

[66] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[67] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 361–370, 1992.

[68] Oracle. Oracle Database Concepts 11*g* Release 1 (11.1), 2008. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/toc.htm`. Referenced 24th April 2009.

[69] M. H. Overmars. Searching in the past I. Technical Report RUU-CS-81-7, University of Utrecht, Utrecht, The Netherlands, 1981.

[70] M. H. Overmars. Searching in the past II: general transforms. Technical Report RUU-CS-81-9, University of Utrecht, Utrecht, The Netherlands, 1981.

[71] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[72] K. Pollari-Malmi, J. Ruuth, and E. Soisalon-Soininen. Concurrency control for B-trees with differential indices. In *Proceedings of the 2000 International Database Engineering and Applications Symposium*, pages 287–296, 2000.

[73] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.

[74] PostgreSQL Global Development Group. PostgreSQL 8.4.2 documentation, 2010. `http://www.postgresql.org/docs/8.4/`. Referenced 4th February 2010.

[75] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 1987. MIT-LCS-TR-205.

[76] B. Salzberg, L. Jiang, D. Lomet, M. Barrena, J. Shan, and E. Kanoulas. A framework for access methods for versioned data. In *Proceedings of the 9th International Conference on Extending Database Technology*, pages 730–747, 2004.

[77] B. Salzberg and D. Lomet. Branched and temporal index structures. Technical Report NU-CCC-95-17, Northeastern University, Boston, MA, 2005.

[78] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[79] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10, 2008.

[80] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[81] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 696–707, 2004.

[82] A. Shoshani and K. Kawagoe. Temporal data management. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 79–88, 1986.

[83] J. Silva and M. Nascimento. An incremental batch-oriented index for bitemporal databases. In *Proceedings of the 7th International*

Workshop on Temporal Representation and Reasoning, pages 133–141, 2000.

[84] S. Sippu and E. Soisalon-Soininen. A theory of transactions on recoverable search trees. In *Proceedings of the 8th International Conference on Database Theory*, pages 83–98, 2001.

[85] R. T. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 236–246, 1985.

[86] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[87] E. Soisalon-Soininen and P. Widmayer. Concurrency and recovery in full-text indexing. In *Proceedings of the 1999 String Processing and Information Retrieval Symposium and International Workshop on Groupware*, pages 192–198, 1999.

[88] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, 1987.

[89] V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass. An extensible notation for spatiotemporal index queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.

[90] V. J. Tsotras and N. Kangelaris. The snapshot index: an I/O-optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, 1995.

[91] T. Tzouramanis, Y. Manolopoulos, and N. Lorentzos. Overlapping B$^+$-trees: an implementation of a transaction time access method. *Data and Knowledge Engineering*, 29(3):381–404, 1999.

[92] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[93] T. Virtanen. Git for computer scientists. `http://eagain.net/articles/git-for-computer-scientists/`, 2009. Referenced 21st October 2009.

[94] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

APPENDIX A

Test Results for Uniform Data Set

The numerical results obtained from the test sets generated with a uniform random distribution are listed in this appendix. The names of the multiversion index structures are explained in Table A.1.

Name	Explanation
CMVBT	Concurrent multiversion B$^+$-tree (Chapter 6)
CMVBT-i	Concurrent multiversion B$^+$-tree, with the maintenance transaction run after each ith transaction has committed (Section 7.1)
TMVBT	Transactional multiversion B$^+$-tree (Chapter 5)
TSB-D	TSB-tree with the deferred split policy (Sections 4.2 and 7.1)
TSB-W	TSB-tree with the WOB-tree split policy (Sections 4.2 and 7.1)
TSB-I	TSB-tree with the isolated-key-split policy (Sections 4.2 and 7.1)
MV-VBT	Versioned B$^+$-tree used as a multiversion index (Section 3.2)

Table A.1. Names of the database index structures.

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	1.02	0.00	6.3
CMVBT-5	3.00	1.02	0.00	4.9
CMVBT-10	3.00	1.02	0.00	5.0
CMVBT-50	3.00	1.02	0.00	5.1
TMVBT	3.00	0.98	0.00	5.5
TSB-D	4.20	1.16	0.00	7.3
TSB-W	4.20	1.17	0.00	6.9
TSB-I	4.20	1.19	0.00	8.1
MV-VBT	3.00	0.99	0.00	3.5

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	4.58	1.03	0.59	9.6
CMVBT-5	4.92	1.03	0.53	9.5
CMVBT-10	5.04	1.02	0.51	6.7
CMVBT-50	5.16	1.03	0.49	7.2
TMVBT	3.71	0.99	0.48	10.0
TSB-D	4.30	1.18	0.49	11.3
TSB-W	4.50	1.17	0.49	11.2
TSB-I	4.50	1.20	0.49	10.7
MV-VBT	3.10	0.99	0.48	3.9

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	6.17	1.03	1.18	13.4
CMVBT-5	5.53	1.03	1.02	12.8
CMVBT-10	5.45	1.03	1.00	12.7
CMVBT-50	5.38	1.03	0.98	14.6
TMVBT	4.42	0.99	0.96	12.8
TSB-D	4.41	1.18	0.98	14.9
TSB-W	4.80	1.18	0.98	15.1
TSB-I	4.80	1.20	0.97	14.7
MV-VBT	3.20	0.99	0.97	4.4

Table A.2. Queries and updates, initial state, five actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	1.02	0.00	5.5
CMVBT-5	3.00	1.02	0.00	4.9
CMVBT-10	3.00	1.02	0.00	4.8
CMVBT-50	3.00	1.02	0.00	5.1
TMVBT	3.00	0.98	0.00	5.2
TSB-D	4.01	1.16	0.00	8.2
TSB-W	4.01	1.17	0.00	7.5
TSB-I	4.01	1.19	0.00	8.0
MV-VBT	3.00	0.99	0.00	3.3

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.24	1.03	0.49	6.3
CMVBT-5	3.81	1.03	0.49	9.4
CMVBT-10	4.34	1.02	0.48	9.6
CMVBT-50	6.32	1.02	0.48	8.7
TMVBT	3.52	0.99	0.48	10.1
TSB-D	4.02	1.17	0.48	13.3
TSB-W	4.03	1.17	0.48	12.1
TSB-I	4.03	1.19	0.48	10.6
MV-VBT	3.01	0.99	0.48	3.7

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.48	1.01	0.98	7.2
CMVBT-5	3.82	1.01	0.98	7.2
CMVBT-10	4.18	1.01	0.98	11.4
CMVBT-50	4.71	1.01	0.97	14.1
TMVBT	4.04	0.99	0.96	13.1
TSB-D	4.03	1.17	0.97	16.2
TSB-W	4.05	1.18	0.98	15.6
TSB-I	4.04	1.20	0.97	15.5
MV-VBT	3.01	0.99	0.97	4.3

Table A.3. Queries and updates, initial state, 100 actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	1.04	0.00	4.9
CMVBT-5	3.00	1.04	0.00	4.6
CMVBT-10	3.00	1.04	0.00	4.7
CMVBT-50	3.00	1.04	0.00	4.7
TMVBT	3.00	0.99	0.00	4.8
TSB-D	4.20	1.25	0.00	6.7
TSB-W	4.20	1.22	0.00	6.2
TSB-I	4.20	1.28	0.00	6.4
MV-VBT	3.00	1.01	0.00	3.3

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	4.25	1.05	0.42	7.3
CMVBT-5	4.59	1.05	0.36	8.0
CMVBT-10	4.70	1.05	0.34	7.6
CMVBT-50	4.82	1.05	0.33	8.1
TMVBT	3.37	1.00	0.31	6.7
TSB-D	4.30	1.26	0.32	9.3
TSB-W	4.49	1.23	0.32	9.4
TSB-I	4.49	1.29	0.32	9.3
MV-VBT	3.10	1.01	0.49	4.1

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	5.48	1.05	0.85	6.8
CMVBT-5	4.84	1.05	0.69	7.1
CMVBT-10	4.76	1.05	0.67	7.0
CMVBT-50	4.70	1.05	0.65	10.5
TMVBT	3.73	1.00	0.64	6.7
TSB-D	4.40	1.25	0.64	13.6
TSB-W	4.78	1.23	0.65	13.2
TSB-I	4.78	1.29	0.64	12.6
MV-VBT	3.20	1.02	0.98	5.3

Table A.4. Queries and updates, 50 % deleted, five actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	1.03	0.00	4.7
CMVBT-5	3.00	1.03	0.00	4.6
CMVBT-10	3.00	1.03	0.00	4.5
CMVBT-50	3.00	1.03	0.00	4.6
TMVBT	3.00	0.99	0.00	4.7
TSB-D	4.01	1.25	0.00	6.9
TSB-W	4.01	1.22	0.00	6.8
TSB-I	4.01	1.28	0.00	7.0
MV-VBT	3.00	1.01	0.00	3.3

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	2.91	1.05	0.32	5.8
CMVBT-5	3.48	1.04	0.32	5.8
CMVBT-10	4.01	1.04	0.32	7.4
CMVBT-50	5.99	1.03	0.31	6.5
TMVBT	3.17	1.00	0.31	6.0
TSB-D	4.02	1.25	0.31	8.2
TSB-W	4.03	1.22	0.31	8.2
TSB-I	4.03	1.28	0.31	9.4
MV-VBT	3.01	1.02	0.49	4.2

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	2.82	1.02	0.64	8.8
CMVBT-5	3.16	1.02	0.64	8.4
CMVBT-10	3.52	1.02	0.64	6.7
CMVBT-50	4.05	1.02	0.64	10.1
TMVBT	3.34	1.00	0.63	6.6
TSB-D	4.02	1.25	0.63	13.1
TSB-W	4.04	1.22	0.63	12.5
TSB-I	4.04	1.29	0.63	12.8
MV-VBT	3.02	1.02	0.98	4.5

Table A.5. Queries and updates, 50 % deleted, 100 actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

del-0	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	240.27	236.69	0.00	44.8
TMVBT	242.47	237.46	0.00	46.0
TSB-D	244.76	239.75	0.94	121.5
TSB-W	295.93	291.69	0.96	119.8
TSB-I	286.19	281.90	0.95	128.7
MV-VBT	4072.29	4067.44	0.00	647.9

del-50	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	211.60	207.65	0.00	33.4
TMVBT	212.03	206.80	0.00	35.0
TSB-D	301.57	297.42	0.96	107.6
TSB-W	305.40	301.23	0.96	109.7
TSB-I	301.18	296.93	0.96	114.7
MV-VBT	5215.10	5212.04	0.00	762.7

del-100	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	0.00	0.00	0.00	0.0
TMVBT	0.00	0.00	0.00	0.0
TSB-D	305.37	301.18	0.96	77.9
TSB-W	306.03	301.86	0.96	77.1
TSB-I	302.60	298.35	0.96	70.3
MV-VBT	6617.91	6615.14	0.00	891.4

Table A.6. Current-version key-range queries. Range size is 5 % of the entire key-space size. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

APPENDIX B

Test Results for Gaussian Data Set

The numerical results obtained from the test sets generated with a Gaussian random distribution are listed in this appendix. The names of the multiversion index structures are explained in Table B.1.

Name	Explanation
CMVBT	Concurrent multiversion B$^+$-tree (Chapter 6)
CMVBT-i	Concurrent multiversion B$^+$-tree, with the maintenance transaction run after each ith transaction has committed (Section 7.1)
TMVBT	Transactional multiversion B$^+$-tree (Chapter 5)
TSB-D	TSB-tree with the deferred split policy (Sections 4.2 and 7.1)
TSB-W	TSB-tree with the WOB-tree split policy (Sections 4.2 and 7.1)
TSB-I	TSB-tree with the isolated-key-split policy (Sections 4.2 and 7.1)
MV-VBT	Versioned B$^+$-tree used as a multiversion index (Section 3.2)

Table B.1. Names of the database index structures.

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	0.58	0.00	6.4
CMVBT-5	3.00	0.58	0.00	4.4
CMVBT-10	3.00	0.58	0.00	4.8
CMVBT-50	3.00	0.58	0.00	4.5
TMVBT	3.00	0.56	0.00	4.8
TSB-D	4.20	0.66	0.00	6.1
TSB-W	4.20	0.70	0.00	5.7
TSB-I	3.20	0.66	0.00	6.4
MV-VBT	3.00	0.60	0.00	2.8

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	4.43	0.59	0.40	7.5
CMVBT-5	4.80	0.59	0.34	6.0
CMVBT-10	4.88	0.59	0.32	7.2
CMVBT-50	4.90	0.59	0.30	7.2
TMVBT	3.71	0.56	0.29	8.0
TSB-D	4.30	0.66	0.29	6.8
TSB-W	4.44	0.69	0.30	7.5
TSB-I	3.44	0.66	0.30	7.4
MV-VBT	3.10	0.60	0.31	2.9

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	5.85	0.61	0.76	9.2
CMVBT-5	5.21	0.61	0.60	9.2
CMVBT-10	5.13	0.61	0.58	8.9
CMVBT-50	5.07	0.61	0.57	10.2
TMVBT	4.42	0.58	0.54	9.5
TSB-D	4.41	0.68	0.55	10.3
TSB-W	4.67	0.71	0.57	10.6
TSB-I	3.67	0.68	0.56	7.2
MV-VBT	3.20	0.63	0.59	3.5

Table B.2. Queries and updates, initial state, five actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	0.58	0.00	4.7
CMVBT-5	3.00	0.58	0.00	4.8
CMVBT-10	3.00	0.58	0.00	4.4
CMVBT-50	3.00	0.58	0.00	4.5
TMVBT	3.00	0.55	0.00	4.6
TSB-D	4.01	0.66	0.00	6.6
TSB-W	4.01	0.69	0.00	5.9
TSB-I	3.01	0.65	0.00	6.0
MV-VBT	3.00	0.60	0.00	2.6

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.02	0.60	0.31	5.2
CMVBT-5	3.58	0.60	0.31	5.3
CMVBT-10	3.97	0.60	0.30	5.7
CMVBT-50	5.96	0.59	0.28	6.5
TMVBT	3.52	0.57	0.29	6.0
TSB-D	4.02	0.67	0.30	8.0
TSB-W	4.02	0.70	0.30	7.8
TSB-I	3.02	0.66	0.30	6.8
MV-VBT	3.01	0.61	0.31	3.1

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.01	0.58	0.55	5.5
CMVBT-5	3.35	0.58	0.55	5.6
CMVBT-10	3.72	0.58	0.55	6.1
CMVBT-50	4.23	0.58	0.54	8.9
TMVBT	4.04	0.56	0.52	9.4
TSB-D	4.03	0.67	0.55	10.0
TSB-W	4.04	0.71	0.56	8.8
TSB-I	3.04	0.67	0.55	9.5
MV-VBT	3.01	0.62	0.58	3.5

Table B.3. Queries and updates, initial state, 100 actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	0.58	0.00	4.2
CMVBT-5	3.00	0.58	0.00	3.9
CMVBT-10	3.00	0.58	0.00	4.0
CMVBT-50	3.00	0.58	0.00	3.9
TMVBT	3.00	0.55	0.00	4.5
TSB-D	4.20	0.71	0.00	5.2
TSB-W	4.20	0.72	0.00	5.1
TSB-I	4.20	0.70	0.00	5.7
MV-VBT	3.00	0.64	0.00	2.8

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	4.15	0.58	0.29	5.6
CMVBT-5	4.52	0.58	0.23	5.6
CMVBT-10	4.61	0.58	0.21	5.6
CMVBT-50	4.63	0.58	0.20	6.0
TMVBT	3.44	0.55	0.19	4.6
TSB-D	4.30	0.70	0.20	6.4
TSB-W	4.44	0.71	0.20	6.7
TSB-I	4.44	0.69	0.20	6.7
MV-VBT	3.10	0.63	0.32	3.1

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	5.34	0.59	0.57	7.5
CMVBT-5	4.70	0.59	0.41	7.6
CMVBT-10	4.62	0.59	0.39	6.3
CMVBT-50	4.56	0.59	0.38	8.6
TMVBT	3.91	0.56	0.36	7.3
TSB-D	4.40	0.71	0.39	8.8
TSB-W	4.67	0.72	0.39	8.9
TSB-I	4.67	0.71	0.39	7.1
MV-VBT	3.20	0.65	0.61	3.6

Table B.4. Queries and updates, 50 % deleted, five actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

0 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	3.00	0.58	0.00	3.8
CMVBT-5	3.00	0.58	0.00	4.0
CMVBT-10	3.00	0.58	0.00	3.9
CMVBT-50	3.00	0.58	0.00	3.9
TMVBT	3.00	0.55	0.00	4.0
TSB-D	4.01	0.70	0.00	5.5
TSB-W	4.01	0.71	0.00	5.7
TSB-I	4.01	0.69	0.00	5.7
MV-VBT	3.00	0.63	0.00	2.7

50 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	2.75	0.59	0.20	5.0
CMVBT-5	3.31	0.59	0.20	5.3
CMVBT-10	3.69	0.59	0.20	4.7
CMVBT-50	5.68	0.58	0.18	5.5
TMVBT	3.25	0.55	0.19	5.4
TSB-D	4.02	0.70	0.20	6.5
TSB-W	4.02	0.72	0.21	7.2
TSB-I	4.02	0.70	0.20	6.6
MV-VBT	3.01	0.64	0.32	3.2

100 % updates	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT-1	2.49	0.58	0.37	4.8
CMVBT-5	2.83	0.58	0.36	5.5
CMVBT-10	3.20	0.58	0.36	4.5
CMVBT-50	3.70	0.58	0.36	8.4
TMVBT	3.52	0.55	0.34	6.2
TSB-D	4.02	0.71	0.38	8.9
TSB-W	4.03	0.72	0.38	9.0
TSB-I	4.04	0.70	0.38	8.7
MV-VBT	3.01	0.65	0.60	4.3

Table B.5. Queries and updates, 50 % deleted, 100 actions/transaction. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).

del-0	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	260.50	257.25	0.00	86.5
TMVBT	259.02	254.84	0.00	82.9
TSB-D	258.12	253.17	0.37	168.2
TSB-W	314.65	310.20	0.39	168.6
TSB-I	299.05	294.96	0.38	177.0
MV-VBT	4360.49	4352.57	0.00	687.4

del-50	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	229.11	225.50	0.00	44.1
TMVBT	230.38	225.50	0.00	44.3
TSB-D	325.61	321.18	0.39	133.5
TSB-W	326.12	321.90	0.39	134.7
TSB-I	320.28	315.92	0.40	136.7
MV-VBT	5399.19	5391.06	0.00	800.0

del-100	*Fixes*	*Reads*	*Writes*	*Time(s)*
CMVBT	0.00	0.00	0.00	0.0
TMVBT	0.00	0.00	0.00	0.0
TSB-D	329.29	324.86	0.39	88.6
TSB-W	326.49	322.26	0.39	83.9
TSB-I	322.05	317.85	0.40	80.9
MV-VBT	7029.21	7021.64	0.00	912.9

Table B.6. Current-version key-range queries. Range size is 5 % of the entire key-space size. The table shows the average buffer fixes, page reads and page writes (per action), and the elapsed real time (for the entire test).