DATABASE INTERNALS

A DEEP DIVE INTO HOW DISTRIBUTED DATA SYSTEMS WORK, 1ST EDITION

BY ALEX PETROV

Contents

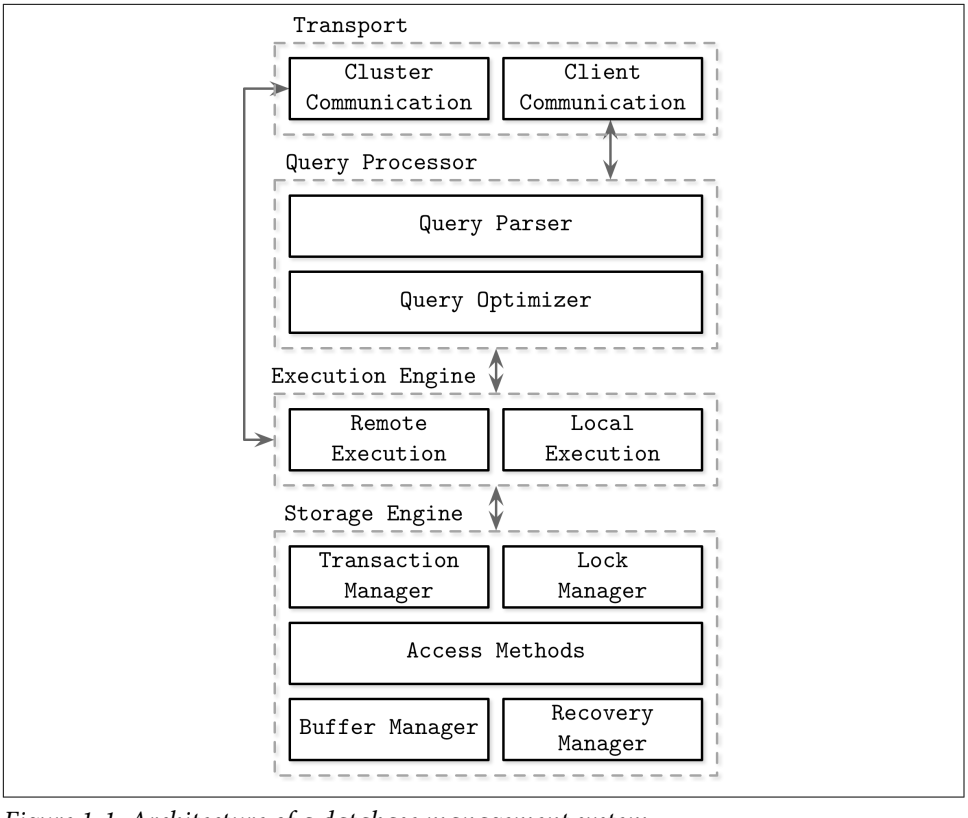

Figure 1-1. Architecture of a database management system

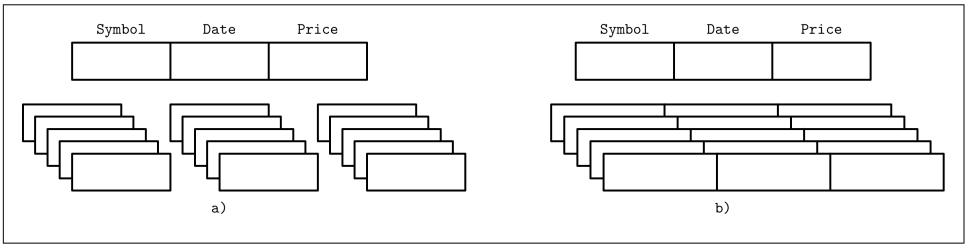

Figure 1-2. Data layout in column- and row-oriented stores

```
{
    "com.cnn.www": {
        contents: {
            t6: html: "<html>..."
            t5: html: "<html>..."
            t3: html: "<html>..."
        }
        anchor: {
            t9: cnnsi.com: "CNN"
            t8: my.look.ca: "CNN.com"
        }
    }
    "com.example.www": {
        contents: {
            t5: html: "<html>..."
        }
        anchor: {}
    }
}
```

Figure 1-3. Conceptual structure of a Webtable

Column Family: contents

Row Key	Timestamp	Qualifier	Value
com.cnn.www	t3	html	"<html>..."
com.cnn.www	t5	html	"<html>..."
com.cnn.www	t6	html	"<html>..."
com.example.www	t5	html	"<html>..."

Column Family: anchor

Row Key	Timestamp	Qualifier	Value
com.cnn.www	t8	cnnsi.com	"CNN"
com.cnn.www	t5	my.look.ca	"CNN.com"

Figure 1-4. Physical structure of a Webtable

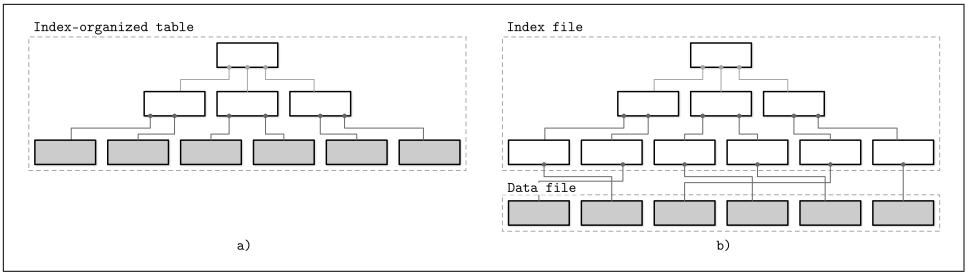

Figure 1-5. Storing data records in an index file versus storing offsets to the data file (index segments shown in white; segments holding data records shown in gray)

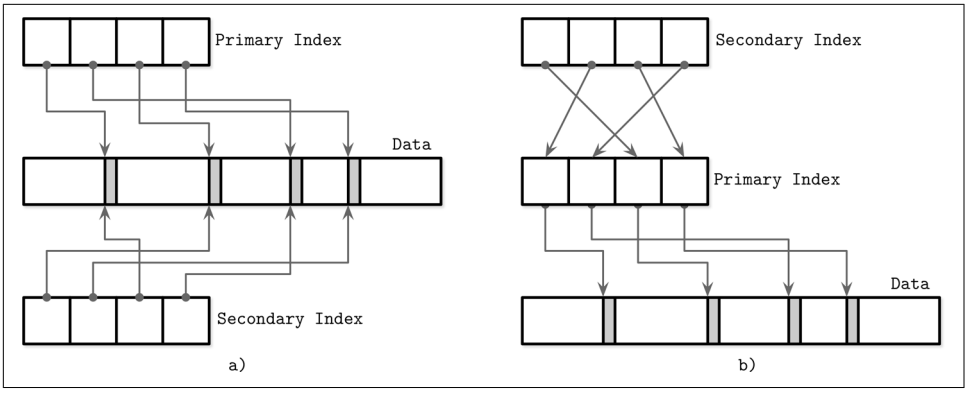

Figure 1-6. Referencing data tuples directly (a) versus using a primary index as indirection (b)

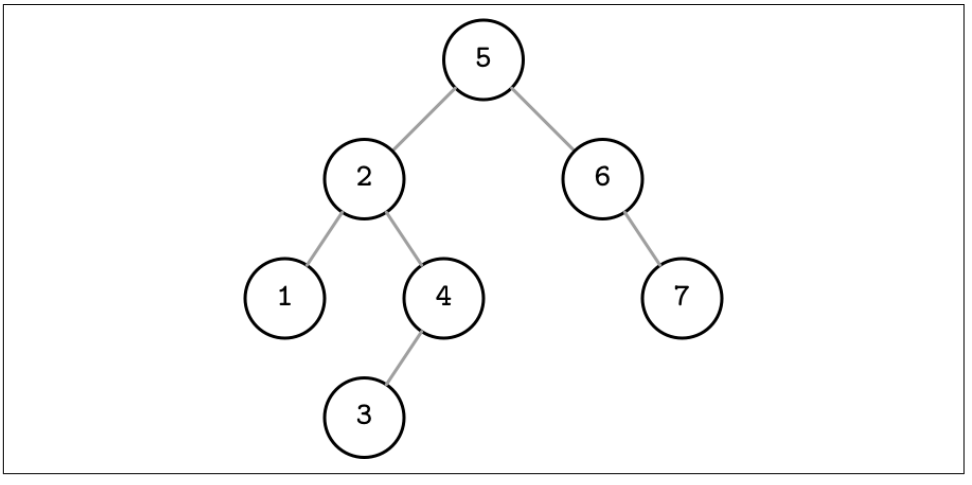

Figure 2-1. Binary search tree

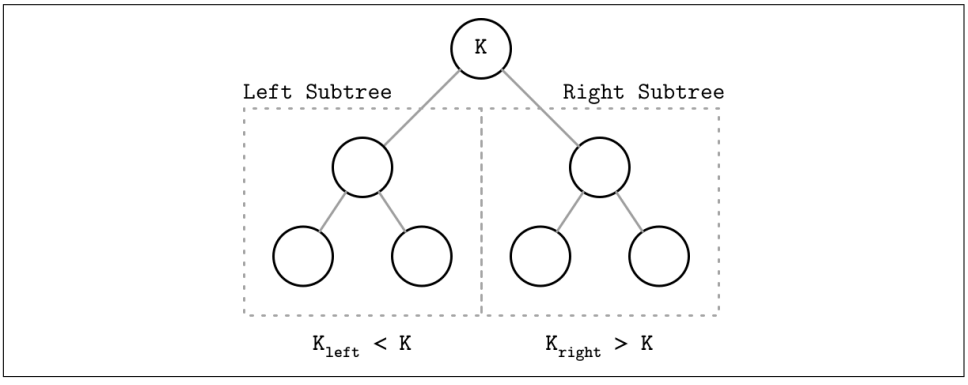

Figure 2-2. Binary tree node invariants

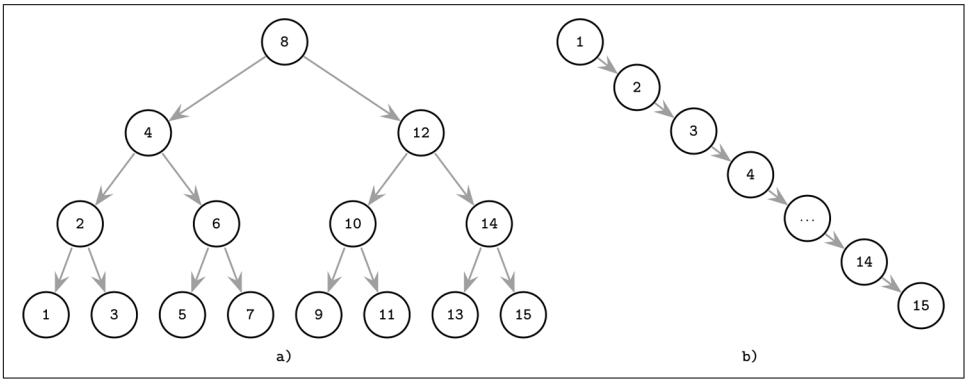

Figure 2-3. Balanced (a) and unbalanced or pathological (b) tree examples

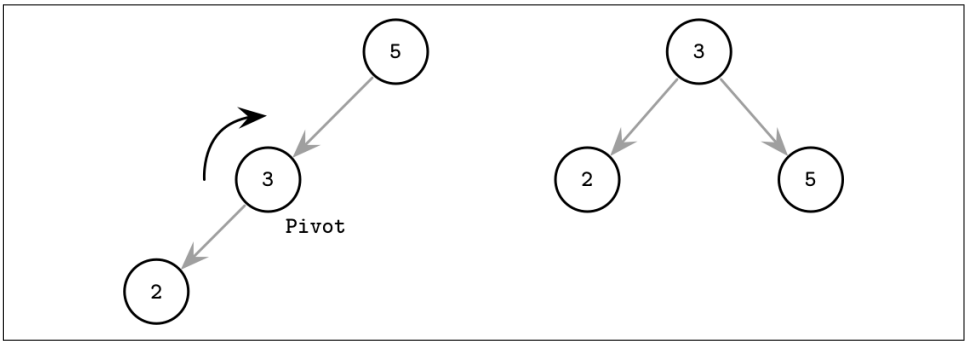

Figure 2-4. Rotation step example

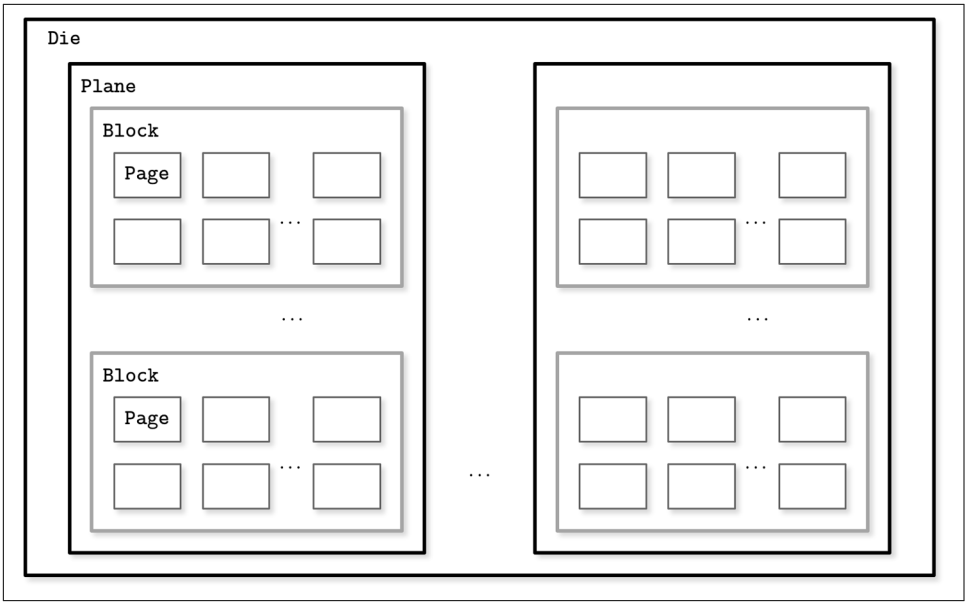

Figure 2-5. SSD organization schematics

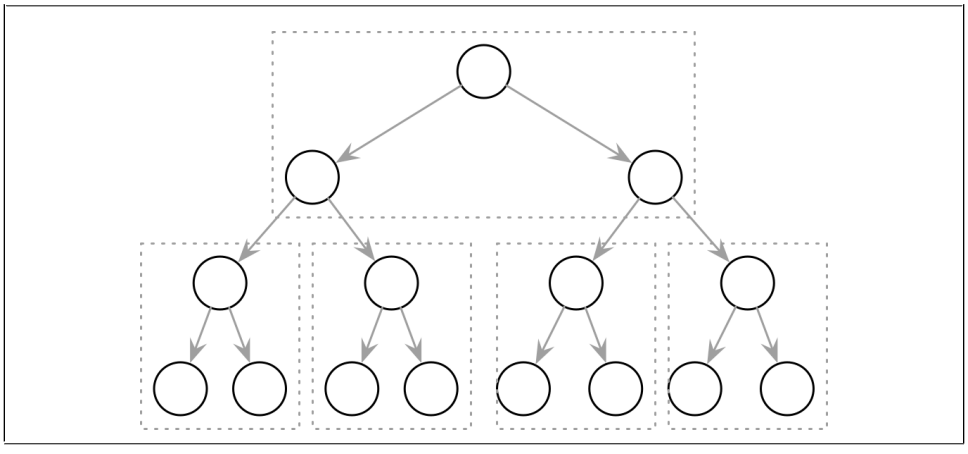

Figure 2-6. Paged binary trees

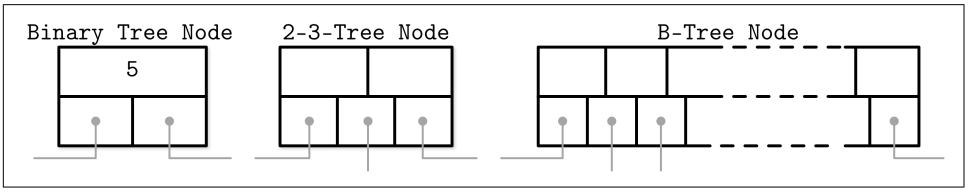

Figure 2-7. Binary tree, 2-3-Tree, and B-Tree nodes side by side

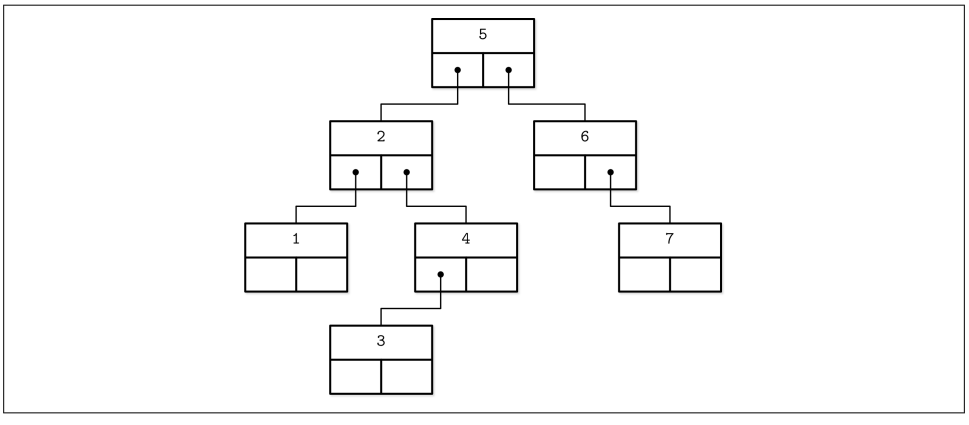

Figure 2-8. Alternative representation of a binary tree

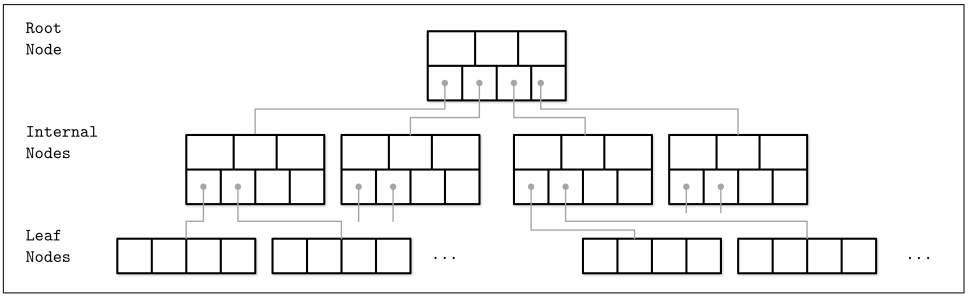

Figure 2-9. B-Tree node hierarchy

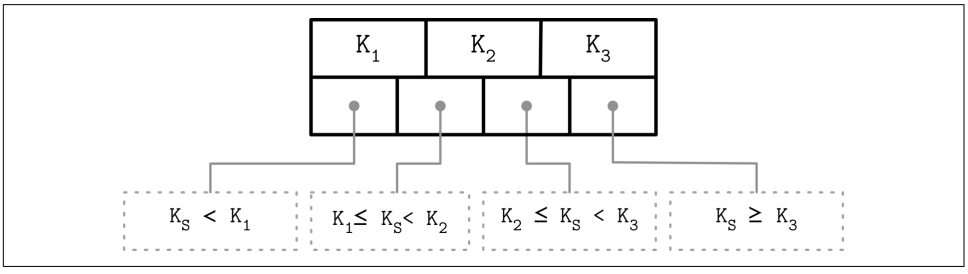

Figure 2-10. How separator keys split a tree into subtrees

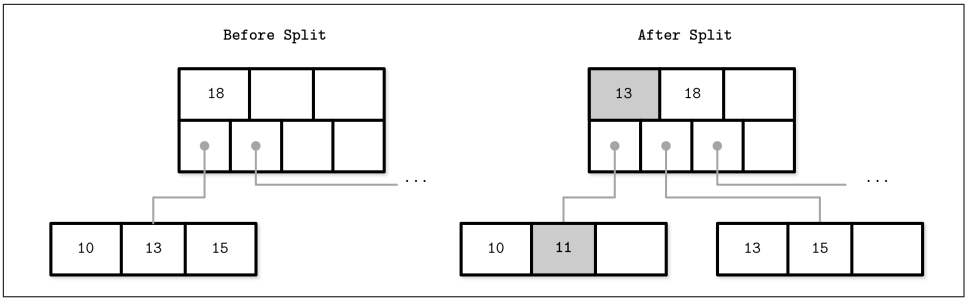

Figure 2-11. Leaf node split during the insertion of 11. New element and promoted key are shown in gray.

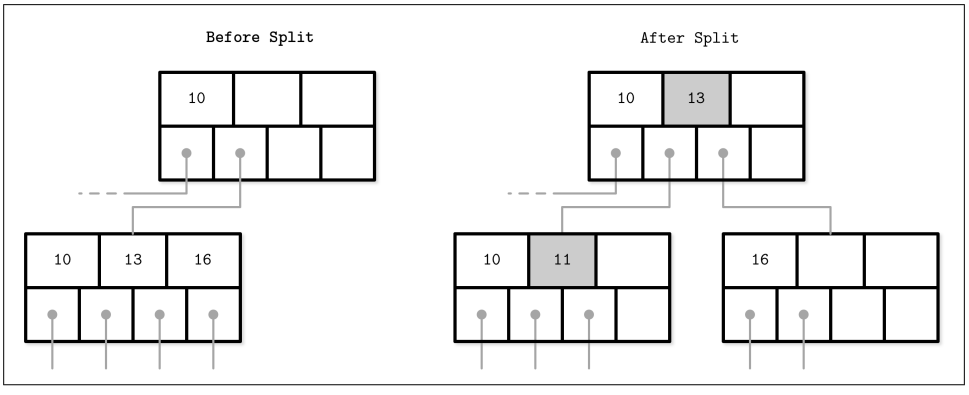

Figure 2-12. Nonleaf node split during the insertion of 11. New element and promoted key are shown in gray.

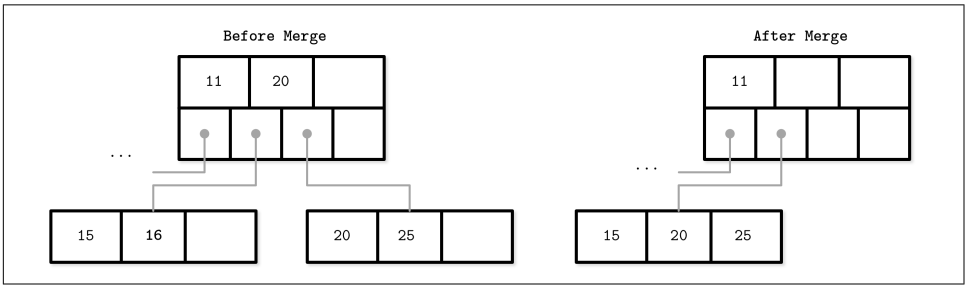

Figure 2-13. Leaf node merge

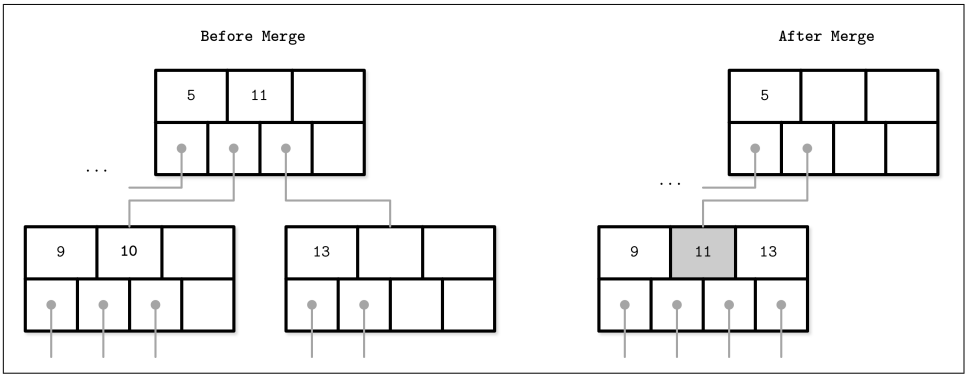

Figure 2-14. Nonleaf node merge

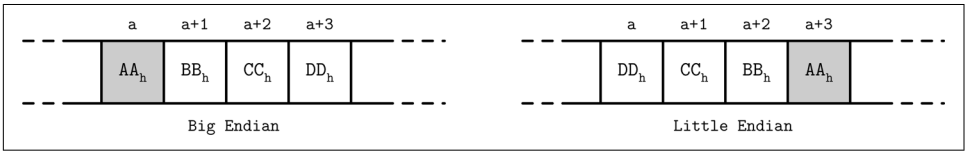

Figure 3-1. Big- and little-endian byte order. The most significant byte is shown in gray. Addresses, denoted by a, grow from left to right.

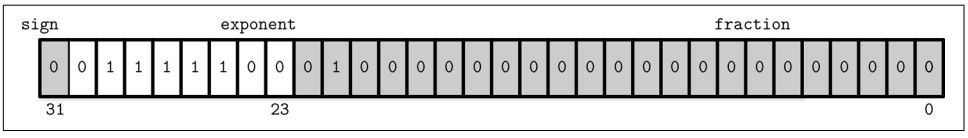

Figure 3-2. Binary representation of single-precision float number

Pascal String

```
String
{
    size    uint_16
    data    byte[size]
}
```

Enums

```
enum NodeType {
   ROOT,     // 0x00h
   INTERNAL, // 0x01h
   LEAF      // 0x02h
};
```

Flags

```
int IS_LEAF_MASK         = 0x01h; // bit #1
int VARIABLE_SIZE_VALUES = 0x02h; // bit #2
int HAS_OVERFLOW_PAGES   = 0x04h; // bit #3
```

Bitmasks

```
// Set the bit
flags |= HAS_OVERFLOW_PAGES;
flags |= (1 << 2);

// Unset the bit
flags &= ~HAS_OVERFLOW_PAGES;
flags &= ~(1 << 2);

// Test whether or not the bit is set
is_set = (flags & HAS_OVERFLOW_PAGES) != 0;
is_set = (flags & (1 << 2)) != 0;
```

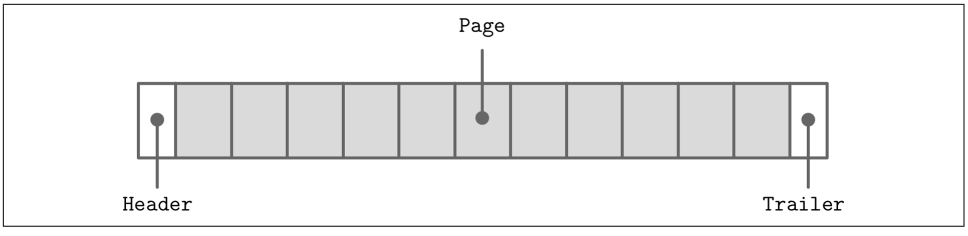

Figure 3-3. File organization

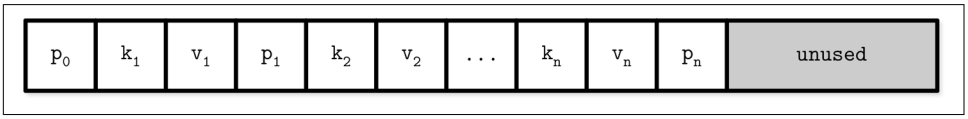

Figure 3-4. Page organization for fixed-size records

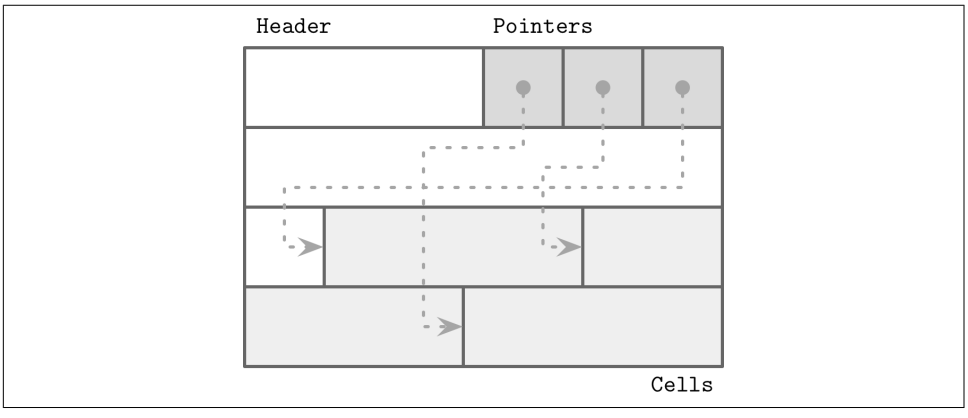

Figure 3-5. Slotted page

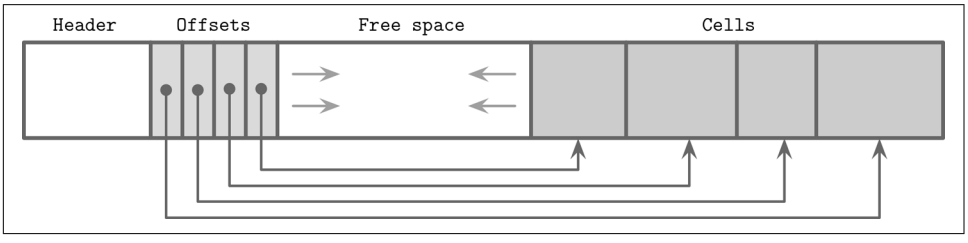

Figure 3-6. Offset and cell growth direction

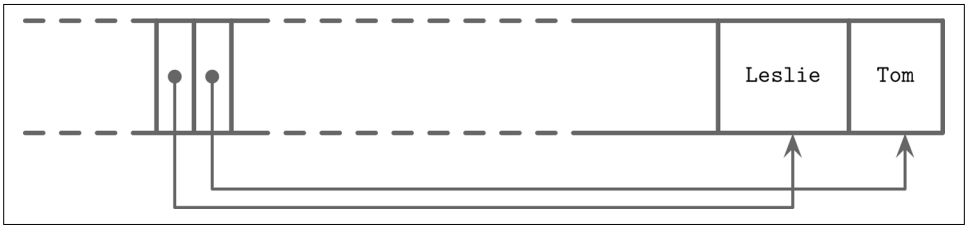

Figure 3-7. Records appended in random order: Tom, Leslie

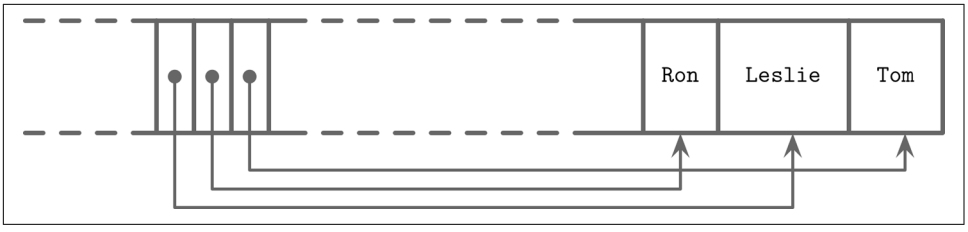

Figure 3-8. Appending one more record: Ron

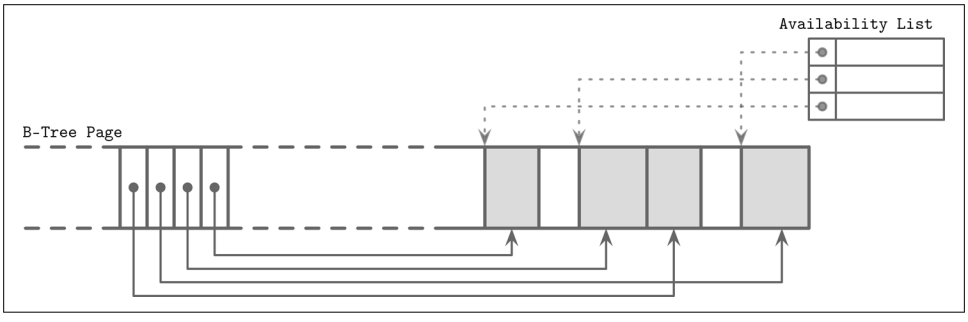

Figure 3-9. Fragmented page and availability list. Occupied pages are shown in gray. Dotted lines represent pointers to unoccupied memory regions from the availability list.

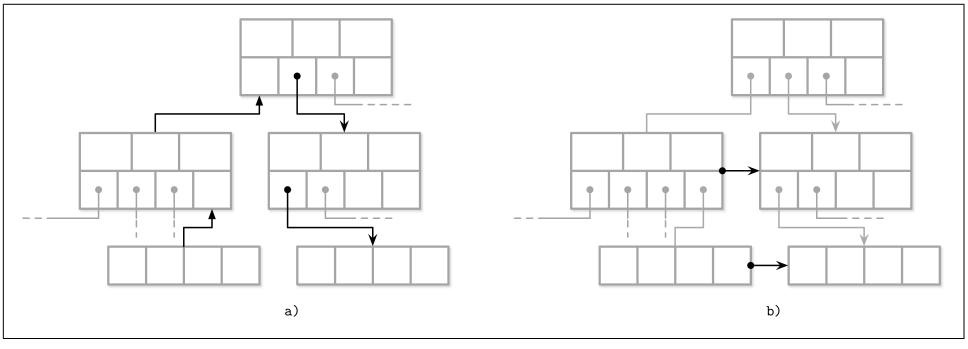

Figure 4-1. Locating a sibling by following parent links (a) versus sibling links (b)

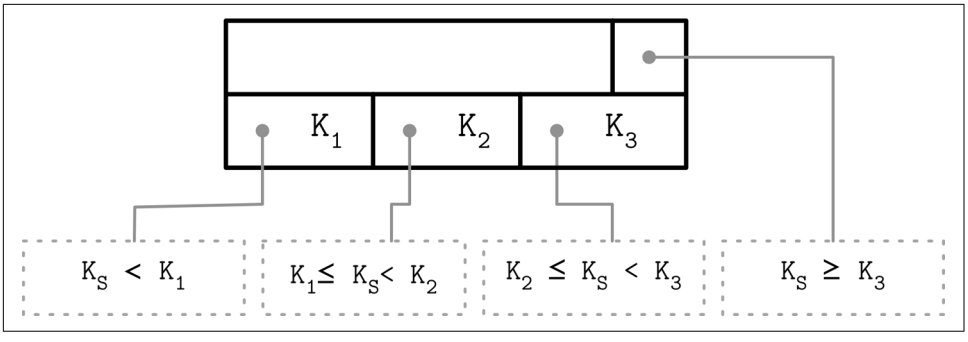

Figure 4-2. Rightmost pointer

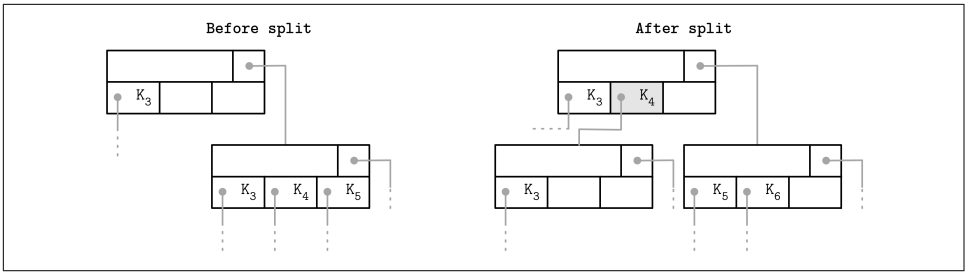

Figure 4-3. Rightmost pointer update during node split. The promoted key is shown in gray.

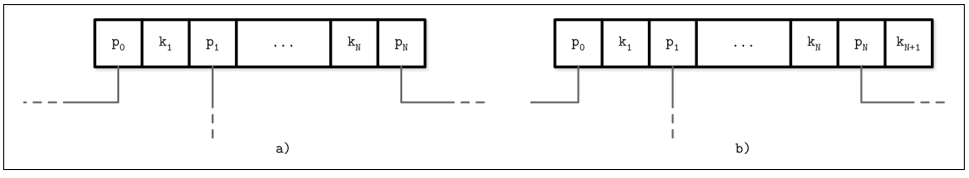

Figure 4-4. B-Trees without (a) and with (b) a high key

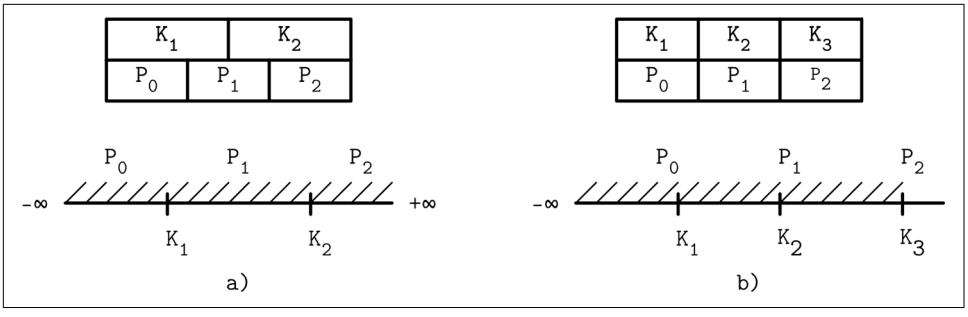

Figure 4-5. Using +∞ as a virtual key (a) versus storing the high key (b)

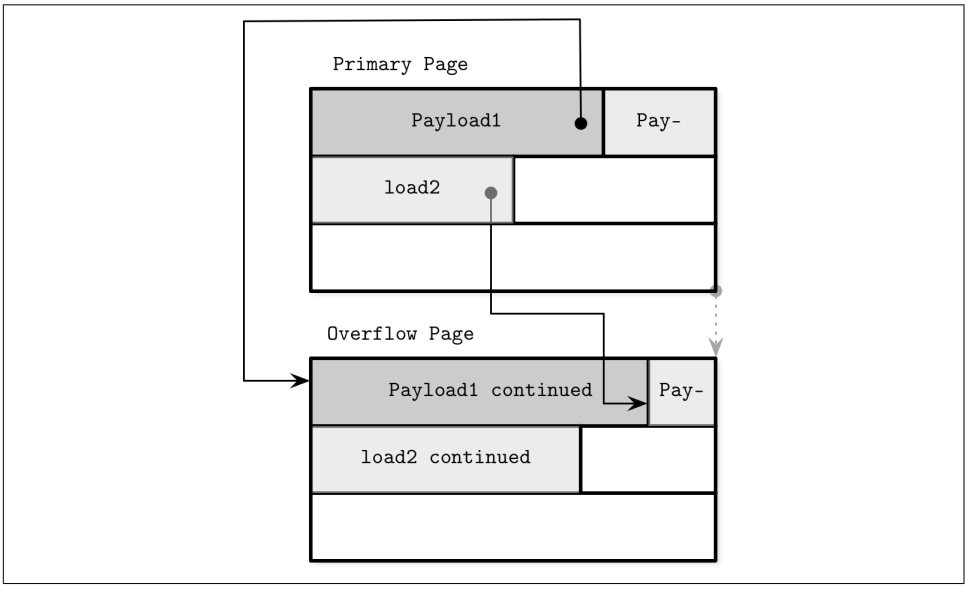

Figure 4-6. Overflow pages

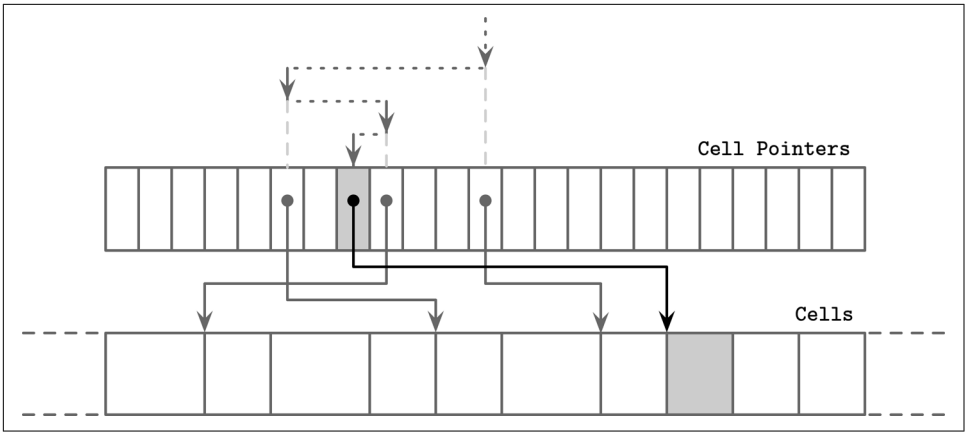

Figure 4-7. Binary search with indirection pointers. The searched element is shown in gray. Dotted arrows represent binary search through cell pointers. Solid lines represent accesses that follow the cell pointers, necessary to compare the cell value with a searched key.

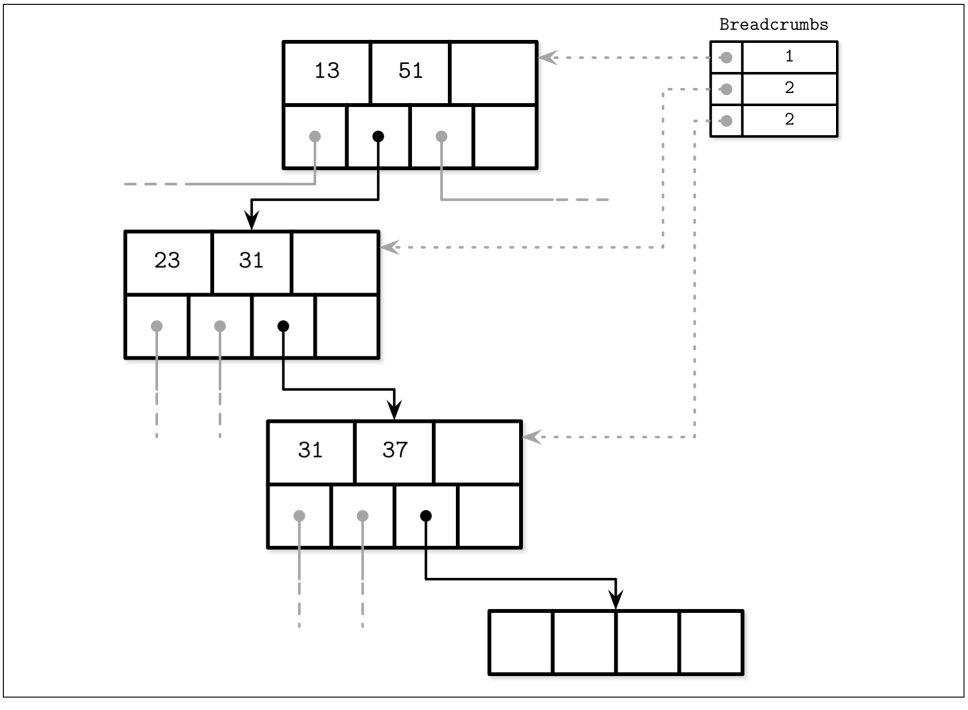

Figure 4-8. Breadcrumbs collected during lookup, containing traversed nodes and cell indices. Dotted lines represent logical links to visited nodes. Numbers in the breadcrumbs table represent indices of the followed child pointers.

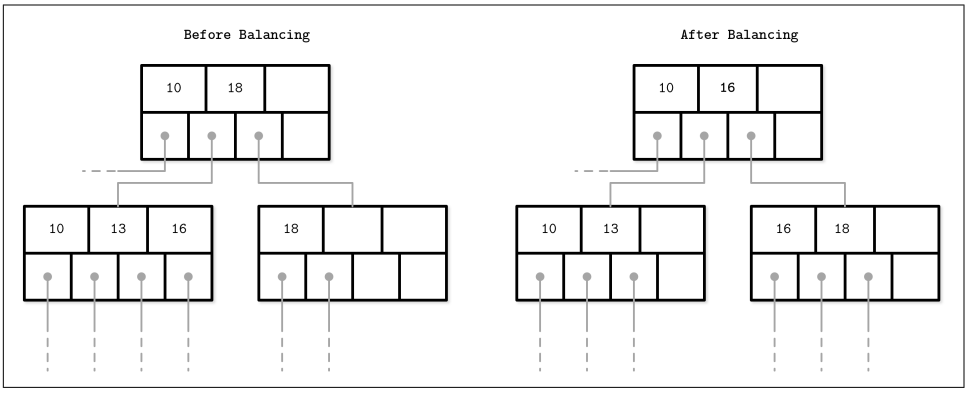

Figure 4-9. B-Tree balancing: Distributing elements between the more occupied node and the less occupied one

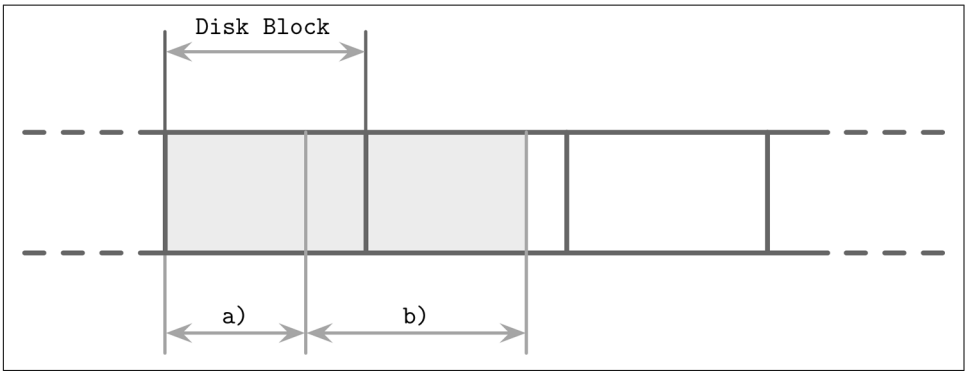

Figure 4-10. Compression and block padding

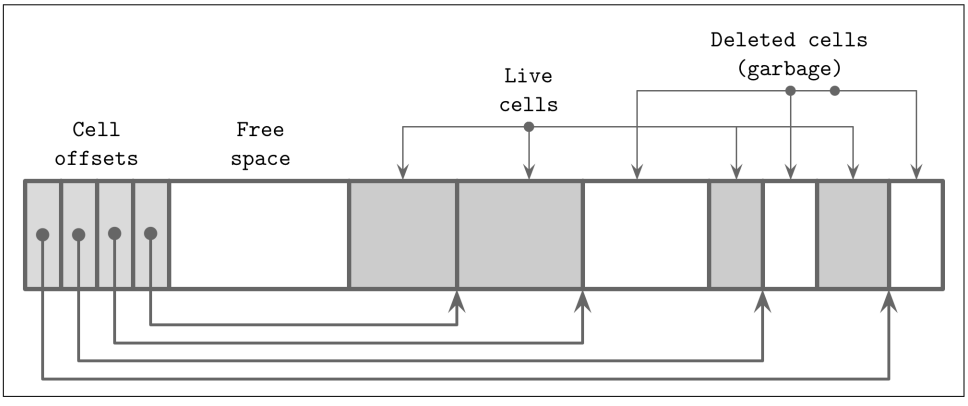

Figure 4-11. An example of a fragmented page

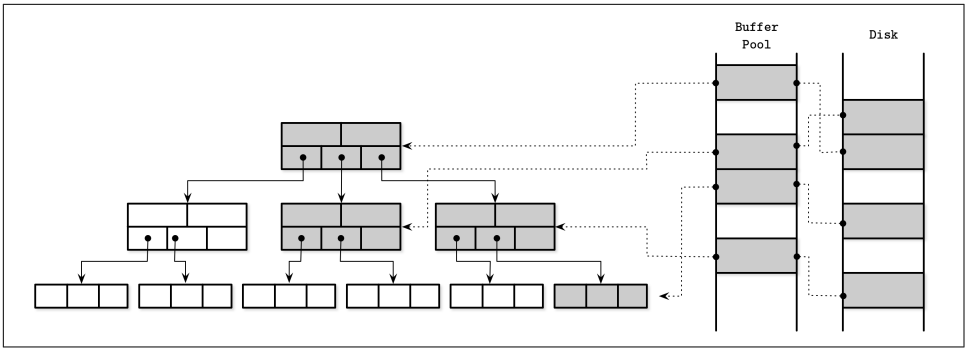

Figure 5-1. Page cache

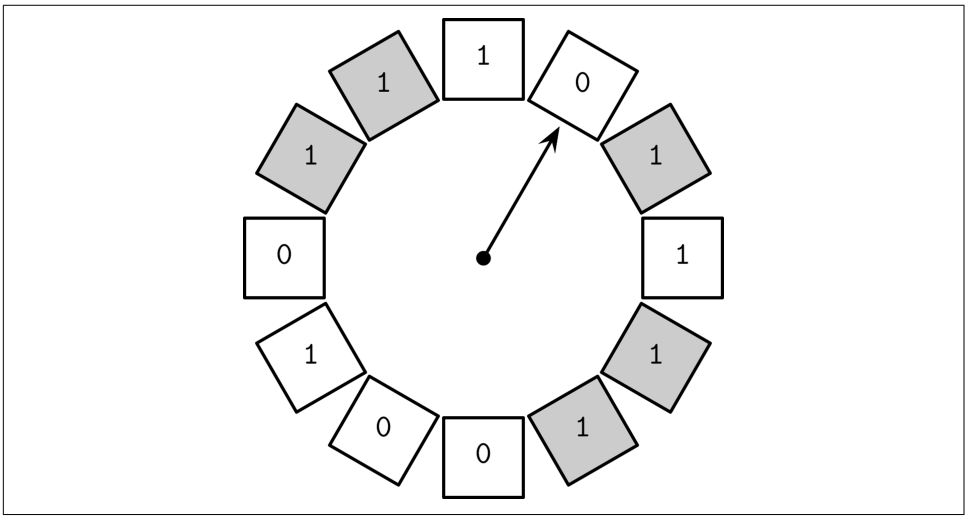

Figure 5-2. CLOCK-sweep example. Counters for currently referenced pages are shown in gray. Counters for unreferenced pages are shown in white. The arrow points to the element that will be checked next.

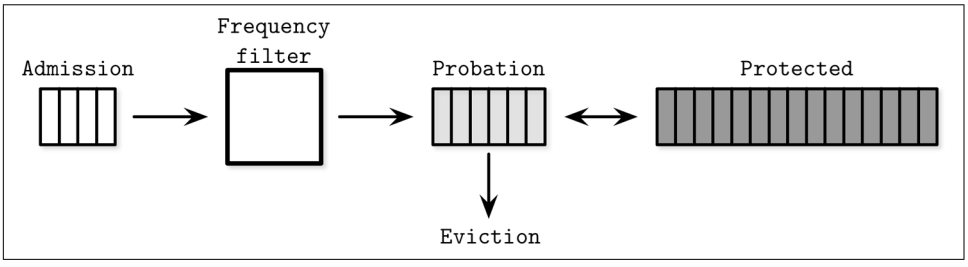

Figure 5-3. TinyLFU admission, protected, and probation queues

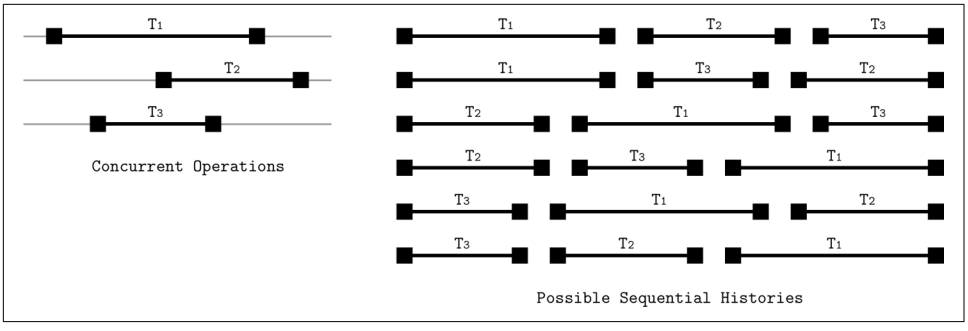

Figure 5-4. Concurrent transactions and their possible sequential execution histories

	Dirty	Non-Repeatable	Phantom
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	-	Allowed	Allowed
Repeatable Read	-	-	Allowed
Serializable	-	-	-

Figure 5-5. Isolation levels and allowed anomalies

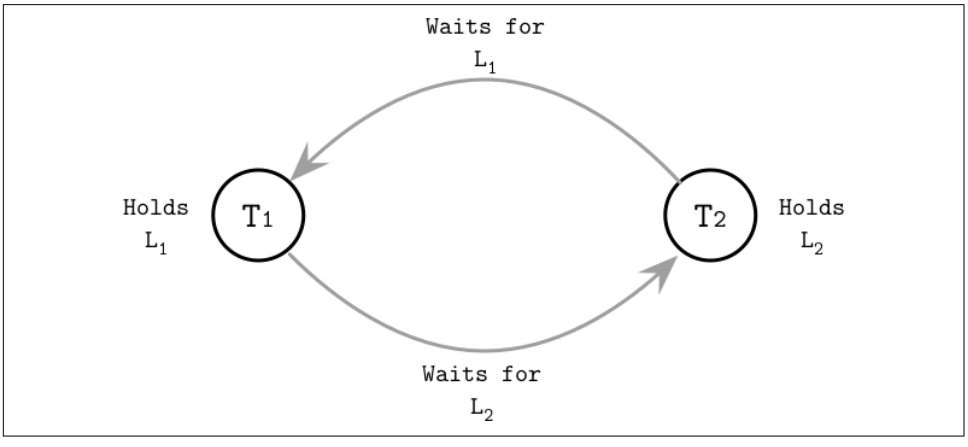

Figure 5-6. Example of a deadlock

	Reader	Writer
Reader	Shared	Exclusive
Writer	Exclusive	Exclusive

Figure 5-7. Readers-writer lock compatibility table

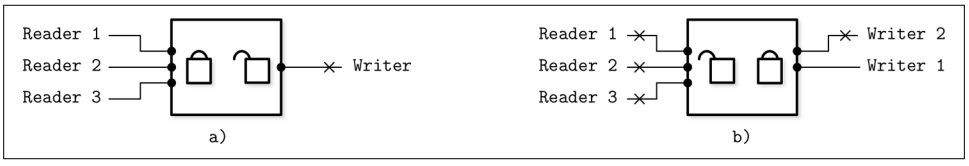

Figure 5-8. Readers-writer locks

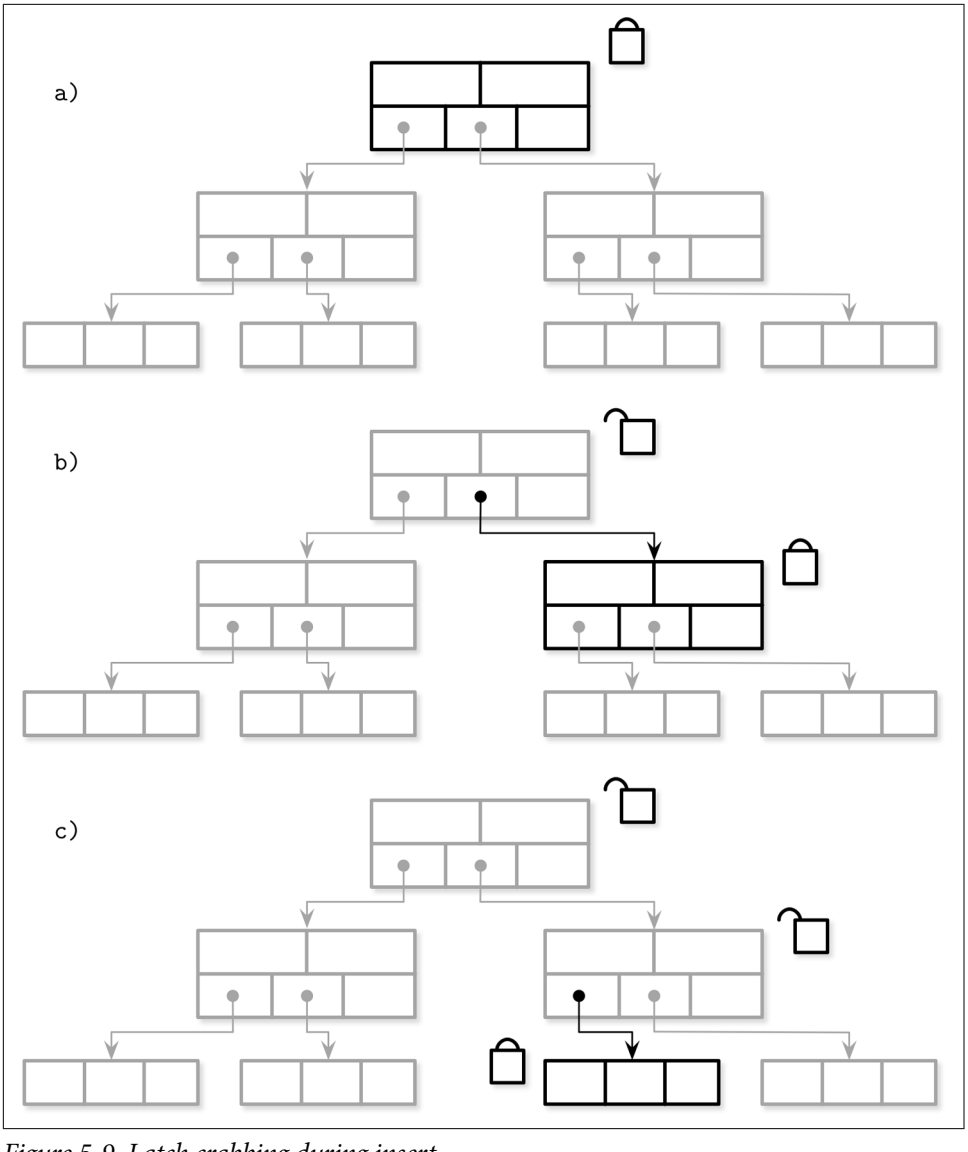

Figure 5-9. Latch crabbing during insert

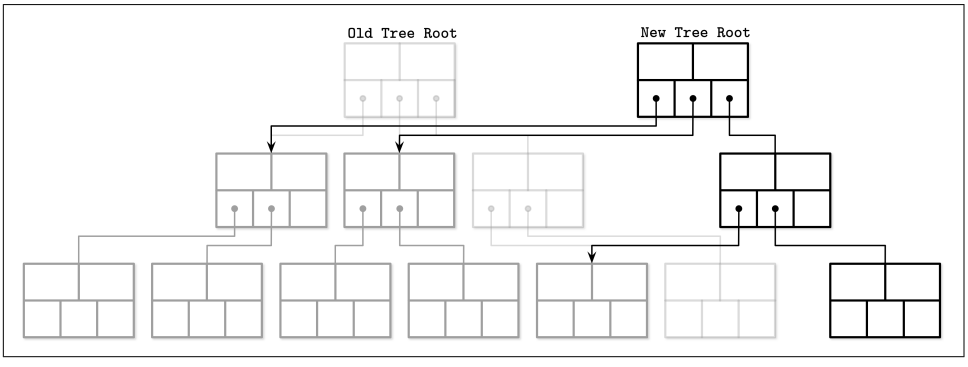

Figure 6-1. Copy-on-write B-Trees

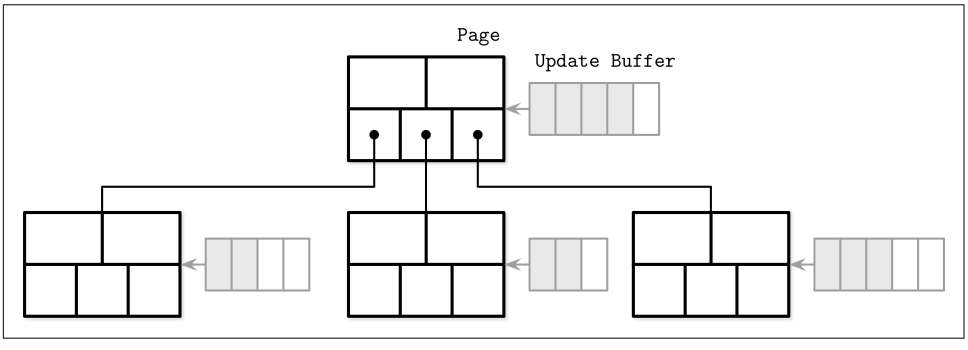

Figure 6-2. WiredTiger: high-level overview

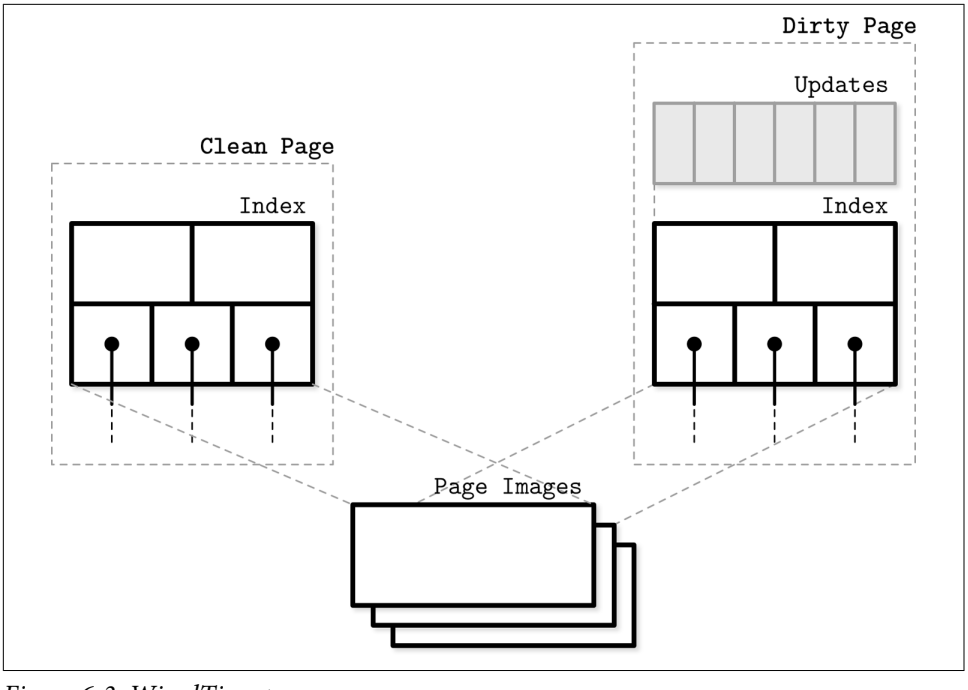

Figure 6-3. WiredTiger pages

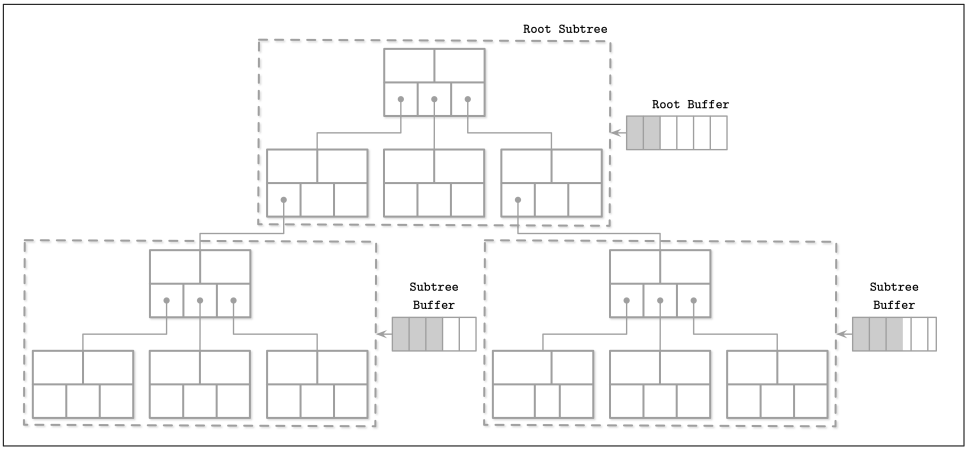

Figure 6-4. LA-Tree

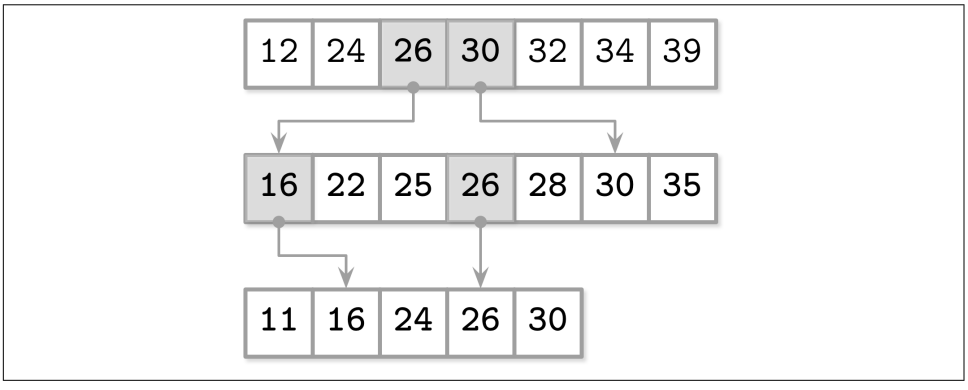

Figure 6-5. Fractional cascading

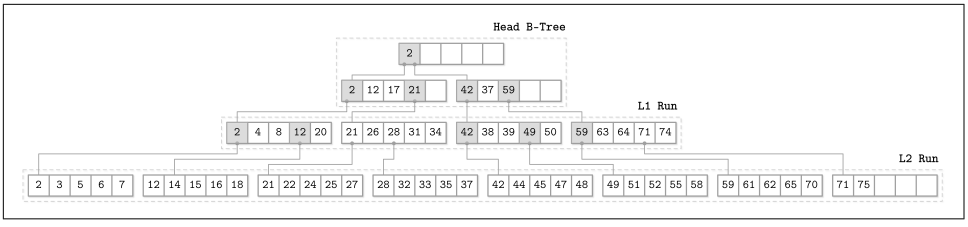

Figure 6-6. Schematic FD-Tree overview

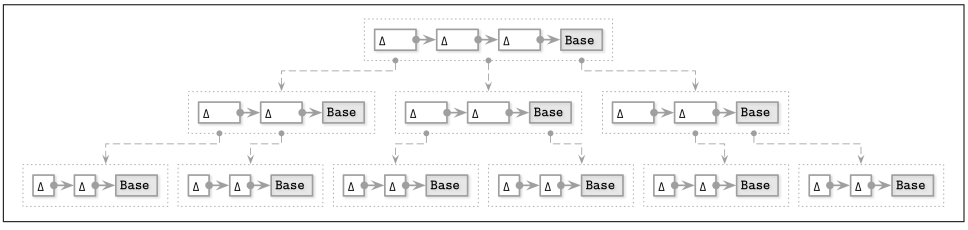

Figure 6-7. Bw-Tree. Dotted lines represent virtual links between the nodes, resolved using the mapping table. Solid lines represent actual data pointers between the nodes.

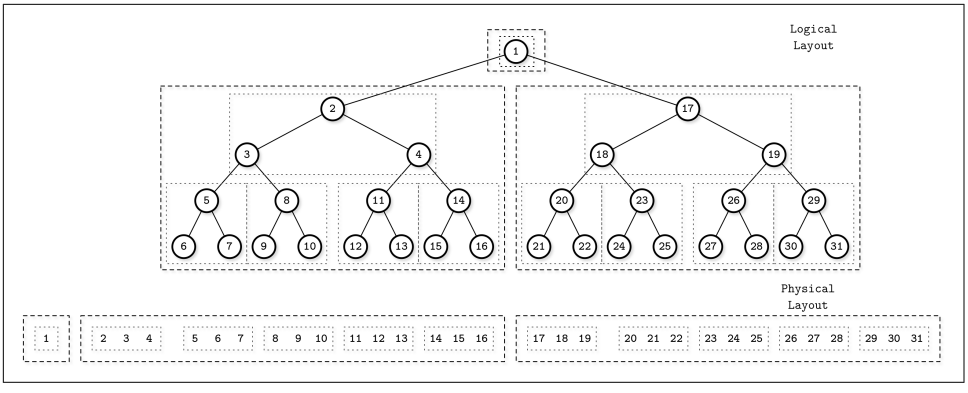

Figure 6-8. van Emde Boas layout

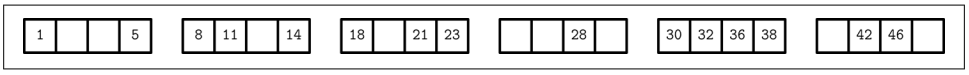

Figure 6-9. Packed array

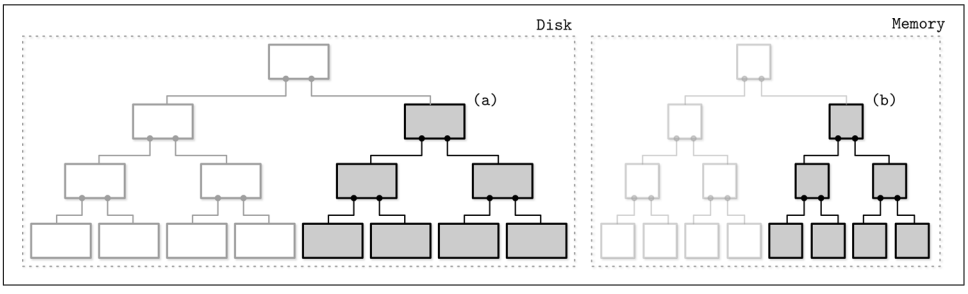

Figure 7-1. Two-component LSM Tree before a flush. Flushing memory- and disk-resident segments are shown in gray.

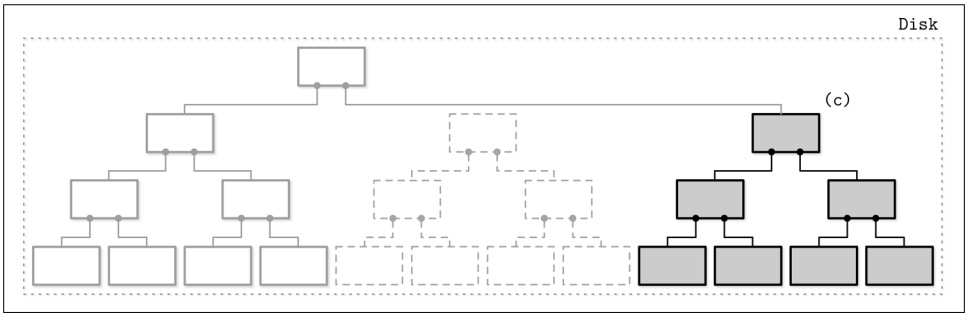

Figure 7-2. Two-component LSM Tree after a flush. Merged contents are shown in gray. Boxes with dashed lines depict discarded on-disk segments.

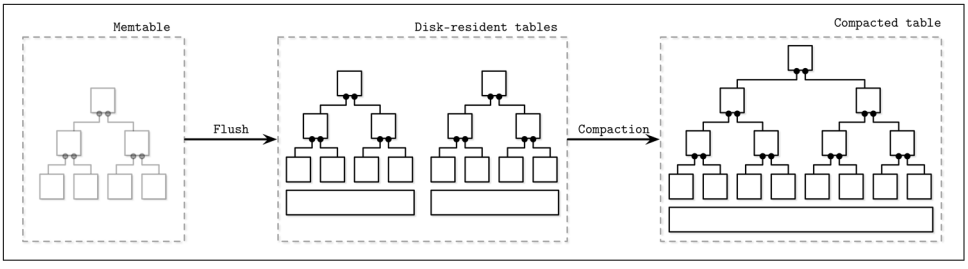

Figure 7-3. Multicomponent LSM Tree data life cycle

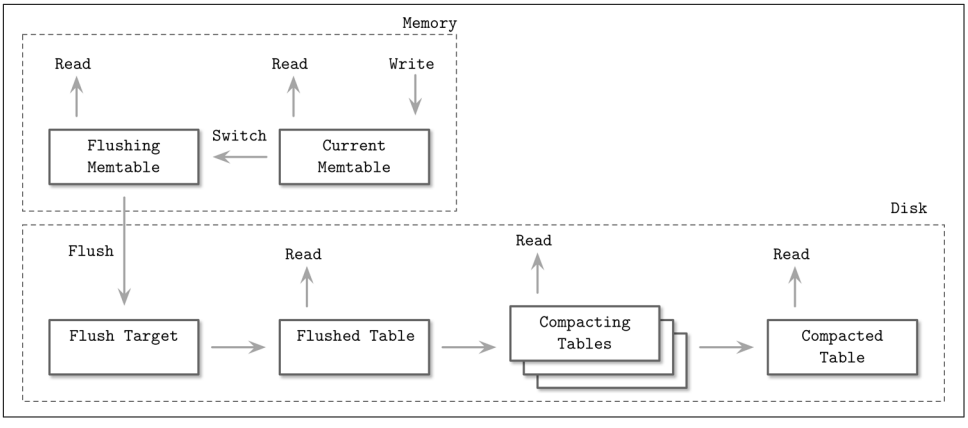

Figure 7-4. LSM component structure

Delete Entry

```
Disk Table          Memtable
| k1 | v1 |         | k1 | <tombstone> |
```

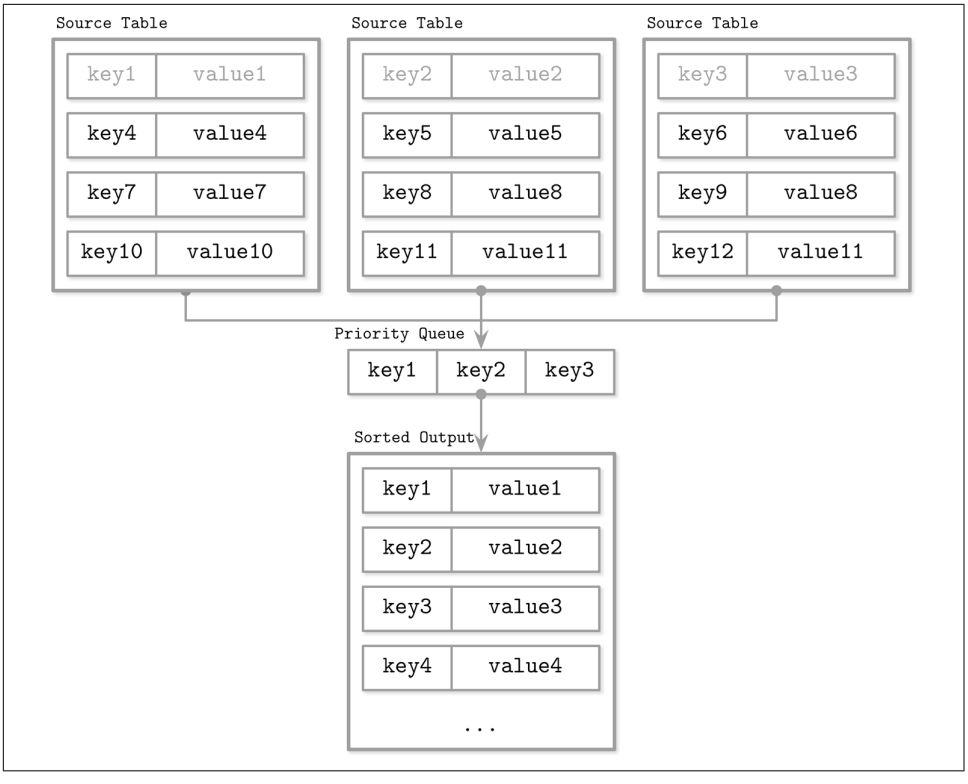

Figure 7-5. LSM merge mechanics

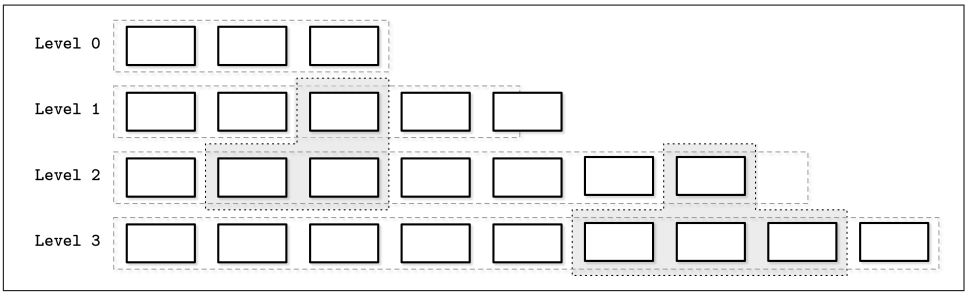

Figure 7-6. Compaction process. Gray boxes with dashed lines represent currently compacting tables. Level-wide boxes represent the target data size limit on the level. Level 1 is over the limit.

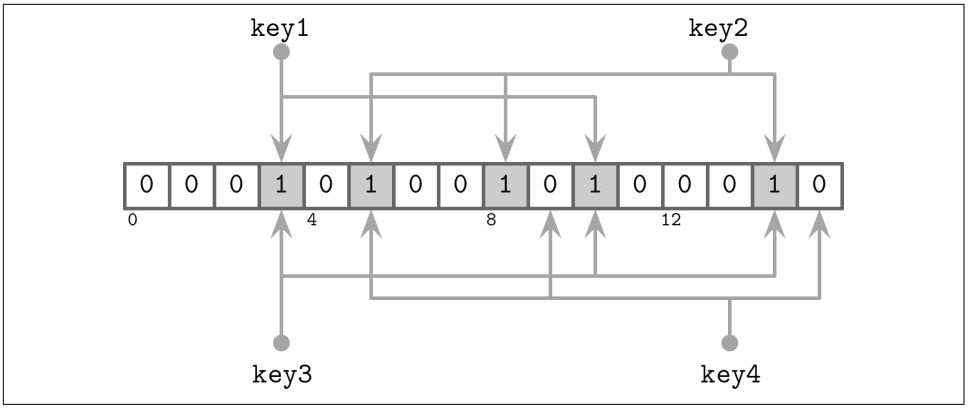

Figure 7-7. Bloom filter

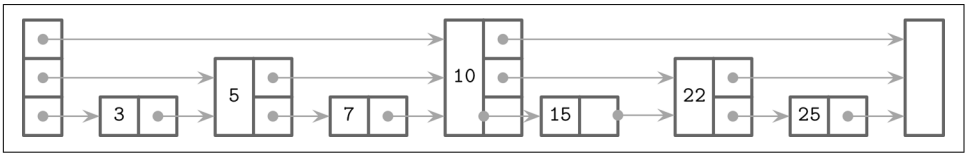

Figure 7-8. Skiplist

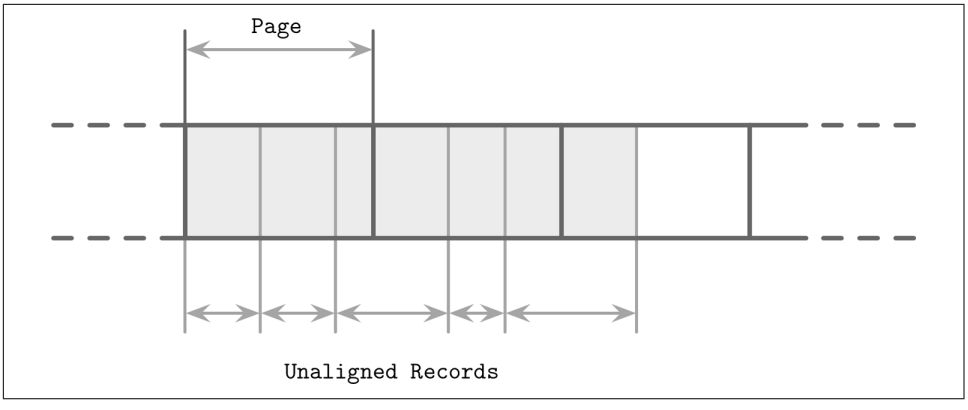

Figure 7-9. Unaligned data records

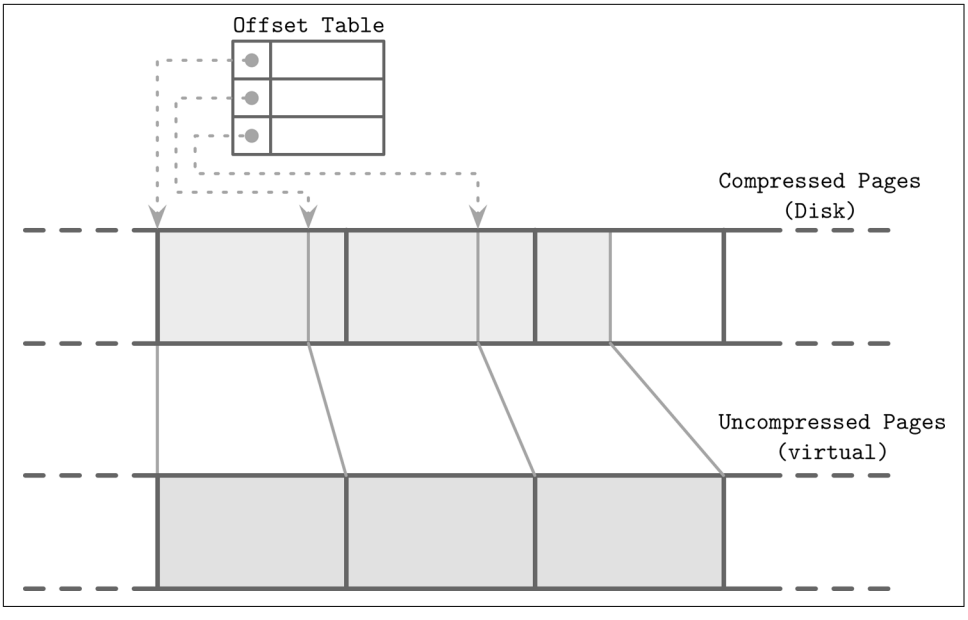

Figure 7-10. Reading compressed blocks. Dotted lines represent pointers from the mapping table to the offsets of compressed pages on disk. Uncompressed pages generally reside in the page cache.

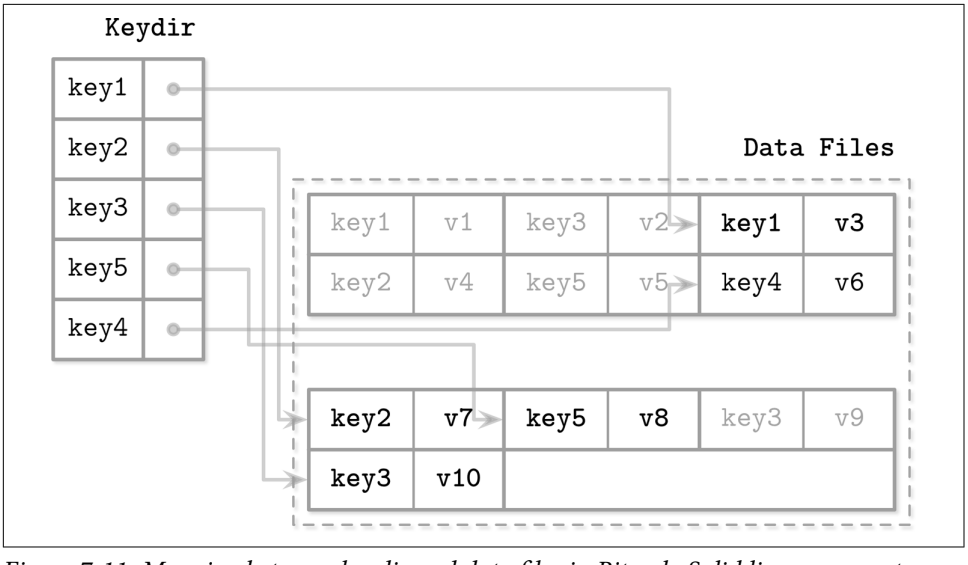

Figure 7-11. Mapping between keydir and data files in Bitcask. Solid lines represent pointers from the key to the latest value associated with it. Shadowed key/value pairs are shown in light gray.

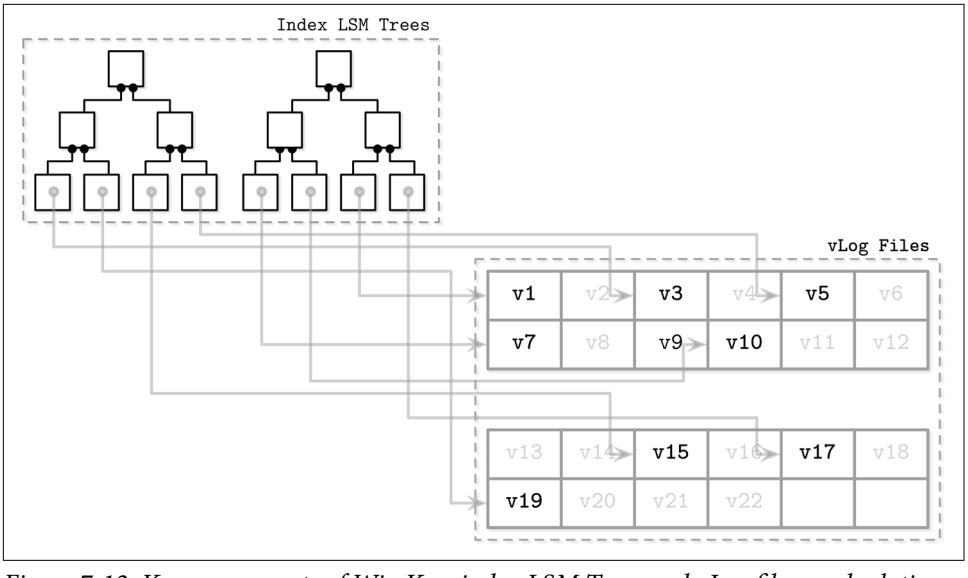

Figure 7-12. Key components of WiscKey: index LSM Trees and vLog files, and relationships between them. Shadowed records in data files (ones that were superseded by later writes or deletes) are shown in gray. Solid lines represent pointers from the key in the LSM tree to the latest value in the log file.

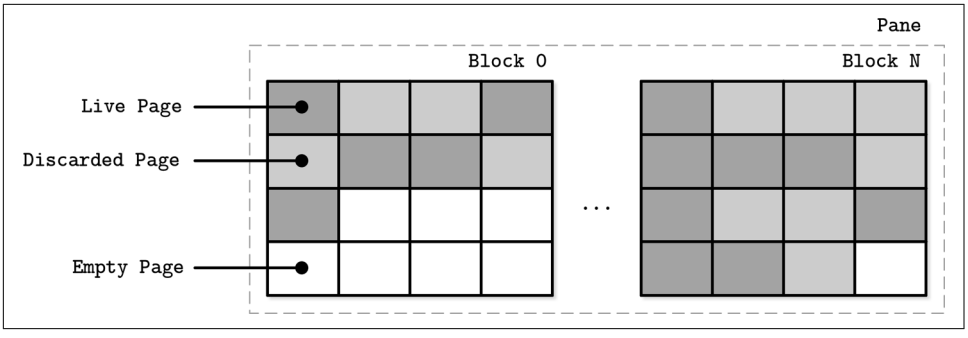

Figure 7-13. SSD pages, grouped into blocks

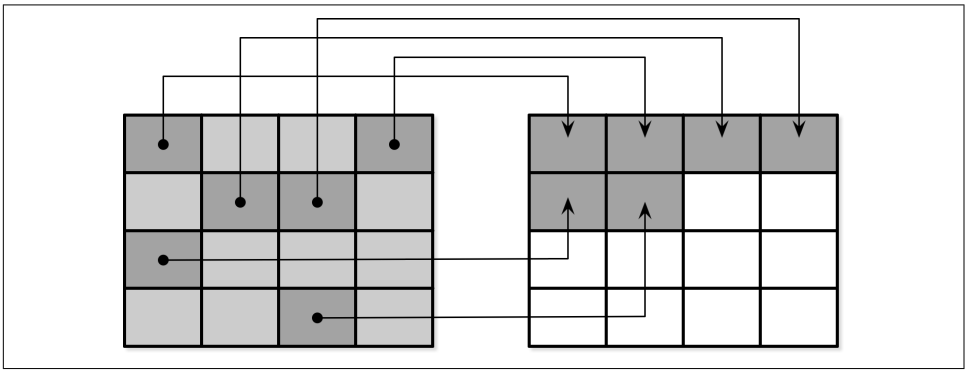

Figure 7-14. Page relocation during garbage collection

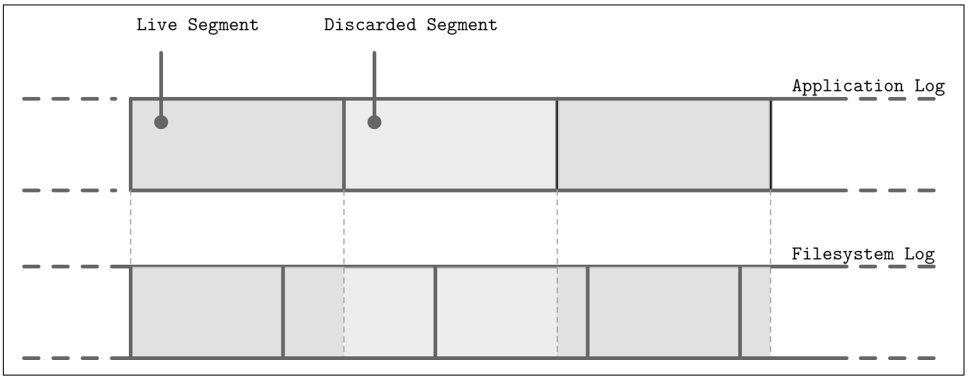

Figure 7-15. Misaligned writes and discarding of a higher-level log segment

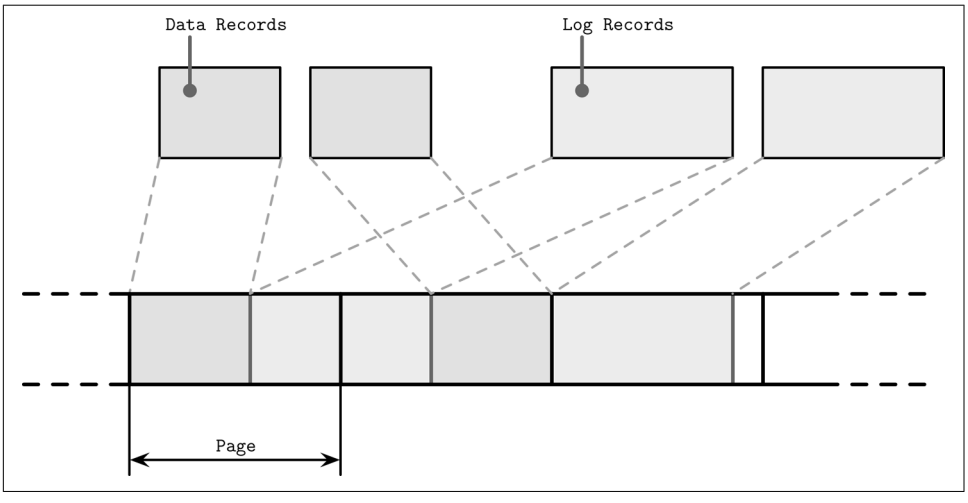

Figure 7-16. Unaligned multistream writes

	Buffered	Mutable	Ordered
B+Trees	No	Yes	Yes
WiredTiger	Yes	Yes	Yes
LA-Trees	Yes	Yes	Yes
COW B-Trees	No	No	Yes
2C LSM Trees	Yes	No	Yes
MC LSM Trees	Yes	No	Yes
FD-Trees	Yes	No	Yes
BitCask	No	No	No
WiscKey	Yes(1)	No	Yes(1)
BW-Trees	No	No	No(2)

Figure I-1. Buffering, immutability, and ordering properties of discussed storage structures. (1) WiscKey uses buffering only for keeping keys sorted order. (2) Only consolidated nodes in Bw-Trees hold ordered records.

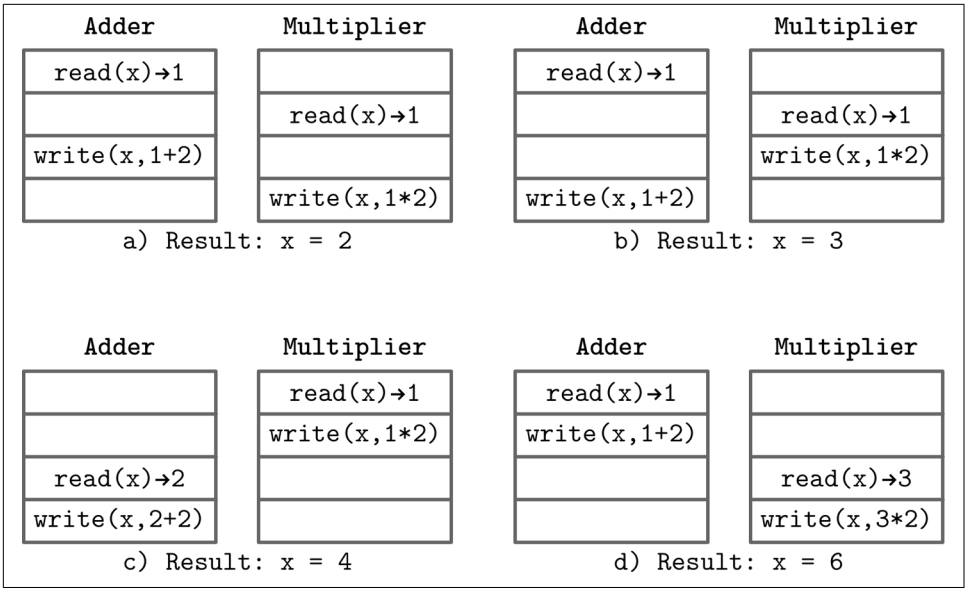

Figure 8-1. Possible interleavings of concurrent executions

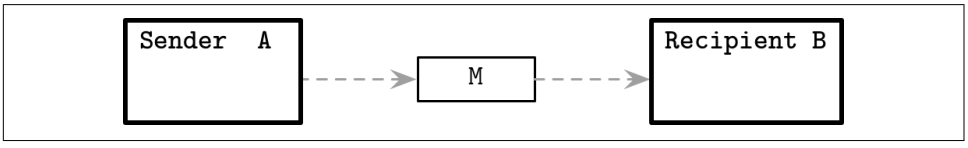

Figure 8-2. Simplest, unreliable form of communication

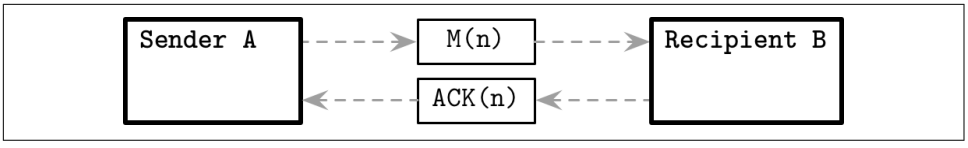

Figure 8-3. Sending a message with an acknowledgment

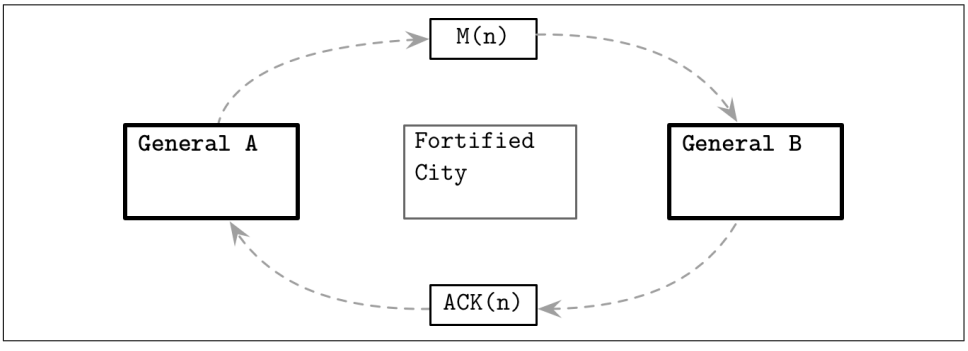

Figure 8-4. Two Generals' Problem illustrated

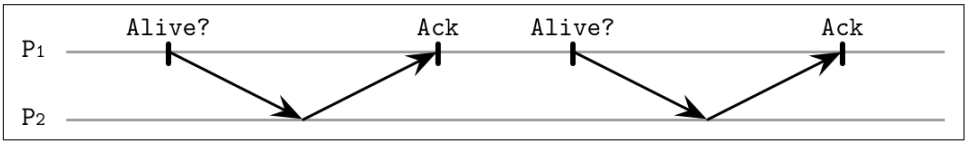

Figure 9-1. Pings for failure detection: normal functioning, no message delays

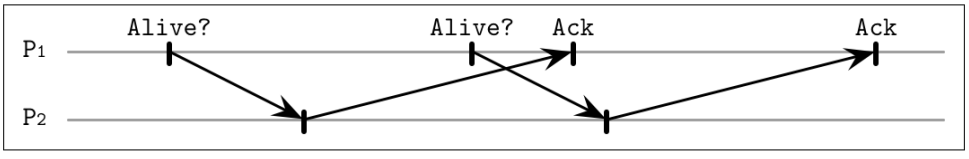

Figure 9-2. Pings for failure detection: responses are delayed, coming after the next message is sent

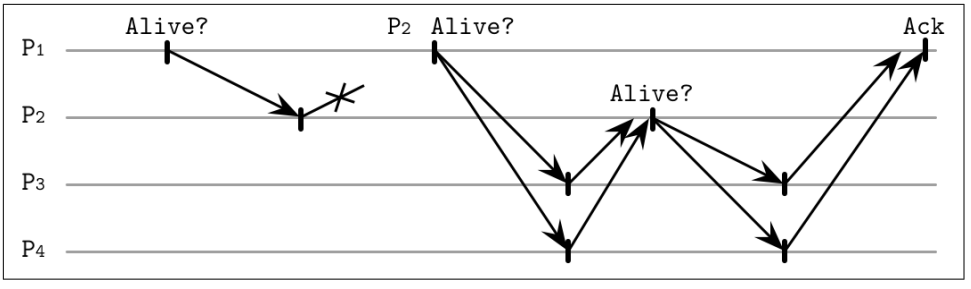

Figure 9-3. “Outsourcing” heartbeats

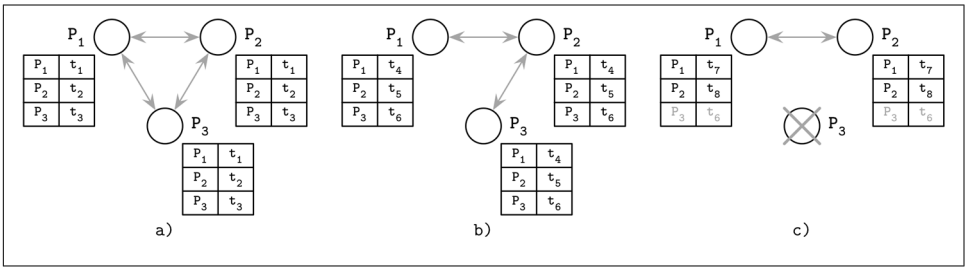

Figure 9-4. Replicated heartbeat table for failure detection

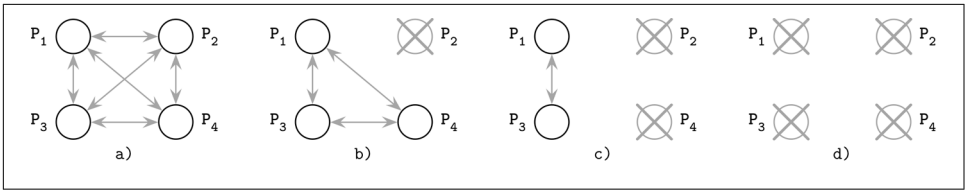

Figure 9-5. FUSE failure detection

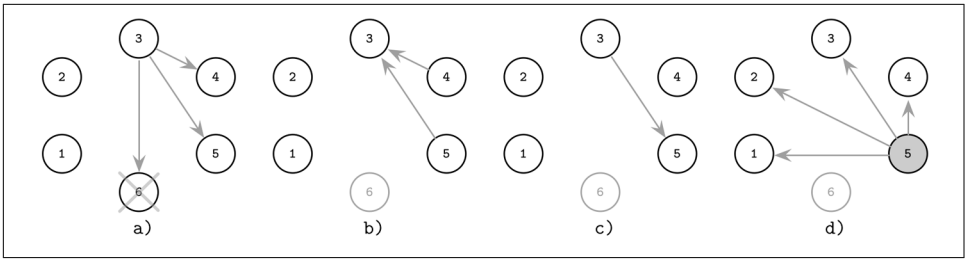

Figure 10-1. Bully algorithm: previous leader (6) fails and process 3 starts the new election

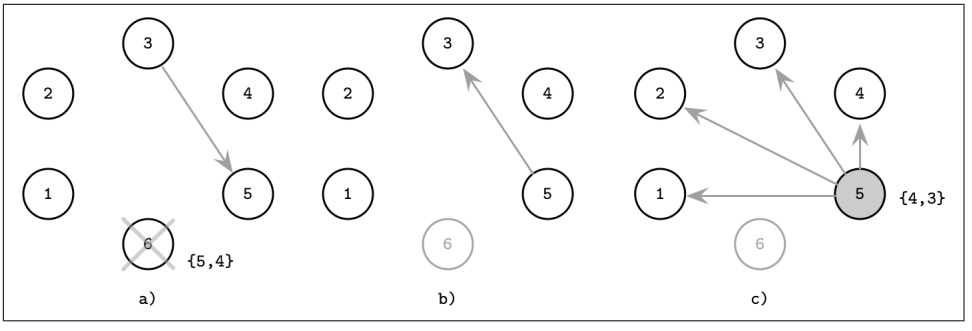

Figure 10-2. Bully algorithm with failover: previous leader (6) fails and process 3 starts the new election by contacting the highest-ranked alternative

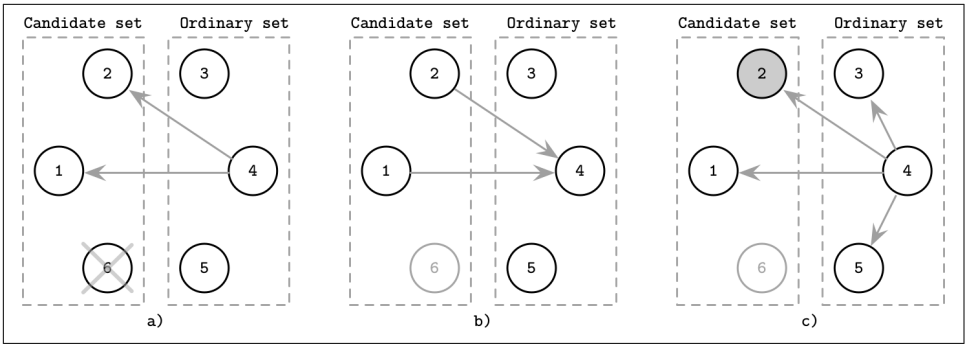

Figure 10-3. Candidate/ordinary modification of the bully algorithm: previous leader (6) fails and process 4 starts the new election

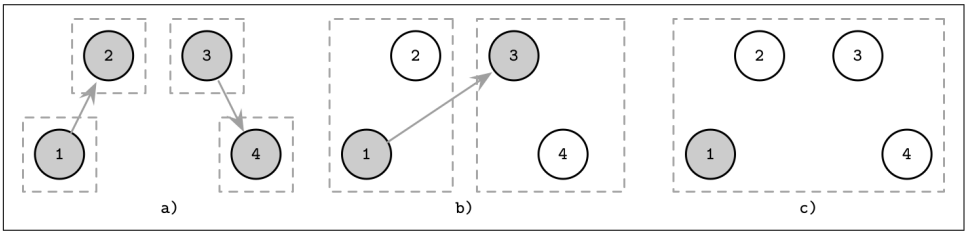

Figure 10-4. Invitation algorithm

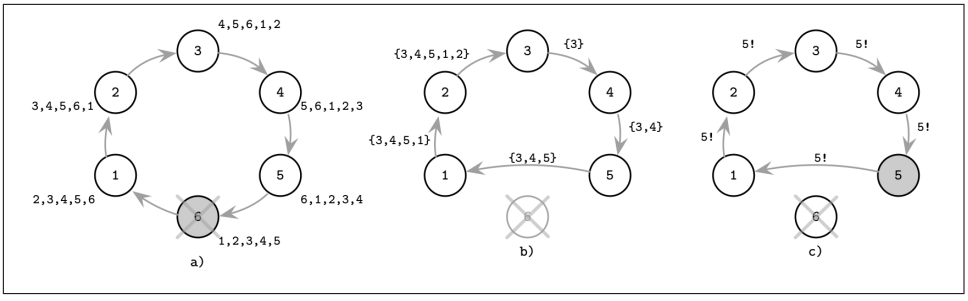

Figure 10-5. Ring algorithm: previous leader (6) fails and 3 starts the election process

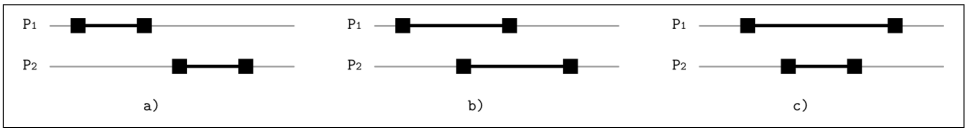

Figure 11-1. Sequential and concurrent operations

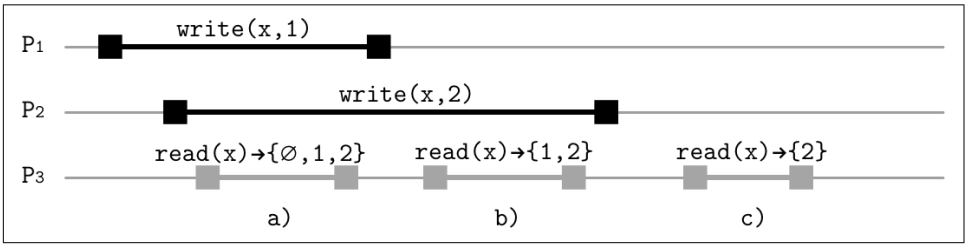

Figure 11-2. Example of linearizability

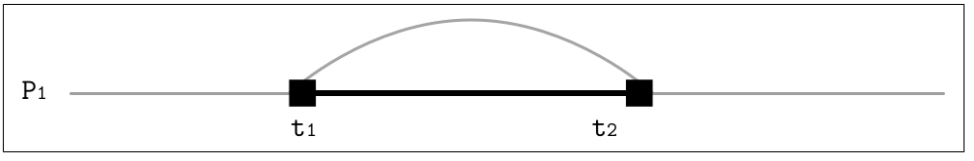

Figure 11-3. Time bounds of a linearizable operation

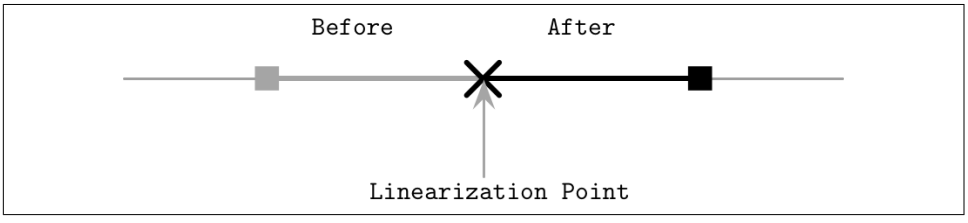

Figure 11-4. Linearization point

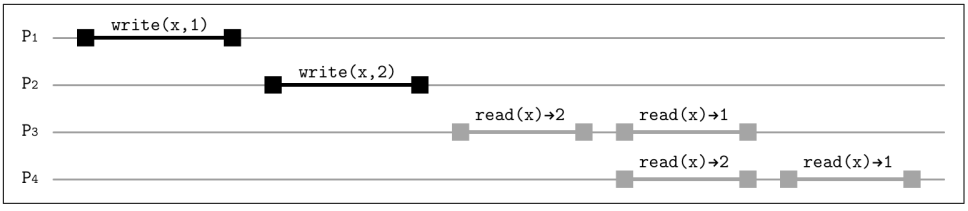

Figure 11-5. Ordering in sequential consistency

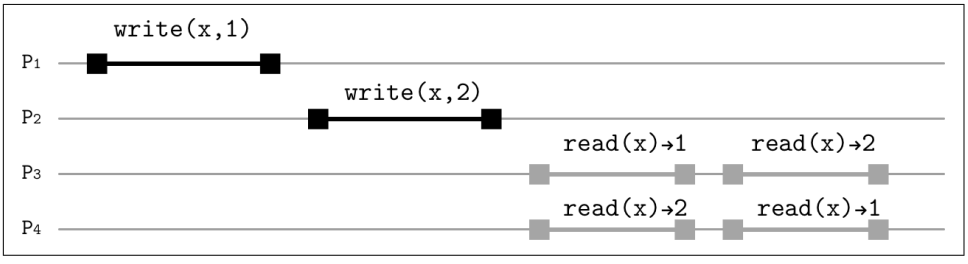

Figure 11-6. Write operations with no causal relationship

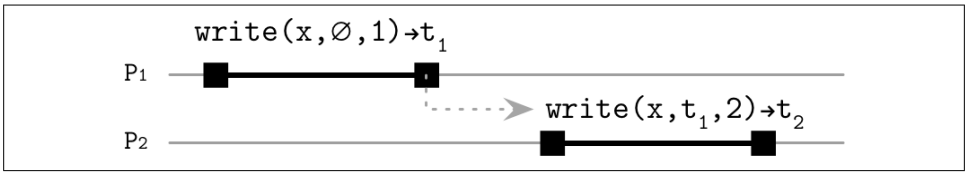

Figure 11-7. Causally related write operations

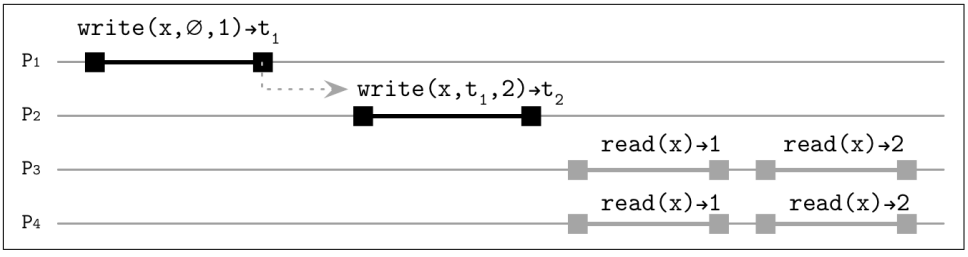

Figure 11-8. Write operations with causal relationship

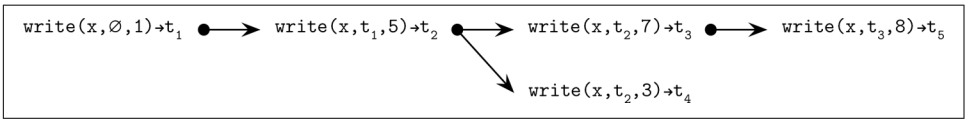

Figure 11-9. Divergent histories under causal consistency

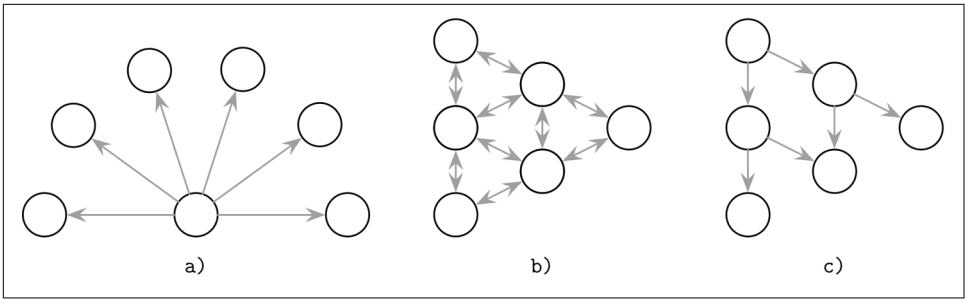

Figure 12-1. Broadcast (a), anti-entropy (b), and gossip (c)

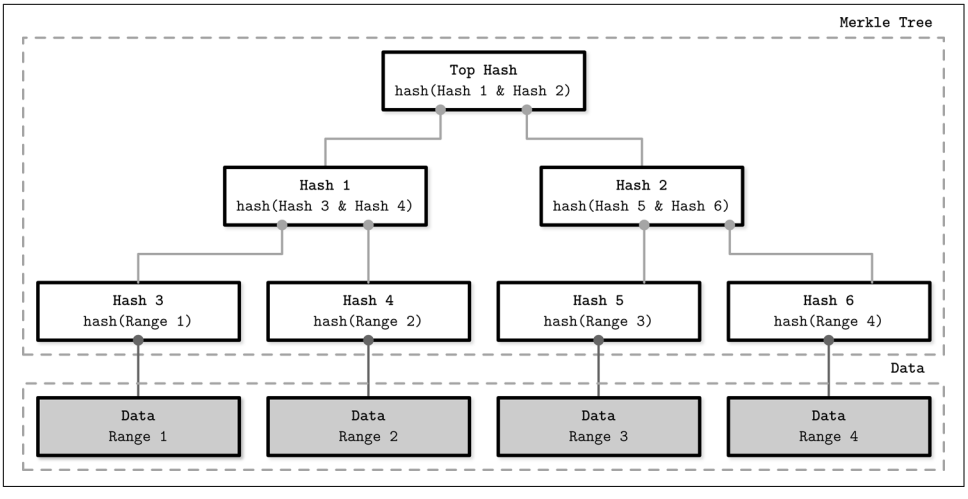

Figure 12-2. Merkle tree. Gray boxes represent data record ranges. White boxes represent a hash tree hierarchy.

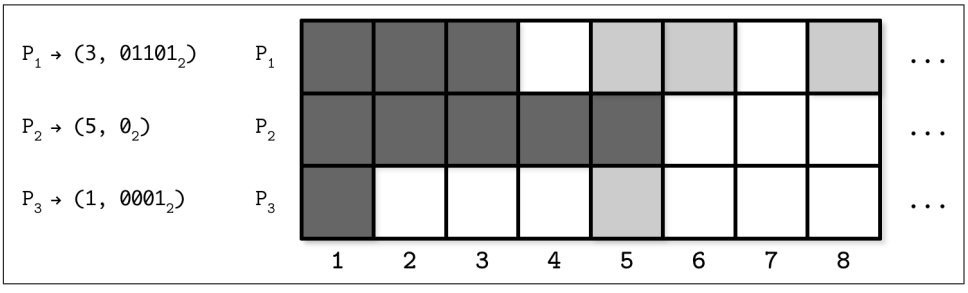

Figure 12-3. Bitmap version vector example

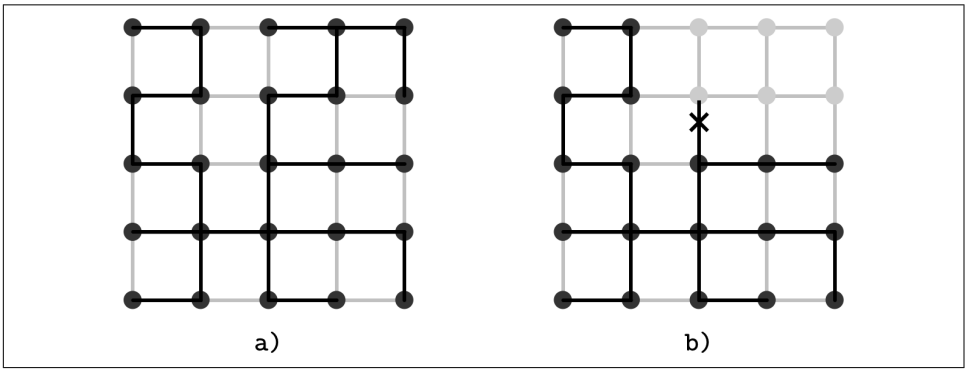

Figure 12-4. Spanning tree. Dark points represent nodes. Dark lines represent an overlay network. Gray lines represent other possible existing connections between the nodes.

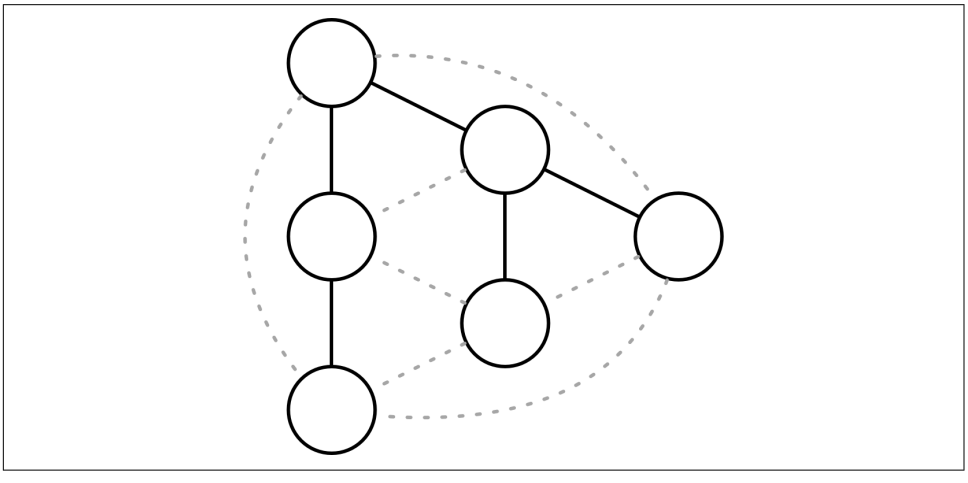

Figure 12-5. Lazy and eager push networks. Solid lines represent a broadcast tree. Dotted lines represent lazy gossip connections.

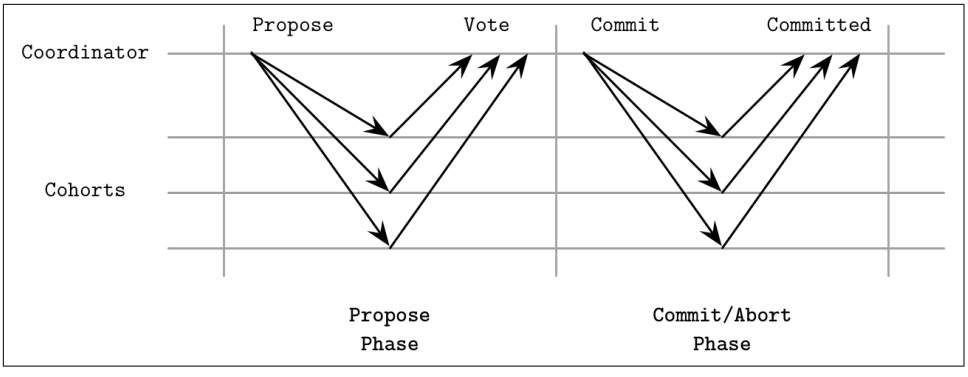

Figure 13-1. Two-phase commit protocol. During the first phase, cohorts are notified about the new transaction. During the second phase, the transaction is committed or aborted.

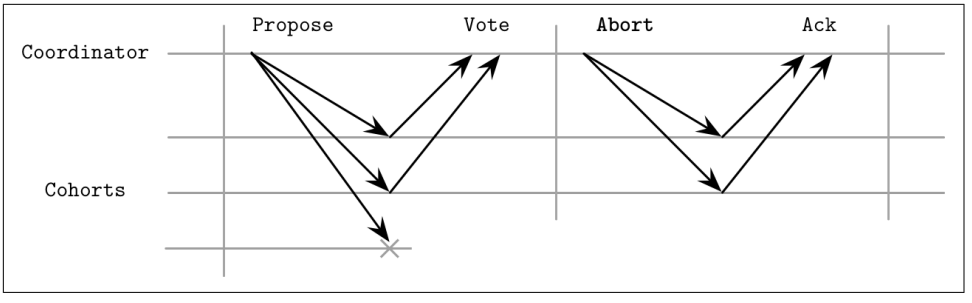

Figure 13-2. Cohort failure during the propose phase

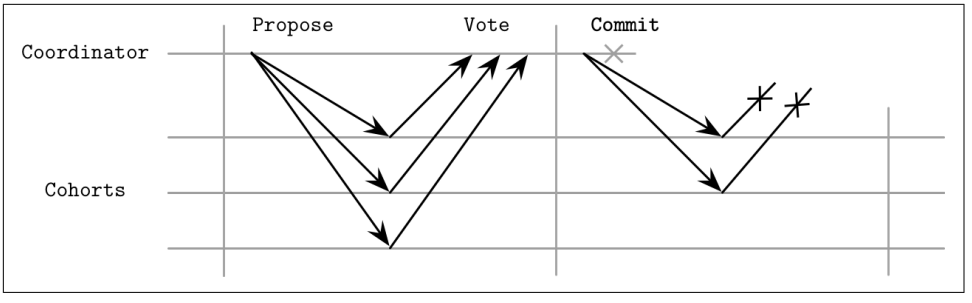

Figure 13-3. Coordinator failure after the propose phase

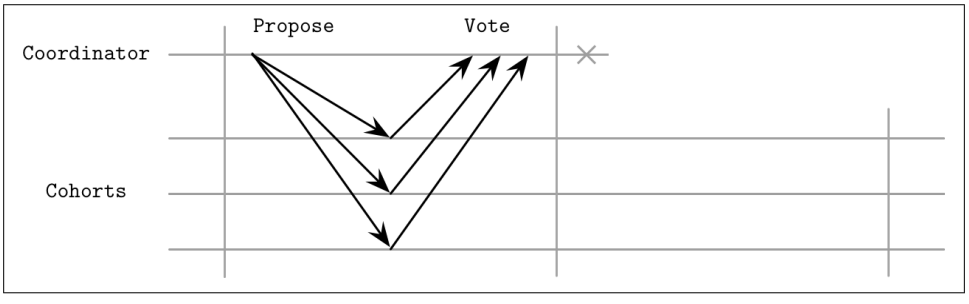

Figure 13-4. Coordinator failure before it could contact any cohorts

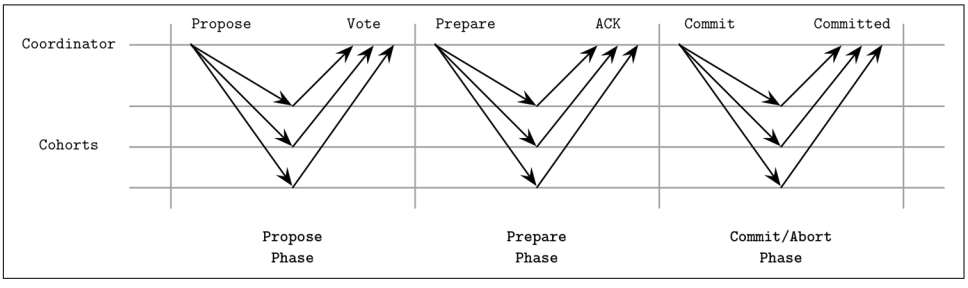

Figure 13-5. Three-phase commit

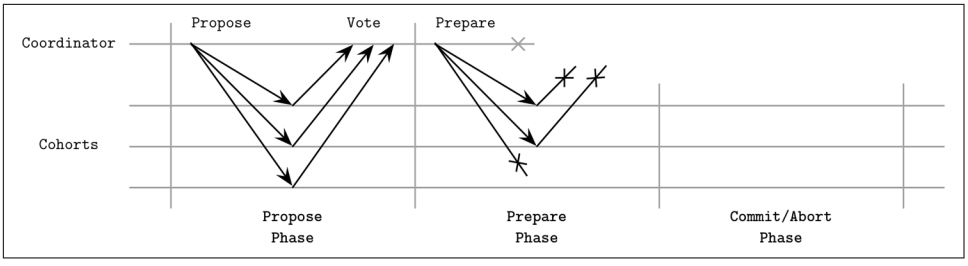

Figure 13-6. Coordinator failure during the second phase

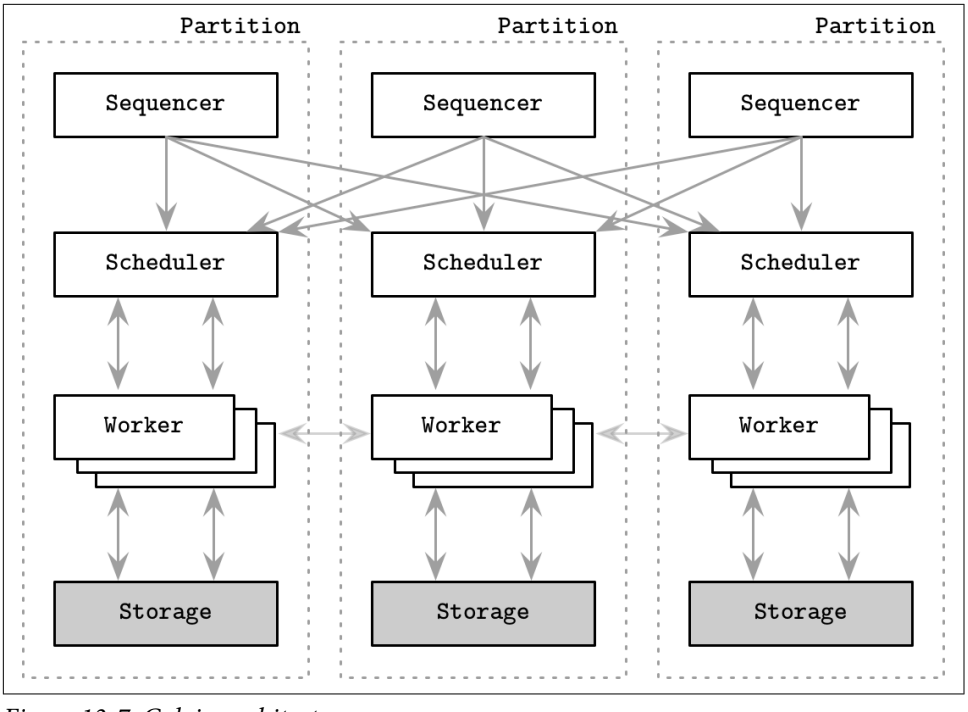

Figure 13-7. Calvin architecture

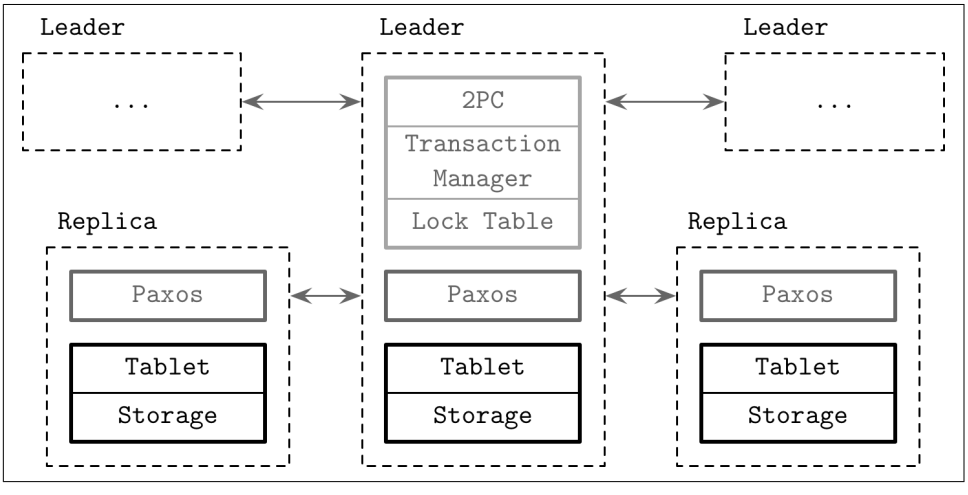

Figure 13-8. Spanner architecture

		Data	Locks	Write Metadata
Account1	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS2	-	-	TS1 is latest
	TS1	$200	-	-

a) Initial State before moving $150 from Account2 to Account1

		Data	Locks	Write Metadata
Account1	TS3	$250	Primary	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS3	$50	Primary at Account1	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

b) State after taking locks and updating accounts

		Data	Locks	Write Metadata
Account1	TS4	-	-	TS3 is latest
	TS3	$250	-	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS4	-	-	TS3 is latest
	TS3	$50	Primary at Account1	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

c) Transaction commit releases locks and updates metadata with latest timestamp

Figure 13-9. Percolator transaction execution steps. Transaction credits $150 from Account2 and debits it to Account1.

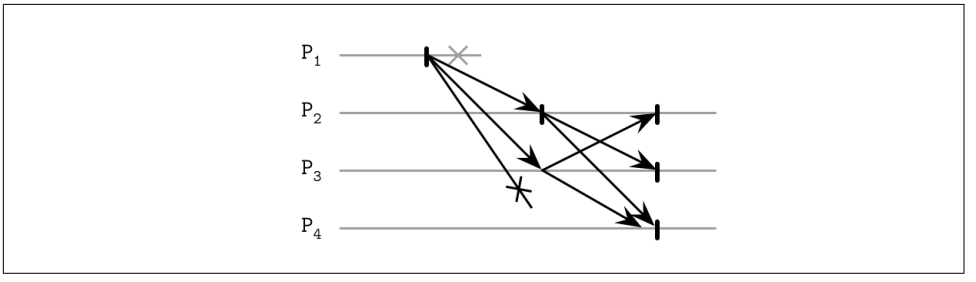

Figure 14-1. Broadcast

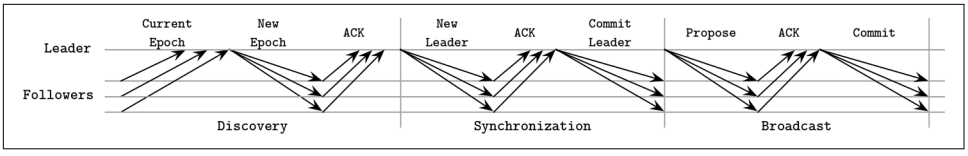

Figure 14-2. ZAB protocol summary

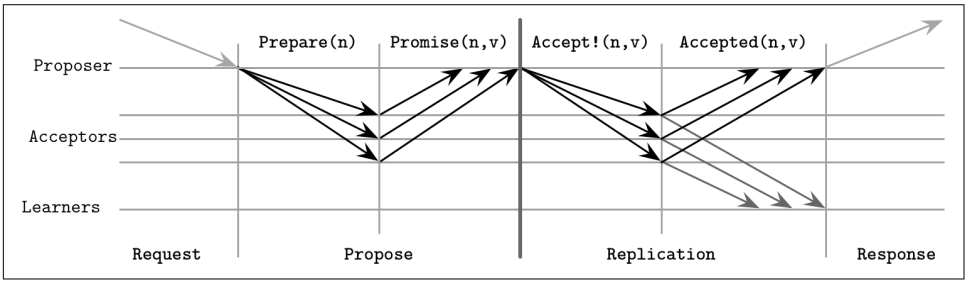

Figure 14-3. Paxos algorithm: normal execution

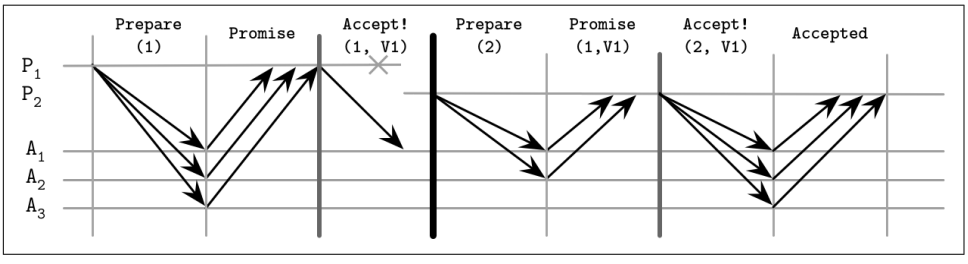

Figure 14-4. Paxos failure scenario: proposer failure, deciding on the old value

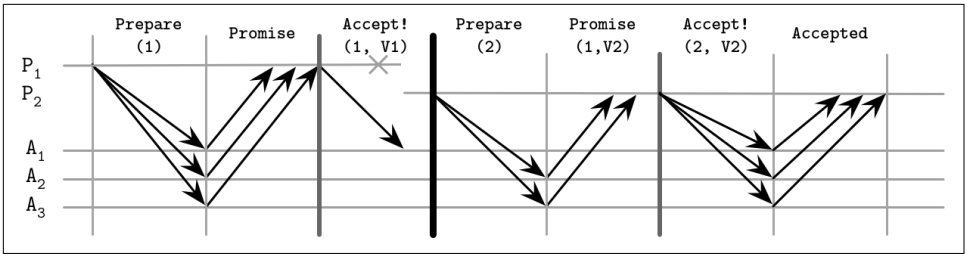

Figure 14-5. Paxos failure scenario: proposer failure, deciding on the new value

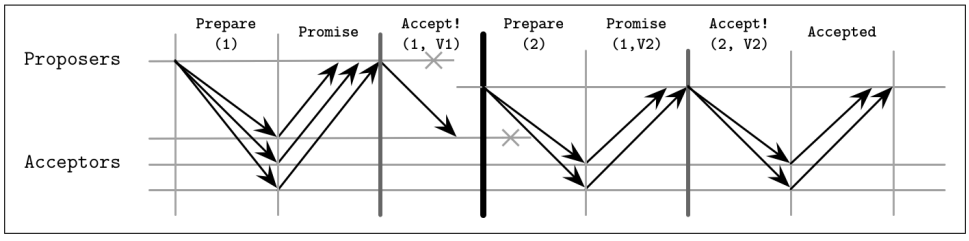

Figure 14-6. Paxos failure scenario: proposer failure, followed by the acceptor failure

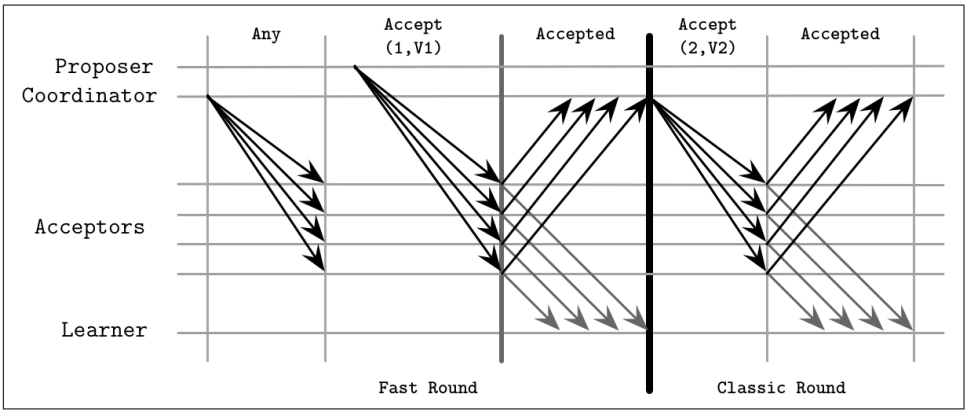

Figure 14-7. Fast Paxos algorithm: fast and classic rounds

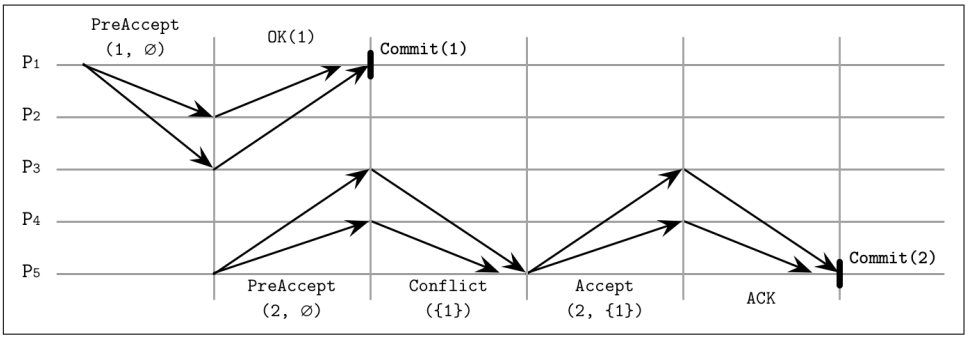

Figure 14-8. EPaxos algorithm run

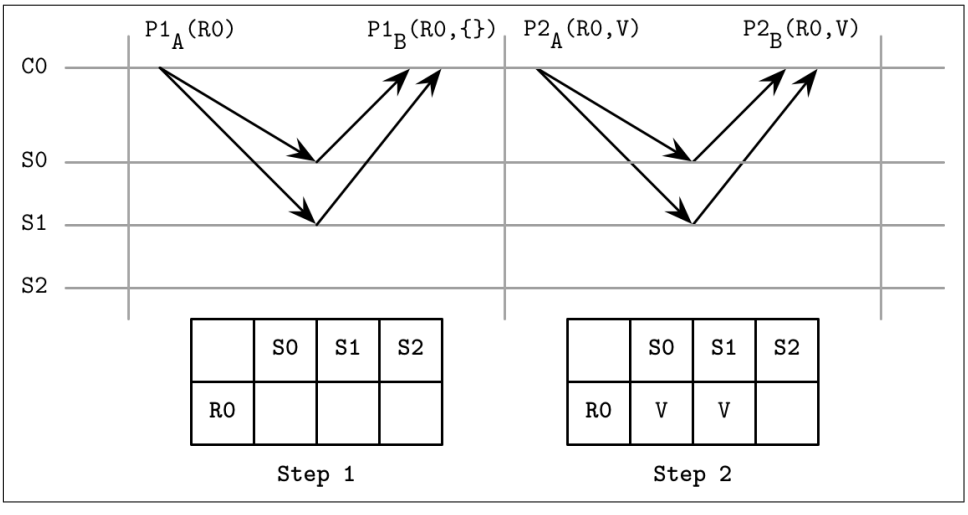

Figure 14-9. Generalization of Paxos

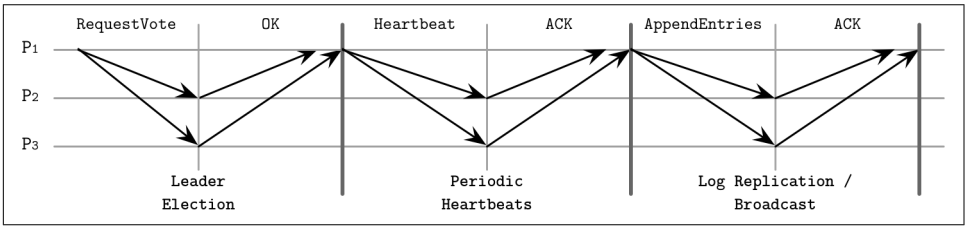

Figure 14-10. Raft consensus algorithm summary

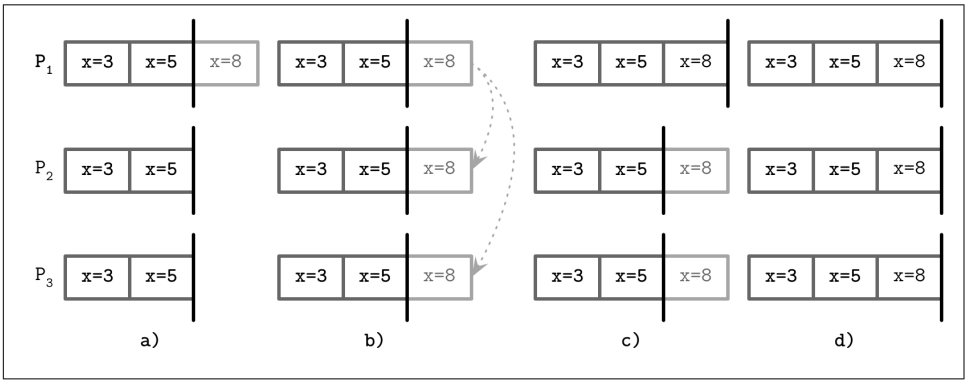

Figure 14-11. Procedure of a commit in Raft with P_1 as a leader

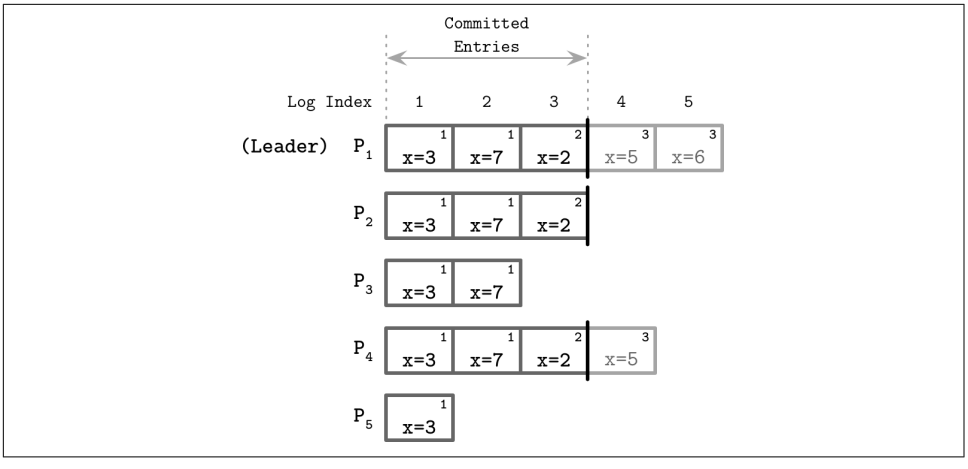

Figure 14-12. Raft state machine

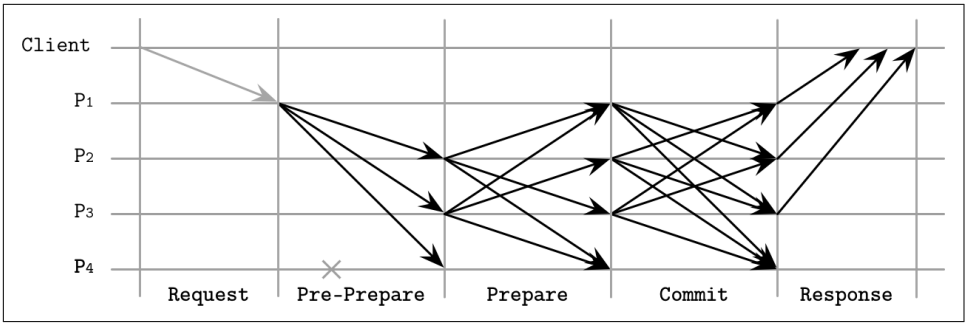

Figure 14-13. PBFT consensus, normal-case operation