PART I
Storage Engines

The primary job of any database management system is reliably storing data and making it available for users. We use databases as a primary source of data, helping us to share it between the different parts of our applications. Instead of finding a way to store and retrieve information and inventing a new way to organize data every time we create a new app, we use databases. This way we can concentrate on application logic instead of infrastructure.

Since the term *database management system* (DBMS) is quite bulky, throughout this book we use more compact terms, *database system* and *database*, to refer to the same concept.

Databases are modular systems and consist of multiple parts: a transport layer accepting requests, a query processor determining the most efficient way to run queries, an execution engine carrying out the operations, and a storage engine (see "DBMS Architecture" on page 8).

The *storage engine* (or database engine) is a software component of a database management system responsible for storing, retrieving, and managing data in memory and on disk, designed to capture a persistent, long-term memory of each node [REED78]. While databases can respond to complex queries, storage engines look at the data more granularly and offer a simple data manipulation API, allowing users to create, update, delete, and retrieve records. One way to look at this is that database management systems are applications built on top of storage engines, offering a schema, a query language, indexing, transactions, and many other useful features.

For flexibility, both keys and values can be arbitrary sequences of bytes with no prescribed form. Their sorting and representation semantics are defined in higher-level subsystems. For example, you can use `int32` (32-bit integer) as a key in one of the tables, and `ascii` (ASCII string) in the other; from the storage engine perspective both keys are just serialized entries.

Storage engines such as BerkeleyDB, LevelDB and its descendant RocksDB, LMDB and its descendant libmdbx, Sophia, HaloDB, and many others were developed independently from the database management systems they're now embedded into. Using pluggable storage engines has enabled database developers to bootstrap database systems using existing storage engines, and concentrate on the other subsystems.

At the same time, clear separation between database system components opens up an opportunity to switch between different engines, potentially better suited for particular use cases. For example, MySQL, a popular database management system, has several storage engines, including InnoDB, MyISAM, and RocksDB (in the MyRocks distribution). MongoDB allows switching between WiredTiger, In-Memory, and the (now-deprecated) MMAPv1 storage engines.

Comparing Databases

Your choice of database system may have long-term consequences. If there's a chance that a database is not a good fit because of performance problems, consistency issues, or operational challenges, it is better to find out about it earlier in the development cycle, since it can be nontrivial to migrate to a different system. In some cases, it may require substantial changes in the application code.

Every database system has strengths and weaknesses. To reduce the risk of an expensive migration, you can invest some time before you decide on a specific database to build confidence in its ability to meet your application's needs.

Trying to compare databases based on their components (e.g., which storage engine they use, how the data is shared, replicated, and distributed, etc.), their rank (an arbitrary popularity value assigned by consultancy agencies such as ThoughtWorks or database comparison websites such as DB-Engines or Database of Databases), or implementation language (C++, Java, or Go, etc.) can lead to invalid and premature conclusions. These methods can be used only for a high-level comparison and can be as coarse as choosing between HBase and SQLite, so even a superficial understanding of how each database works and what's inside it can help you land a more weighted conclusion.

Every comparison should start by clearly defining the goal, because even the slightest bias may completely invalidate the entire investigation. If you're searching for a database that would be a good fit for the workloads you have (or are planning to facilitate), the best thing you can do is to simulate these workloads against different

database systems, measure the performance metrics that are important for you, and compare results. Some issues, especially when it comes to performance and scalability, start showing only after some time or as the capacity grows. To detect potential problems, it is best to have long-running tests in an environment that simulates the real-world production setup as closely as possible.

Simulating real-world workloads not only helps you understand how the database performs, but also helps you learn how to operate, debug, and find out how friendly and helpful its community is. Database choice is always a combination of these factors, and performance often turns out *not* to be the most important aspect: it's usually much better to use a database that slowly saves the data than one that quickly loses it.

To compare databases, it's helpful to understand the use case in great detail and define the current and anticipated variables, such as:

- Schema and record sizes
- Number of clients
- Types of queries and access patterns
- Rates of the read and write queries
- Expected changes in any of these variables

Knowing these variables can help to answer the following questions:

- Does the database support the required queries?
- Is this database able to handle the amount of data we're planning to store?
- How many read and write operations can a single node handle?
- How many nodes should the system have?
- How do we expand the cluster given the expected growth rate?
- What is the maintenance process?

Having these questions answered, you can construct a test cluster and simulate your workloads. Most databases already have stress tools that can be used to reconstruct specific use cases. If there's no standard stress tool to generate realistic randomized workloads in the database ecosystem, it might be a red flag. If something prevents you from using default tools, you can try one of the existing general-purpose tools, or implement one from scratch.

If the tests show positive results, it may be helpful to familiarize yourself with the database code. Looking at the code, it is often useful to first understand the parts of the database, how to find the code for different components, and then navigate through those. Having even a rough idea about the database codebase helps you bet-

ter understand the log records it produces, its configuration parameters, and helps you find issues in the application that uses it and even in the database code itself.

It'd be great if we could use databases as black boxes and never have to take a look inside them, but the practice shows that sooner or later a bug, an outage, a performance regression, or some other problem pops up, and it's better to be prepared for it. If you know and understand database internals, you can reduce business risks and improve chances for a quick recovery.

One of the popular tools used for benchmarking, performance evaluation, and comparison is Yahoo! Cloud Serving Benchmark (YCSB). YCSB offers a framework and a common set of workloads that can be applied to different data stores. Just like anything generic, this tool should be used with caution, since it's easy to make wrong conclusions. To make a fair comparison and make an educated decision, it is necessary to invest enough time to understand the real-world conditions under which the database has to perform, and tailor benchmarks accordingly.

TPC-C Benchmark

The Transaction Processing Performance Council (TPC) has a set of benchmarks that database vendors use for comparing and advertising performance of their products. TPC-C is an online transaction processing (OLTP) benchmark, a mixture of read-only and update transactions that simulate common application workloads.

This benchmark concerns itself with the performance and correctness of executed concurrent transactions. The main performance indicator is *throughput*: the number of transactions the database system is able to process per minute. Executed transactions are required to preserve ACID properties and conform to the set of properties defined by the benchmark itself.

This benchmark does not concentrate on any particular business segment, but provides an abstract set of actions important for most of the applications for which OLTP databases are suitable. It includes several tables and entities such as warehouses, stock (inventory), customers and orders, specifying table layouts, details of transactions that can be performed against these tables, the minimum number of rows per table, and data durability constraints.

This doesn't mean that benchmarks can be used *only* to compare databases. Benchmarks can be useful to define and test details of the service-level agreement,[1] under-

1 The service-level agreement (or SLA) is a commitment by the service provider about the quality of provided services. Among other things, the SLA can include information about latency, throughput, jitter, and the number and frequency of failures.

standing system requirements, capacity planning, and more. The more knowledge you have about the database before using it, the more time you'll save when running it in production.

Choosing a database is a long-term decision, and it's best to keep track of newly released versions, understand what exactly has changed and why, and have an upgrade strategy. New releases usually contain improvements and fixes for bugs and security issues, but may introduce new bugs, performance regressions, or unexpected behavior, so testing new versions before rolling them out is also critical. Checking how database implementers were handling upgrades previously might give you a good idea about what to expect in the future. Past smooth upgrades do not guarantee that future ones will be as smooth, but complicated upgrades in the past might be a sign that future ones won't be easy, either.

Understanding Trade-Offs

As users, we can see how databases behave under different conditions, but when working on databases, we have to make choices that influence this behavior directly.

Designing a storage engine is definitely more complicated than just implementing a textbook data structure: there are many details and edge cases that are hard to get right from the start. We need to design the physical data layout and organize pointers, decide on the serialization format, understand how data is going to be garbage-collected, how the storage engine fits into the semantics of the database system as a whole, figure out how to make it work in a concurrent environment, and, finally, make sure we never lose any data, under any circumstances.

Not only there are many things to decide upon, but most of these decisions involve trade-offs. For example, if we save records in the order they were inserted into the database, we can store them quicker, but if we retrieve them in their lexicographical order, we have to re-sort them before returning results to the client. As you will see throughout this book, there are many different approaches to storage engine design, and every implementation has its own upsides and downsides.

When looking at different storage engines, we discuss their benefits and shortcomings. If there was an absolutely optimal storage engine for every conceivable use case, everyone would just use it. But since it does not exist, we need to choose wisely, based on the workloads and use cases we're trying to facilitate.

There are many storage engines, using all sorts of data structures, implemented in different languages, ranging from low-level ones, such as C, to high-level ones, such as Java. All storage engines face the same challenges and constraints. To draw a parallel with city planning, it is possible to build a city for a specific population and choose to build *up* or build *out*. In both cases, the same number of people will fit into the city, but these approaches lead to radically different lifestyles. When building the city up,

people live in apartments and population density is likely to lead to more traffic in a smaller area; in a more spread-out city, people are more likely to live in houses, but commuting will require covering larger distances.

Similarly, design decisions made by storage engine developers make them better suited for different things: some are optimized for low read or write latency, some try to maximize density (the amount of stored data per node), and some concentrate on operational simplicity.

You can find complete algorithms that can be used for the implementation and other additional references in the chapter summaries. Reading this book should make you well equipped to work productively with these sources and give you a solid understanding of the existing alternatives to concepts described there.

CHAPTER 1
Introduction and Overview

Database management systems can serve different purposes: some are used primarily for temporary *hot* data, some serve as a long-lived *cold* storage, some allow complex analytical queries, some only allow accessing values by the key, some are optimized to store time-series data, and some store large blobs efficiently. To understand differences and draw distinctions, we start with a short classification and overview, as this helps us to understand the scope of further discussions.

Terminology can sometimes be ambiguous and hard to understand without a complete context. For example, distinctions between *column* and *wide column* stores that have little or nothing to do with each other, or how *clustered* and *nonclustered indexes* relate to *index-organized tables*. This chapter aims to disambiguate these terms and find their precise definitions.

We start with an overview of database management system architecture (see "DBMS Architecture" on page 8), and discuss system components and their responsibilities. After that, we discuss the distinctions among the database management systems in terms of a storage medium (see "Memory- Versus Disk-Based DBMS" on page 10), and layout (see "Column- Versus Row-Oriented DBMS" on page 12).

These two groups do not present a full taxonomy of database management systems and there are many other ways they're classified. For example, some sources group DBMSs into three major categories:

Online transaction processing (OLTP) databases
: These handle a large number of user-facing requests and transactions. Queries are often predefined and short-lived.

Online analytical processing (OLAP) databases
: These handle complex aggregations. OLAP databases are often used for analytics and data warehousing, and are capable of handling complex, long-running ad hoc queries.

Hybrid transactional and analytical processing (HTAP)
: These databases combine properties of both OLTP and OLAP stores.

There are many other terms and classifications: key-value stores, relational databases, document-oriented stores, and graph databases. These concepts are not defined here, since the reader is assumed to have a high-level knowledge and understanding of their functionality. Because the concepts we discuss here are widely applicable and are used in most of the mentioned types of stores in some capacity, complete taxonomy is not necessary or important for further discussion.

Since Part I of this book focuses on the storage and indexing structures, we need to understand the high-level data organization approaches, and the relationship between the data and index files (see "Data Files and Index Files" on page 17).

Finally, in "Buffering, Immutability, and Ordering" on page 21, we discuss three techniques widely used to develop efficient storage structures and how applying these techniques influences their design and implementation.

DBMS Architecture

There's no common blueprint for database management system design. Every database is built slightly differently, and component boundaries are somewhat hard to see and define. Even if these boundaries exist on paper (e.g., in project documentation), in code seemingly independent components may be coupled because of performance optimizations, handling edge cases, or architectural decisions.

Sources that describe database management system architecture (for example, [HELLERSTEIN07], [WEIKUM01], [ELMASRI11], and [MOLINA08]), define components and relationships between them differently. The architecture presented in Figure 1-1 demonstrates some of the common themes in these representations.

Database management systems use a *client/server model*, where database system instances (*nodes*) take the role of servers, and application instances take the role of clients.

Client requests arrive through the *transport* subsystem. Requests come in the form of queries, most often expressed in some query language. The transport subsystem is also responsible for communication with other nodes in the database cluster.

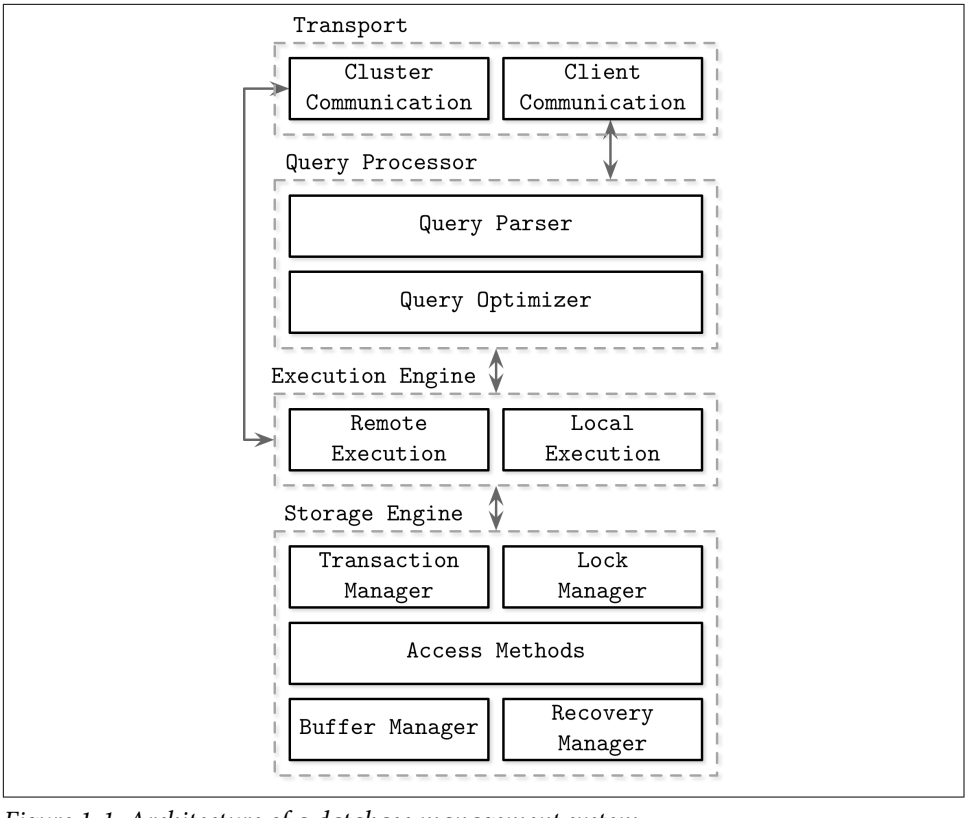

Figure 1-1. Architecture of a database management system

Upon receipt, the transport subsystem hands the query over to a *query processor*, which parses, interprets, and validates it. Later, access control checks are performed, as they can be done fully only after the query is interpreted.

The parsed query is passed to the *query optimizer*, which first eliminates impossible and redundant parts of the query, and then attempts to find the most efficient way to execute it based on internal statistics (index cardinality, approximate intersection size, etc.) and data placement (which nodes in the cluster hold the data and the costs associated with its transfer). The optimizer handles both relational operations required for query resolution, usually presented as a dependency tree, and optimizations, such as index ordering, cardinality estimation, and choosing access methods.

The query is usually presented in the form of an *execution plan* (or *query plan*): a sequence of operations that have to be carried out for its results to be considered complete. Since the same query can be satisfied using different execution plans that can vary in efficiency, the optimizer picks the best available plan.

The execution plan is handled by the *execution engine*, which collects the results of the execution of local and remote operations. *Remote execution* can involve writing and reading data to and from other nodes in the cluster, and replication.

Local queries (coming directly from clients or from other nodes) are executed by the *storage engine*. The storage engine has several components with dedicated responsibilities:

Transaction manager
: This manager schedules transactions and ensures they cannot leave the database in a logically inconsistent state.

Lock manager
: This manager locks on the database objects for the running transactions, ensuring that concurrent operations do not violate physical data integrity.

Access methods (storage structures)
: These manage access and organizing data on disk. Access methods include heap files and storage structures such as B-Trees (see "Ubiquitous B-Trees" on page 33) or LSM Trees (see "LSM Trees" on page 130).

Buffer manager
: This manager caches data pages in memory (see "Buffer Management" on page 81).

Recovery manager
: This manager maintains the operation log and restoring the system state in case of a failure (see "Recovery" on page 88).

Together, transaction and lock managers are responsible for concurrency control (see "Concurrency Control" on page 93): they guarantee logical and physical data integrity while ensuring that concurrent operations are executed as efficiently as possible.

Memory- Versus Disk-Based DBMS

Database systems store data in memory and on disk. *In-memory database management systems* (sometimes called *main memory DBMS*) store data *primarily* in memory and use the disk for recovery and logging. *Disk-based* DBMS hold *most* of the data on disk and use memory for caching disk contents or as a temporary storage. Both types of systems use the disk to a certain extent, but main memory databases store their contents almost exclusively in RAM.

Accessing memory has been and remains several orders of magnitude faster than accessing disk,[1] so it is compelling to use memory as the primary storage, and it becomes more economically feasible to do so as memory prices go down. However, RAM prices still remain high compared to persistent storage devices such as SSDs and HDDs.

Main memory database systems are different from their disk-based counterparts not only in terms of a primary storage medium, but also in which data structures, organization, and optimization techniques they use.

Databases using memory as a primary data store do this mainly because of performance, comparatively low access costs, and access granularity. Programming for main memory is also significantly simpler than doing so for the disk. Operating systems abstract memory management and allow us to think in terms of allocating and freeing arbitrarily sized memory chunks. On disk, we have to manage data references, serialization formats, freed memory, and fragmentation manually.

The main limiting factors on the growth of in-memory databases are RAM volatility (in other words, lack of durability) and costs. Since RAM contents are not persistent, software errors, crashes, hardware failures, and power outages can result in data loss. There are ways to ensure durability, such as uninterrupted power supplies and battery-backed RAM, but they require additional hardware resources and operational expertise. In practice, it all comes down to the fact that disks are easier to maintain and have significantly lower prices.

The situation is likely to change as the availability and popularity of Non-Volatile Memory (NVM) [ARULRAJ17] technologies grow. NVM storage reduces or completely eliminates (depending on the exact technology) asymmetry between read and write latencies, further improves read and write performance, and allows byte-addressable access.

Durability in Memory-Based Stores

In-memory database systems maintain backups on disk to provide durability and prevent loss of the volatile data. Some databases store data exclusively in memory, without any durability guarantees, but we do not discuss them in the scope of this book.

Before the operation can be considered complete, its results have to be written to a sequential log file. We discuss write-ahead logs in more detail in "Recovery" on page 88. To avoid replaying complete log contents during startup or after a crash, in-memory stores maintain a *backup copy*. The backup copy is maintained as a sorted

1 You can find a visualization and comparison of disk, memory access latencies, and many other relevant numbers over the years at *https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html*.

disk-based structure, and modifications to this structure are often asynchronous (decoupled from client requests) and applied in batches to reduce the number of I/O operations. During recovery, database contents can be restored from the backup and logs.

Log records are usually applied to backup in batches. After the batch of log records is processed, backup holds a database *snapshot* for a specific point in time, and log contents up to this point can be discarded. This process is called *checkpointing*. It reduces recovery times by keeping the disk-resident database most up-to-date with log entries without requiring clients to block until the backup is updated.

It is unfair to say that the in-memory database is the equivalent of an on-disk database with a huge page cache (see "Buffer Management" on page 81). Even though pages are *cached* in memory, serialization format and data layout incur additional overhead and do not permit the same degree of optimization that in-memory stores can achieve.

Disk-based databases use specialized storage structures, optimized for disk access. In memory, pointers can be followed comparatively quickly, and random memory access is significantly faster than the random disk access. Disk-based storage structures often have a form of wide and short trees (see "Trees for Disk-Based Storage" on page 28), while memory-based implementations can choose from a larger pool of data structures and perform optimizations that would otherwise be impossible or difficult to implement on disk [MOLINA92]. Similarly, handling variable-size data on disk requires special attention, while in memory it's often a matter of referencing the value with a pointer.

For some use cases, it is reasonable to assume that an entire dataset is going to fit in memory. Some datasets are bounded by their real-world representations, such as student records for schools, customer records for corporations, or inventory in an online store. Each record takes up not more than a few Kb, and their number is limited.

Column- Versus Row-Oriented DBMS

Most database systems store a set of *data records*, consisting of *columns* and *rows* in *tables*. *Field* is an intersection of a column and a row: a single value of some type. Fields belonging to the same column usually have the same data type. For example, if we define a table holding user records, all names would be of the same type and belong to the same column. A collection of values that belong logically to the same record (usually identified by the key) constitutes a row.

One of the ways to classify databases is by how the data is stored on disk: row- or column-wise. Tables can be partitioned either horizontally (storing values belonging

to the same row together), or vertically (storing values belonging to the same column together). Figure 1-2 depicts this distinction: (a) shows the values partitioned column-wise, and (b) shows the values partitioned row-wise.

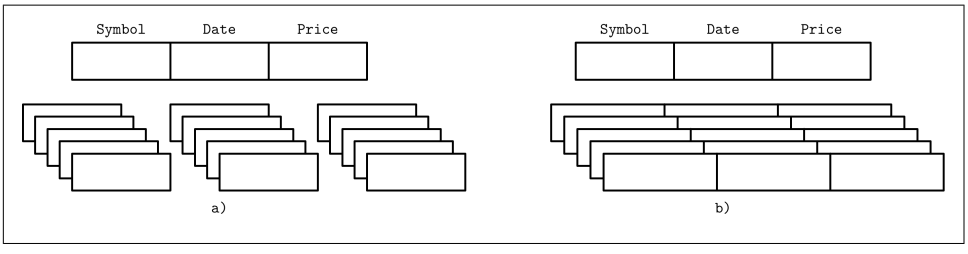

Figure 1-2. Data layout in column- and row-oriented stores

Examples of row-oriented database management systems are abundant: MySQL, PostgreSQL, and most of the traditional relational databases. The two pioneer open source column-oriented stores are MonetDB and C-Store (C-Store is an open source predecessor to Vertica).

Row-Oriented Data Layout

Row-oriented database management systems store data in records or *rows*. Their layout is quite close to the tabular data representation, where every row has the same set of fields. For example, a row-oriented database can efficiently store user entries, holding names, birth dates, and phone numbers:

```
| ID | Name  | Birth Date  | Phone Number   |
| 10 | John  | 01 Aug 1981 | +1 111 222 333 |
| 20 | Sam   | 14 Sep 1988 | +1 555 888 999 |
| 30 | Keith | 07 Jan 1984 | +1 333 444 555 |
```

This approach works well for cases where several fields constitute the record (name, birth date, and a phone number) uniquely identified by the key (in this example, a monotonically incremented number). All fields representing a single user record are often read together. When creating records (for example, when the user fills out a registration form), we write them together as well. At the same time, each field can be modified individually.

Since row-oriented stores are most useful in scenarios when we have to access data by row, storing entire rows together improves spatial locality[2] [DENNING68].

Because data on a persistent medium such as a disk is typically accessed block-wise (in other words, a minimal unit of disk access is a block), a single block will contain

2 Spatial locality is one of the Principles of Locality, stating that if a memory location is accessed, its nearby memory locations will be accessed in the near future.

data for all columns. This is great for cases when we'd like to access an entire user record, but makes queries accessing individual fields of multiple user records (for example, queries fetching only the phone numbers) more expensive, since data for the other fields will be paged in as well.

Column-Oriented Data Layout

Column-oriented database management systems partition data *vertically* (i.e., by column) instead of storing it in rows. Here, values for the same column are stored contiguously on disk (as opposed to storing rows contiguously as in the previous example). For example, if we store historical stock market prices, price quotes are stored together. Storing values for different columns in separate files or file segments allows efficient queries by column, since they can be read in one pass rather than consuming entire rows and discarding data for columns that weren't queried.

Column-oriented stores are a good fit for analytical workloads that compute aggregates, such as finding trends, computing average values, etc. Processing complex aggregates can be used in cases when logical records have multiple fields, but some of them (in this case, price quotes) have different importance and are often consumed together.

From a logical perspective, the data representing stock market price quotes can still be expressed as a table:

```
| ID | Symbol | Date        | Price     |
| 1  | DOW    | 08 Aug 2018 | 24,314.65 |
| 2  | DOW    | 09 Aug 2018 | 24,136.16 |
| 3  | S&P    | 08 Aug 2018 | 2,414.45  |
| 4  | S&P    | 09 Aug 2018 | 2,232.32  |
```

However, the physical column-based database layout looks entirely different. Values belonging to the same row are stored closely together:

```
Symbol: 1:DOW; 2:DOW; 3:S&P; 4:S&P
Date:   1:08 Aug 2018; 2:09 Aug 2018; 3:08 Aug 2018; 4:09 Aug 2018
Price:  1:24,314.65; 2:24,136.16; 3:2,414.45; 4:2,232.32
```

To reconstruct data tuples, which might be useful for joins, filtering, and multirow aggregates, we need to preserve some metadata on the column level to identify which data points from other columns it is associated with. If you do this explicitly, each value will have to hold a key, which introduces duplication and increases the amount of stored data. Some column stores use implicit identifiers (*virtual IDs*) instead and use the position of the value (in other words, its offset) to map it back to the related values [ABADI13].

During the last several years, likely due to a rising demand to run complex analytical queries over growing datasets, we've seen many new column-oriented file formats

such as Apache Parquet, Apache ORC, RCFile, as well as column-oriented stores, such as Apache Kudu, ClickHouse, and many others [ROY12].

Distinctions and Optimizations

It is not sufficient to say that distinctions between row and column stores are only in the way the data is stored. Choosing the data layout is just one of the steps in a series of possible optimizations that columnar stores are targeting.

Reading multiple values for the same column in one run significantly improves cache utilization and computational efficiency. On modern CPUs, vectorized instructions can be used to process multiple data points with a single CPU instruction[3] [DREPPER07].

Storing values that have the same data type together (e.g., numbers with other numbers, strings with other strings) offers a better compression ratio. We can use different compression algorithms depending on the data type and pick the most effective compression method for each case.

To decide whether to use a column- or a row-oriented store, you need to understand your *access patterns*. If the read data is consumed in records (i.e., most or all of the columns are requested) and the workload consists mostly of point queries and range scans, the row-oriented approach is likely to yield better results. If scans span many rows, or compute aggregate over a subset of columns, it is worth considering a column-oriented approach.

Wide Column Stores

Column-oriented databases should not be mixed up with *wide column stores*, such as BigTable or HBase, where data is represented as a multidimensional map, columns are grouped into *column families* (usually storing data of the same type), and inside each column family, data is stored row-wise. This layout is best for storing data retrieved by a key or a sequence of keys.

A canonical example from the Bigtable paper [CHANG06] is a Webtable. A Webtable stores snapshots of web page contents, their attributes, and the relations among them at a specific timestamp. Pages are identified by the reversed URL, and all attributes (such as page *content* and *anchors*, representing links between pages) are identified by the timestamps at which these snapshots were taken. In a simplified way, it can be represented as a nested map, as Figure 1-3 shows.

3 Vectorized instructions, or Single Instruction Multiple Data (SIMD), describes a class of CPU instructions that perform the same operation on multiple data points.

```
{
    "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

Data is stored in a multidimensional sorted map with hierarchical indexes: we can locate the data related to a specific web page by its reversed URL and its contents or anchors by the timestamp. Each row is indexed by its *row key*. Related columns are grouped together in *column families*—`contents` and `anchor` in this example—which are stored on disk separately. Each column inside a column family is identified by the *column key*, which is a combination of the column family name and a qualifier (`html`, `cnnsi.com`, `my.look.ca` in this example). Column families store multiple versions of data by timestamp. This layout allows us to quickly locate the higher-level entries (web pages, in this case) and their parameters (versions of content and links to the other pages).

While it is useful to understand the conceptual representation of wide column stores, their physical layout is somewhat different. A schematic representation of the data layout in column families is shown in Figure 1-4: column families are stored separately, but in each column family, the data belonging to the same key is stored together.

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

Data Files and Index Files

The primary goal of a database system is to store data and to allow quick access to it. But how is the data organized? Why do we need a database management system and not just a bunch of files? How does file organization improve efficiency?

Database systems do use files for storing the data, but instead of relying on filesystem hierarchies of directories and files for locating records, they compose files using implementation-specific formats. The main reasons to use specialized file organization over flat files are:

Storage efficiency
: Files are organized in a way that minimizes storage overhead per stored data record.

Access efficiency
: Records can be located in the smallest possible number of steps.

Update efficiency
: Record updates are performed in a way that minimizes the number of changes on disk.

Database systems store *data records*, consisting of multiple fields, in tables, where each table is usually represented as a separate file. Each record in the table can be looked up using a *search key*. To locate a record, database systems use *indexes*: auxiliary data structures that allow it to efficiently locate data records without scanning an entire table on every access. Indexes are built using a subset of fields identifying the record.

A database system usually separates *data files* and *index files*: data files store data records, while index files store record metadata and use it to locate records in data

files. Index files are typically smaller than the data files. Files are partitioned into *pages*, which typically have the size of a single or multiple disk blocks. Pages can be organized as sequences of records or as a *slotted pages* (see "Slotted Pages" on page 52).

New records (insertions) and updates to the existing records are represented by key/value pairs. Most modern storage systems *do not* delete data from pages explicitly. Instead, they use *deletion markers* (also called *tombstones*), which contain deletion metadata, such as a key and a timestamp. Space occupied by the records *shadowed* by their updates or deletion markers is reclaimed during garbage collection, which reads the pages, writes the live (i.e., nonshadowed) records to the new place, and discards the shadowed ones.

Data Files

Data files (sometimes called *primary files*) can be implemented as *index-organized tables* (IOT), *heap-organized tables* (heap files), or *hash-organized tables* (hashed files).

Records in heap files are not required to follow any particular order, and most of the time they are placed in a write order. This way, no additional work or file reorganization is required when new pages are appended. Heap files require additional index structures, pointing to the locations where data records are stored, to make them searchable.

In hashed files, records are stored in buckets, and the hash value of the key determines which bucket a record belongs to. Records in the bucket can be stored in append order or sorted by key to improve lookup speed.

Index-organized tables (IOTs) store data records in the index itself. Since records are stored in key order, range scans in IOTs can be implemented by sequentially scanning its contents.

Storing data records in the index allows us to reduce the number of disk seeks by at least one, since after traversing the index and locating the searched key, we do not have to address a separate file to find the associated data record.

When records are stored in a separate file, index files hold *data entries*, uniquely identifying data records and containing enough information to locate them in the data file. For example, we can store file *offsets* (sometimes called *row locators*), locations of data records in the data file, or bucket IDs in the case of hash files. In index-organized tables, data entries hold actual data records.

Index Files

An index is a structure that organizes data records on disk in a way that facilitates efficient retrieval operations. Index files are organized as specialized structures that

map keys to locations in data files where the records identified by these keys (in the case of heap files) or primary keys (in the case of index-organized tables) are stored.

An index on a *primary* (data) file is called the *primary index*. However, in most cases we can also assume that the primary index is built over a primary key or a set of keys identified as primary. All other indexes are called *secondary*.

Secondary indexes can point directly to the data record, or simply store its primary key. A pointer to a data record can hold an offset to a heap file or an index-organized table. Multiple secondary indexes can point to the same record, allowing a single data record to be identified by different fields and located through different indexes. While primary index files hold a unique entry per search key, secondary indexes may hold several entries per search key [MOLINA08].

If the order of data records follows the search key order, this index is called *clustered* (also known as clustering). Data records in the clustered case are usually stored in the same file or in a *clustered file*, where the key order is preserved. If the data is stored in a separate file, and its order does not follow the key order, the index is called *nonclustered* (sometimes called unclustered).

Figure 1-5 shows the difference between the two approaches:

- a) Two indexes reference data entries directly from secondary index files.
- b) A secondary index goes through the indirection layer of a primary index to locate the data entries.

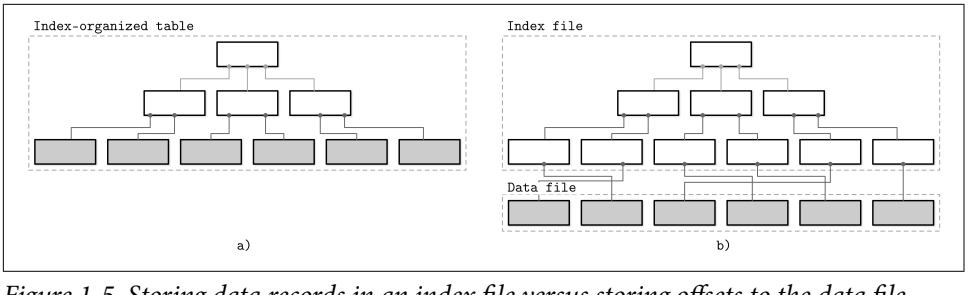

Figure 1-5. Storing data records in an index file versus storing offsets to the data file (index segments shown in white; segments holding data records shown in gray)

Index-organized tables store information in index order and are clustered by definition. Primary indexes are *most often* clustered. Secondary indexes are nonclustered by definition, since they're used to facilitate access by keys other than the primary one. Clustered indexes can be both index-organized or have separate index and data files.

Many database systems have an inherent and explicit *primary key*, a set of columns that uniquely identify the database record. In cases when the primary key is not specified, the storage engine can create an *implicit* primary key (for example, MySQL InnoDB adds a new auto-increment column and fills in its values automatically).

This terminology is used in different kinds of database systems: relational database systems (such as MySQL and PostgreSQL), Dynamo-based NoSQL stores (such as Apache Cassandra and in Riak), and document stores (such as MongoDB). There can be some project-specific naming, but most often there's a clear mapping to this terminology.

Primary Index as an Indirection

There are different opinions in the database community on whether data records should be referenced directly (through file offset) or via the primary key index.[4]

Both approaches have their pros and cons and are better discussed in the scope of a complete implementation. By referencing data directly, we can reduce the number of disk seeks, but have to pay a cost of updating the pointers whenever the record is updated or relocated during a maintenance process. Using indirection in the form of a primary index allows us to reduce the cost of pointer updates, but has a higher cost on a read path.

Updating just a couple of indexes might work if the workload mostly consists of reads, but this approach does not work well for write-heavy workloads with multiple indexes. To reduce the costs of pointer updates, instead of payload offsets, some implementations use primary keys for indirection. For example, MySQL InnoDB uses a primary index and performs two lookups: one in the secondary index, and one in a primary index when performing a query [TARIQ11]. This adds an overhead of a primary index lookup instead of following the offset directly from the secondary index.

Figure 1-6 shows how the two approaches are different:

- a) Two indexes reference data entries directly from secondary index files.
- b) A secondary index goes through the indirection layer of a primary index to locate the data entries.

4 The original post that has stirred up the discussion was controversial and one-sided, but you can refer to the presentation comparing MySQL and PostgreSQL index and storage formats, which references the original source as well.

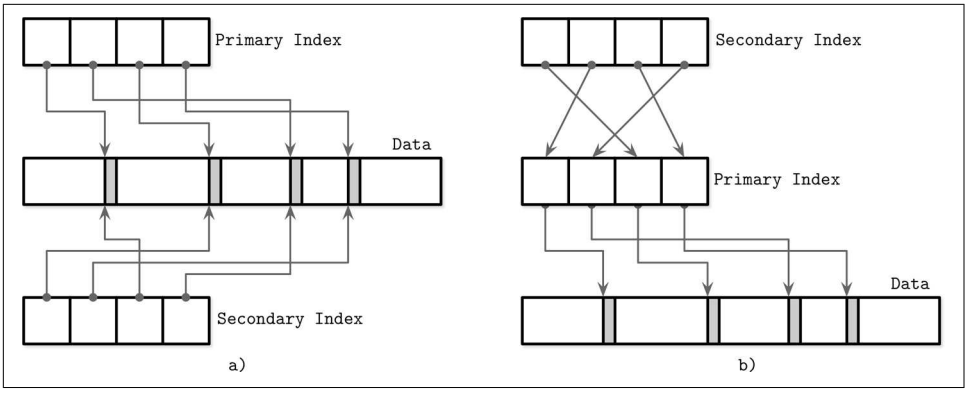

Figure 1-6. Referencing data tuples directly (a) versus using a primary index as indirection (b)

It is also possible to use a hybrid approach and store both data file offsets and primary keys. First, you check if the data offset is still valid and pay the extra cost of going through the primary key index if it has changed, updating the index file after finding a new offset.

Buffering, Immutability, and Ordering

A storage engine is based on some data structure. However, these structures do not describe the semantics of caching, recovery, transactionality, and other things that storage engines add on top of them.

In the next chapters, we will start the discussion with B-Trees (see "Ubiquitous B-Trees" on page 33) and try to understand why there are so many B-Tree variants, and why new database storage structures keep emerging.

Storage structures have three common variables: they use *buffering* (or avoid using it), use *immutable* (or mutable) files, and store values *in order* (or out of order). Most of the distinctions and optimizations in storage structures discussed in this book are related to one of these three concepts.

Buffering
: This defines whether or not the storage structure chooses to collect a certain amount of data in memory before putting it on disk. Of course, every on-disk structure has to use buffering to *some* degree, since the smallest unit of data transfer to and from the disk is a *block*, and it is desirable to write full blocks. Here, we're talking about avoidable buffering, something storage engine implementers *choose* to do. One of the first optimizations we discuss in this book is adding in-memory buffers to B-Tree nodes to amortize I/O costs (see "Lazy B-Trees" on page 114). However, this is not the only way we can apply buffering.

For example, two-component LSM Trees (see "Two-component LSM Tree" on page 132), despite their similarities with B-Trees, use buffering in an entirely different way, and combine buffering with immutability.

Mutability (or immutability)
: This defines whether or not the storage structure reads parts of the file, updates them, and writes the updated results at the same location in the file. Immutable structures are *append-only*: once written, file contents are not modified. Instead, modifications are appended to the end of the file. There are other ways to implement immutability. One of them is *copy-on-write* (see "Copy-on-Write" on page 112), where the modified page, holding the updated version of the record, is written to the *new* location in the file, instead of its original location. Often the distinction between LSM and B-Trees is drawn as immutable against in-place update storage, but there are structures (for example, "Bw-Trees" on page 120) that are inspired by B-Trees but are immutable.

Ordering
: This is defined as whether or not the *data records* are stored in the key order in the pages on disk. In other words, the keys that sort closely are stored in contiguous segments on disk. Ordering often defines whether or not we can efficiently scan the *range* of records, not only locate the individual data records. Storing data out of order (most often, in insertion order) opens up for some write-time optimizations. For example, Bitcask (see "Bitcask" on page 153) and WiscKey (see "WiscKey" on page 154) store data records directly in append-only files.

Of course, a brief discussion of these three concepts is not enough to show their power, and we'll continue this discussion throughout the rest of the book.

Summary

In this chapter, we've discussed the architecture of a database management system and covered its primary components.

To highlight the importance of disk-based structures and their difference from in-memory ones, we discussed memory- and disk-based stores. We came to the conclusion that disk-based structures are important for both types of stores, but are used for different purposes.

To understand how access patterns influence database system design, we discussed column- and row-oriented database management systems and the primary factors that set them apart from each other. To start a conversation about *how the data is stored*, we covered data and index files.

Lastly, we introduced three core concepts: buffering, immutability, and ordering. We will use them throughout this book to highlight properties of the storage engines that use them.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Database architecture

Hellerstein, Joseph M., Michael Stonebraker, and James Hamilton. 2007. "Architecture of a Database System." *Foundations and Trends in Databases* 1, no. 2 (February): 141-259. *https://doi.org/10.1561/1900000002*.

Column-oriented DBMS

Abadi, Daniel, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. Hanover, MA: Now Publishers Inc.

In-memory DBMS

Faerber, Frans, Alfons Kemper, and Per-Åke Alfons. 2017. *Main Memory Database Systems*. Hanover, MA: Now Publishers Inc.

CHAPTER 2
B-Tree Basics

In the previous chapter, we separated storage structures in two groups: *mutable* and *immutable* ones, and identified immutability as one of the core concepts influencing their design and implementation. Most of the mutable storage structures use an *in-place update* mechanism. During insert, delete, or update operations, data records are updated directly in their locations in the target file.

Storage engines often allow multiple versions of the same data record to be present in the database; for example, when using multiversion concurrency control (see "Multi-version Concurrency Control" on page 99) or slotted page organization (see "Slotted Pages" on page 52). For the sake of simplicity, for now we assume that each key is associated only with one data record, which has a unique location.

One of the most popular storage structures is a B-Tree. Many open source database systems are B-Tree based, and over the years they've proven to cover the majority of use cases.

B-Trees are not a recent invention: they were introduced by Rudolph Bayer and Edward M. McCreight back in 1971 and gained popularity over the years. By 1979, there were already quite a few variants of B-Trees. Douglas Comer collected and systematized some of them [COMER79].

Before we dive into B-Trees, let's first talk about why we should consider alternatives to traditional search trees, such as, for example, binary search trees, 2-3-Trees, and AVL Trees [KNUTH98]. For that, let's recall what binary search trees are.

Binary Search Trees

A *binary search tree* (BST) is a sorted in-memory data structure, used for efficient key-value lookups. BSTs consist of multiple nodes. Each tree node is represented by a key, a value associated with this key, and two child pointers (hence the name binary). BSTs start from a single node, called a *root node*. There can be only one root in the tree. Figure 2-1 shows an example of a binary search tree.

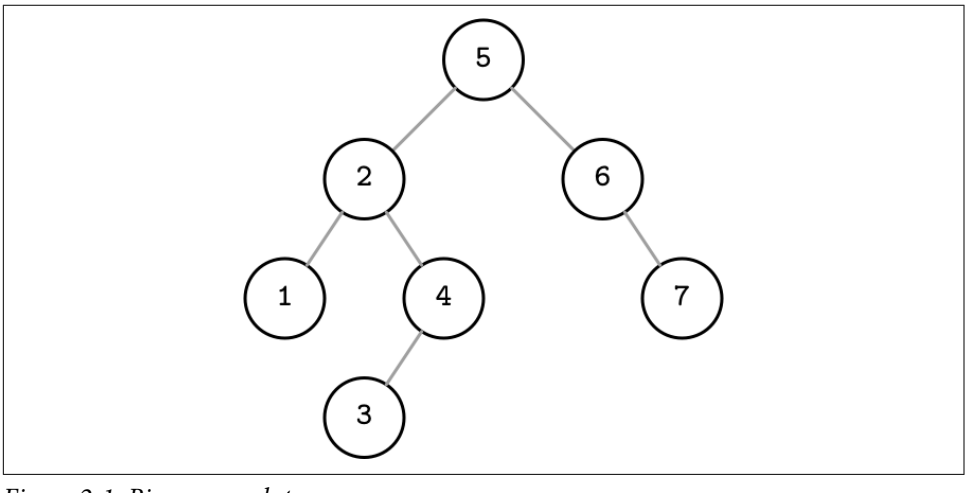

Figure 2-1. Binary search tree

Each node splits the search space into left and right *subtrees*, as Figure 2-2 shows: a node key is *greater than* any key stored in its left subtree and *less than* any key stored in its right subtree [SEDGEWICK11].

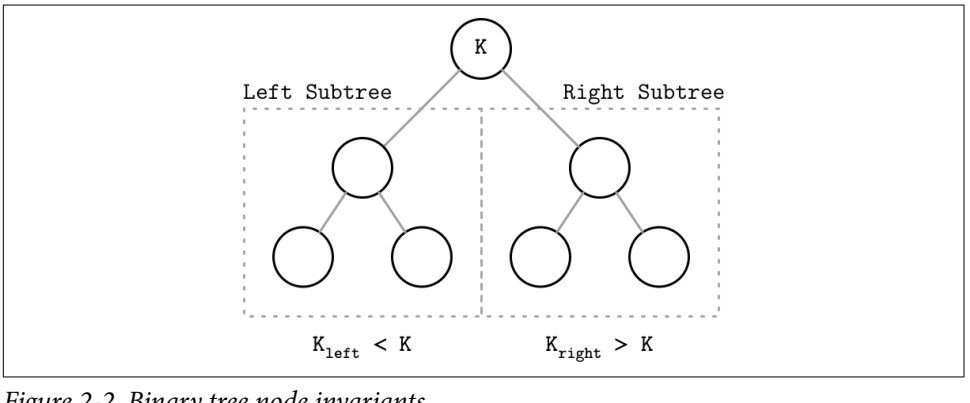

Figure 2-2. Binary tree node invariants

Following left pointers from the root of the tree down to the leaf level (the level where nodes have no children) locates the node holding the smallest key within the tree and a value associated with it. Similarly, following right pointers locates the node holding the largest key within the tree and a value associated with it. Values are allowed to be stored in all nodes in the tree. Searches start from the root node, and may terminate before reaching the bottom level of the tree if the searched key was found on a higher level.

Tree Balancing

Insert operations do not follow any specific pattern, and element insertion might lead to the situation where the tree is unbalanced (i.e., one of its branches is longer than the other one). The worst-case scenario is shown in Figure 2-3 (b), where we end up with a *pathological* tree, which looks more like a linked list, and instead of desired logarithmic complexity, we get linear, as illustrated in Figure 2-3 (a).

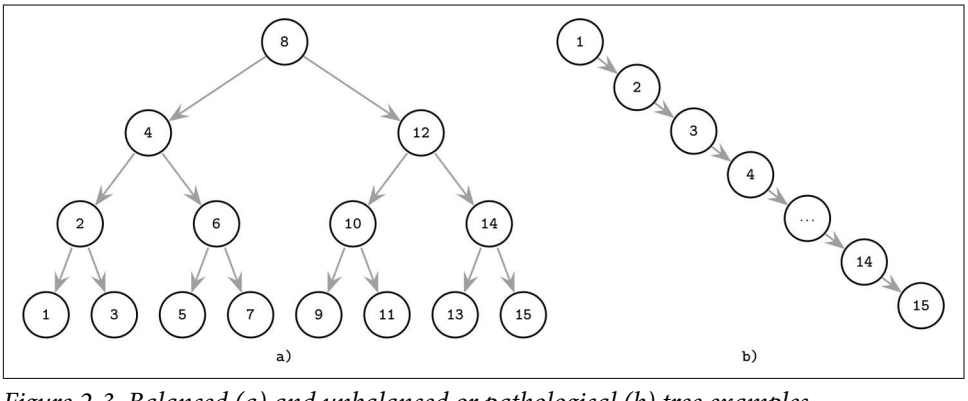

Figure 2-3. Balanced (a) and unbalanced or pathological (b) tree examples

This example might slightly exaggerate the problem, but it illustrates why the tree needs to be balanced: even though it's somewhat unlikely that all the items end up on one side of the tree, at least some of them certainly will, which will significantly slow down searches.

The *balanced* tree is defined as one that has a height of $\log_2 N$, where `N` is the total number of items in the tree, and the difference in height between the two subtrees is not greater than one[1] [KNUTH98]. Without balancing, we lose performance benefits of the binary search tree structure, and allow insertions and deletions order to determine tree shape.

1 This property is imposed by AVL Trees and several other data structures. More generally, binary search trees keep the difference in heights between subtrees within a small constant factor.

In the balanced tree, following the left or right node pointer reduces the search space in half on average, so lookup complexity is logarithmic: $O(\log_2 N)$. If the tree is not balanced, worst-case complexity goes up to $O(N)$, since we might end up in the situation where all elements end up on one side of the tree.

Instead of adding new elements to one of the tree branches and making it longer, while the other one remains empty (as shown in Figure 2-3 (b)), the tree is *balanced* after each operation. Balancing is done by reorganizing nodes in a way that minimizes tree height and keeps the number of nodes on each side within bounds.

One of the ways to keep the tree balanced is to perform a rotation step after nodes are added or removed. If the insert operation leaves a branch unbalanced (two consecutive nodes in the branch have only one child), we can *rotate* nodes around the middle one. In the example shown in Figure 2-4, during rotation the middle node (3), known as a rotation *pivot*, is promoted one level higher, and its parent becomes its right child.

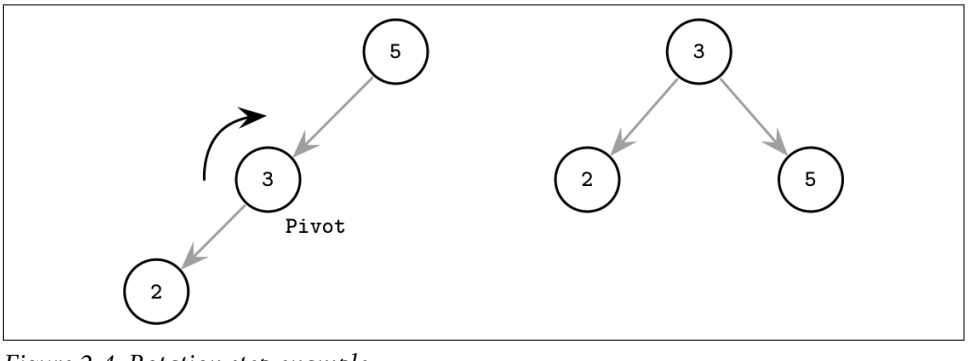

Figure 2-4. Rotation step example

Trees for Disk-Based Storage

As previously mentioned, unbalanced trees have a worst-case complexity of $O(N)$. Balanced trees give us an average $O(\log_2 N)$. At the same time, due to low *fanout* (fanout is the maximum allowed number of children per node), we have to perform balancing, relocate nodes, and update pointers rather frequently. Increased maintenance costs make BSTs impractical as on-disk data structures [NIEVERGELT74].

If we wanted to maintain a BST on disk, we'd face several problems. One problem is locality: since elements are added in random order, there's no guarantee that a newly created node is written close to its parent, which means that node child pointers may span across several disk pages. We can improve the situation to a certain extent by modifying the tree layout and using paged binary trees (see "Paged Binary Trees" on page 33).

Another problem, closely related to the cost of following child pointers, is tree height. Since binary trees have a fanout of just two, height is a binary logarithm of the number of the elements in the tree, and we have to perform $O(\log_2 N)$ seeks to locate the searched element and, subsequently, perform the same number of disk transfers. 2-3-Trees and other low-fanout trees have a similar limitation: while they are useful as in-memory data structures, small node size makes them impractical for external storage [COMER79].

A naive on-disk BST implementation would require as many disk seeks as comparisons, since there's no built-in concept of locality. This sets us on a course to look for a data structure that would exhibit this property.

Considering these factors, a version of the tree that would be better suited for disk implementation has to exhibit the following properties:

- *High fanout* to improve locality of the neighboring keys.
- *Low height* to reduce the number of seeks during traversal.

Fanout and height are inversely correlated: the higher the fanout, the lower the height. If fanout is high, each node can hold more children, reducing the number of nodes and, subsequently, reducing height.

Disk-Based Structures

We've talked about memory and disk-based storage (see "Memory- Versus Disk-Based DBMS" on page 10) in general terms. We can draw the same distinction for specific data structures: some are better suited to be used on disk and some work better in memory.

As we have discussed, not every data structure that satisfies space and complexity requirements can be effectively used for on-disk storage. Data structures used in databases have to be adapted to account for persistent medium limitations.

On-disk data structures are often used when the amounts of data are so large that keeping an entire dataset in memory is impossible or not feasible. Only a fraction of the data can be *cached* in memory at any time, and the rest has to be stored on disk in a manner that allows efficiently accessing it.

Hard Disk Drives

Most traditional algorithms were developed when spinning disks were the most widespread persistent storage medium, which significantly influenced their design. Later, new developments in storage media, such as flash drives, inspired new algorithms and modifications to the existing ones, exploiting the capabilities of the new hardware. These days, new types of data structures are emerging, optimized to work with nonvolatile byte-addressable storage (for example, [XIA17] [KANNAN18]).

On spinning disks, *seeks* increase costs of random reads because they require disk rotation and mechanical head movements to position the read/write head to the desired location. However, once the expensive part is done, reading or writing contiguous bytes (i.e., sequential operations) is *relatively* cheap.

The smallest transfer unit of a spinning drive is a *sector*, so when some operation is performed, at least an entire sector can be read or written. Sector sizes typically range from 512 bytes to 4 Kb.

Head positioning is the most expensive part of an operation on the HDD. This is one of the reasons we often hear about the positive effects of *sequential* I/O: reading and writing contiguous memory segments from disk.

Solid State Drives

Solid state drives (SSDs) do not have moving parts: there's no disk that spins, or head that has to be positioned for the read. A typical SSD is built of *memory cells*, connected into *strings* (typically 32 to 64 cells per string), strings are combined into *arrays*, arrays are combined into *pages*, and pages are combined into *blocks* [LARRIVEE15].

Depending on the exact technology used, a cell can hold one or multiple bits of data. Pages vary in size between devices, but typically their sizes range from 2 to 16 Kb. Blocks typically contain 64 to 512 pages. Blocks are organized into planes and, finally, planes are placed on a *die*. SSDs can have one or more dies. Figure 2-5 shows this hierarchy.

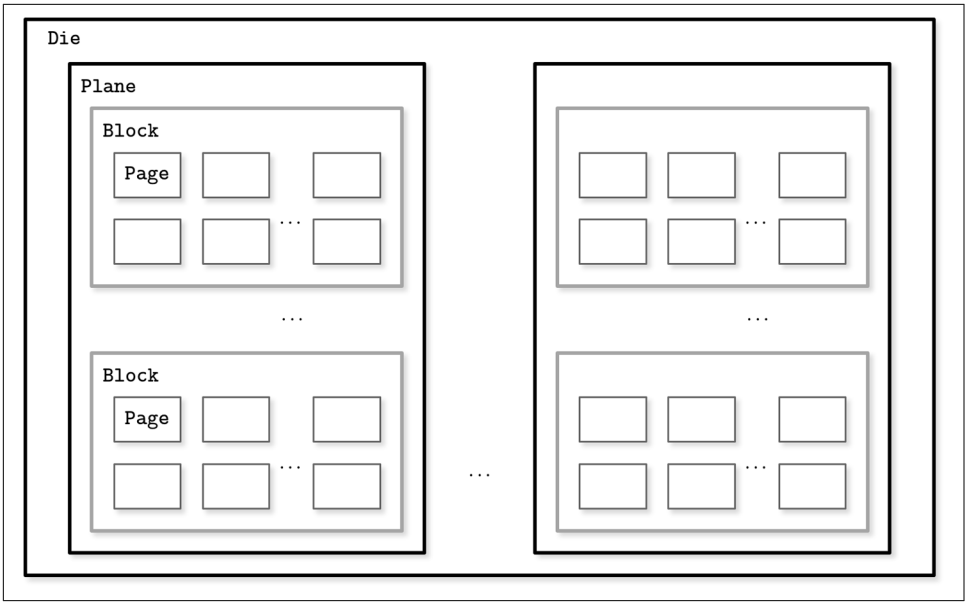

Figure 2-5. SSD organization schematics

The smallest unit that can be written (programmed) or read is a page. However, we can only make changes to the empty memory cells (i.e., to ones that have been erased before the write). The smallest erase entity is not a page, but a block that holds multiple pages, which is why it is often called an *erase block*. Pages in an empty block have to be written sequentially.

The part of a flash memory controller responsible for mapping page IDs to their physical locations, tracking empty, written, and discarded pages, is called the Flash Translation Layer (FTL) (see "Flash Translation Layer" on page 157 for more about FTL). It is also responsible for *garbage collection*, during which FTL finds blocks it can safely erase. Some blocks might still contain live pages. In this case, it relocates live pages from these blocks to new locations and remaps page IDs to point there. After this, it erases the now-unused blocks, making them available for writes.

Since in both device types (HDDs and SSDs) we are addressing chunks of memory rather than individual bytes (i.e., accessing data block-wise), most operating systems have a *block device* abstraction [CESATI05]. It hides an internal disk structure and buffers I/O operations internally, so when we're reading a *single word* from a block device, the *whole block* containing it is read. This is a constraint we cannot ignore and should always take into account when working with disk-resident data structures.

In SSDs, we don't have a strong emphasis on random versus sequential I/O, as in HDDs, because the difference in latencies between random and sequential reads is

not as large. There is *still* some difference caused by prefetching, reading contiguous pages, and internal parallelism [GOOSSAERT14].

Even though garbage collection is usually a background operation, its effects may negatively impact write performance, especially in cases of random and unaligned write workloads.

Writing only full blocks, and combining subsequent writes to the same block, can help to reduce the number of required I/O operations. We discuss buffering and immutability as ways to achieve that in later chapters.

On-Disk Structures

Besides the cost of disk access itself, the main limitation and design condition for building efficient on-disk structures is the fact that the smallest unit of disk operation is a block. To follow a pointer to the specific location within the block, we have to fetch an entire block. Since we already have to do that, we can change the layout of the data structure to take advantage of it.

We've mentioned pointers several times throughout this chapter already, but this word has slightly different semantics for on-disk structures. On disk, most of the time we manage the data layout manually (unless, for example, we're using memory mapped files). This is still similar to regular pointer operations, but we have to compute the target pointer addresses and follow the pointers explicitly.

Most of the time, on-disk offsets are precomputed (in cases when the pointer is written on disk before the part it points to) or cached in memory until they are flushed on the disk. Creating long dependency chains in on-disk structures greatly increases code and structure complexity, so it is preferred to keep the number of pointers and their spans to a minimum.

In summary, on-disk structures are designed with their target storage specifics in mind and generally optimize for fewer disk accesses. We can do this by improving locality, optimizing the internal representation of the structure, and reducing the number of out-of-page pointers.

In "Binary Search Trees" on page 26, we came to the conclusion that *high fanout* and *low height* are desired properties for an optimal on-disk data structure. We've also just discussed additional space overhead coming from pointers, and maintenance overhead from remapping these pointers as a result of balancing. B-Trees combine these ideas: increase node fanout, and reduce tree height, the number of node pointers, and the frequency of balancing operations.

Paged Binary Trees

Laying out a binary tree by grouping nodes into pages, as Figure 2-6 shows, improves the situation with locality. To find the next node, it's only necessary to follow a pointer in an already fetched page. However, there's still some overhead incurred by the nodes and pointers between them. Laying the structure out on disk and its further maintenance are nontrivial endeavors, especially if keys and values are not presorted and added in random order. Balancing requires page reorganization, which in turn causes pointer updates.

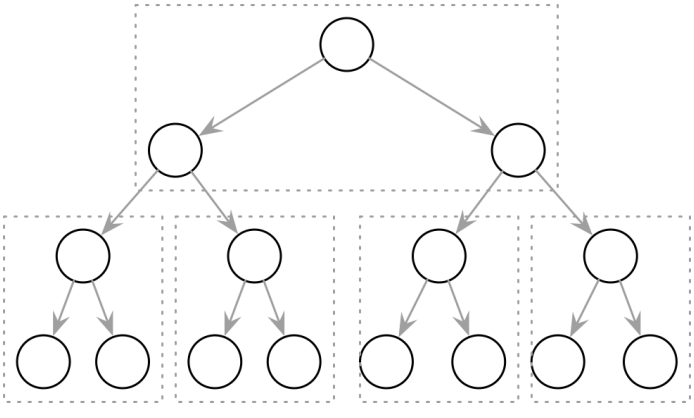

Figure 2-6. Paged binary trees

Ubiquitous B-Trees

> We are braver than a bee, and a… longer than a tree…
>
> —Winnie the Pooh

B-Trees can be thought of as a vast catalog room in the library: you first have to pick the correct cabinet, then the correct shelf in that cabinet, then the correct drawer on the shelf, and then browse through the cards in the drawer to find the one you're searching for. Similarly, a B-Tree builds a hierarchy that helps to navigate and locate the searched items quickly.

As we discussed in "Binary Search Trees" on page 26, B-Trees build upon the foundation of balanced search trees and are different in that they have higher fanout (have more child nodes) and smaller height.

In most of the literature, binary tree nodes are drawn as circles. Since each node is responsible just for one key and splits the range into two parts, this level of detail is

sufficient and intuitive. At the same time, B-Tree nodes are often drawn as rectangles, and pointer blocks are also shown explicitly to highlight the relationship between child nodes and separator keys. Figure 2-7 shows binary tree, 2-3-Tree, and B-Tree nodes side by side, which helps to understand the similarities and differences between them.

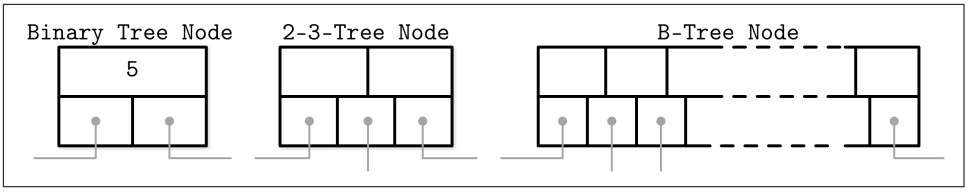

Figure 2-7. Binary tree, 2-3-Tree, and B-Tree nodes side by side

Nothing prevents us from depicting binary trees in the same way. Both structures have similar pointer-following semantics, and differences start showing in how the balance is maintained. Figure 2-8 shows that and hints at similarities between BSTs and B-Trees: in both cases, keys split the tree into subtrees, and are used for navigating the tree and finding searched keys. You can compare it to Figure 2-1.

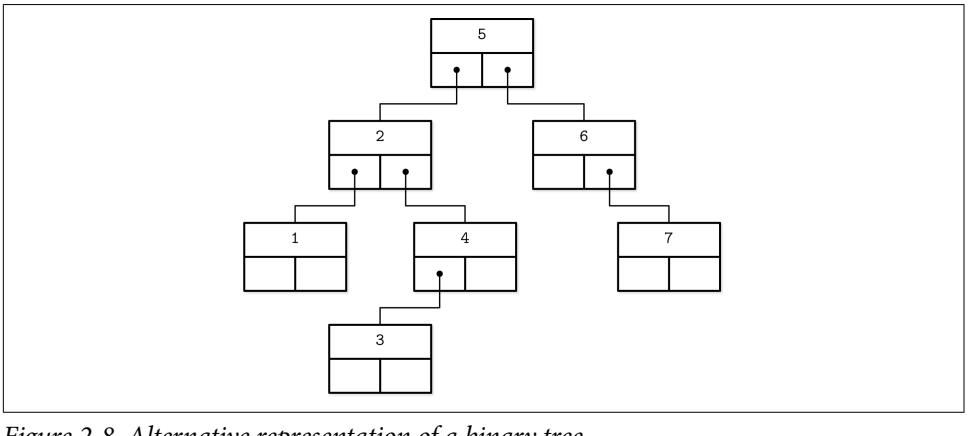

Figure 2-8. Alternative representation of a binary tree

B-Trees are *sorted*: keys inside the B-Tree nodes are stored in order. Because of that, to locate a searched key, we can use an algorithm like binary search. This also implies that lookups in B-Trees have logarithmic complexity. For example, finding a searched key among 4 billion (4×10^9) items takes about 32 comparisons (see "B-Tree Lookup Complexity" on page 37 for more on this subject). If we had to make a disk seek for each one of these comparisons, it would significantly slow us down, but since B-Tree nodes store dozens or even hundreds of items, we only have to make one disk seek per level jump. We'll discuss a lookup algorithm in more detail later in this chapter.

Using B-Trees, we can efficiently execute both *point* and *range* queries. Point queries, expressed by the equality (=) predicate in most query languages, locate a single item. On the other hand, range queries, expressed by comparison (<, >, ≤, and ≥) predicates, are used to query multiple data items in order.

B-Tree Hierarchy

B-Trees consist of multiple nodes. Each node holds up to `N` keys and `N + 1` pointers to the child nodes. These nodes are logically grouped into three groups:

Root node
: This has no parents and is the top of the tree.

Leaf nodes
: These are the bottom layer nodes that have no child nodes.

Internal nodes
: These are all other nodes, connecting root with leaves. There is usually more than one level of internal nodes.

This hierarchy is shown in Figure 2-9.

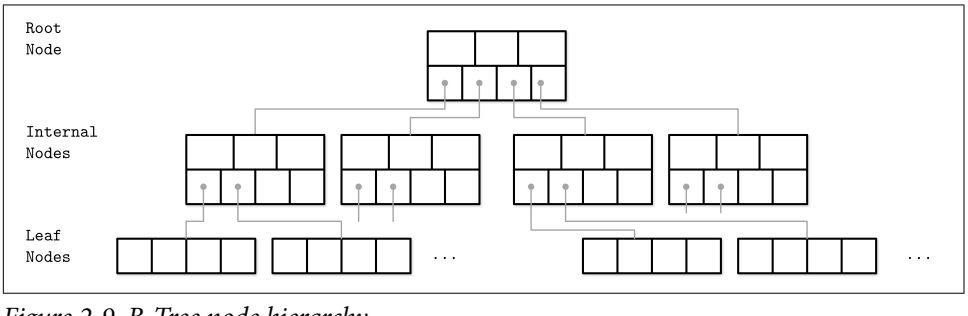

Figure 2-9. B-Tree node hierarchy

Since B-Trees are a *page* organization technique (i.e., they are used to organize and navigate fixed-size pages), we often use terms *node* and *page* interchangeably.

The relation between the node capacity and the number of keys it actually holds is called *occupancy*.

B-Trees are characterized by their *fanout*: the number of keys stored in each node. Higher fanout helps to amortize the cost of structural changes required to keep the tree balanced and to reduce the number of seeks by storing keys and pointers to child nodes in a single block or multiple consecutive blocks. Balancing operations (namely, *splits* and *merges*) are triggered when the nodes are full or nearly empty.

B⁺-Trees

We're using the term *B-Tree* as an umbrella for a family of data structures that share all or most of the mentioned properties. A more precise name for the described data structure is B⁺-Tree. [KNUTH98] refers to trees with a high fanout as *multiway* trees.

B-Trees allow storing values on any level: in root, internal, and leaf nodes. B⁺-Trees store values *only* in leaf nodes. Internal nodes store only *separator keys* used to guide the search algorithm to the associated value stored on the leaf level.

Since values in B⁺-Trees are stored only on the leaf level, all operations (inserting, updating, removing, and retrieving data records) affect only leaf nodes and propagate to higher levels only during splits and merges.

B⁺-Trees became widespread, and we refer to them as B-Trees, similar to other literature the subject. For example, in [GRAEFE11] B⁺-Trees are referred to as a default design, and MySQL InnoDB refers to its B⁺-Tree implementation as B-tree.

Separator Keys

Keys stored in B-Tree nodes are called *index entries*, *separator keys*, or *divider cells*. They split the tree into *subtrees* (also called *branches* or *subranges*), holding corresponding key ranges. Keys are stored in sorted order to allow binary search. A subtree is found by locating a key and following a corresponding pointer from the higher to the lower level.

The first pointer in the node points to the subtree holding items *less than* the first key, and the last pointer in the node points to the subtree holding items *greater than or equal* to the last key. Other pointers are reference subtrees *between* the two keys: $K_{i-1} \leq K_s < K_i$, where K is a set of keys, and K_s is a key that belongs to the subtree. Figure 2-10 shows these invariants.

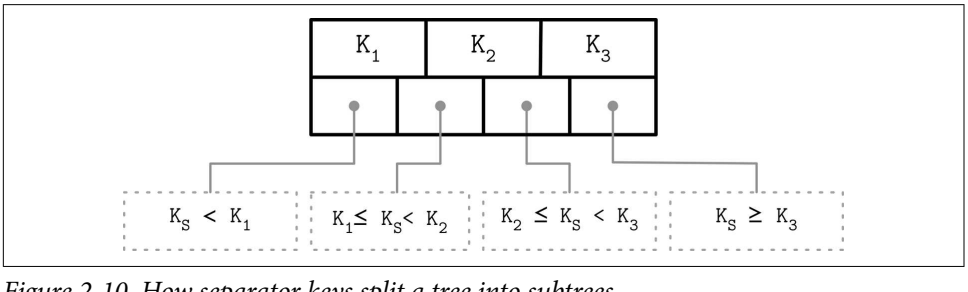

Figure 2-10. How separator keys split a tree into subtrees

Some B-Tree variants also have sibling node pointers, most often on the leaf level, to simplify range scans. These pointers help avoid going back to the parent to find the next sibling. Some implementations have pointers in both directions, forming a double-linked list on the leaf level, which makes the reverse iteration possible.

What sets B-Trees apart is that, rather than being built from top to bottom (as binary search trees), they're constructed the other way around—from bottom to top. The number of leaf nodes grows, which increases the number of internal nodes and tree height.

Since B-Trees reserve extra space inside nodes for future insertions and updates, tree storage utilization can get as low as 50%, but is usually considerably higher. Higher occupancy does not influence B-Tree performance negatively.

B-Tree Lookup Complexity

B-Tree lookup complexity can be viewed from two standpoints: the number of block transfers and the number of comparisons done during the lookup.

In terms of number of transfers, the logarithm base is `N` (number of keys per node). There are `K` times more nodes on each new level, and following a child pointer reduces the search space by the factor of `N`. During lookup, at most $\log_K M$ (where `M` is a total number of items in the B-Tree) pages are addressed to find a searched key. The number of child pointers that have to be followed on the root-to-leaf pass is also equal to the number of levels, in other words, the height `h` of the tree.

From the perspective of number of comparisons, the logarithm base is `2`, since searching a key inside each node is done using binary search. Every comparison halves the search space, so complexity is $\log_2 M$.

Knowing the distinction between the number of seeks and the number of comparisons helps us gain the intuition about how searches are performed and understand what lookup complexity is, from both perspectives.

In textbooks and articles,[2] B-Tree lookup complexity is generally referenced as $\log M$. Logarithm base is generally not used in complexity analysis, since changing the base simply adds a constant factor, and multiplication by a constant factor does not change complexity. For example, given the nonzero constant factor `c`, $O(|c| \times n) == O(n)$ [KNUTH97].

2 For example, [KNUTH98].

B-Tree Lookup Algorithm

Now that we have covered the structure and internal organization of B-Trees, we can define algorithms for lookups, insertions, and removals. To find an item in a B-Tree, we have to perform a single traversal from root to leaf. The objective of this search is to find a searched key or its predecessor. Finding an exact match is used for point queries, updates, and deletions; finding its predecessor is useful for range scans and inserts.

The algorithm starts from the root and performs a binary search, comparing the searched key with the keys stored in the root node until it finds the first separator key that is greater than the searched value. This locates a searched subtree. As we've discussed previously, index keys split the tree into subtrees with boundaries *between* two neighboring keys. As soon as we find the subtree, we follow the pointer that corresponds to it and continue the same search process (locate the separator key, follow the pointer) until we reach a target leaf node, where we either find the searched key or conclude it is not present by locating its predecessor.

On each level, we get a more detailed view of the tree: we start on the most coarse-grained level (the root of the tree) and descend to the next level where keys represent more precise, detailed ranges, until we finally reach leaves, where the data records are located.

During the point query, the search is done after finding or failing to find the searched key. During the range scan, iteration starts from the closest found key-value pair and continues by following sibling pointers until the end of the range is reached or the range predicate is exhausted.

Counting Keys

Across the literature, you can find different ways to describe key and child offset counts. [BAYER72] mentions the device-dependent natural number `k` that represents an optimal page size. Pages, in this case, can hold between `k` and `2k` keys, but can be partially filled and hold at least `k + 1` and at most `2k + 1` pointers to child nodes. The root page can hold between `1` and `2k` keys. Later, a number `l` is introduced, and it is said that any nonleaf page can have `l + 1` keys.

Other sources, for example [GRAEFE11], describe nodes that can hold up to `N` *separator keys* and `N + 1` *pointers*, with otherwise similar semantics and invariants.

Both approaches bring us to the same result, and differences are only used to emphasize the contents of each source. In this book, we stick to `N` as the number of keys (or key-value pairs, in the case of the leaf nodes) for clarity.

B-Tree Node Splits

To insert the value into a B-Tree, we first have to locate the target leaf and find the insertion point. For that, we use the algorithm described in the previous section. After the leaf is located, the key and value are appended to it. Updates in B-Trees work by locating a target leaf node using a lookup algorithm and associating a new value with an existing key.

If the target node doesn't have enough room available, we say that the node has *overflowed* [NICHOLS66] and has to be split in two to fit the new data. More precisely, the node is split if the following conditions hold:

- For leaf nodes: if the node can hold up to `N` key-value pairs, and inserting one more key-value pair brings it *over* its maximum capacity `N`.
- For nonleaf nodes: if the node can hold up to `N + 1` pointers, and inserting one more pointer brings it *over* its maximum capacity `N + 1`.

Splits are done by allocating the new node, transferring half the elements from the splitting node to it, and adding its first key and pointer to the parent node. In this case, we say that the key is *promoted*. The index at which the split is performed is called the *split point* (also called the midpoint). All elements after the split point (including split point in the case of nonleaf node split) are transferred to the newly created sibling node, and the rest of the elements remain in the splitting node.

If the parent node is full and does not have space available for the promoted key and pointer to the newly created node, it has to be split as well. This operation might propagate recursively all the way to the root.

As soon as the tree reaches its capacity (i.e., split propagates all the way up to the root), we have to split the root node. When the root node is split, a new root, holding a split point key, is allocated. The old root (now holding only half the entries) is demoted to the next level along with its newly created sibling, increasing the tree height by one. The tree height changes when the root node is split and the new root is allocated, or when two nodes are merged to form a new root. On the leaf and internal node levels, the tree only grows *horizontally*.

Figure 2-11 shows a fully occupied *leaf* node during insertion of the new element `11`. We draw the line in the middle of the full node, leave half the elements in the node, and move the rest of elements to the new one. A split point value is placed into the parent node to serve as a separator key.

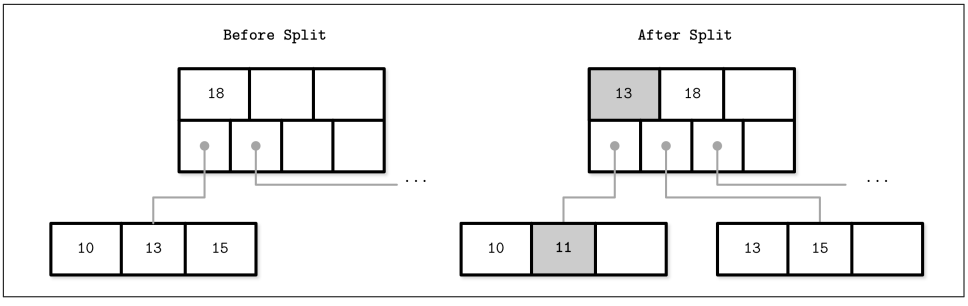

Figure 2-11. Leaf node split during the insertion of 11. New element and promoted key are shown in gray.

Figure 2-12 shows the split process of a fully occupied *nonleaf* (i.e., root or internal) node during insertion of the new element `11`. To perform a split, we first create a new node and move elements starting from index `N/2 + 1` to it. The split point key is promoted to the parent.

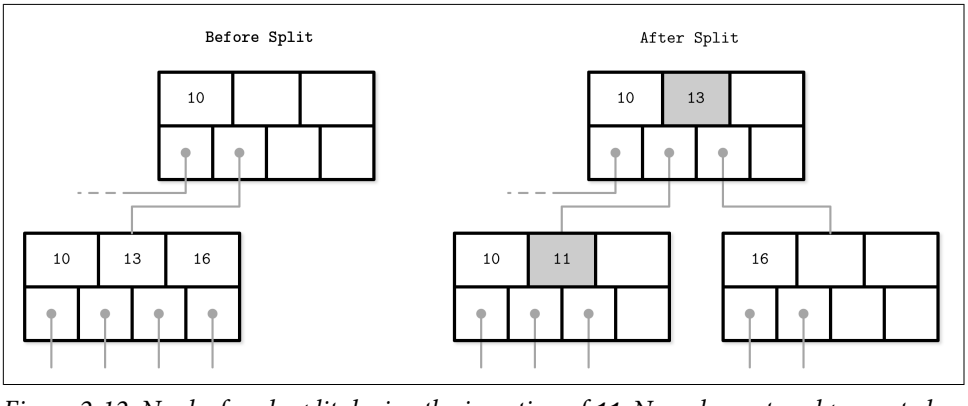

Figure 2-12. Nonleaf node split during the insertion of 11. New element and promoted key are shown in gray.

Since nonleaf node splits are always a manifestation of splits propagating from the levels below, we have an additional pointer (to the newly created node on the next level). If the parent does not have enough space, it has to be split as well.

It doesn't matter whether the leaf or nonleaf node is split (i.e., whether the node holds keys and values or just the keys). In the case of leaf split, keys are moved together with their associated values.

When the split is done, we have two nodes and have to pick the correct one to finish insertion. For that, we can use the separator key invariants. If the inserted key is less than the promoted one, we finish the operation by inserting to the split node. Otherwise, we insert to the newly created one.

To summarize, node splits are done in four steps:

1. Allocate a new node.
2. Copy half the elements from the splitting node to the new one.
3. Place the new element into the corresponding node.
4. At the parent of the split node, add a separator key and a pointer to the new node.

B-Tree Node Merges

Deletions are done by first locating the target leaf. When the leaf is located, the key and the value associated with it are removed.

If neighboring nodes have too few values (i.e., their occupancy falls under a threshold), the sibling nodes are merged. This situation is called *underflow*. [BAYER72] describes two underflow scenarios: if two adjacent nodes have a common parent and their contents fit into a single node, their contents should be merged (concatenated); if their contents do not fit into a single node, keys are redistributed between them to restore balance (see "Rebalancing" on page 70). More precisely, two nodes are merged if the following conditions hold:

- For leaf nodes: if a node can hold up to `N` key-value pairs, and a combined number of key-value pairs in two neighboring nodes is less than or equal to `N`.
- For nonleaf nodes: if a node can hold up to `N + 1` pointers, and a combined number of pointers in two neighboring nodes is less than or equal to `N + 1`.

Figure 2-13 shows the merge during deletion of element `16`. To do this, we move elements from one of the siblings to the other one. Generally, elements from the *right* sibling are moved to the *left* one, but it can be done the other way around as long as the key order is preserved.

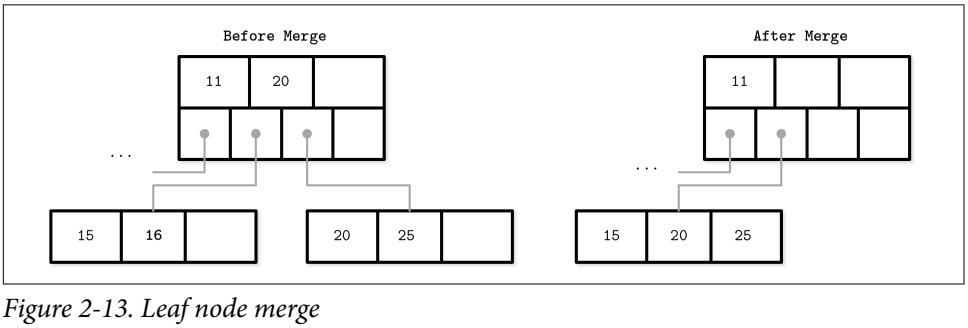

Figure 2-13. Leaf node merge

Figure 2-14 shows two sibling nonleaf nodes that have to be merged during deletion of element `10`. If we combine their elements, they fit into one node, so we can have one node instead of two. During the merge of nonleaf nodes, we have to pull the corresponding separator key from the parent (i.e., demote it). The number of pointers is reduced by one because the merge is a result of the propagation of the pointer deletion from the lower level, caused by the page removal. Just as with splits, merges can propagate all the way to the root level.

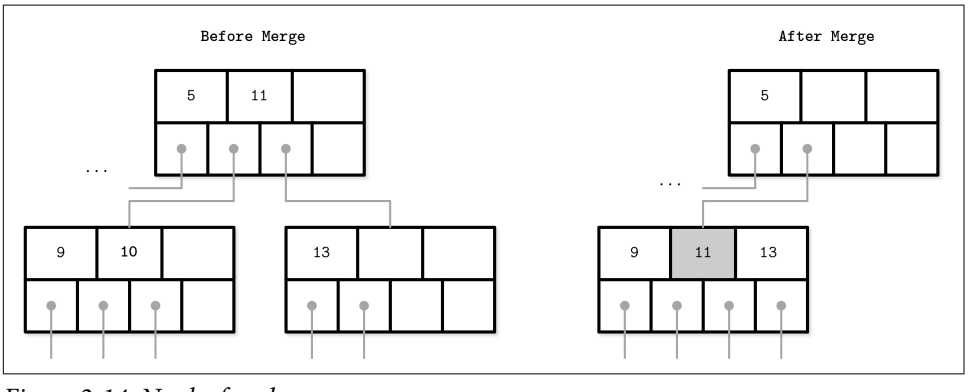

Figure 2-14. Nonleaf node merge

To summarize, node merges are done in three steps, assuming the element is already removed:

1. Copy all elements from the *right* node to the *left* one.
2. Remove the *right* node pointer from the parent (or *demote* it in the case of a nonleaf merge).
3. Remove the right node.

One of the techniques often implemented in B-Trees to reduce the number of splits and merges is rebalancing, which we discuss in "Rebalancing" on page 70.

Summary

In this chapter, we started with a motivation to create specialized structures for on-disk storage. Binary search trees might have similar complexity characteristics, but still fall short of being suitable for disk because of low fanout and a large number of relocations and pointer updates caused by balancing. B-Trees solve both problems by increasing the number of items stored in each node (high fanout) and less frequent balancing operations.

After that, we discussed internal B-Tree structure and outlines of algorithms for lookup, insert, and delete operations. Split and merge operations help to restructure

the tree to keep it balanced while adding and removing elements. We keep the tree depth to a minimum and add items to the existing nodes while there's still some free space in them.

We can use this knowledge to create in-memory B-Trees. To create a disk-based implementation, we need to go into details of how to lay out B-Tree nodes on disk and compose on-disk layout using data-encoding formats.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Binary search trees

Sedgewick, Robert and Kevin Wayne. 2011. *Algorithms (4th Ed.)*. Boston: Pearson.

Knuth, Donald E. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston: Addison-Wesley Longman.

Algorithms for splits and merges in B-Trees

Elmasri, Ramez and Shamkant Navathe. 2011. *Fundamentals of Database Systems (6th Ed.)*. Boston: Pearson.

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. 2010. *Database Systems Concepts (6th Ed.)*. New York: McGraw-Hill.

CHAPTER 3
File Formats

With the basic semantics of B-Trees covered, we are now ready to explore how exactly B-Trees and other structures are implemented on disk. We access the disk in a way that is different from how we access main memory: from an application developer's perspective, memory accesses are mostly transparent. Because of virtual memory [BHATTACHARJEE17], we do not have to manage offsets manually. Disks are accessed using system calls (see *https://databass.dev/links/54*). We usually have to specify the offset inside the target file, and then interpret on-disk representation into a form suitable for main memory.

This means that efficient on-disk structures have to be designed with this distinction in mind. To do that, we have to come up with a file format that's easy to construct, modify, and interpret. In this chapter, we'll discuss general principles and practices that help us to design all sorts of on-disk structures, not only B-Trees.

There are numerous possibilities for B-Tree implementations, and here we discuss several useful techniques. Details may vary between implementations, but the general principles remain the same. Understanding the basic mechanics of B-Trees, such as splits and merges, is necessary, but they are insufficient for the actual implementation. There are many things that have to play together for the final result to be useful.

The semantics of pointer management in on-disk structures are somewhat different from in-memory ones. It is useful to think of on-disk B-Trees as a page management mechanism: algorithms have to compose and navigate *pages*. Pages and pointers to them have to be calculated and placed accordingly.

Since most of the complexity in B-Trees comes from mutability, we discuss details of page layouts, splitting, relocations, and other concepts applicable to mutable data structures. Later, when talking about LSM Trees (see "LSM Trees" on page 130), we focus on sorting and maintenance, since that's where most LSM complexity comes from.

Motivation

Creating a file format is in many ways similar to how we create data structures in languages with an unmanaged memory model. We allocate a block of data and slice it any way we like, using fixed-size primitives and structures. If we want to reference a larger chunk of memory or a structure with variable size, we use pointers.

Languages with an unmanaged memory model allow us to allocate more memory any time we need (within reasonable bounds) without us having to think or worry about whether or not there's a contiguous memory segment available, whether or not it is fragmented, or what happens after we free it. On disk, we have to take care of garbage collection and fragmentation ourselves.

Data layout is much less important in memory than on disk. For a disk-resident data structure to be efficient, we need to lay out data on disk in ways that allow quick access to it, and consider the specifics of a persistent storage medium, come up with binary data formats, and find a means to serialize and deserialize data efficiently.

Anyone who has ever used a low-level language such as C without additional libraries knows the constraints. Structures have a predefined size and are allocated and freed explicitly. Manually implementing memory allocation and tracking is even more challenging, since it is only possible to operate with memory segments of predefined size, and it is necessary to track which segments are already released and which ones are still in use.

When storing data in main memory, most of the problems with memory layout do not exist, are easier to solve, or can be solved using third-party libraries. For example, handling variable-length fields and oversize data is much more straightforward, since we can use memory allocation and pointers, and do not need to lay them out in any special way. There still are cases when developers design specialized main memory data layouts to take advantage of CPU cache lines, prefetching, and other hardware-related specifics, but this is mainly done for optimization purposes [FOWLER11].

Even though the operating system and filesystem take over some of the responsibilities, implementing on-disk structures requires attention to more details and has more pitfalls.

Binary Encoding

To store data on disk efficiently, it needs to be encoded using a format that is compact and easy to serialize and deserialize. When talking about binary formats, you hear the word *layout* quite often. Since we do not have primitives such as `malloc` and `free`, but only `read` and `write`, we have to think of accesses differently and prepare data accordingly.

Here, we discuss the main principles used to create efficient page layouts. These principles apply to any binary format: you can use similar guidelines to create file and serialization formats or communication protocols.

Before we can organize records into pages, we need to understand how to represent keys and data records in binary form, how to combine multiple values into more complex structures, and how to implement variable-size types and arrays.

Primitive Types

Keys and values have a *type*, such as `integer`, `date`, or `string`, and can be represented (serialized to and deserialized from) in their raw binary forms.

Most numeric data types are represented as fixed-size values. When working with multibyte numeric values, it is important to use the same *byte-order* (*endianness*) for both encoding and decoding. Endianness determines the sequential order of bytes:

Big-endian
: The order starts from the most-significant byte (MSB), followed by the bytes in *decreasing* significance order. In other words, MSB has the *lowest* address.

Little-endian
: The order starts from the least-significant byte (LSB), followed by the bytes in *increasing* significance order.

Figure 3-1 illustrates this. The hexadecimal 32-bit integer `0xAABBCCDD`, where `AA` is the MSB, is shown using both big- and little-endian byte order.

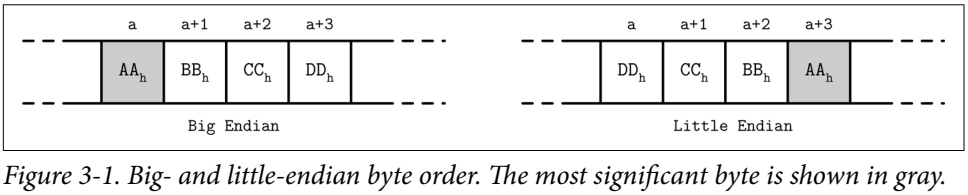

Figure 3-1. Big- and little-endian byte order. The most significant byte is shown in gray. Addresses, denoted by a, grow from left to right.

For example, to reconstruct a 64-bit integer with a corresponding byte order, RocksDB has platform-specific definitions that help to identify target platform byte order.[1] If the target platform endianness does not match value endianness (`EncodeFixed64WithEndian` looks up `kLittleEndian` value and compares it with value endianness), it reverses the bytes using `EndianTransform`, which reads values byte-wise in reverse order and appends them to the result.

Records consist of primitives like numbers, strings, booleans, and their combinations. However, when transferring data over the network or storing it on disk, we can only use byte sequences. This means that, in order to send or write the record, we have to *serialize* it (convert it to an interpretable sequence of bytes) and, before we can use it after receiving or reading, we have to *deserialize* it (translate the sequence of bytes back to the original record).

In binary data formats, we always start with primitives that serve as building blocks for more complex structures. Different numeric types may vary in size. `byte` value is 8 bits, `short` is 2 bytes (16 bits), `int` is 4 bytes (32 bits), and `long` is 8 bytes (64 bits).

Floating-point numbers (such as `float` and `double`) are represented by their *sign*, *fraction*, and *exponent*. The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) standard describes widely accepted floating-point number representation. A 32-bit `float` represents a single-precision value. For example, a floating-point number `0.15652` has a binary representation, as shown in Figure 3-2. The first 23 bits represent a fraction, the following 8 bits represent an exponent, and 1 bit represents a sign (whether or not the number is negative).

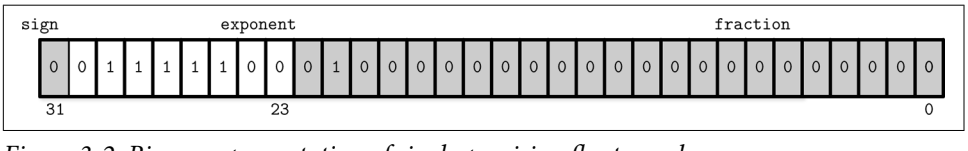

Figure 3-2. Binary representation of single-precision float number

Since a floating-point value is calculated using fractions, the number this representation yields is just an approximation. Discussing a complete conversion algorithm is out of the scope of this book, and we only cover representation basics.

The `double` represents a double-precision floating-point value [SAVARD05]. Most programming languages have means for encoding and decoding floating-point values to and from their binary representation in their standard libaries.

1 Depending on the platform (macOS, Solaris, Aix, or one of the BSD flavors, or Windows), the `kLittleEndian` variable is set to whether or not the platform supports little-endian.

Strings and Variable-Size Data

All primitive numeric types have a fixed size. Composing more complex values together is much like `struct`[2] in C. You can combine primitive values into structures and use fixed-size arrays or pointers to other memory regions.

Strings and other variable-size data types (such as arrays of fixed-size data) can be serialized as a number, representing the length of the array or string, followed by `size` bytes: the actual data. For strings, this representation is often called *UCSD String* or *Pascal String*, named after the popular implementation of the Pascal programming language. We can express it in pseudocode as follows:

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

An alternative to Pascal strings is *null-terminated strings*, where the reader consumes the string byte-wise until the end-of-string symbol is reached. The Pascal string approach has several advantages: it allows finding out a length of a string in constant time, instead of iterating through string contents, and a language-specific string can be composed by slicing `size` bytes from memory and passing the byte array to a string constructor.

Bit-Packed Data: Booleans, Enums, and Flags

Booleans can be represented either by using a single byte, or encoding `true` and `false` as `1` and `0` values. Since a boolean has only two values, using an entire byte for its representation is wasteful, and developers often batch boolean values together in groups of eight, each boolean occupying just one bit. We say that every `1` bit is *set* and every `0` bit is *unset* or *empty*.

Enums, short for *enumerated types*, can be represented as integers and are often used in binary formats and communication protocols. Enums are used to represent often-repeated low-cardinality values. For example, we can encode a B-Tree node type using an enum:

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

2 It's worth noting that compilers can add padding to structures, which is also architecture dependent. This may break the assumptions about the exact byte offsets and locations. You can read more about structure packing here: *https://databass.dev/links/58*.

Another closely related concept is *flags*, kind of a combination of packed booleans and enums. Flags can represent nonmutually exclusive named boolean parameters. For example, we can use flags to denote whether or not the page holds value cells, whether the values are fixed-size or variable-size, and whether or not there are overflow pages associated with this node. Since every bit represents a flag value, we can only use power-of-two values for masks (since powers of two in binary always have a single set bit; for example, 2^3 `==` `8` `==` `1000b`, 2^4 `==` `16` `==` `0001 0000b`, etc.):

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

Just like packed booleans, flag values can be read and written from the packed value using *bitmasks* and bitwise operators. For example, in order to set a bit responsible for one of the flags, we can use bitwise `OR` (`|`) and a bitmask. Instead of a bitmask, we can use *bitshift* (`<<`) and a bit index. To unset the bit, we can use bitwise `AND` (`&`) and the bitwise negation operator (`~`). To test whether or not the bit `n` is set, we can compare the result of a bitwise `AND` with `0`:

```
// 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;
```

General Principles

Usually, you start designing a file format by deciding how the addressing is going to be done: whether the file is going to be split into same-sized pages, which are represented by a single block or multiple contiguous blocks. Most in-place update storage structures use pages of the same size, since it significantly simplifies read and write access. Append-only storage structures often write data page-wise, too: records are appended one after the other and, as soon as the page fills up in memory, it is flushed on disk.

The file usually starts with a fixed-size *header* and may end with a fixed-size *trailer*, which hold auxiliary information that should be accessed quickly or is required for decoding the rest of the file. The rest of the file is split into pages. Figure 3-3 shows this file organization schematically.

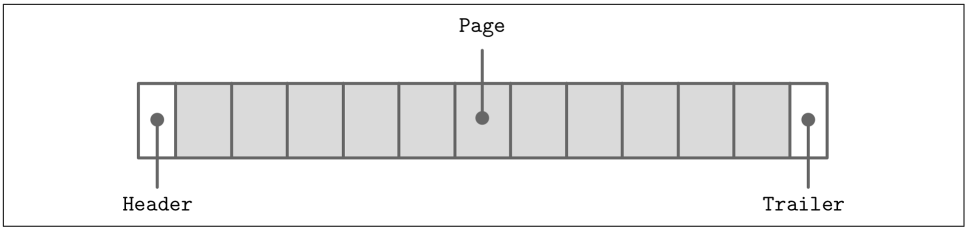

Figure 3-3. File organization

Many data stores have a fixed schema, specifying the number, order, and type of fields the table can hold. Having a fixed schema helps to reduce the amount of data stored on disk: instead of repeatedly writing field names, we can use their positional identifiers.

If we wanted to design a format for the company directory, storing names, birth dates, tax numbers, and genders for each employee, we could use several approaches. We could store the fixed-size fields (such as birth date and tax number) in the head of the structure, followed by the variable-size ones:

```
Fixed-size fields:
| (4 bytes) employee_id                |
| (4 bytes) tax_number                 |
| (3 bytes) date                       |
| (1 byte)  gender                     |
| (2 bytes) first_name_length          |
| (2 bytes) last_name_length           |

Variable-size fields:
| (first_name_length bytes) first_name |
| (last_name_length bytes) last_name   |
```

Now, to access `first_name`, we can slice `first_name_length` bytes after the fixed-size area. To access `last_name`, we can locate its starting position by checking the sizes of the variable-size fields that precede it. To avoid calculations involving multiple fields, we can encode both *offset* and *length* to the fixed-size area. In this case, we can locate any variable-size field separately.

Building more complex structures usually involves building hierarchies: fields composed out of primitives, cells composed of fields, pages composed of cells, sections composed of pages, regions composed of sections, and so on. There are no strict rules you have to follow here, and it all depends on what kind of data you need to create a format for.

Database files often consist of multiple parts, with a lookup table aiding navigation and pointing to the start offsets of these parts written either in the file header, trailer, or in the separate file.

Page Structure

Database systems store data records in data and index files. These files are partitioned into fixed-size units called *pages*, which often have a size of multiple filesystem blocks. Page sizes usually range from 4 to 16 Kb.

Let's take a look at the example of an on-disk B-Tree node. From a structure perspective, in B-Trees, we distinguish between the *leaf nodes* that hold keys and data records pairs, and *nonleaf nodes* that hold keys and pointers to other nodes. Each B-Tree node occupies one page or multiple pages linked together, so in the context of B-Trees the terms *node* and *page* (and even *block*) are often used interchangeably.

The original B-Tree paper [BAYER72] describes a simple page organization for fixed-size data records, where each page is just a concatenation of triplets, as shown in Figure 3-4: keys are denoted by `k`, associated values are denoted by `v`, and pointers to child pages are denoted by `p`.

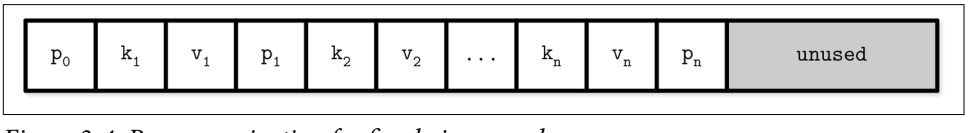

Figure 3-4. Page organization for fixed-size records

This approach is easy to follow, but has some downsides:

- Appending a key anywhere but the right side requires relocating elements.
- It doesn't allow managing or accessing variable-size records efficiently and works only for fixed-size data.

Slotted Pages

When storing variable-size records, the main problem is free space management: reclaiming the space occupied by removed records. If we attempt to put a record of size `n` into the space previously occupied by the record of size `m`, unless `m == n` or we can find another record that has a size exactly `m - n`, this space will remain unused. Similarly, a segment of size `m` cannot be used to store a record of size `k` if `k` is larger than `m`, so it will be inserted without reclaiming the unused space.

To simplify space management for variable-size records, we can split the page into fixed-size segments. However, we end up wasting space if we do that, too. For example, if we use a segment size of 64 bytes, unless the record size is a multiple of 64, we waste `64 - (n modulo 64)` bytes, where `n` is the size of the inserted record. In other words, unless the record is a multiple of 64, one of the blocks will be only partially filled.

Space reclamation can be done by simply rewriting the page and moving the records around, but we need to preserve record offsets, since out-of-page pointers might be using these offsets. It is desirable to do that while minimizing space waste, too.

To summarize, we need a page format that allows us to:

- Store variable-size records with a minimal overhead.
- Reclaim space occupied by the removed records.
- Reference records in the page without regard to their exact locations.

To efficiently store variable-size records such as strings, binary large objects (BLOBs), etc., we can use an organization technique called *slotted page* (i.e., a page with slots) [SILBERSCHATZ10] or *slot directory* [RAMAKRISHNAN03]. This approach is used by many databases, for example, PostgreSQL.

We organize the page into a collection of *slots* or *cells* and split out pointers and cells in two independent memory regions residing on different sides of the page. This means that we only need to reorganize pointers addressing the cells to preserve the order, and deleting a record can be done either by nullifying its pointer or removing it.

A slotted page has a fixed-size header that holds important information about the page and cells (see "Page Header" on page 61). Cells may differ in size and can hold arbitrary data: keys, pointers, data records, etc. Figure 3-5 shows a slotted page organization, where every page has a maintenance region (header), cells, and pointers to them.

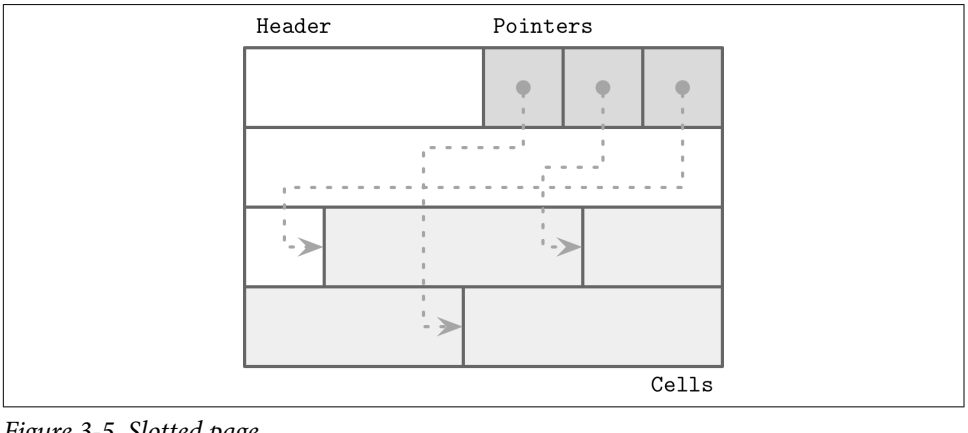

Figure 3-5. Slotted page

Let's see how this approach fixes the problems we stated in the beginning of this section:

- Minimal overhead: the only overhead incurred by slotted pages is a pointer array holding offsets to the exact positions where the records are stored.
- Space reclamation: space can be reclaimed by defragmenting and rewriting the page.
- Dynamic layout: from outside the page, slots are referenced only by their IDs, so the exact location is internal to the page.

Cell Layout

Using flags, enums, and primitive values, we can start designing the cell layout, then combine cells into pages, and compose a tree out of the pages. On a cell level, we have a distinction between key and key-value cells. Key cells hold a separator key and a pointer to the page *between* two neighboring pointers. Key-value cells hold keys and data records associated with them.

We assume that all cells within the page are uniform (for example, all cells can hold either just keys or both keys and values; similarly, all cells hold either fixed-size or variable-size data, but not a mix of both). This means we can store metadata describing cells once on the page level, instead of duplicating it in every cell.

To compose a key cell, we need to know:

- Cell type (can be inferred from the page metadata)
- Key size
- ID of the child page this cell is pointing to
- Key bytes

A variable-size key cell layout might look something like this (a fixed-size one would have no size specifier on the cell level):

```
0                4                8
+----------------+---------------+-------------+
| [int] key_size | [int] page_id | [bytes] key |
+----------------+---------------+-------------+
```

We have grouped fixed-size data fields together, followed by `key_size` bytes. This is not strictly necessary but can simplify offset calculation, since all fixed-size fields can be accessed by using static, precomputed offsets, and we need to calculate the offsets only for the variable-size data.

The key-value cells hold data records instead of the child page IDs. Otherwise, their structure is similar:

- Cell type (can be inferred from page metadata)
- Key size
- Value size
- Key bytes
- Data record bytes

```
0              1               5 ...
+--------------+---------------+
| [byte] flags | [int] key_size |
+--------------+---------------+

5                 9                    .. + key_size
+-----------------+--------------------+----------------------+
| [int] value_size |     [bytes] key    | [bytes] data_record  |
+-----------------+--------------------+----------------------+
```

You might have noticed the distinction between the *offset* and *page ID* here. Since pages have a fixed size and are managed by the page cache (see "Buffer Management" on page 81), we only need to store the page ID, which is later translated to the actual offset in the file using the lookup table. *Cell offsets* are page-local and are relative to the page start offset: this way we can use a smaller cardinality integer to keep the representation more compact.

Variable-Size Data

It is not necessary for the key and value in the cell to have a fixed size. Both the key and value can have a variable size. Their locations can be calculated from the fixed-size cell header using offsets.

To locate the key, we skip the header and read `key_size` bytes. Similarly, to locate the value, we can skip the header plus `key_size` more bytes and read `value_size` bytes.

There are different ways to do the same; for example, by storing a total size and calculating the value size by subtraction. It all boils down to having enough information to slice the cell into subparts and reconstruct the encoded data.

Combining Cells into Slotted Pages

To organize cells into pages, we can use the *slotted page* technique that we discussed in "Page Structure" on page 52. We append cells to the right side of the page (toward its end) and keep cell offsets/pointers in the left side of the page, as shown in Figure 3-6.

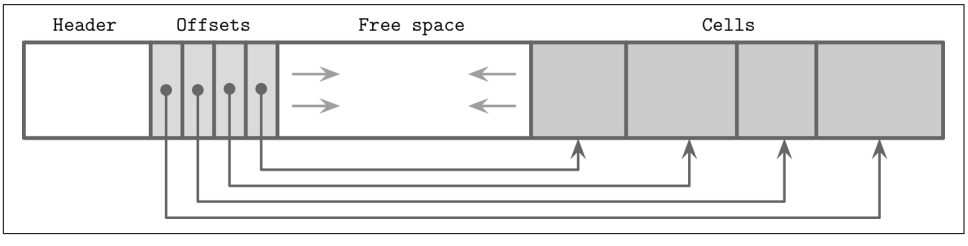

Figure 3-6. Offset and cell growth direction

Keys can be inserted out of order and their logical sorted order is kept by sorting cell offset pointers in key order. This design allows appending cells to the page with minimal effort, since cells don't have to be relocated during insert, update, or delete operations.

Let's consider an example of a page that holds names. Two names are added to the page, and their insertion order is: *Tom* and *Leslie*. As you can see in Figure 3-7, their *logical* order (in this case, alphabetical), does *not* match *insertion* order (order in which they were appended to the page). Cells are laid out in insertion order, but offsets are re-sorted to allow using binary search.

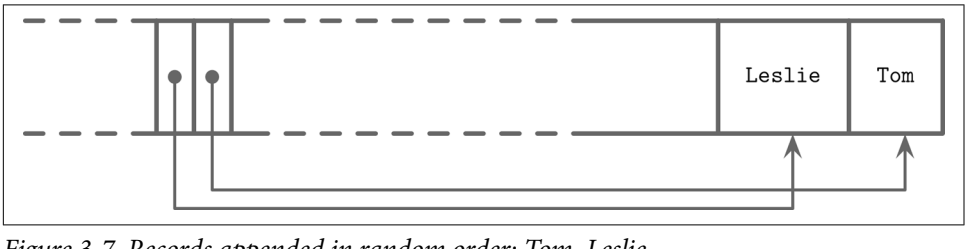

Figure 3-7. Records appended in random order: Tom, Leslie

Now, we'd like to add one more name to this page: *Ron*. New data is appended at the upper boundary of the free space of the page, but cell offsets have to preserve the lexicographical key order: *Leslie*, *Ron*, *Tom*. To do that, we have to reorder cell offsets: pointers after the *insertion point* are shifted to the right to make space for the new pointer to the Ron cell, as you can see in Figure 3-8.

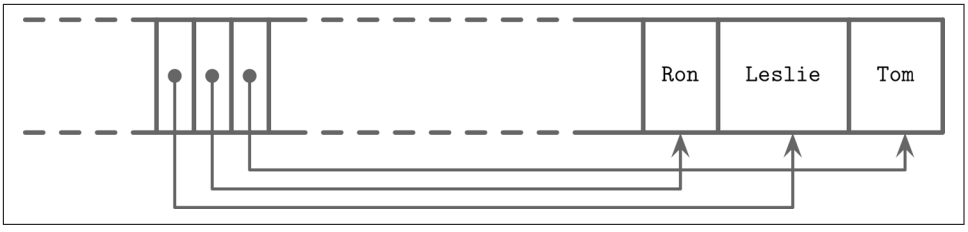

Figure 3-8. Appending one more record: Ron

Managing Variable-Size Data

Removing an item from the page does not have to remove the actual cell and shift other cells to reoccupy the freed space. Instead, the cell can be marked as deleted and an in-memory *availability list* can be updated with the amount of freed memory and a pointer to the freed value. The availability list stores offsets of freed segments and their sizes. When inserting a new cell, we first check the availability list to find if there's a segment where it may fit. You can see an example of the fragmented page with available segments in Figure 3-9.

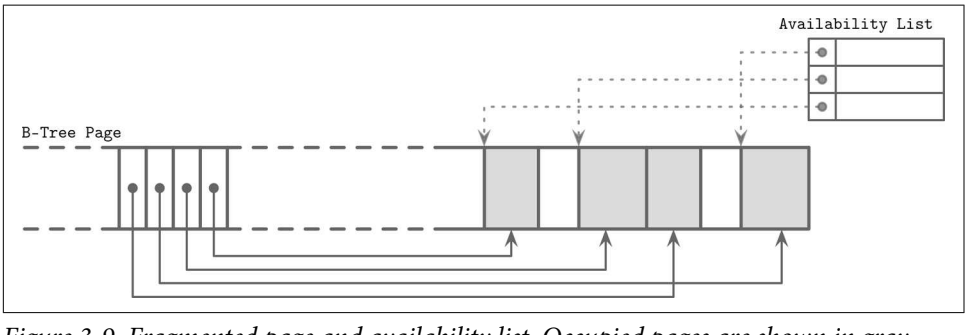

Figure 3-9. Fragmented page and availability list. Occupied pages are shown in gray. Dotted lines represent pointers to unoccupied memory regions from the availability list.

SQLite calls unoccupied segments *freeblocks* and stores a pointer to the first freeblock in the page header. Additionally, it stores a total number of available bytes within the page to quickly check whether or not we can fit a new element into the page after defragmenting it.

Fit is calculated based on the *strategy*:

First fit
: This might cause a larger overhead, since the space remaining after reusing the first suitable segment might be too small to fit any other cell, so it will be effectively wasted.

Best fit
: For best fit, we try to find a segment for which insertion leaves the smallest remainder.

If we cannot find enough consecutive bytes to fit the new cell but there are enough fragmented bytes available, live cells are read and rewritten, defragmenting the page and reclaiming space for new writes. If there's not enough free space even after defragmentation, we have to create an overflow page (see "Overflow Pages" on page 65).

To improve locality (especially when keys are small in size), some implementations store keys and values separately on the leaf level. Keeping keys together can improve the locality during the search. After the searched key is located, its value can be found in a value cell with a corresponding index. With variable-size keys, this requires us to calculate and store an additional value cell pointer.

In summary, to simplify B-Tree layout, we assume that each node occupies a single page. A page consists of a fixed-size header, cell pointer block, and cells. Cells hold keys and pointers to the pages representing child nodes or associated data records. B-Trees use simple pointer hierarchies: page identifiers to locate the child nodes in the tree file, and cell offsets to locate cells within the page.

Versioning

Database systems constantly evolve, and developers work to add features, and to fix bugs and performance issues. As a result of that, the binary file format can change. Most of the time, any storage engine version has to support more than one serialization format (e.g., current and one or more legacy formats for backward compatibility). To support that, we have to be able to find out which version of the file we're up against.

This can be done in several ways. For example, Apache Cassandra is using version prefixes in filenames. This way, you can tell which version the file has without even opening it. As of version 4.0, a data file name has the `na` prefix, such as *na-1-big-Data.db*. Older files have different prefixes: files written in version 3.0 have the `ma` prefix.

Alternatively, the version can be stored in a separate file. For example, PostgreSQL stores the version in the *PG_VERSION* file.

The version can also be stored directly in the index file header. In this case, a part of the header (or an entire header) has to be encoded in a format that does not change between versions. After finding out which version the file is encoded with, we can

create a version-specific reader to interpret the contents. Some file formats identify the version using magic numbers, which we discuss in more detail in "Magic Numbers" on page 62.

Checksumming

Files on disk may get damaged or corrupted by software bugs and hardware failures. To identify these problems preemptively and avoid propagating corrupt data to other subsystems or even nodes, we can use checksums and cyclic redundancy checks (CRCs).

Some sources make no distinction between cryptographic and noncryptographic hash functions, CRCs, and checksums. What they all have in common is that they reduce a large chunk of data to a small number, but their use cases, purposes, and guarantees are different.

Checksums provide the weakest form of guarantee and aren't able to detect corruption in multiple bits. They're usually computed by using `XOR` with parity checks or summation [KOOPMAN15].

CRCs can help detect burst errors (e.g., when multiple consecutive bits got corrupted) and their implementations usually use lookup tables and polynomial division [STONE98]. Multibit errors are crucial to detect, since a significant percentage of failures in communication networks and storage devices manifest this way.

Noncryptographic hashes and CRCs should not be used to verify whether or not the data has been tampered with. For this, you should always use strong cryptographic hashes designed for security. The main goal of CRC is to make sure that there were no unintended and accidental changes in data. These algorithms are not designed to resist attacks and intentional changes in data.

Before writing the data on disk, we compute its checksum and write it together with the data. When reading it back, we compute the checksum again and compare it with the written one. If there's a checksum mismatch, we know that corruption has occurred and we should not use the data that was read.

Since computing a checksum over the whole file is often impractical and it is unlikely we're going to read the entire content every time we access it, page checksums are usually computed on pages and placed in the page header. This way, checksums can be more robust (since they are performed on a small subset of the data), and the whole file doesn't have to be discarded if corruption is contained in a single page.

Summary

In this chapter, we learned about binary data organization: how to serialize primitive data types, combine them into cells, build slotted pages out of cells, and navigate these structures.

We learned how to handle variable-size data types such as strings, byte sequences, and arrays, and compose special cells that hold a size of values contained in them.

We discussed the slotted page format, which allows us to reference individual cells from outside the page by cell ID, store records in the insertion order, and preserve the key order by sorting cell offsets.

These principles can be used to compose binary formats for on-disk structures and network protocols.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

File organization techniques

Folk, Michael J., Greg Riccardi, and Bill Zoellick. 1997. *File Structures: An Object-Oriented Approach with C++ (3rd Ed.)*. Boston: Addison-Wesley Longman.

Giampaolo, Dominic. 1998. *Practical File System Design with the Be File System (1st Ed.)*. San Francisco: Morgan Kaufmann.

Vitter, Jeffrey Scott. 2008. "Algorithms and data structures for external memory." *Foundations and Trends in Theoretical Computer Science* 2, no. 4 (January): 305-474. *https://doi.org/10.1561/0400000014*.

CHAPTER 4

Implementing B-Trees

In the previous chapter, we talked about general principles of binary format composition, and learned how to create cells, build hierarchies, and connect them to pages using pointers. These concepts are applicable for both in-place update and append-only storage structures. In this chapter, we discuss some concepts specific to B-Trees.

The sections in this chapter are split into three logical groups. First, we discuss organization: how to establish relationships between keys and pointers, and how to implement headers and links between pages.

Next, we discuss processes that occur during root-to-leaf descends, namely how to perform binary search and how to collect breadcrumbs and keep track of parent nodes in case we later have to split or merge nodes.

Lastly, we discuss optimization techniques (rebalancing, right-only appends, and bulk loading), maintenance processes, and garbage collection.

Page Header

The page header holds information about the page that can be used for navigation, maintenance, and optimizations. It usually contains flags that describe page contents and layout, number of cells in the page, lower and upper offsets marking the empty space (used to append cell offsets and data), and other useful metadata.

For example, PostgreSQL stores the page size and layout version in the header. In MySQL InnoDB, page header holds the number of heap records, level, and some other implementation-specific values. In SQLite, page header stores the number of cells and a rightmost pointer.

Magic Numbers

One of the values often placed in the file or page header is a magic number. Usually, it's a multibyte block, containing a constant value that can be used to signal that the block represents a page, specify its kind, or identify its version.

Magic numbers are often used for validation and sanity checks [GIAMPAOLO98]. It's very improbable that the byte sequence at a random offset would exactly match the magic number. If it did match, there's a good chance the offset is correct. For example, to verify that the page is loaded and aligned correctly, during write we can place the magic number `50 41 47 45` (hex for `PAGE`) into the header. During the read, we validate the page by comparing the four bytes from the read header with the expected byte sequence.

Sibling Links

Some implementations store forward and backward links, pointing to the left and right sibling pages. These links help to locate neighboring nodes without having to ascend back to the parent. This approach adds some complexity to split and merge operations, as the sibling offsets have to be updated as well. For example, when a non-rightmost node is split, its right sibling's backward pointer (previously pointing to the node that was split) has to be re-bound to point to the newly created node.

In Figure 4-1 you can see that to locate a sibling node, unless the siblings are linked, we have to refer to the parent node. This operation might ascend all the way up to the root, since the direct parent can only help to address *its* own children. If we store sibling links directly in the header, we can simply follow them to locate the previous or next node on the same level.

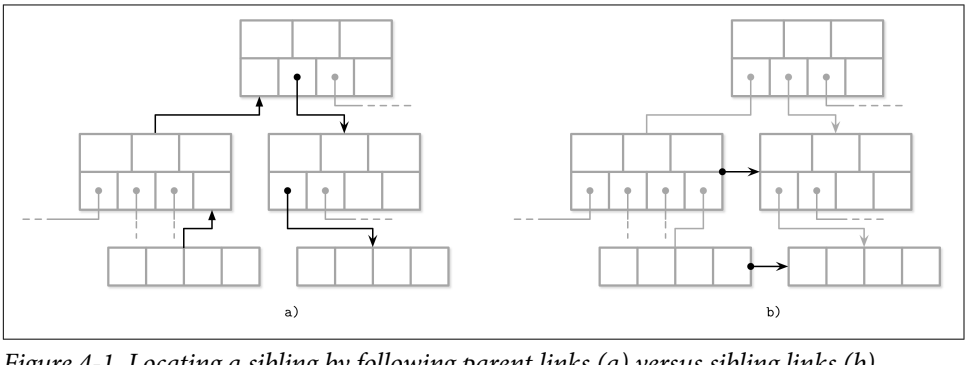

Figure 4-1. Locating a sibling by following parent links (a) versus sibling links (b)

One of the downsides of storing sibling links is that they have to be updated during splits and merges. Since updates have to happen in a sibling node, not in a splitting/merging node, it may require additional locking. We discuss how sibling links can be useful in a concurrent B-Tree implementation in "Blink-Trees" on page 107.

Rightmost Pointers

B-Tree separator keys have strict invariants: they're used to split the tree into subtrees and navigate them, so there is always one more pointer to child pages than there are keys. That's where the `+1` mentioned in "Counting Keys" on page 38 is coming from.

In "Separator Keys" on page 36, we described separator key invariants. In many implementations, nodes look more like the ones displayed in Figure 4-2: each separator key has a child pointer, while the last pointer is stored separately, since it's not paired with any key. You can compare this to Figure 2-10.

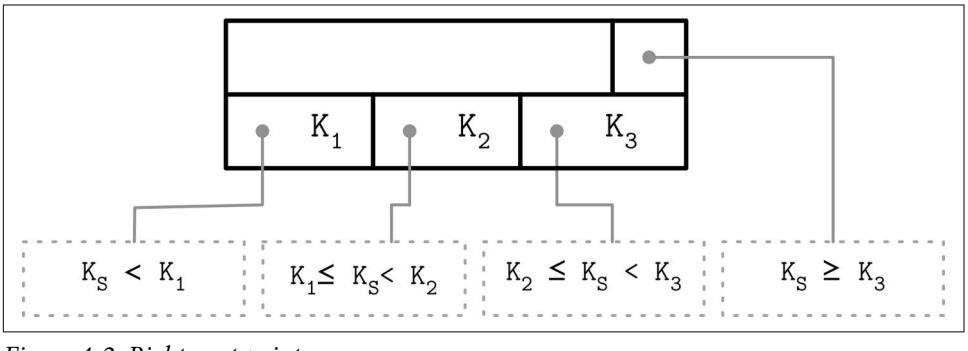

Figure 4-2. Rightmost pointer

This extra pointer can be stored in the header as, for example, it is implemented in SQLite.

If the rightmost child is split and the new cell is appended to its parent, the rightmost child pointer has to be reassigned. As shown in Figure 4-3, after the split, the cell appended to the parent (shown in gray) holds the promoted key and points to the split node. The pointer to the new node is assigned instead of the previous rightmost pointer. A similar approach is described and implemented in SQLite.[1]

1 You can find this algorithm in the `balance_deeper` function in the project repository.

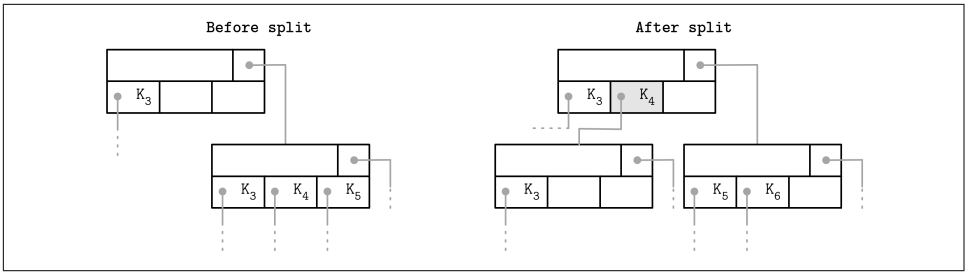

Figure 4-3. Rightmost pointer update during node split. The promoted key is shown in gray.

Node High Keys

We can take a slightly different approach and store the rightmost pointer in the cell along with the node *high key*. The high key represents the highest possible key that can be present in the subtree under the current node. This approach is used by PostgreSQL and is called B^{link}-Trees (for concurrency implications of this approach, see "Blink-Trees" on page 107).

B-Trees have `N` keys (denoted with K_i) and `N + 1` pointers (denoted with P_i). In each subtree, keys are bounded by $K_{i-1} \leq K_s < K_i$. The $K_0 = -\infty$ is implicit and is not present in the node.

B^{link}-Trees add a K_{N+1} key to each node. It specifies an upper bound of keys that can be stored in the subtree to which the pointer P_N points, and therefore is an upper bound of values that can be stored in the current subtree. Both approaches are shown in Figure 4-4: (a) shows a node *without* a high key, and (b) shows a node with a high key.

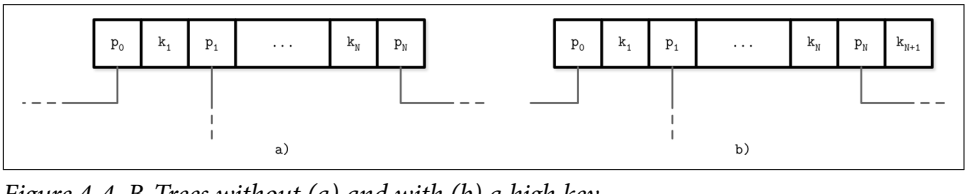

Figure 4-4. B-Trees without (a) and with (b) a high key

In this case, pointers can be stored pairwise, and each cell can have a corresponding pointer, which might simplify rightmost pointer handling as there are not as many edge cases to consider.

In Figure 4-5, you can see schematic page structure for both approaches and how the search space is split differently for these cases: going up to $+\infty$ in the first case, and up to the upper bound of K_3 in the second.

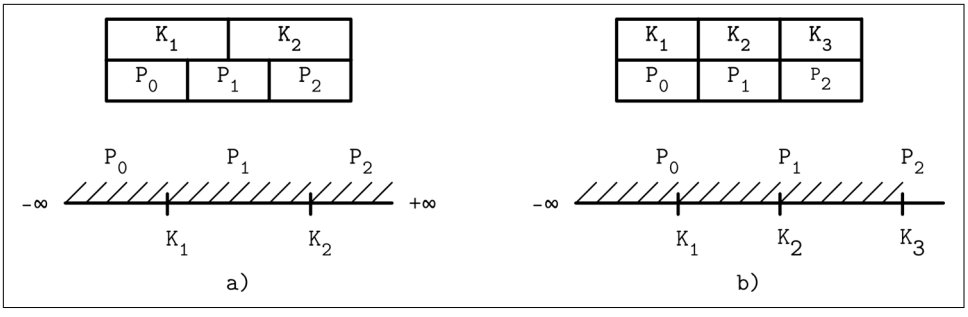

Figure 4-5. Using +∞ as a virtual key (a) versus storing the high key (b)

Overflow Pages

Node size and tree fanout values are fixed and do not change dynamically. It would also be difficult to come up with a value that would be universally optimal: if variable-size values are present in the tree and they are large enough, only a few of them can fit into the page. If the values are tiny, we end up wasting the reserved space.

The B-Tree algorithm specifies that every node keeps a specific number of items. Since some values have different sizes, we may end up in a situation where, according to the B-Tree algorithm, the node is not *full* yet, but there's *no more free space* on the fixed-size *page* that holds this node. Resizing the page requires copying already written data to the new region and is often impractical. However, we still need to find a way to increase or extend the page size.

To implement variable-size nodes without copying data to the new contiguous region, we can build nodes from multiple linked pages. For example, the default page size is 4 K, and after inserting a few values, its data size has grown over 4 K. Instead of allowing arbitrary sizes, nodes are allowed to grow in 4 K increments, so we allocate a 4 K extension page and link it from the original one. These linked page extensions are called *overflow pages*. For clarity, we call the original page the *primary page* in the scope of this section.

Most B-Tree implementations allow storing only up to a fixed number of payload bytes in the B-Tree node directly and *spilling* the rest to the overflow page. This value is calculated by dividing the node size by fanout. Using this approach, we cannot end up in a situation where the page has no free space, as it will always have at least `max_payload_size` bytes. For more information on overflow pages in SQLite, see the SQLite source code repository; also check out the MySQL InnoDB documentation.

When the inserted payload is larger than `max_payload_size`, the node is checked for whether or not it already has any associated overflow pages. If an overflow page already exists and has enough space available, extra bytes from the payload are spilled there. Otherwise, a new overflow page is allocated.

In Figure 4-6, you can see a primary page and an overflow page with records pointing from the primary page to the overflow one, where their payload continues.

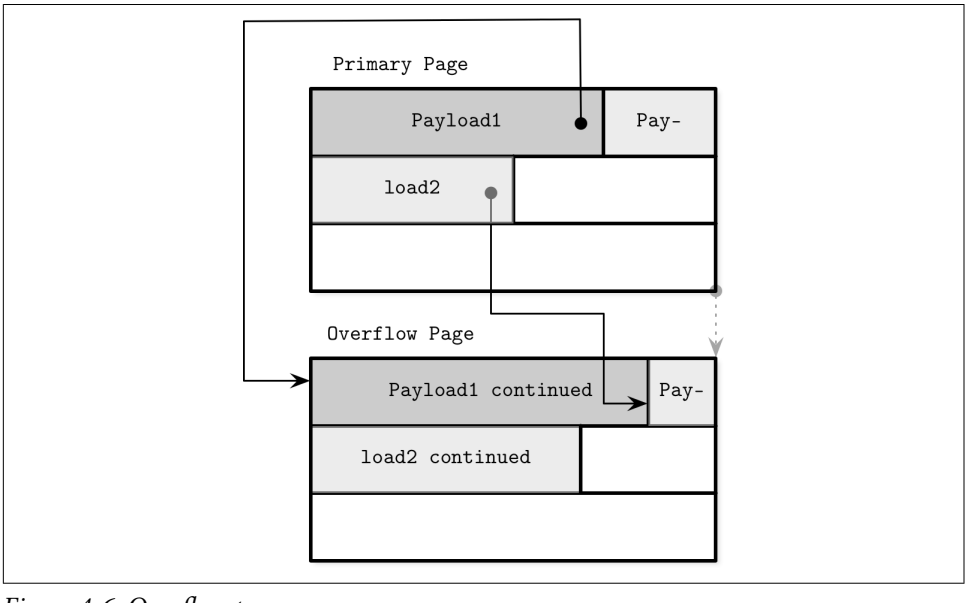

Figure 4-6. Overflow pages

Overflow pages require some extra bookkeeping, since they may get fragmented as well as primary pages, and we have to be able to reclaim this space to write new data, or discard the overflow page if it's not needed anymore.

When the first overflow page is allocated, its page ID is stored in the header of the primary page. If a single overflow page is not enough, multiple overflow pages are linked together by storing the next overflow page ID in the previous one's header. Several pages may have to be traversed to locate the overflow part for the given payload.

Since keys usually have high cardinality, storing a portion of a key makes sense, as most of the comparisons can be made on the trimmed key part that resides in the primary page.

For data records, we have to locate their overflow parts to return them to the user. However, this doesn't matter much, since it's an infrequent operation. If all data records are oversize, it is worth considering specialized blob storage for large values.

Binary Search

We've already discussed the B-Tree lookup algorithm (see "B-Tree Lookup Algorithm" on page 38) and mentioned that we locate a searched key within the node using the *binary search* algorithm. Binary search works *only* for sorted data. If keys are not ordered, they can't be binary searched. This is why keeping keys in order and maintaining a sorted invariant is essential.

The binary search algorithm receives an array of sorted items and a searched key, and returns a number. If the returned number is positive, we know that the searched key was found and the number specifies its position in the input array. A negative return value indicates that the searched key is not present in the input array and gives us an *insertion point*.

The insertion point is the index of the first element that is *greater than* the given key. An absolute value of this number is the index at which the searched key can be inserted to preserve order. Insertion can be done by shifting elements over one position, starting from an insertion point, to make space for the inserted element [SEDGEWICK11].

The majority of searches on higher levels do not result in exact matches, and we're interested in the search direction, in which case we have to find the first value that is greater than the searched one and follow the corresponding child link into the associated subtree.

Binary Search with Indirection Pointers

Cells in the B-Tree page are stored in the insertion order, and only cell offsets preserve the logical element order. To perform binary search through page cells, we pick the middle cell offset, follow its pointer to locate the cell, compare the key from this cell with the searched key to decide whether the search should continue left or right, and continue this process recursively until the searched element or the insertion point is found, as shown in Figure 4-7.

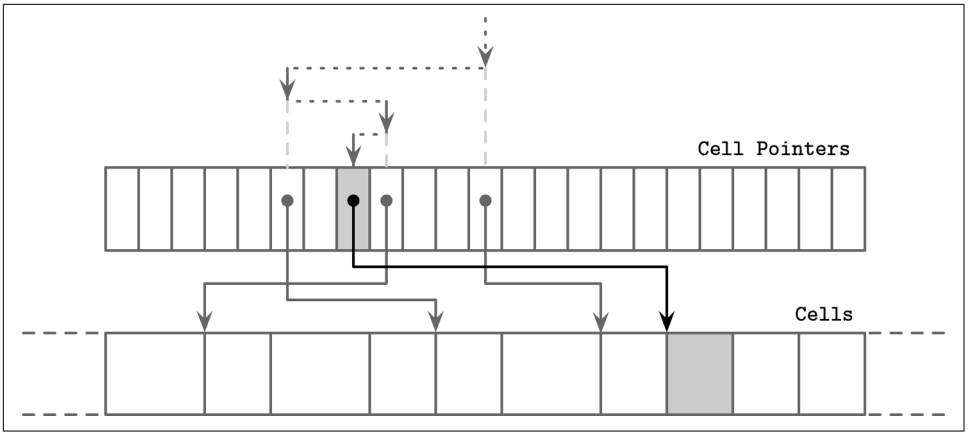

Figure 4-7. Binary search with indirection pointers. The searched element is shown in gray. Dotted arrows represent binary search through cell pointers. Solid lines represent accesses that follow the cell pointers, necessary to compare the cell value with a searched key.

Propagating Splits and Merges

As we've discussed in previous chapters, B-Tree splits and merges can propagate to higher levels. For that, we need to be able to traverse a chain back to the root node from the splitting leaf or a pair of merging leaves.

B-Tree nodes may include parent node pointers. Since pages from lower levels are always paged in when they're referenced from a higher level, it is not even necessary to persist this information on disk.

Just like sibling pointers (see "Sibling Links" on page 62), parent pointers have to be updated whenever the parent changes. This happens in all the cases when the separator key with the page identifier is transferred from one node to another: during the parent node splits, merges, or rebalancing of the parent node.

Some implementations (for example, WiredTiger) use parent pointers for leaf traversal to avoid deadlocks, which may happen when using sibling pointers (see [MILLER78], [LEHMAN81]). Instead of using sibling pointers to traverse leaf nodes, the algorithm employs parent pointers, much like we saw in Figure 4-1.

To address and locate a sibling, we can follow a pointer from the parent node and recursively descend back to the lower level. Whenever we reach the end of the parent node after traversing all the siblings sharing the parent, the search continues upward recursively, eventually reaching up to the root and continuing back down to the leaf level.

Breadcrumbs

Instead of storing and maintaining parent node pointers, it is possible to keep track of nodes traversed on the path to the target leaf node, and follow the chain of parent nodes in reverse order in case of cascading splits during inserts, or merges during deletes.

During operations that may result in structural changes of the B-Tree (insert or delete), we first traverse the tree from the root to the leaf to find the target node and the insertion point. Since we do not always know up front whether or not the operation will result in a split or merge (at least not until the target leaf node is located), we have to collect *breadcrumbs*.

Breadcrumbs contain references to the nodes followed from the root and are used to backtrack them in reverse when propagating splits or merges. The most natural data structure for this is a stack. For example, PostgreSQL stores breadcrumbs in a stack, internally referenced as BTStack.[2]

If the node is split or merged, breadcrumbs can be used to find insertion points for the keys pulled to the parent and to walk back up the tree to propagate structural changes to the higher-level nodes, if necessary. This stack is maintained in memory.

Figure 4-8 shows an example of root-to-leaf traversal, collecting breadcrumbs containing pointers to the visited nodes and cell indices. If the target leaf node is split, the item on top of the stack is popped to locate its immediate parent. If the parent node has enough space, a new cell is appended to it at the cell index from the breadcrumb (assuming the index is still valid). Otherwise, the parent node is split as well. This process continues recursively until either the stack is empty and we have reached the root, or there was no split on the level.

2 You can read more about it in the project repository: *https://databass.dev/links/21*.

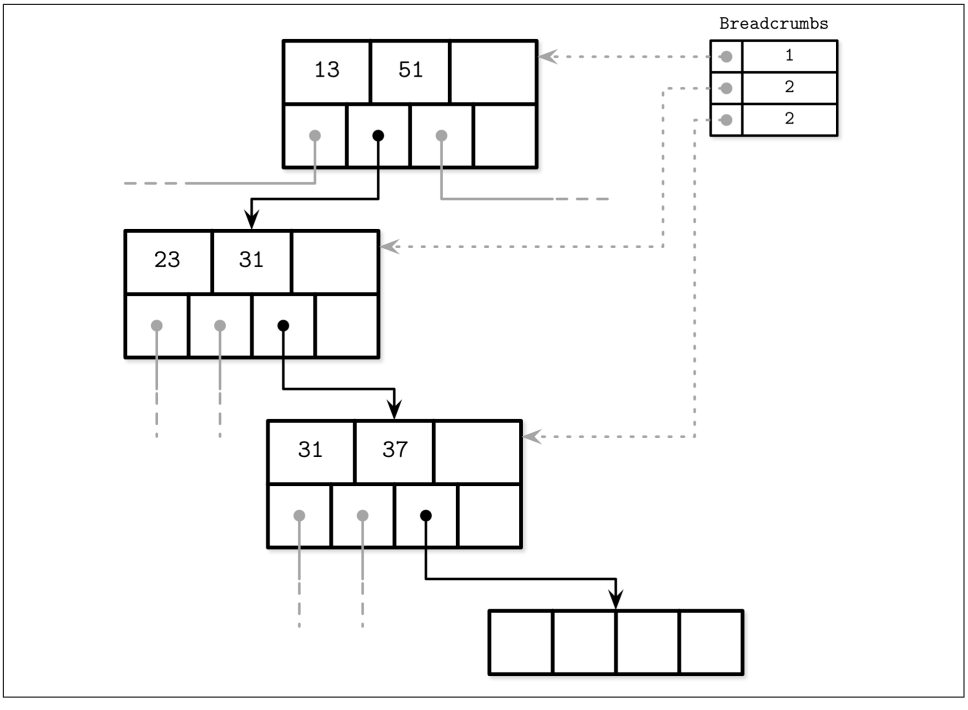

Figure 4-8. Breadcrumbs collected during lookup, containing traversed nodes and cell indices. Dotted lines represent logical links to visited nodes. Numbers in the breadcrumbs table represent indices of the followed child pointers.

Rebalancing

Some B-Tree implementations attempt to postpone split and merge operations to amortize their costs by *rebalancing* elements within the level, or moving elements from more occupied nodes to less occupied ones for as long as possible before finally performing a split or merge. This helps to improve node occupancy and may reduce the number of levels within the tree at a potentially higher maintenance cost of rebalancing.

Load balancing can be performed during insert and delete operations [GRAEFE11]. To improve space utilization, instead of splitting the node on overflow, we can transfer some of the elements to one of the sibling nodes and make space for the insertion. Similarly, during delete, instead of merging the sibling nodes, we may choose to move some of the elements from the neighboring nodes to ensure the node is at least half full.

B*-Trees keep distributing data between the neighboring nodes until both siblings are full [KNUTH98]. Then, instead of splitting a single node into two half-empty ones,

the algorithm splits two nodes into three nodes, each of which is two-thirds full. SQLite uses this variant in the implementation. This approach improves an average occupancy by postponing splits, but requires additional tracking and balancing logic. Higher utilization also means more efficient searches, because the height of the tree is smaller and fewer pages have to be traversed on the path to the searched leaf.

Figure 4-9 shows distributing elements between the neighboring nodes, where the left sibling contains more elements than the right one. Elements from the more occupied node are moved to the less occupied one. Since balancing changes the min/max invariant of the sibling nodes, we have to update keys and pointers at the parent node to preserve it.

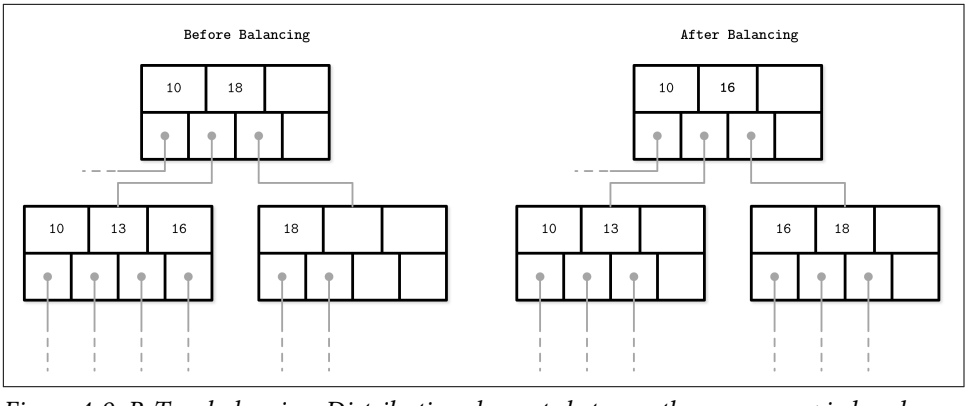

Figure 4-9. B-Tree balancing: Distributing elements between the more occupied node and the less occupied one

Load balancing is a useful technique used in many database implementations. For example, SQLite implements the *balance-siblings* algorithm, which is somewhat close to what we have described in this section. Balancing might add some complexity to the code, but since its use cases are isolated, it can be implemented as an optimization at a later stage.

Right-Only Appends

Many database systems use auto-incremented monotonically increasing values as primary index keys. This case opens up an opportunity for an optimization, since all the insertions are happening toward the end of the index (in the rightmost leaf), so most of the splits occur on the rightmost node on each level. Moreover, since the keys are monotonically incremented, given that the ratio of appends versus updates and deletes is low, nonleaf pages are also less fragmented than in the case of randomly ordered keys.

PostgreSQL is calling this case a *fastpath*. When the inserted key is strictly greater than the first key in the rightmost page, and the rightmost page has enough space to hold the newly inserted entry, the new entry is inserted into the appropriate location in the cached rightmost leaf, and the whole read path can be skipped.

SQLite has a similar concept and calls it *quickbalance*. When the entry is inserted on the far right end and the target node is full (i.e., it becomes the largest entry in the tree upon insertion), instead of rebalancing or splitting the node, it allocates the new rightmost node and adds its pointer to the parent (for more on implementing balancing in SQLite, see "Rebalancing" on page 70). Even though this leaves the newly created page nearly empty (instead of half empty in the case of a node split), it is very likely that the node will get filled up shortly.

Bulk Loading

If we have presorted data and want to bulk load it, or have to rebuild the tree (for example, for defragmentation), we can take the idea with right-only appends even further. Since the data required for tree creation is already sorted, during bulk loading we only need to append the items at the rightmost location in the tree.

In this case, we can avoid splits and merges altogether and compose the tree from the bottom up, writing it out level by level, or writing out higher-level nodes as soon as we have enough pointers to already written lower-level nodes.

One approach for implementing bulk loading is to write presorted data on the leaf level page-wise (rather then inserting individual elements). After the leaf page is written, we propagate its first key to the parent and use a normal algorithm for building higher B-Tree levels [RAMAKRISHNAN03]. Since appended keys are given in the sorted order, all splits in this case occur on the rightmost node.

Since B-Trees are always built starting from the bottom (leaf) level, the complete leaf level can be written out before any higher-level nodes are composed. This allows having all child pointers at hand by the time the higher levels are constructed. The main benefits of this approach are that we do not have to perform any splits or merges on disk and, at the same time, have to keep only a minimal part of the tree (i.e., all parents of the currently filling leaf node) in memory for the time of construction.

Immutable B-Trees can be created in the same manner but, unlike mutable B-Trees, they require no space overhead for subsequent modifications, since all operations on a tree are final. All pages can be completely filled up, improving occupancy and resulting into better performance.

Compression

Storing the raw, uncompressed data can induce significant overhead, and many databases offer ways to compress it to save space. The apparent trade-off here is between access speed and compression ratio: larger compression ratios can improve data size, allowing you to fetch more data in a single access, but might require more RAM and CPU cycles to compress and decompress it.

Compression can be done at different granularity levels. Even though compressing entire files can yield better compression ratios, it has limited application as a whole file has to be recompressed on an update, and more granular compression is usually better-suited for larger datasets. Compressing an entire index file is both impractical and hard to implement efficiently: to address a particular page, the whole file (or its section containing compression metadata) has to be accessed (in order to locate a compressed section), decompressed, and made available.

An alternative is to compress data page-wise. It fits our discussion well, since the algorithms we've been discussing so far use fixed-size pages. Pages can be compressed and uncompressed independently from one another, allowing you to couple compression with page loading and flushing. However, a compressed page in this case can occupy only a fraction of a disk block and, since transfers are usually done in units of disk blocks, it might be necessary to page in extra bytes [RAY95]. In Figure 4-10, you can see a compressed page (a) taking less space than the disk block. When we load this page, we also page in additional bytes that belong to the other page. With pages that span multiple disk blocks, like (b) in the same image, we have to read an additional block.

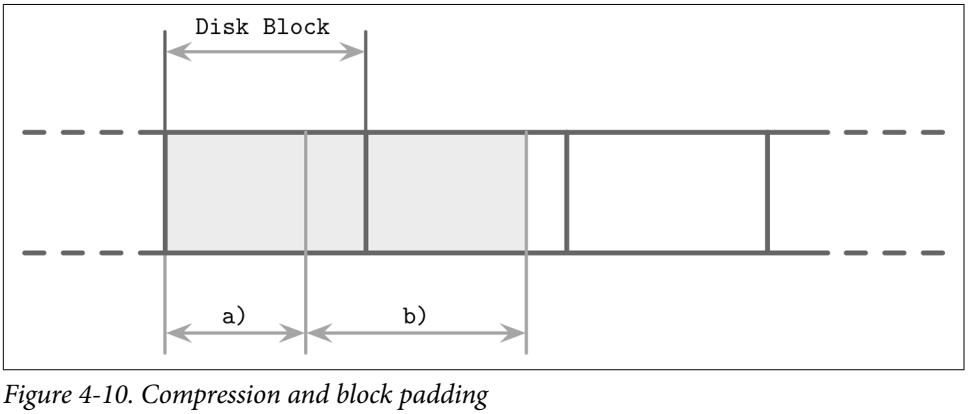

Figure 4-10. Compression and block padding

Another approach is to compress data only, either row-wise (compressing entire data records) or column-wise (compressing columns individually). In this case, page management and compression are decoupled.

Most of the open source databases reviewed while writing this book have pluggable compression methods, using available libraries such as Snappy, zLib, lz4, and many others.

As compression algorithms yield different results depending on a dataset and potential objectives (e.g., compression ratio, performance, or memory overhead), we will not go into comparison and implementation details in this book. There are many overviews available that evaluate different compression algorithms for different block sizes (for example, Squash Compression Benchmark), usually focusing on four metrics: memory overhead, compression performance, decompression performance, and compression ratio. These metrics are important to consider when picking a compression library.

Vacuum and Maintenance

So far we've been mostly talking about user-facing operations in B-Trees. However, there are other processes that happen in parallel with queries that maintain storage integrity, reclaim space, reduce overhead, and keep pages in order. Performing these operations in the background allows us to save some time and avoid paying the price of cleanup during inserts, updates, and deletes.

The described design of slotted pages (see "Slotted Pages" on page 52) requires maintenance to be performed on pages to keep them in good shape. For example, subsequent splits and merges in internal nodes or inserts, updates, and deletes on the leaf level can result in a page that has enough *logical* space but does not have enough *contiguous* space, since it is fragmented. Figure 4-11 shows an example of such a situation: the page still has some logical space available, but it's fragmented and is split between the two deleted (garbage) records and some remaining free space between the header/cell pointers and cells.

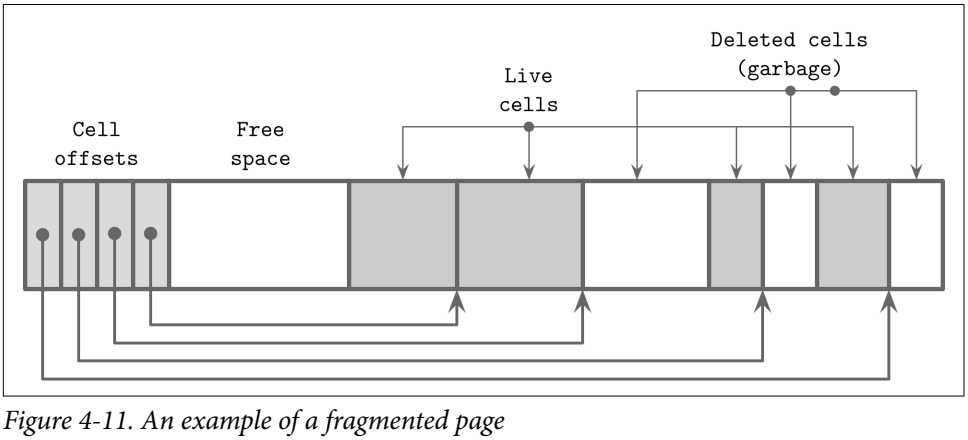

Figure 4-11. An example of a fragmented page

B-Trees are navigated from the root level. Data records that can be reached by following pointers down from the root node are *live* (addressable). Nonaddressable data records are said to be *garbage*: these records are not referenced anywhere and cannot be read or interpreted, so their contents are as good as nullified.

You can see this distinction in Figure 4-11: cells that still have pointers to them are addressable, unlike the removed or overwritten ones. Zero-filling of garbage areas is often skipped for performance reasons, as eventually these areas are overwritten by the new data anyway.

Fragmentation Caused by Updates and Deletes

Let's consider under which circumstances pages get into the state where they have nonaddressable data and have to be compacted. On the leaf level, deletes only remove cell offsets from the header, leaving the cell itself intact. After this is done, the cell is not *addressable* anymore, its contents will not appear in the query results, and nullifying it or moving neighboring cells is not necessary.

When the page is split, only offsets are trimmed and, since the rest of the page is not addressable, cells whose offsets were truncated are not reachable, so they will be overwritten whenever the new data arrives, or garbage-collected when the vacuum process kicks in.

Some databases rely on garbage collection, and leave removed and updated cells in place for multiversion concurrency control (see "Multiversion Concurrency Control" on page 99). Cells remain accessible for the concurrently executing transactions until the update is complete, and can be collected as soon as no other thread accesses them. Some databases maintain structures that track *ghost* records, which are collected as soon as all transactions that may have seen them complete [WEIKUM01].

Since deletes only discard cell offsets and do not relocate remaining cells or physically remove the target cells to occupy the freed space, freed bytes might end up scattered across the page. In this case, we say that the page is *fragmented* and requires defragmentation.

To make a write, we often need a contiguous block of free bytes where the cell fits. To put the freed fragments back together and fix this situation, we have to *rewrite* the page.

Insert operations leave tuples in their insertion order. This does not have as significant an impact, but having naturally sorted tuples can help with cache prefetch during sequential reads.

Updates are mostly applicable to the leaf level: internal page keys are used for guided navigation and only define subtree boundaries. Additionally, updates are performed on a per-key basis, and generally do not result in structural changes in the tree, apart from the creation of overflow pages. On the leaf level, however, update operations do not change cell order and attempt to avoid page rewrite. This means that multiple versions of the cell, only one of which is addressable, may end up being stored.

Page Defragmentation

The process that takes care of space reclamation and page rewrites is called *compaction*, *vacuum*, or just *maintenance*. Page rewrites can be done synchronously on write if the page does not have enough free physical space (to avoid creating unnecessary overflow pages), but compaction is mostly referred to as a distinct, asynchronous process of walking through pages, performing garbage collection, and rewriting their contents.

This process reclaims the space occupied by dead cells, and rewrites cells in their logical order. When pages are rewritten, they may also get relocated to new positions in the file. Unused in-memory pages become available and are returned to the page cache. IDs of the newly available on-disk pages are added to the *free page list* (sometimes called a *freelist*[3]). This information has to be persisted to survive node crashes and restarts, and to make sure free space is not lost or leaked.

Summary

In this chapter, we discussed the concepts specific to on-disk B-Tree implementations, such as:

Page header
: What information is usually stored there.

Rightmost pointers
: These are not paired with separator keys, and how to handle them.

High keys
: Determine the maximum allowed key that can be stored in the node.

Overflow pages
: Allow you to store oversize and variable-size records using fixed-size pages.

3 For example, SQLite maintains a list of pages that are not used by the database, where *trunk* pages are held in a linked list and hold addresses of freed pages.

After that, we went through some details related to root-to-leaf traversals:

- How to perform binary search with indirection pointers
- How to keep track of tree hierarchies using parent pointers or breadcrumbs

Lastly, we went through some optimization and maintenance techniques:

Rebalancing
: Moves elements between neighboring nodes to reduce a number of splits and merges.

Right-only appends
: Appends the new rightmost cell instead of splitting it under the assumption that it will quickly fill up.

Bulk loading
: A technique for efficiently building B-Trees from scratch from sorted data.

Garbage collection
: A process that rewrites pages, puts cells in key order, and reclaims space occupied by unaddressable cells.

These concepts should bridge the gap between the basic B-Tree algorithm and a real-world implementation, and help you better understand how B-Tree–based storage systems work.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Disk-based B-Trees
: Graefe, Goetz. 2011. "Modern B-Tree Techniques." *Foundations and Trends in Databases* 3, no. 4 (April): 203-402. *https://doi.org/10.1561/1900000028*.

  Healey, Christopher G. 2016. *Disk-Based Algorithms for Big Data (1st Ed.)*. Boca Raton: CRC Press.

CHAPTER 5
Transaction Processing and Recovery

In this book, we've taken a bottom-up approach to database system concepts: we first learned about storage structures. Now, we're ready to move to the higher-level components responsible for buffer management, lock management, and recovery, which are the prerequisites for understanding database transactions.

A *transaction* is an indivisible logical unit of work in a database management system, allowing you to represent multiple operations as a single step. Operations executed by transactions include reading and writing database records. A database transaction has to preserve atomicity, consistency, isolation, and durability. These properties are commonly referred as *ACID* [HAERDER83]:

Atomicity
: Transaction steps are *indivisible*, which means that either *all* the steps associated with the transaction execute successfully or none of them do. In other words, transactions should not be applied partially. Each transaction can either *commit* (make all changes from write operations executed during the transaction visible), or *abort* (roll back all transaction side effects that haven't yet been made visible). Commit is a final operation. After an abort, the transaction can be retried.

Consistency
: Consistency is an application-specific guarantee; a transaction should only bring the database from one valid state to another valid state, maintaining all database invariants (such as constraints, referential integrity, and others). Consistency is the most weakly defined property, possibly because it is the only property that is controlled by the user and not only by the database itself.

Isolation
: Multiple concurrently executing transactions should be able to run without interference, as if there were no other transactions executing at the same time.

Isolation defines *when* the changes to the database state may become visible, and what changes may become visible to the concurrent transactions. Many databases use isolation levels that are weaker than the given definition of isolation for performance reasons. Depending on the methods and approaches used for concurrency control, changes made by a transaction may or may not be visible to other concurrent transactions (see "Isolation Levels" on page 96).

Durability
: Once a transaction has been committed, all database state modifications have to be persisted on disk and be able to survive power outages, system failures, and crashes.

Implementing transactions in a database system, in addition to a storage structure that organizes and persists data on disk, requires several components to work together. On the node locally, the *transaction manager* coordinates, schedules, and tracks transactions and their individual steps.

The *lock manager* guards access to these resources and prevents concurrent accesses that would violate data integrity. Whenever a lock is requested, the lock manager checks if it is already held by any other transaction in shared or exclusive mode, and grants access to it if the requested access level results in no contradiction. Since exclusive locks can be held by at most one transaction at any given moment, other transactions requesting them have to wait until locks are released, or abort and retry later. As soon as the lock is released or whenever the transaction terminates, the lock manager notifies one of the pending transactions, letting it acquire the lock and continue.

The *page cache* serves as an intermediary between persistent storage (disk) and the rest of the storage engine. It stages state changes in main memory and serves as a cache for the pages that haven't been synchronized with persistent storage. All changes to a database state are first applied to the cached pages.

The *log manager* holds a history of operations (log entries) applied to cached pages but not yet synchronized with persistent storage to guarantee they won't be lost in case of a crash. In other words, the log is used to reapply these operations and reconstruct the cached state during startup. Log entries can also be used to undo changes done by the aborted transactions.

Distributed (multipartition) transactions require additional coordination and remote execution. We discuss distributed transaction protocols in Chapter 13.

Buffer Management

Most databases are built using a two-level memory hierarchy: slower persistent storage (disk) and faster main memory (RAM). To reduce the number of accesses to persistent storage, pages are *cached* in memory. When the page is requested again by the storage layer, its cached copy is returned.

Cached pages available in memory can be reused under the assumption that no other process has modified the data on disk. This approach is sometimes referenced as *virtual disk* [BAYER72]. A virtual disk read accesses physical storage only if no copy of the page is already available in memory. A more common name for the same concept is *page cache* or *buffer pool*. The page cache is responsible for caching pages read from disk in memory. In case of a database system crash or unorderly shutdown, cached contents are lost.

Since the term *page cache* better reflects the purpose of this structure, this book defaults to this name. The term *buffer pool* sounds like its primary purpose is to pool and reuse *empty* buffers, without sharing their contents, which can be a useful part of a page cache or even as a separate component, but does not reflect the entire purpose as precisely.

The problem of caching pages is not limited in scope to databases. Operating systems have the concept of a page cache, too. Operating systems utilize *unused* memory segments to transparently cache disk contents to improve performance of I/O syscalls.

Uncached pages are said to be *paged in* when they're loaded from disk. If any changes are made to the cached page, it is said to be *dirty*, until these changes are *flushed* back on disk.

Since the memory region where cached pages are held is usually substantially smaller than an entire dataset, the page cache eventually fills up and, in order to page in a new page, one of the cached pages has to be *evicted*.

In Figure 5-1, you can see the relation between the logical representation of B-Tree pages, their cached versions, and the pages on disk. The page cache loads pages into free slots out of order, so there's no direct mapping between how pages are ordered on disk and in memory.

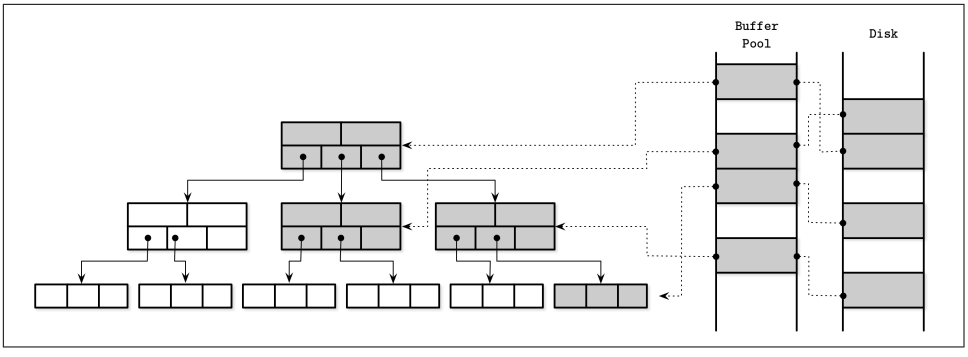

Figure 5-1. Page cache

The primary functions of a page cache can be summarized as:

- It keeps cached page contents in memory.
- It allows modifications to on-disk pages to be *buffered* together and performed against their cached versions.
- When a requested page isn't present in memory and there's enough space available for it, it is *paged in* by the page cache, and its cached version is returned.
- If an already cached page is requested, its cached version is returned.
- If there's not enough space available for the new page, some other page is *evicted* and its contents are *flushed* to disk.

Bypassing the Kernel Page Cache

Many database systems open files using `O_DIRECT` flag. This flag allows I/O system calls to bypass the kernel page cache, access the disk directly, and use database-specific buffer management. This is sometimes frowned upon by the operating systems folks.

Linus Torvalds has criticized usage of `O_DIRECT` since it's not asynchronous and has no readahead or other means for instructing the kernel about access patterns. However, until operating systems start offering better mechanisms, `O_DIRECT` is still going to be useful.

We can gain some control over how the kernel evicts pages from its cache is by using `fadvise`, but this only allows us to ask the kernel to consider our opinion and does not guarantee it will actually happen. To avoid syscalls when performing I/O, we can use memory mapping, but then we lose control over caching.

Caching Semantics

All changes made to buffers are kept in memory until they are eventually written back to disk. As no other process is allowed to make changes to the backing file, this synchronization is a one-way process: from memory to disk, and not vice versa. The page cache allows the database to have more control over memory management and disk accesses. You can think of it as an application-specific equivalent of the kernel page cache: it accesses the block device directly, implements similar functionality, and serves a similar purpose. It abstracts disk accesses and decouples logical write operations from the physical ones.

Caching pages helps to keep the tree partially in memory without making additional changes to the algorithm and materializing objects in memory. All we have to do is replace disk accesses by the calls to the page cache.

When the storage engine accesses (in other words, requests) the page, we first check if its contents are already cached, in which case the cached page contents are returned. If the page contents are not yet cached, the cache translates the logical page address or page number to its physical address, loads its contents in memory, and returns its cached version to the storage engine. Once returned, the buffer with cached page contents is said to be *referenced*, and the storage engine has to hand it back to the page cache or dereference it once it's done. The page cache can be instructed to avoid evicting pages by *pinning* them.

If the page is modified (for example, a cell was appended to it), it is marked as dirty. A dirty flag set on the page indicates that its contents are out of sync with the disk and have to be flushed for durability.

Cache Eviction

Keeping caches populated is good: we can serve more reads without going to persistent storage, and more same-page writes can be buffered together. However, the page cache has a limited capacity and, sooner or later, to serve the new contents, old pages have to be evicted. If page contents are in sync with the disk (i.e., were already flushed or were never modified) and the page is not pinned or referenced, it can be evicted right away. Dirty pages have to be *flushed* before they can be evicted. Referenced pages should not be evicted while some other thread is using them.

Since triggering a flush on every eviction might be bad for performance, some databases use a separate background process that cycles through the dirty pages that are likely to be evicted, updating their disk versions. For example, PostgreSQL has a background flush writer that does just that.

Another important property to keep in mind is *durability*: if the database has crashed, all data that was not flushed is lost. To make sure that all changes are persisted, flushes are coordinated by the *checkpoint* process. The checkpoint process controls

the write-ahead log (WAL) and page cache, and ensures that they work in lockstep. Only log records associated with operations applied to cached pages that were flushed can be discarded from the WAL. Dirty pages cannot be evicted until this process completes.

This means there is always a trade-off between several objectives:

- Postpone flushes to reduce the number of disk accesses
- Preemptively flush pages to allow quick eviction
- Pick pages for eviction and flush in the optimal order
- Keep cache size within its memory bounds
- Avoid losing the data as it is not persisted to the primary storage

We explore several techniques that help us to improve the first three characteristics while keeping us within the boundaries of the other two.

Locking Pages in Cache

Having to perform disk I/O on each read or write is impractical: subsequent reads may request the same page, just as subsequent writes may modify the same page. Since B-Tree gets "narrower" toward the top, higher-level nodes (ones that are closer to the root) are hit for most of the reads. Splits and merges also eventually propagate to the higher-level nodes. This means there's always at least a part of a tree that can significantly benefit from being cached.

We can "lock" pages that have a high probability of being used in the nearest time. Locking pages in the cache is called *pinning*. Pinned pages are kept in memory for a longer time, which helps to reduce the number of disk accesses and improve performance [GRAEFE11].

Since each lower B-Tree node level has exponentially more nodes than the higher one, and higher-level nodes represent just a small fraction of the tree, this part of the tree can reside in memory permanently, and other parts can be paged in on demand. This means that, in order to perform a query, we won't have to make `h` disk accesses (as discussed in "B-Tree Lookup Complexity" on page 37, `h` is the height of the tree), but only hit the disk for the lower levels, for which pages are not cached.

Operations performed against a subtree may result in structural changes that contradict each other—for example, multiple delete operations causing merges followed by writes causing splits, or vice versa. Likewise for structural changes that propagate from different subtrees (structural changes occurring close to each other in time, in different parts of the tree, propagating up). These operations can be buffered together by applying changes only in memory, which can reduce the number of disk writes

and amortize the operation costs, since only one write can be performed instead of multiple writes.

Prefetching and Immediate Eviction

The page cache also allows the storage engine to have fine-grained control over prefetching and eviction. It can be instructed to load pages ahead of time, before they are accessed. For example, when the leaf nodes are traversed in a range scan, the next leaves can be preloaded. Similarly, if a maintenance process loads the page, it can be evicted immediately after the process finishes, since it's unlikely to be useful for the in-flight queries. Some databases, for example, PostgreSQL, use a circular buffer (in other words, FIFO page replacement policy) for large sequential scans.

Page Replacement

When cache capacity is reached, to load new pages, old ones have to be evicted. However, unless we evict pages that are least likely to be accessed again soon, we might end up loading them several times subsequently even though we could've just kept them in memory for all that time. We need to find a way to estimate the likelihood of subsequent page access to optimize this.

For this, we can say that pages should be evicted according to the *eviction policy* (also sometimes called the *page-replacement* policy). It attempts to find pages that are least likely to be accessed again any time soon. When the page is evicted from the cache, the new page can be loaded in its place.

For a page cache implementation to be performant, it needs an efficient page-replacement algorithm. An ideal page-replacement strategy would require a crystal ball that would predict the order in which pages are going to be accessed and evict only pages that will not be touched for the longest time. Since requests do not necessarily follow any specific pattern or distribution, precisely predicting behavior can be complicated, but using a right page replacement strategy can help to reduce the number of evictions.

It seems logical that we can reduce the number of evictions by simply using a larger cache. However, this does not appear to be the case. One of the examples demonstrating this dilemma this is called *Bélády's anomaly* [BEDALY69]. It shows that increasing the number of pages might increase the number of evictions if the used page-replacement algorithm is not optimal. When pages that might be required soon are evicted and then loaded again, pages start competing for space in the cache. Because of that, we need to wisely consider the algorithm we're using, so that it would improve the situation, not make it worse.

FIFO and LRU

The most naïve page-replacement strategy is first in, first out (*FIFO*). FIFO maintains a queue of page IDs in their insertion order, adding new pages to the tail of the queue. Whenever the page cache is full, it takes the element from the head of the queue to find the page that was paged in at the farthest point in time. Since it does not account for subsequent page accesses, only for page-in events, this proves to be impractical for the most real-world systems. For example, the root and topmost-level pages are paged in first and, according to this algorithm, are the first candidates for eviction, even though it's clear from the tree structure that these pages are likely to paged in again soon, if not immediately.

A natural extension of the FIFO algorithm is *least-recently used* (LRU) [TANENBAUM14]. It also maintains a queue of eviction candidates in insertion order, but allows you to place a page back to the tail of the queue on repeated accesses, as if this was the first time it was paged in. However, updating references and relinking nodes on every access can become expensive in a concurrent environment.

There are other LRU-based cache eviction strategies. For example, 2Q (Two-Queue LRU) maintains two queues and puts pages into the first queue during the initial access and moves them to the second *hot* queue on subsequent accesses, allowing you to distinguish between the recently and frequently accessed pages [JONSON94]. LRU-K identifies frequently referenced pages by keeping track of the last `K` accesses, and using this information to estimate access times on a page basis [ONEIL93].

CLOCK

In some situations, efficiency may be more important than precision. *CLOCK* algorithm variants are often used as compact, cache-friendly, and concurrent alternatives to LRU [SOUNDARARARJAN06]. Linux, for example, uses a variant of the CLOCK algorithm.

CLOCK-sweep holds references to pages and associated access bits in a circular buffer. Some variants use counters instead of bits to account for frequency. Every time the page is accessed, its access bit is set to `1`. The algorithm works by going around the circular buffer, checking access bits:

- If the access bit is `1`, and the page is unreferenced, it is set to `0`, and the next page is inspected.
- If the access bit is already `0`, the page becomes a *candidate* and is scheduled for eviction.
- If the page is currently referenced, its access bit remains unchanged. It is assumed that the access bit of an accessed page cannot be 0, so it cannot be evicted. This makes referenced pages less likely to be replaced.

Figure 5-2 shows a circular buffer with access bits.

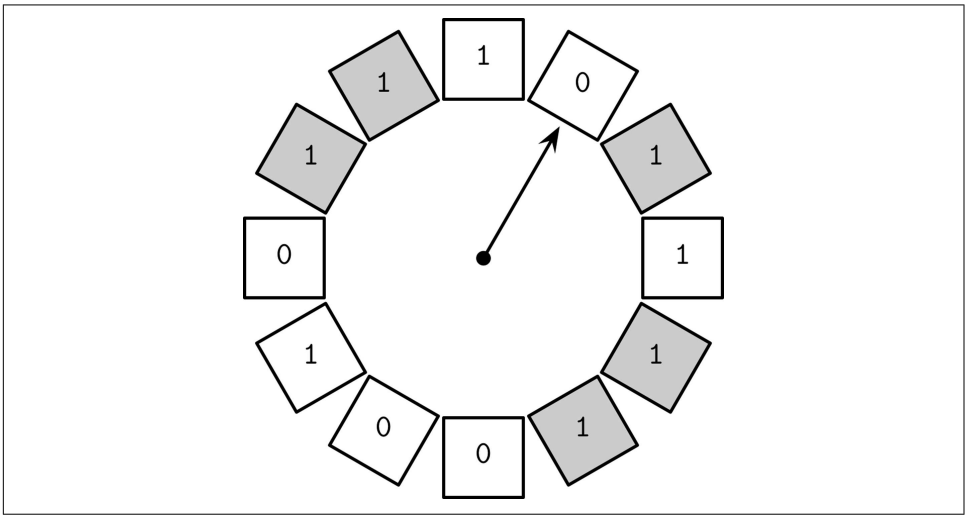

Figure 5-2. CLOCK-sweep example. Counters for currently referenced pages are shown in gray. Counters for unreferenced pages are shown in white. The arrow points to the element that will be checked next.

An advantage of using a circular buffer is that both the clock hand pointer and contents can be modified using compare-and-swap operations, and do not require additional locking mechanisms. The algorithm is easy to understand and implement and is often used in both textbooks [TANENBAUM14] and real-wold systems.

LRU is not always the best replacement strategy for a database system. Sometimes, it may be more practical to consider *usage frequency* rather than *recency* as a predictive factor. In the end, for a database system under a heavy load, recency might not be very indicative as it only represents the order in which items were accessed.

LFU

To improve the situation, we can start tracking *page reference events* rather than *page-in events*. One of the approaches allowing us to do this tracks least-frequently used (LFU) pages.

TinyLFU, a frequency-based page-eviction policy [EINZIGER17], does precisely this: instead of evicting pages based on *page-in recency*, it orders pages by *usage frequency*. It is implemented in the popular Java library called Caffeine.

TinyLFU uses a frequency histogram [CORMODE11] to maintain compact cache access history, since preserving an entire history might be prohibitively expensive for practical purposes.

Elements can be in one of the three queues:

- *Admission*, maintaining newly added elements, implemented using LRU policy.
- *Probation*, holding elements most likely to get evicted.
- *Protected*, holding elements that are to stay in the queue for a longer time.

Rather than choosing which elements to evict every time, this approach chooses which ones to promote for retention. Only the items that have a frequency larger than the item that would be evicted as a result of promoting them, can be moved to the probation queue. On subsequent accesses, items can get moved from probation to the protected queue. If the protected queue is full, one of the elements from it may have to be placed back into probation. More frequently accessed items have a higher chance of retention, and less frequently used ones are more likely to be evicted.

Figure 5-3 shows the logical connections between the admission, probation, and protected queues, the frequency filter, and eviction.

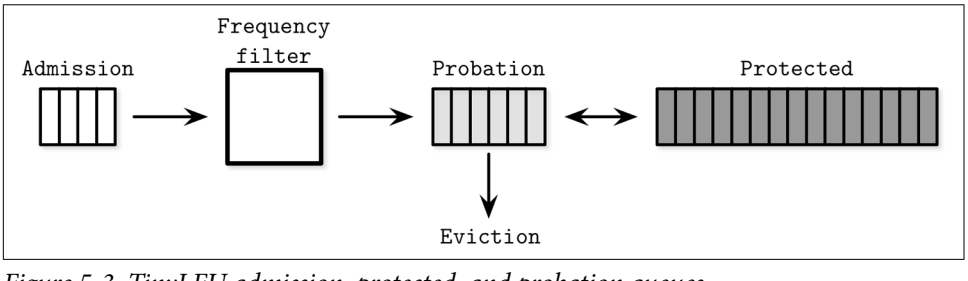

Figure 5-3. TinyLFU admission, protected, and probation queues

There are many other algorithms that can be used for optimal cache eviction. The choice of a page-replacement strategy has a significant impact on latency and the number of performed I/O operations, and has to be taken into consideration.

Recovery

Database systems are built on top of several hardware and software layers that can have their own stability and reliability problems. Database systems themselves, as well as the underlying software and hardware components, may fail. Database implementers have to consider these failure scenarios and make sure that the data that was "promised" to be written is, in fact, written.

A *write-ahead log* (WAL for short, also known as a *commit log*) is an append-only auxiliary disk-resident structure used for crash and transaction recovery. The page cache allows buffering changes to page contents in memory. Until the cached contents are flushed back to disk, the only disk-resident copy preserving the operation

history is stored in the WAL. Many database systems use append-only write-ahead logs; for example, PostgreSQL and MySQL.

The main functionality of a write-ahead log can be summarized as:

- Allow the page cache to buffer updates to disk-resident pages while ensuring durability semantics in the larger context of a database system.
- Persist all operations on disk until the cached copies of pages affected by these operations are synchronized on disk. Every operation that modifies the database state has to be logged on disk *before* the contents of the associated pages can be modified.
- Allow lost in-memory changes to be reconstructed from the operation log in case of a crash.

In addition to this functionality, the write-ahead log plays an important role in transaction processing. It is hard to overstate the importance of the WAL as it ensures that data makes it to the persistent storage and is available in case of a crash, as uncommitted data is replayed from the log and the pre-crash database state is fully restored. In this section, we will often refer to ARIES (Algorithm for Recovery and Isolation Exploiting Semantics), a state-of-the-art recovery algorithm that is widely used and cited [MOHAN92].

PostgreSQL Versus fsync()

PostgreSQL uses checkpoints to ensure that index and data files have been updated with all information up to a certain record in the logfile. Flushing all dirty (modified) pages at once is done periodically by the checkpoint process. Synchronizing dirty page contents with disk is done by making the `fsync()` kernel call, which is supposed to sync dirty pages to disk, and unset the *dirty* flag on the kernel pages. As you would expect, `fsync` returns with an error if it isn't able to flush pages on disk.

In Linux and a few other operating systems, `fsync` unsets the dirty flag *even* from unsuccessfully flushed pages after I/O errors. Additionally, errors will be reported only to the file descriptors that were open at the time of failure, so `fsync` will *not* return any errors that have occurred *before* the descriptor it was called upon was opened [CORBET18].

Since the checkpointer doesn't keep all files open at any given point in time, it may happen that it misses error notifications. Because dirty page flags are cleared, the checkpointer will assume that data has successfully made it on disk while, in fact, it might have not been written.

A combination of these behaviors can be a source of data loss or database corruption in the presence of potentially recoverable failures. Such behaviors can be difficult to detect and some of the states they lead to may be unrecoverable. Sometimes, even

triggering such behavior can be nontrivial. When working on recovery mechanisms, we should always take extra care and think through and attempt to test every possible failure scenario.

Log Semantics

The write-ahead log is append-only and its written contents are immutable, so all writes to the log are sequential. Since the WAL is an immutable, append-only data structure, readers can safely access its contents up to the latest write threshold while the writer continues appending data to the log tail.

The WAL consists of log records. Every record has a unique, monotonically increasing *log sequence number* (LSN). Usually, the LSN is represented by an internal counter or a timestamp. Since log records do not necessarily occupy an entire disk block, their contents are cached in the *log buffer* and are flushed on disk in a *force* operation. Forces happen as the log buffers fill up, and can be requested by the transaction manager or a page cache. All log records have to be flushed on disk in LSN order.

Besides individual operation records, the WAL holds records indicating transaction completion. A transaction can't be considered committed until the log is forced up to the LSN of its commit record.

To make sure the system can continue functioning correctly after a crash during rollback or recovery, some systems use *compensation log records* (CLR) during undo and store them in the log.

The WAL is usually coupled with a primary storage structure by the interface that allows *trimming* it whenever a *checkpoint* is reached. Logging is one of the most critical correctness aspects of the database, which is somewhat tricky to get right: even the slightest disagreements between log trimming and ensuring that the data has made it to the primary storage structure may cause data loss.

Checkpoints are a way for a log to know that log records up to a certain mark are fully persisted and aren't required anymore, which significantly reduces the amount of work required during the database startup. A process that forces *all* dirty pages to be flushed on disk is generally called a *sync checkpoint*, as it fully synchronizes the primary storage structure.

Flushing the entire contents on disk is rather impractical and would require pausing all running operations until the checkpoint is done, so most database systems implement *fuzzy checkpoints*. In this case, the `last_checkpoint` pointer stored in the log header contains the information about the last successful checkpoint. A fuzzy checkpoint begins with a special `begin_checkpoint` log record specifying its start, and ends with `end_checkpoint` log record, containing information about the dirty pages, and the contents of a transaction table. Until all the pages specified by this record are

flushed, the checkpoint is considered to be *incomplete*. Pages are flushed asynchronously and, once this is done, the `last_checkpoint` record is updated with the LSN of the `begin_checkpoint` record and, in case of a crash, the recovery process will start from there [MOHAN92].

Operation Versus Data Log

Some database systems, for example System R [CHAMBERLIN81], use *shadow paging*: a copy-on-write technique ensuring data durability and transaction atomicity. New contents are placed into the new unpublished *shadow* page and made visible with a pointer flip, from the old page to the one holding updated contents.

Any state change can be represented by a before-image and an after-image or by corresponding redo and undo operations. Applying a *redo* operation to a *before-image* produces an *after-image*. Similarly, applying an *undo* operation to an *after-image* produces a *before-image*.

We can use a physical log (that stores complete page state or byte-wise changes to it) or a logical log (that stores operations that have to be performed against the current state) to move records or pages from one state to the other, both backward and forward in time. It is important to track the *exact* state of the pages that physical and logical log records can be applied to.

Physical logging records before and after images, requiring entire pages affected by the operation to be logged. A logical log specifies which operations have to be applied to the page, such as `"insert a data record X for key Y"`, and a corresponding undo operation, such as `"remove the value associated with Y"`.

In practice, many database systems use a combination of these two approaches, using logical logging to perform an undo (for concurrency and performance) and physical logging to perform a redo (to improve recovery time) [MOHAN92].

Steal and Force Policies

To determine when the changes made in memory have to be flushed on disk, database management systems define steal/no-steal and force/no-force policies. These policies are *mostly* applicable to the page cache, but they're better discussed in the context of recovery, since they have a significant impact on which recovery approaches can be used in combination with them.

A recovery method that allows flushing a page modified by the transaction even before the transaction has committed is called a *steal* policy. A *no-steal* policy does not allow flushing any uncommitted transaction contents on disk. To *steal* a dirty page here means flushing its in-memory contents to disk and loading a different page from disk in its place.

A *force* policy requires all pages modified by the transactions to be flushed on disk *before* the transaction commits. On the other hand, a *no-force* policy allows a transaction to commit even if some pages modified during this transaction were not yet flushed on disk. To *force* a dirty page here means to flush it on disk before the commit.

Steal and force policies are important to understand, since they have implications for transaction undo and redo. *Undo* rolls back updates to forced pages for committed transactions, while *redo* applies changes performed by committed transactions on disk.

Using the *no-steal* policy allows implementing recovery using only redo entries: old copy is contained in the page on disk and modification is stored in the log [WEIKUM01]. With *no-force*, we potentially can buffer several updates to pages by *deferring* them. Since page contents have to be cached in memory for that time, a larger page cache may be needed.

When the *force* policy is used, crash recovery doesn't need any additional work to reconstruct the results of committed transactions, since pages modified by these transactions are already flushed. A major drawback of using this approach is that transactions take longer to commit due to the necessary I/O.

More generally, *until* the transaction commits, we need to have enough information to undo its results. If any pages touched by the transaction are flushed, we need to keep undo information in the log until it commits to be able to roll it back. Otherwise, we have to keep redo records in the log until it commits. In both cases, transaction *cannot* commit until either undo or redo records are written to the logfile.

ARIES

ARIES is a *steal/no-force* recovery algorithm. It uses physical redo to improve performance during recovery (since changes can be installed quicker) and logical undo to improve concurrency during normal operation (since logical undo operations can be applied to pages independently). It uses WAL records to implement *repeating history* during recovery, to completely reconstruct the database state before undoing uncommitted transactions, and creates compensation log records during undo [MOHAN92].

When the database system restarts after the crash, recovery proceeds in three phases:

1. The *analysis* phase identifies dirty pages in the page cache and transactions that were in progress at the time of a crash. Information about dirty pages is used to identify the starting point for the redo phase. A list of in-progress transactions is used during the undo phase to roll back incomplete transactions.

2. The *redo* phase repeats the history up to the point of a crash and restores the database to the previous state. This phase is done for incomplete transactions as well as ones that were committed but whose contents weren't flushed to persistent storage.
3. The *undo* phase rolls back all incomplete transactions and restores the database to the last consistent state. All operations are rolled back in reverse chronological order. In case the database crashes again during recovery, operations that undo transactions are logged as well to avoid repeating them.

ARIES uses LSNs for identifying log records, tracks pages modified by running transactions in the dirty page table, and uses physical redo, logical undo, and fuzzy checkpointing. Even though the paper describing this system was released in 1992, most concepts, approaches, and paradigms are still relevant in transaction processing and recovery today.

Concurrency Control

When discussing database management system architecture in "DBMS Architecture" on page 8, we mentioned that the transaction manager and lock manager work together to handle *concurrency control*. Concurrency control is a set of techniques for handling interactions between concurrently executing transactions. These techniques can be roughly grouped into the following categories:

Optimistic concurrency control (OCC)
: Allows transactions to execute concurrent read and write operations, and determines whether or not the result of the combined execution is serializable. In other words, transactions do not block each other, maintain histories of their operations, and check these histories for possible conflicts before commit. If execution results in a conflict, one of the conflicting transactions is aborted.

Multiversion concurrency control (MVCC)
: Guarantees a consistent view of the database at some point in the past identified by the timestamp by allowing multiple timestamped versions of the record to be present. MVCC can be implemented using validation techniques, allowing only one of the updating or committing transactions to win, as well as with lockless techniques such as timestamp ordering, or lock-based ones, such as two-phase locking.

Pessimistic (also known as conservative) concurrency control (PCC)
: There are both lock-based and nonlocking conservative methods, which differ in how they manage and grant access to shared resources. Lock-based approaches require transactions to maintain locks on database records to prevent other transactions from modifying locked records and assessing records that are being modified until the transaction releases its locks. Nonlocking approaches maintain

read and write operation lists and restrict execution, depending on the schedule of unfinished transactions. Pessimistic schedules can result in a deadlock when multiple transactions wait for each other to release a lock in order to proceed.

In this chapter, we concentrate on node-local concurrency control techniques. In Chapter 13, you can find information about distributed transactions and other approaches, such as deterministic concurrency control (see "Distributed Transactions with Calvin" on page 266).

Before we can further discuss concurrency control, we need to define a set of problems we're trying to solve and discuss how transaction operations overlap and what consequences this overlapping has.

Serializability

Transactions consist of read and write operations executed against the database state, and business logic (transformations, applied to the read contents). A *schedule* is a list of operations required to execute a set of transactions from the database-system perspective (i.e., only ones that interact with the database state, such as read, write, commit, or abort operations), since all other operations are assumed to be side-effect free (in other words, have no impact on the database state) [MOLINA08].

A schedule is *complete* if contains all operations from every transaction executed in it. *Correct* schedules are logical equivalents to the original lists of operations, but their parts can be executed in parallel or get reordered for optimization purposes, as long as this does not violate ACID properties and the correctness of the results of individual transactions [WEIKUM01].

A schedule is said to be *serial* when transactions in it are executed completely independently and without any interleaving: every preceding transaction is fully executed before the next one starts. Serial execution is easy to reason about, as contrasted with all possible interleavings between several multistep transactions. However, always executing transactions one after another would significantly limit the system throughput and hurt performance.

We need to find a way to execute transaction operations concurrently, while maintaining the correctness and simplicity of a serial schedule. We can achieve this with *serializable* schedules. A schedule is serializable if it is equivalent to *some* complete serial schedule over the same set of transactions. In other words, it produces the same result as if we executed a set of transactions one after another in *some* order. Figure 5-4 shows three concurrent transactions, and possible execution histories (`3! = 6` possibilities, in every possible order).

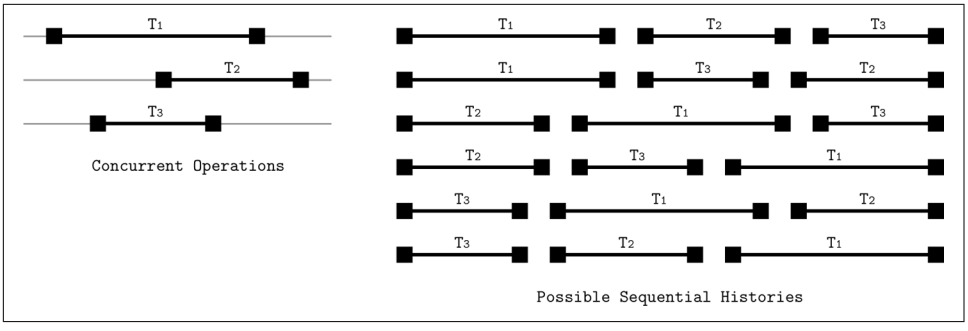

Figure 5-4. Concurrent transactions and their possible sequential execution histories

Transaction Isolation

Transactional database systems allow different isolation levels. An *isolation level* specifies how and when parts of the transaction can and should become visible to other transactions. In other words, isolation levels describe the degree to which transactions are isolated from other concurrently executing transactions, and what kinds of anomalies can be encountered during execution.

Achieving isolation comes at a cost: to prevent incomplete or temporary writes from propagating over transaction boundaries, we need additional coordination and synchronization, which negatively impacts the performance.

Read and Write Anomalies

The SQL standard [MELTON06] refers to and describes *read anomalies* that can occur during execution of concurrent transactions: dirty, nonrepeatable, and phantom reads.

A *dirty read* is a situation in which a transaction can read uncommitted changes from other transactions. For example, transaction T_1 updates a user record with a new value for the address field, and transaction T_2 reads the updated address before T_1 commits. Transaction T_1 aborts and rolls back its execution results. However, T_2 has already been able to read this value, so it has accessed the value that has never been committed.

A *nonrepeatable read* (sometimes called a fuzzy read) is a situation in which a transaction queries the *same row* twice and gets different results. For example, this can happen even if transaction T_1 reads a row, then transaction T_2 modifies it *and commits* this change. If T_1 requests the same row again before finishing its execution, the result will differ from the previous run.

If we use range reads during the transaction (i.e., read not a single data record, but a range of records), we might see *phantom records*. A *phantom read* is when a

transaction queries the same *set of rows* twice and receives different results. It is similar to a nonrepeatable read, but holds for range queries.

There are also *write anomalies* with similar semantics: lost update, dirty write, and write skew.

A *lost update* occurs when transactions T_1 and T_2 both attempt to update the value of `V`. T_1 and T_2 read the value of `V`. T_1 updates `V` and commts, and T_2 updates `V` after that and commits as well. Since the transactions are not aware about each other's existence, if both of them are allowed to commit, the results of T_1 will be overwritten by the results of T_2, and the update from T_1 will be lost.

A *dirty write* is a situation in which one of the transactions takes an uncommitted value (i.e., dirty read), modifies it, and saves it. In other words, when transaction results are based on the values that have never been committed.

A *write skew* occurs when each individual transaction respects the required invariants, but their combination does not satisfy these invariants. For example, transactions T_1 and T_2 modify values of two accounts A_1 and A_2. A_1 starts with `100$` and A_2 starts with `150$`. The account value is allowed to be negative, as long as the sum of the two accounts is nonnegative: A_1 + A_2 >= `0`. T_1 and T_2 each attempt to withdraw `200$` from A_1 and A_2, respectively. Since at the time these transactions start A_1 + A_2 = `250$`, `250$` is available in total. Both transactions assume they're preserving the invariant and are allowed to commit. After the commit, A_1 has `-100$` and A_2 has `-50$`, which clearly violates the requirement to keep a sum of the accounts positive [FEKETE04].

Isolation Levels

The lowest (in other words, weakest) isolation level is *read uncommitted*. Under this isolation level, the transactional system allows one transaction to observe uncommitted changes of other concurrent transactions. In other words, dirty reads are allowed.

We can avoid some of the anomalies. For example, we can make sure that any read performed by the specific transaction can only read *already committed* changes. However, it is not guaranteed that if the transaction attempts to read the same data record once again at a later stage, it will see the same value. If there was a committed modification between two reads, two queries in the same transaction would yield different results. In other words, dirty reads are not permitted, but phantom and nonrepeatable reads are. This isolation level is called *read committed*. If we further disallow nonrepeatable reads, we get a *repeatable read* isolation level.

The strongest isolation level is serializability. As we already discussed in "Serializability" on page 94, it guarantees that transaction outcomes will appear in *some* order as if transactions were executed *serially* (i.e., without overlapping in time). Disallowing concurrent execution would have a substantial negative impact on the database per-

formance. Transactions can get reordered, as long as their internal invariants hold and can be executed concurrently, but their outcomes have to appear in *some* serial order.

Figure 5-5 shows isolation levels and the anomalies they allow.

	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

Transactions that do not have dependencies can be executed in any order since their results are fully independent. Unlike linearizability (which we discuss in the context of distributed systems; see "Linearizability" on page 223), serializability is a property of *multiple* operations executed in *arbitrary* order. It does not imply or attempt to impose any particular order on executing transactions. *Isolation* in ACID terms means serializability [BAILIS14a]. Unfortunately, implementing serializability requires coordination. In other words, transactions executing concurrently have to coordinate to preserve invariants and impose a serial order on conflicting executions [BAILIS14b].

Some databases use *snapshot isolation*. Under snapshot isolation, a transaction can observe the state changes performed by all transactions that were committed by the time it has started. Each transaction takes a snapshot of data and executes queries against it. This snapshot cannot change during transaction execution. The transaction commits only if the values it has modified did *not* change while it was executing. Otherwise, it is aborted and rolled back.

If two transactions attempt to modify the same value, only one of them is allowed to commit. This precludes a *lost update* anomaly. For example, transactions T_1 and T_2 both attempt to modify `V`. They read the current value of `V` from the snapshot that contains changes from all transactions that were committed before they started. Whichever transaction attempts to commit first, will commit, and the other one will have to abort. The failed transactions will retry instead of overwriting the value.

A *write skew* anomaly is possible under snapshot isolation, since if two transactions read from local state, modify independent records, and preserve local invariants, they both are allowed to commit [FEKETE04]. We discuss snapshot isolation in more detail in the context of distributed transactions in "Distributed Transactions with Percolator" on page 272.

Optimistic Concurrency Control

Optimistic concurrency control assumes that transaction conflicts occur rarely and, instead of using locks and blocking transaction execution, we can validate transactions to prevent read/write conflicts with concurrently executing transactions and ensure serializability before committing their results. Generally, transaction execution is split into three phases [WEIKUM01]:

Read phase
: The transaction executes its steps in its own private context, without making any of the changes visible to other transactions. After this step, all transaction dependencies (*read set*) are known, as well as the side effects the transaction produces (*write set*).

Validation phase
: Read and write sets of concurrent transactions are checked for the presence of possible conflicts between their operations that might violate serializability. If some of the data the transaction was reading is now out-of-date, or it would overwrite some of the values written by transactions that committed during its read phase, its private context is cleared and the read phase is restarted. In other words, the validation phase determines whether or not committing the transaction preserves ACID properties.

Write phase
: If the validation phase hasn't determined any conflicts, the transaction can commit its write set from the private context to the database state.

Validation can be done by checking for conflicts with the transactions that have already been committed (*backward-oriented*), or with the transactions that are currently in the validation phase (*forward-oriented*). Validation and write phases of different transactions should be done atomically. No transaction is allowed to commit while some other transaction is being validated. Since validation and write phases are generally shorter than the read phase, this is an acceptable compromise.

Backward-oriented concurrency control ensures that for any pair of transactions T_1 and T_2, the following properties hold:

- T_1 was committed before the read phase of T_2 began, so T_2 is allowed to commit.
- T_1 was committed before the T_2 write phase, and the write set of T_1 doesn't intersect with the T_2 read set. In other words, T_1 hasn't written any values T_2 should have seen.
- The read phase of T_1 has completed before the read phase of T_2, and the write set of T_2 doesn't intersect with the read or write sets of T_1. In other words,

transactions have operated on independent sets of data records, so both are allowed to commit.

This approach is efficient if validation usually succeeds and transactions don't have to be retried, since retries have a significant negative impact on performance. Of course, optimistic concurrency still has a *critical section*, which transactions can enter one at a time. Another approach that allows nonexclusive ownership for some operations is to use readers-writer locks (to allow shared access for readers) and upgradeable locks (to allow conversion of shared locks to exclusive when needed).

Multiversion Concurrency Control

Multiversion concurrency control is a way to achieve transactional consistency in database management systems by allowing multiple record versions and using monotonically incremented transaction IDs or timestamps. This allows reads and writes to proceed with a minimal coordination on the storage level, since reads can continue accessing older values until the new ones are committed.

MVCC distinguishes between *committed* and *uncommitted* versions, which correspond to value versions of committed and uncommitted transactions. The last committed version of the value is assumed to be *current*. Generally, the goal of the transaction manager in this case is to have at most one uncommitted value at a time.

Depending on the isolation level implemented by the database system, read operations may or may not be allowed to access uncommitted values [WEIKUM01]. Multiversion concurrency can be implemented using locking, scheduling, and conflict resolution techniques (such as two-phase locking), or timestamp ordering. One of the major use cases for MVCC for implementing snapshot isolation [HELLERSTEIN07].

Pessimistic Concurrency Control

Pessimistic concurrency control schemes are more conservative than optimistic ones. These schemes determine transaction conflicts while they're running and block or abort their execution.

One of the simplest pessimistic (lock-free) concurrency control schemes is *timestamp ordering*, where each transaction has a timestamp. Whether or not transaction operations are allowed to be executed is determined by whether or not any transaction with an *earlier* timestamp has already been committed. To implement that, the transaction manager has to maintain `max_read_timestamp` and `max_write_timestamp` per value, describing read and write operations executed by concurrent transactions.

Read operations that attempt to read a value with a timestamp lower than `max_write_timestamp` cause the transaction they belong to be aborted, since there's already a newer value, and allowing this operation would violate the transaction order.

Similarly, *write* operations with a timestamp lower than `max_read_timestamp` would conflict with a more recent read. However, *write* operations with a timestamp lower than `max_write_timestamp` are allowed, since we can safely ignore the outdated written values. This conjecture is commonly called the *Thomas Write Rule* [THOMAS79]. As soon as read or write operations are performed, the corresponding maximum timestamp values are updated. Aborted transactions restart with a *new* timestamp, since otherwise they're guaranteed to be aborted again [RAMAKRISHNAN03].

Lock-Based Concurrency Control

Lock-based concurrency control schemes are a form of pessimistic concurrency control that uses explicit locks on the database objects rather than resolving schedules, like protocols such as timestamp ordering do. Some of the downsides of using locks are contention and scalability issues [REN16].

One of the most widespread lock-based techniques is *two-phase locking* (2PL), which separates lock management into two phases:

- The *growing phase* (also called the *expanding phase*), during which all locks required by the transaction are acquired and no locks are released.
- The *shrinking phase*, during which all locks acquired during the growing phase are released.

A rule that follows from these two definitions is that a transaction cannot acquire any locks as soon as it has released at least one of them. It's important to note that 2PL does not preclude transactions from executing steps during either one of these phases; however, some 2PL variants (such as conservative 2PL) do impose these limitations.

Despite similar names, two-phase locking is a concept that is entirely different from two-phase commit (see "Two-Phase Commit" on page 259). Two-phase commit is a protocol used for distributed multipartition transactions, while two-phase locking is a concurrency control mechanism often used to implement serializability.

Deadlocks

In locking protocols, transactions attempt to acquire locks on the database objects and, in case a lock cannot be granted immediately, a transaction has to wait until the lock is released. A situation may occur when two transactions, while attempting to acquire locks they require in order to proceed with execution, end up waiting for each other to release the other locks they hold. This situation is called a *deadlock*.

Figure 5-6 shows an example of a deadlock: T_1 holds lock L_1 and waits for lock L_2 to be released, while T_2 holds lock L_2 and waits for L_1 to be released.

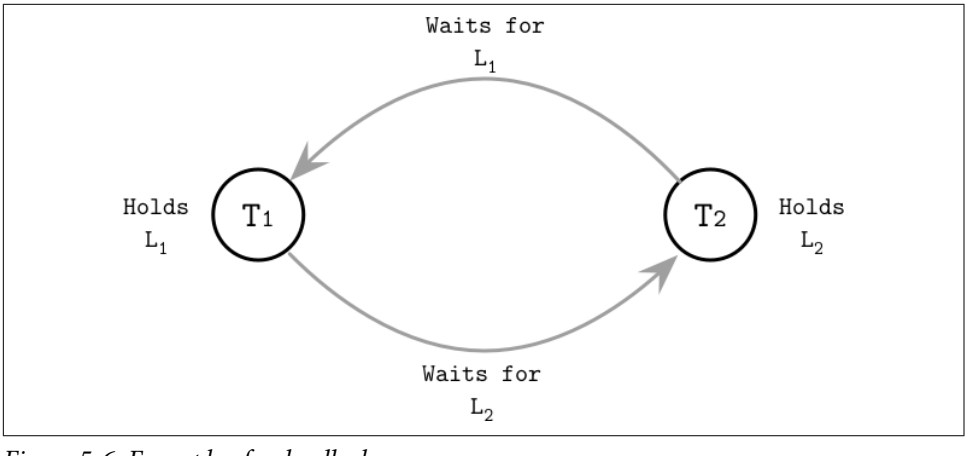

Figure 5-6. Example of a deadlock

The simplest way to handle deadlocks is to introduce timeouts and abort long-running transactions under the assumption that they might be in a deadlock. Another strategy, conservative 2PL, requires transactions to acquire all the locks before they can execute any of their operations and abort if they cannot. However, these approaches significantly limit system concurrency, and database systems mostly use a transaction manager to *detect* or *avoid* (in other words, prevent) deadlocks.

Detecting deadlocks is generally done using a *waits-for graph*, which tracks relationships between the in-flight transactions and establishes waits-for relationships between them.

Cycles in this graph indicate the presence of a deadlock: transaction T_1 is waiting for T_2 which, in turn, waits for T_1. Deadlock detection can be done *periodically* (once per time interval) or *continuously* (every time the waits-for graph is updated) [WEIKUM01]. One of the transactions (usually, the one that attempted to acquire the lock more recently) is aborted.

To *avoid* deadlocks and restrict lock acquisition to cases that will not result in a deadlock, the transaction manager can use transaction timestamps to determine their *priority*. A lower timestamp usually implies higher priority and vice versa.

If transaction T_1 attempts to acquire a lock currently held by T_2, and T_1 has higher priority (it started before T_2), we can use one of the following restrictions to avoid deadlocks [RAMAKRISHNAN03]:

Wait-die
: T_1 is allowed to block and *wait* for the lock. Otherwise, T_1 is aborted and restarted. In other words, a transaction can be blocked only by a transaction with a higher timestamp.

Wound-wait
: T_2 is aborted and restarted (T_1 *wounds* T_2). Otherwise (if T_2 has started before T_1), T_1 is allowed to wait. In other words, a transaction can be blocked only by a transaction with a lower timestamp.

Transaction processing requires a scheduler to handle deadlocks. At the same time, latches (see "Latches" on page 103) rely on the programmer to ensure that deadlocks cannot happen and do not rely on deadlock avoidance mechanisms.

Locks

If two transactions are submitted concurrently, modifying overlapping segments of data, neither one of them should observe partial results of the other one, hence maintaining logical consistency. Similarly, two threads from the same transaction have to observe the same database contents, and have access to each other's data.

In transaction processing, there's a distinction between the mechanisms that guard the logical and physical data integrity. The two concepts responsible logical and physical integrity are, correspondingly, *locks* and *latches*. The naming is somewhat unfortunate since what's called a latch here is usually referred to as a lock in systems programming, but we'll clarify the distinction and implications in this section.

Locks are used to isolate and schedule overlapping transactions and manage database contents but not the internal storage structure, and are acquired on the key. Locks can guard either a specific key (whether it's existing or nonexisting) or a range of keys. Locks are generally stored and managed outside of the tree implementation and represent a higher-level concept, managed by the database lock manager.

Locks are more heavyweight than latches and are held for the duration of the transaction.

Latches

On the other hand, latches guard the *physical* representation: leaf page contents are modified during insert, update, and delete operations. Nonleaf page contents and a tree structure are modified during operations resulting in splits and merges that propagate from leaf under- and overflows. Latches guard the physical tree representation (page contents and the tree structure) during these operations and are obtained on the page level. Any page has to be latched to allow safe concurrent access to it. Lockless concurrency control techniques still have to use latches.

Since a single modification on the leaf level might propagate to higher levels of the B-Tree, latches might have to be obtained on multiple levels. Executing queries should not be able to observe pages in an inconsistent state, such as incomplete writes or partial node splits, during which data might be present in both the source and target node, or not yet propagated to the parent.

The same rules apply to parent or sibling pointer updates. A general rule is to hold a latch for the smallest possible duration—namely, when the page is read or updated—to increase concurrency.

Interferences between concurrent operations can be roughly grouped into three categories:

- *Concurrent reads*, when several threads access the same page without modifying it.
- *Concurrent updates*, when several threads attempt to make modifications to the same page.
- *Reading while writing*, when one of the threads is trying to modify the page contents, and the other one is trying to access the same page for a read.

These scenarios also apply to accesses that overlap with database maintenance (such as background processes, as described in "Vacuum and Maintenance" on page 74).

Readers-writer lock

The simplest latch implementation would grant exclusive read/write access to the requesting thread. However, most of the time, we do not need to isolate *all* the processes from each other. For example, reads can access pages concurrently without causing any trouble, so we only need to make sure that multiple concurrent *writers* do not overlap, and *readers* do not overlap with *writers*. To achieve this level of granularity, we can use a *readers-writer lock* or RW lock.

An RW lock allows multiple readers to access the object concurrently, and only writers (which we usually have fewer of) have to obtain exclusive access to the object. Figure 5-7 shows the compatibility table for readers-writer locks: only readers can

share lock ownership, while all other combinations of readers and writers should obtain exclusive ownership.

	Reader	Writer
Reader	Shared	Exclusive
Writer	Exclusive	Exclusive

Figure 5-7. Readers-writer lock compatibility table

In Figure 5-8 (a), we have multiple readers accessing the object, while the writer is waiting for its turn, since it can't modify the page while readers access it. In Figure 5-8 (b), `writer 1` holds an exclusive lock on the object, while another writer and three readers have to wait.

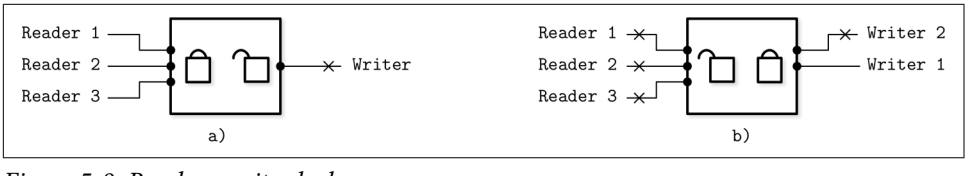

Figure 5-8. Readers-writer locks

Since two overlapping reads attempting to access the same page do not require synchronization other than preventing the page from being fetched from disk by the page cache twice, reads can be safely executed concurrently in shared mode. As soon as writes come into play, we need to isolate them from both concurrent reads and other writes.

Busy-Wait and Queueing Techniques

To manage shared access to pages, we can either use blocking algorithms, which deschedule threads and wake them up as soon as they can proceed, or use busy-wait algorithms. Busy-wait algorithms allow threads to wait for insignificant amounts of time instead of handing control back to the scheduler.

Queuing is usually implemented using compare-and-swap instructions, used to perform operations guaranteeing lock acquisition and queue update atomicity. If the queue is empty, the thread obtains access immediately. Otherwise, the thread appends itself to the waiting queue and spins on the variable that can be updated only by the thread preceding it in the queue. This helps to reduce the amount of CPU traffic for lock acquisition and release [MELLORCRUMMEY91].

Latch crabbing

The most straightforward approach for latch acquisition is to grab all the latches on the way from the root to the target leaf. This creates a concurrency bottleneck and can be avoided in most cases. The time during which a latch is held should be minimized. One of the optimizations that can be used to achieve that is called *latch crabbing* (or latch coupling) [RAMAKRISHNAN03].

Latch crabbing is a rather simple method that allows holding latches for less time and releasing them as soon as it's clear that the executing operation does not require them anymore. On the read path, as soon as the child node is located and its latch is acquired, the parent node's latch can be released.

During insert, the parent latch can be released if the operation is guaranteed not to result in structural changes that can propagate to it. In other words, the parent latch can be released if the child node is not full.

Similarly, during deletes, if the child node holds enough elements and the operation will not cause sibling nodes to merge, the latch on the parent node is released.

Figure 5-9 shows a root-to-leaf pass during insert:

- a) The write latch is acquired on the root level.
- b) The next-level node is located, and its write latch is acquired. The node is checked for potential structural changes. Since the node is not full, the parent latch can be released.
- c) The operation descends to the next level. The write latch is acquired, the target leaf node is checked for potential structural changes, and the parent latch is released.

This approach is optimistic: most insert and delete operations do not cause structural changes that propagate multiple levels up. In fact, the probability of structural changes decreases at higher levels. Most of the operations only require the latch on the target node, and the number of cases when the parent latch has to be retained is relatively small.

If the child page is still not loaded in the page cache, we can either latch a future loading page, or release a parent latch and restart the root-to-leaf pass after the page is loaded to reduce contention. Restarting root-to-leaf traversal sounds rather expensive, but in reality, we have to perform it rather infrequently, and can employ mechanisms to detect whether or not there were any structural changes at higher levels since the time of traversal [GRAEFE10].

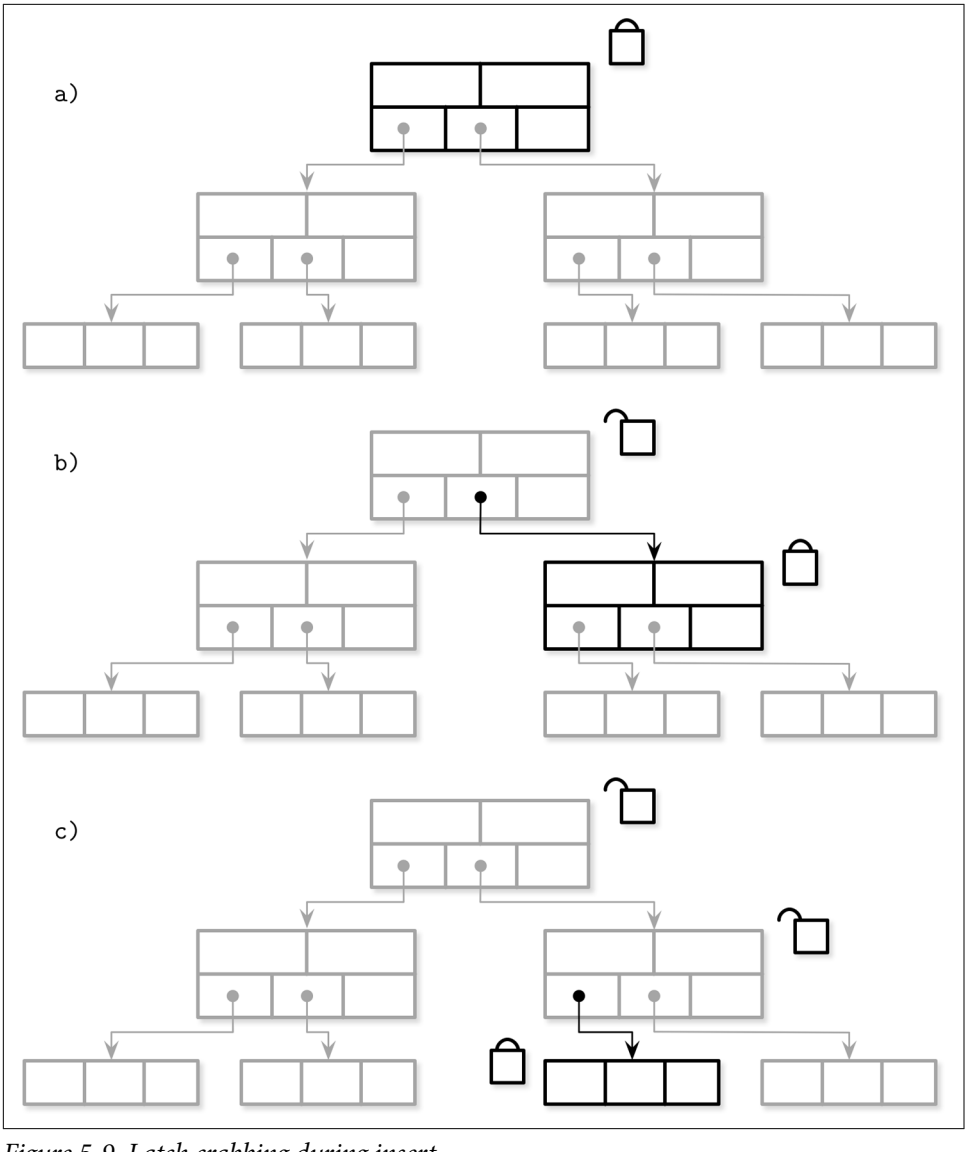

Figure 5-9. Latch crabbing during insert

Latch Upgrading and Pointer Chasing

Instead of acquiring latches during traversal in an exclusive mode right away, *latch upgrading* can be employed instead. This approach involves acquisition of shared locks along the search path and *upgrading* them to exclusive locks when necessary.

Write operations first acquire exclusive locks only at the leaf level. If the leaf has to be split or merged, the algorithm walks up the tree and attempts to *upgrade* a shared lock that the parent holds, acquiring exclusive ownership of latches for the affected portion of the tree (i.e., nodes that will also be split or merged as a result of that operation). Since multiple threads might attempt to acquire exclusive locks on one of the higher levels, one of them has to wait or restart.

You might have noticed that the mechanisms described so far all start by acquiring a latch on the root node. Every request has to go through the root node, and it quickly becomes a bottleneck. At the same time, the root is always the last to be split, since all of its children have to fill up first. This means that the root node can *always* be latched optimistically, and the price of a retry (*pointer chasing*) is seldom paid.

Blink-Trees

B^{link}-Trees build on top of B*-Trees (see "Rebalancing" on page 70) and add *high keys* (see "Node High Keys" on page 64) and *sibling link* pointers [LEHMAN81]. A high key indicates the highest possible subtree key. Every node but root in a B^{link}-Tree has two pointers: a child pointer descending from the parent and a sibling link from the left node residing on the same level.

B^{link}-Trees allow a state called *half-split*, where the node is already referenced by the sibling pointer, but not by the child pointer from its parent. Half-split is identified by checking the node high key. If the search key exceeds the high key of the node (which violates the high key invariant), the lookup algorithm concludes that the structure has been changed concurrently and follows the sibling link to proceed with the search.

The pointer has to be quickly added to the parent guarantee the best performance, but the search process doesn't have to be aborted and restarted, since all elements in the tree are accessible. The advantage here is that we do not have to hold the parent lock when descending to the child level, even if the child is going to be split: we can make a new node visible through its sibling link and update the parent pointer lazily without sacrificing correctness [GRAEFE10].

While this is slightly less efficient than descending directly from the parent and requires accessing an extra page, this results in correct root-to-leaf descent while simplifying concurrent access. Since splits are a relatively infrequent operation and B-Trees rarely shrink, this case is exceptional, and its cost is insignificant. This approach has quite a few benefits: it reduces contention, prevents holding a parent lock during

splits, and reduces the number of locks held during tree structure modification to a constant number. More importantly, it allows reads concurrent to structural tree changes, and prevents deadlocks otherwise resulting from concurrent modifications ascending to the parent nodes.

Summary

In this chapter, we discussed the storage engine components responsible for transaction processing and recovery. When implementing transaction processing, we are presented with two problems:

- To improve efficiency, we need to allow concurrent transaction execution.
- To preserve correctness, we have to ensure that concurrently executing transactions preserve ACID properties.

Concurrent transaction execution can cause different kinds of read and write anomalies. Presence or absence of these anomalies is described and limited by implementing different isolation levels. Concurrency control approaches determine how transactions are scheduled and executed.

The page cache is responsible for reducing the number of disk accesses: it caches pages in memory and allows read and write access to them. When the cache reaches its capacity, pages are evicted and flushed back on disk. To make sure that unflushed changes are not lost in case of node crashes and to support transaction rollback, we use write-ahead logs. The page cache and write-ahead logs are coordinated using force and steal policies, ensuring that every transaction can be executed efficiently and rolled back without sacrificing durability.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Transaction processing and recovery, generally

Weikum, Gerhard, and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco: Morgan Kaufmann Publishers Inc.

Bernstein, Philip A. and Eric Newcomer. 2009. *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann.

Graefe, Goetz, Guy, Wey & Sauer, Caetano. 2016. "Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, (2nd Ed.)" in *Synthesis Lectures on Data Management* 8, 1-113. 10.2200/S00710ED2V01Y201603DTM044.

Mohan, C., Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." *Transactions on Database Systems* 17, no. 1 (March): 94-162. *https://doi.org/10.1145/128765.128770*.

Concurrency control in B-Trees

Wang, Paul. 1991. "An In-Depth Analysis of Concurrent B-Tree Algorithms." MIT Technical Report. *https://apps.dtic.mil/dtic/tr/fulltext/u2/a232287.pdf*.

Goetz Graefe. 2010. A survey of B-tree locking techniques. ACM Trans. Database Syst. 35, 3, Article 16 (July 2010), 26 pages.

Parallel and concurrent data structures

McKenney, Paul E. 2012. "Is Parallel Programming Hard, And, If So,What Can You Do About It?" *https://arxiv.org/abs/1701.00854*.

Herlihy, Maurice and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint (1st Ed.)*. San Francisco: Morgan Kaufmann.

Chronological developments in the field of transaction processing

Diaconu, Cristian, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, 1243-1254. New York: Association for Computing Machinery. *https://doi.org/10.1145/2463676.2463710*.

Kimura, Hideaki. 2015. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM." In *Proceedings of the 2015 ACM SIGMOD International Conference on*

Management of Data (SIGMOD '15), 691-706. https://doi.org/10.1145/2723372.2746480.

Yu, Xiangyao, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. "TicToc: Time Traveling Optimistic Concurrency Control." In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, 1629-1642. https://doi.org/10.1145/2882903.2882935.

Kim, Kangnyeon, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. "ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads." In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, 1675-1687. https://doi.org/10.1145/2882903.2882905.

Lim, Hyeontaek, Michael Kaminsky, and David G. Andersen. 2017. "Cicada: Dependably Fast Multi-Core In-Memory Transactions." In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, 21-35. https://doi.org/10.1145/3035918.3064015.

CHAPTER 6
B-Tree Variants

B-Tree variants have a few things in common: tree structure, balancing through splits and merges, and lookup and delete algorithms. Other details, related to concurrency, on-disk page representation, links between sibling nodes, and maintenance processes, may vary between implementations.

In this chapter, we'll discuss several techniques that can be used to implement efficient B-Trees and structures that employ them:

- *Copy-on-write B-Trees* are structured like B-Trees, but their nodes are immutable and are not updated in place. Instead, pages are copied, updated, and written to new locations.
- *Lazy B-Trees* reduce the number of I/O requests from subsequent same-node writes by *buffering* updates to nodes. In the next chapter, we also cover two-component LSM trees (see "Two-component LSM Tree" on page 132), which take buffering a step further to implement fully immutable B-Trees.
- *FD-Trees* take a different approach to buffering, somewhat similar to LSM Trees (see "LSM Trees" on page 130). FD-Trees buffer updates in a small B-Tree. As soon as this tree fills up, its contents are written into an immutable run. Updates propagate between *levels* of immutable runs in a cascading manner, from higher levels to lower ones.
- *Bw-Trees* separate B-Tree nodes into several smaller parts that are written in an append-only manner. This reduces costs of small writes by batching updates to the different nodes together.
- *Cache-oblivious B-Trees* allow treating on-disk data structures in a way that is very similar to how we build in-memory ones.

Copy-on-Write

Some databases, rather than building complex latching mechanisms, use the *copy-on-write* technique to guarantee data integrity in the presence of concurrent operations. In this case, whenever the page is about to be modified, its contents are copied, the copied page is modified instead of the original one, and a parallel tree hierarchy is created.

Old tree versions remain accessible for readers that run concurrently to the writer, while writers accessing modified pages have to wait until preceding write operations are complete. After the new page hierarchy is created, the pointer to the topmost page is atomically updated. In Figure 6-1, you can see a new tree being created parallel to the old one, reusing the untouched pages.

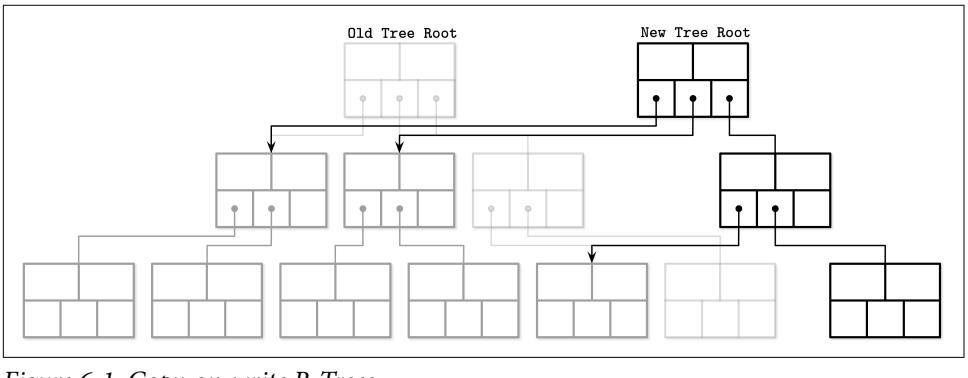

Figure 6-1. Copy-on-write B-Trees

An obvious downside of this approach is that it requires more space (even though old versions are retained only for brief time periods, since pages can be reclaimed immediately after concurrent operations using the old pages complete) and processor time, as entire page contents have to be copied. Since B-Trees are generally shallow, the simplicity and advantages of this approach often still outweigh the downsides.

The biggest advantage of this approach is that readers require no synchronization, because written pages are immutable and can be accessed without additional latching. Since writes are performed against copied pages, readers do not block writers. No operation can observe a page in an incomplete state, and a system crash cannot leave pages in a corrupted state, since the topmost pointer is switched only when all page modifications are done.

Implementing Copy-on-Write: LMDB

One of the storage engines using copy-on-write is the Lightning Memory-Mapped Database (LMDB), which is a key-value store used by the OpenLDAP project. Due to its design and architecture, LMDB doesn't require a page cache, a write-ahead log, checkpointing, or compaction.[1]

LMDB is implemented as a single-level data store, which means that read and write operations are satisfied directly through the memory map, without additional application-level caching in between. This also means that pages require no additional materialization and reads can be served directly from the memory map without copying data to the intermediate buffer. During the update, every branch node on the path from the root to the target leaf is copied and potentially modified: nodes for which updates propagate are changed, and the rest of the nodes remain intact.

LMDB holds only two versions of the root node: the latest version, and the one where new changes are going to be committed. This is sufficient since all writes have to go through the root node. After the new root is created, the old one becomes unavailable for new reads and writes. As soon as the reads referencing old tree sections complete, their pages are reclaimed and can be reused. Because of LMDB's append-only design, it does not use sibling pointers and has to ascend back to the parent node during sequential scans.

With this design, leaving stale data in copied nodes is impractical: there is already a copy that can be used for MVCC and satisfy ongoing read transactions. The database structure is inherently multiversioned, and readers can run without any locks as they do not interfere with writers in any way.

Abstracting Node Updates

To update the page on disk, one way or the other, we have to first update its in-memory representation. However, there are a few ways to represent a node in memory: we can access the cached version of the node directly, do it through the wrapper object, or create its in-memory representation native to the implementation language.

In languages with an unmanaged memory model, raw binary data stored in B-Tree nodes can be reinterpreted and native pointers can be used to manipulate it. In this case, the node is defined in terms of structures, which use raw binary data behind the pointer and runtime casts. Most often, they point to the memory area managed by the page cache or use memory mapping.

1 To learn more about LMDB, see the code comments and the presentation.

Alternatively, B-Tree nodes can be materialized into objects or structures native to the language. These structures can be used for inserts, updates, and deletes. During flush, changes are applied to pages in memory and, subsequently, on disk. This approach has the advantage of simplifying concurrent accesses since changes to underlying raw pages are managed separately from accesses to intermediate objects, but results in a higher memory overhead, since we have to store two versions (raw binary and language-native) of the same page in memory.

The third approach is to provide access to the buffer backing the node through the wrapper object that materializes changes in the B-Tree as soon as they're performed. This approach is most often used in languages with a managed memory model. Wrapper objects apply the changes to the backing buffers.

Managing on-disk pages, their cached versions, and their in-memory representations separately allows them to have different life cycles. For example, we can buffer insert, update, and delete operations, and reconcile changes made in memory with the original on-disk versions during reads.

Lazy B-Trees

Some algorithms (in the scope of this book, we call them lazy B-Trees[2]) reduce costs of updating the B-Tree and use more lightweight, concurrency- and update-friendly in-memory structures to buffer updates and propagate them with a delay.

WiredTiger

Let's take a look at how we can use buffering to implement a lazy B-Tree. For that, we can materialize B-Tree nodes in memory as soon as they are paged in and use this structure to store updates until we're ready to flush them.

A similar approach is used by WiredTiger, a now-default MongoDB storage engine. Its row store B-Tree implementation uses different formats for in-memory and on-disk pages. Before in-memory pages are persisted, they have to go through the reconciliation process.

2 This is not a commonly recognized name, but since the B-Tree variants we're discussing here share one property—buffering B-Tree updates in intermediate structures instead of applying them to the tree directly—we'll use the term *lazy*, which rather precisely defines this property.

In Figure 6-2, you can see a schematic representation of WiredTiger pages and their composition in a B-Tree. A *clean* page consists of just an index, initially constructed from the on-disk page image. Updates are first saved into the *update buffer*.

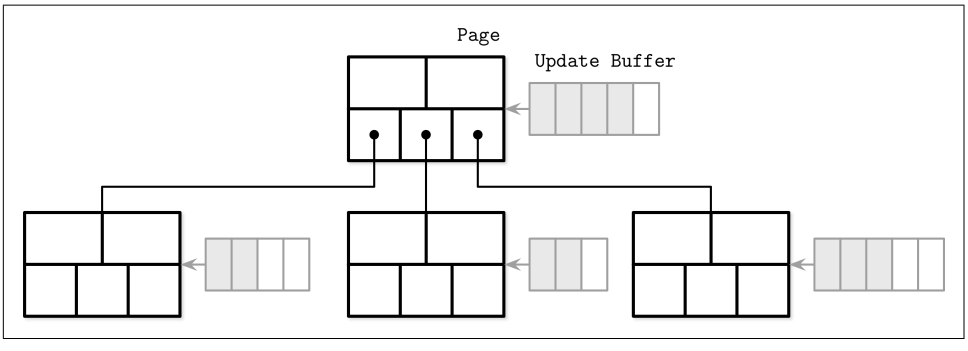

Figure 6-2. WiredTiger: high-level overview

Update buffers are accessed during reads: their contents are merged with the original on-disk page contents to return the most recent data. When the page is flushed, update buffer contents are reconciled with page contents and persisted on disk, overwriting the original page. If the size of the reconciled page is greater than the maximum, it is split into multiple pages. Update buffers are implemented using skiplists, which have a complexity similar to search trees [PAPADAKIS93] but have a better concurrency profile [PUGH90a].

Figure 6-3 shows that both clean and dirty pages in WiredTiger have in-memory versions, and reference a base image on disk. Dirty pages have an update buffer in addition to that.

The main advantage here is that the page updates and structural modifications (splits and merges) are performed by the background thread, and read/write processes do not have to wait for them to complete.

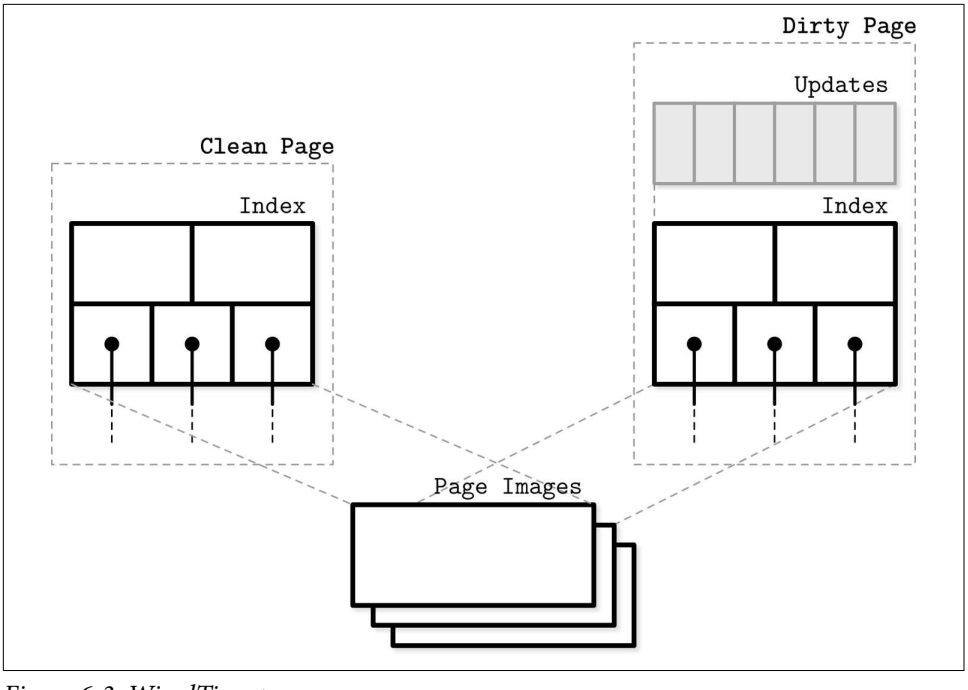

Figure 6-3. WiredTiger pages

Lazy-Adaptive Tree

Rather than buffering updates to individual nodes, we can group nodes into subtrees, and attach an update buffer for batching operations *to each subtree*. Update buffers in this case will track all operations performed against the subtree top node and its descendants. This algorithm is called *Lazy-Adaptive Tree* (LA-Tree) [AGRAWAL09].

When inserting a data record, a new entry is first added to the root node update buffer. When this buffer becomes full, it is emptied by copying and propagating the changes to the buffers in the lower tree levels. This operation can continue recursively if the lower levels fill up as well, until it finally reaches the leaf nodes.

In Figure 6-4, you see an LA-Tree with cascaded buffers for nodes grouped in corresponding subtrees. Gray boxes represent changes that propagated from the root buffer.

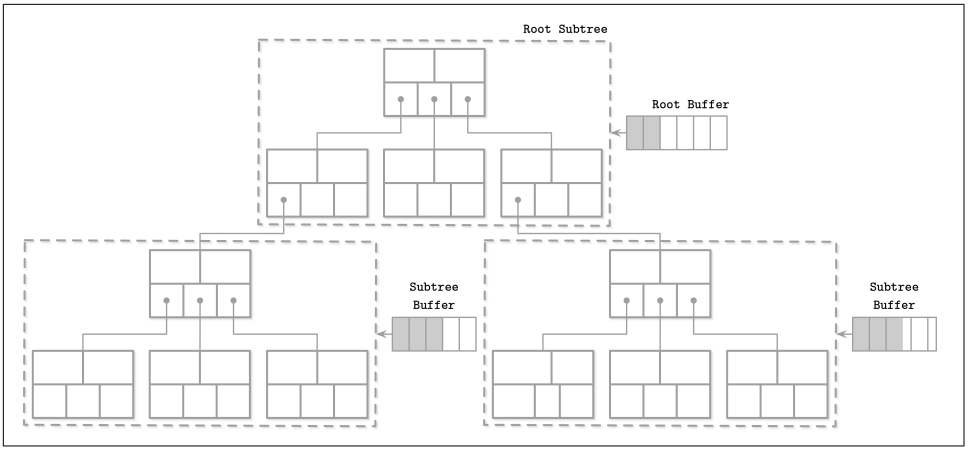

Figure 6-4. LA-Tree

Buffers have hierarchical dependencies and are *cascaded*: all the updates propagate from higher-level buffers to the lower-level ones. When the updates reach the leaf level, batched insert, update, and delete operations are performed there, applying all changes to the tree contents and its structure at once. Instead of performing subsequent updates on pages separately, pages can be updated in a single run, requiring fewer disk accesses and structural changes, since splits and merges propagate to the higher levels in batches as well.

The buffering approaches described here optimize tree update time by batching write operations, but in slightly different ways. Both algorithms require additional lookups in in-memory buffering structures and merge/reconciliation with stale disk data.

FD-Trees

Buffering is one of the ideas that is widely used in database storage: it helps to avoid many small random writes and performs a single larger write instead. On HDDs, random writes are slow because of the head positioning. On SSDs, there are no moving parts, but the extra write I/O imposes an additional garbage collection penalty.

Maintaining a B-Tree requires a lot of random writes—leaf-level writes, splits, and merges propagating to the parents—but what if we could avoid random writes and node updates altogether?

So far we've discussed buffering updates to individual nodes or groups of nodes by creating auxiliary buffers. An alternative approach is to group updates targeting *different nodes* together by using append-only storage and merge processes, an idea that has also inspired LSM Trees (see "LSM Trees" on page 130). This means that any write we perform does not require locating a target node for the write: all updates are

simply appended. One of the examples of using this approach for indexing is called Flash Disk Tree (FD-Tree) [LI10].

An FD-Tree consists of a small mutable *head tree* and multiple immutable sorted runs. This approach limits the surface area, where random write I/O is required, to the head tree: a small B-Tree buffering the updates. As soon as the head tree fills up, its contents are transferred to the immutable *run*. If the size of the newly written run exceeds the threshold, its contents are merged with the next level, gradually propagating data records from upper to lower levels.

Fractional Cascading

To maintain pointers between the levels, FD-Trees use a technique called *fractional cascading* [CHAZELLE86]. This approach helps to reduce the cost of locating an item in the cascade of sorted arrays: you perform `log n` steps to find the searched item in the first array, but subsequent searches are significantly cheaper, since they start the search from the closest match from the previous level.

Shortcuts between the levels are made by building *bridges* between the neighbor-level arrays to minimize the *gaps*: element groups without pointers from higher levels. Bridges are built by *pulling* elements from lower levels to the higher ones, if they don't already exist there, and pointing to the location of the pulled element in the lower-level array.

Since [CHAZELLE86] solves a search problem in computational geometry, it describes bidirectional bridges, and an algorithm for restoring the gap size invariant that we won't be covering here. We describe only the parts that are applicable to database storage and FD-Trees in particular.

We could create a mapping from every element of the higher-level array to the closest element on the next level, but that would cause too much overhead for pointers and their maintenance. If we were to map only the items that already exist on a higher level, we could end up in a situation where the gaps between the elements are too large. To solve this problem, we pull every `N`th item from the lower-level array to the higher one.

For example, if we have multiple sorted arrays:

```
A1 = [12, 24, 32, 34, 39]
A2 = [22, 25, 28, 30, 35]
A3 = [11, 16, 24, 26, 30]
```

We can bridge the gaps between elements by pulling every other element from the array with a higher index to the one with a lower index in order to simplify searches:

```
A1 = [12, 24, 25, 30, 32, 34, 39]
A2 = [16, 22, 25, 26, 28, 30, 35]
A3 = [11, 16, 24, 26, 30]
```

Now, we can use these pulled elements to create *bridges* (or *fences* as the FD-Tree paper calls them): pointers from higher-level elements to their counterparts on the lower levels, as Figure 6-5 shows.

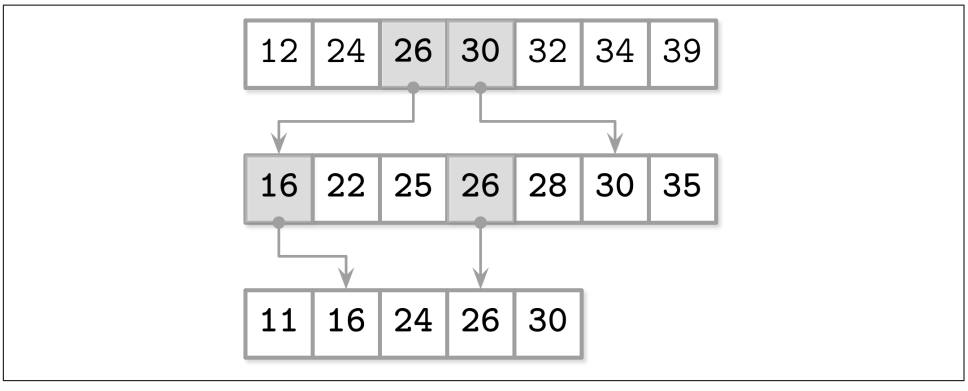

Figure 6-5. Fractional cascading

To search for elements in *all* these arrays, we perform a binary search on the highest level, and the search space on the next level is reduced significantly, since now we are forwarded to the approximate location of the searched item by following a bridge. This allows us to connect multiple sorted runs and reduce the costs of searching in them.

Logarithmic Runs

An FD-Tree combines fractional cascading with creating *logarithmically sized sorted runs*: immutable sorted arrays with sizes increasing by a factor of `k`, created by merging the previous level with the current one.

The highest-level run is created when the head tree becomes full: its leaf contents are written to the first level. As soon as the head tree fills up again, its contents are merged with the first-level items. The merged result replaces the old version of the first run. The lower-level runs are created when the sizes of the higher-level ones reach a threshold. If a lower-level run already exists, it is replaced by the result of merging its contents with the contents of a higher level. This process is quite similar to compaction in LSM Trees, where immutable table contents are merged to create larger tables.

Figure 6-6 shows a schematic representation of an FD-Tree, with a head B-Tree on the top, two logarithmic runs `L1` and `L2`, and bridges between them.

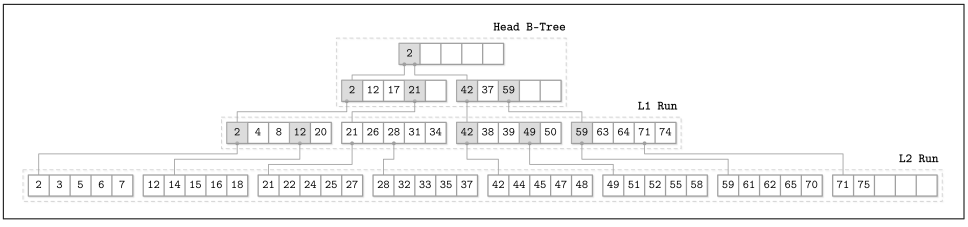

Figure 6-6. Schematic FD-Tree overview

To keep items in all sorted runs addressable, FD-Trees use an adapted version of fractional cascading, where *head elements* from lower-level pages are propagated as pointers to the higher levels. Using these pointers, the cost of searching in lower-level trees is reduced, since the search was already partially done on a higher level and can continue from the closest match.

Since FD-Trees do not update pages in place, and it may happen that data records for the same key are present on several levels, the FD-Trees delete work by inserting tombstones (the FD-Tree paper calls them *filter entries*) that indicate that the data record associated with a corresponding key is marked for deletion, and all data records for that key in the lower levels have to be discarded. When tombstones propagate all the way to the lowest level, they can be discarded, since it is guaranteed that there are no items they can shadow anymore.

Bw-Trees

Write amplification is one of the most significant problems with in-place update implementations of B-Trees: subsequent updates to a B-Tree page may require updating a disk-resident page copy on every update. The second problem is space amplification: we reserve extra space to make updates possible. This also means that for each transferred *useful* byte carrying the requested data, we have to transfer some empty bytes and the rest of the page. The third problem is complexity in solving concurrency problems and dealing with latches.

To solve all three problems at once, we have to take an approach entirely different from the ones we've discussed so far. Buffering updates helps with write and space amplification, but offers no solution to concurrency issues.

We can batch updates to different nodes by using append-only storage, link nodes together into chains, and use an in-memory data structure that allows *installing* pointers between the nodes with a single compare-and-swap operation, making the tree lock-free. This approach is called a *Buzzword-Tree* (Bw-Tree) [LEVANDOSKI14] .

Update Chains

A Bw-Tree writes a *base node* separately from its modifications. Modifications (*delta nodes*) form a chain: a linked list from the newest modification, through older ones, with the base node in the end. Each update can be stored separately, without needing to rewrite the existing node on disk. Delta nodes can represent inserts, updates (which are indistinguishable from inserts), or deletes.

Since the sizes of base and delta nodes are unlikely to be page aligned, it makes sense to store them contiguously, and because neither base nor delta nodes are modified during update (all modifications just prepend a node to the existing linked list), we do not need to reserve any extra space.

Having a node as a logical, rather than physical, entity is an interesting paradigm change: we do not need to pre-allocate space, require nodes to have a fixed size, or even keep them in contiguous memory segments. This certainly has a downside: during a read, all deltas have to be traversed and applied to the base node to reconstruct the actual node state. This is somewhat similar to what LA-Trees do (see "Lazy-Adaptive Tree" on page 116): keeping updates separate from the main structure and replaying them on read.

Taming Concurrency with Compare-and-Swap

It would be quite costly to maintain an on-disk tree structure that allows prepending items to child nodes: it would require us to constantly update parent nodes with pointers to the freshest delta. This is why Bw-Tree nodes, consisting of a chain of deltas and the base node, have logical identifiers and use an in-memory *mapping table* from the identifiers to their locations on disk. Using this mapping also helps us to get rid of latches: instead of having exclusive ownership during write time, the Bw-Tree uses compare-and-swap operations on physical offsets in the mapping table.

Figure 6-7 shows a simple Bw-Tree. Each logical node consists of a single base node and multiple linked delta nodes.

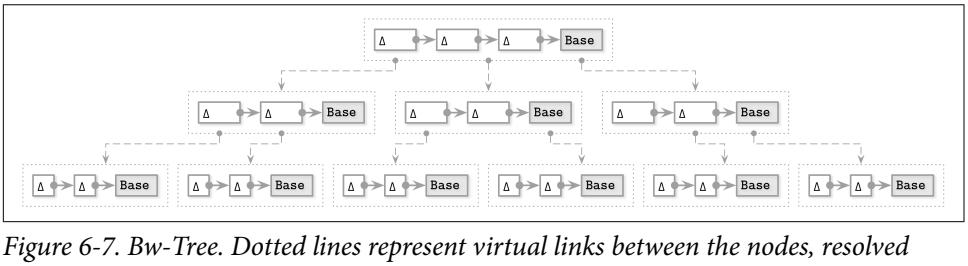

Figure 6-7. Bw-Tree. Dotted lines represent virtual links between the nodes, resolved using the mapping table. Solid lines represent actual data pointers between the nodes.

To update a Bw-Tree node, the algorithm executes the following steps:

1. The target logical *leaf* node is located by traversing the tree from root to leaf. The mapping table contains virtual links to target base nodes or the latest delta nodes in the update chain.
2. A new delta node is created with a pointer to the base node (or to the latest delta node) located during step 1.
3. The mapping table is updated with a pointer to the new delta node created during step 2.

An update operation during step 3 can be done using compare-and-swap, which is an atomic operation, so all reads, concurrent to the pointer update, are ordered either *before* or *after* the write, without blocking either the readers or the writer. Reads ordered *before* follow the old pointer and do not see the new delta node, since it was not yet installed. Reads ordered *after* follow the new pointer, and observe the update. If two threads attempt to install a new delta node to the same logical node, only one of them can succeed, and the other one has to retry the operation.

Structural Modification Operations

A Bw-Tree is logically structured like a B-Tree, which means that nodes still might grow to be too large (overflow) or shrink to be almost empty (underflow) and require structure modification operations (SMOs), such as splits and merges. The semantics of splits and merges here are similar to those of B-Trees (see "B-Tree Node Splits" on page 39 and "B-Tree Node Merges" on page 41), but their implementation is different.

Split SMOs start by consolidating the logical contents of the splitting node, applying deltas to its base node, and creating a new page containing elements to the right of the split point. After this, the process proceeds in two steps [WANG18]:

1. *Split*—A special *split delta* node is appended to the splitting node to notify the readers about the ongoing split. The split delta node holds a midpoint separator key to invalidate records in the splitting node, and a link to the new logical sibling node.
2. *Parent update*—At this point, the situation is similar to that of the B^{link}-Tree *half-split* (see "Blink-Trees" on page 107), since the node is available through the split delta node pointer, but is not yet referenced by the parent, and readers have to go through the old node and then traverse the sibling pointer to reach the newly created sibling node. A new node is added as a child to the parent node, so that readers can directly reach it instead of being redirected through the splitting node, and the split completes.

Updating the parent pointer is a performance optimization: all nodes and their elements remain accessible even if the parent pointer is never updated. Bw-Trees are latch-free, so any thread can encounter an incomplete SMO. The thread is required to cooperate by picking up and finishing a multistep SMO before proceeding. The next thread will follow the installed parent pointer and won't have to go through the sibling pointer.

Merge SMOs work in a similar way:

1. *Remove sibling*—A special *remove delta* node is created and appended to the *right* sibling, indicating the start of the merge SMO and marking the right sibling for deletion.
2. *Merge*—A *merge delta* node is created on the *left* sibling to point to the contents of the right sibling and making it a logical part of the left sibling.
3. *Parent update*—At that point, the right sibling node contents are accessible from the left one. To finish the merge process, the link to the right sibling has to be removed from the parent.

Concurrent SMOs require an additional *abort delta* node to be installed on the parent to prevent concurrent splits and merges [WANG18]. An abort delta works similarly to a write lock: only one thread can have write access at a time, and any thread that attempts to append a new record to this delta node will abort. On SMO completion, the abort delta can be removed from the parent.

The Bw-Tree height grows during the root node splits. When the root node gets too big, it is split in two, and a new root is created in place of the old one, with the old root and a newly created sibling as its children.

Consolidation and Garbage Collection

Delta chains can get arbitrarily long without any additional action. Since reads are getting more expensive as the delta chain gets longer, we need to try to keep the delta chain length within reasonable bounds. When it reaches a configurable threshold, we rebuild the node by merging the base node contents with all of the deltas, consolidating them to one new base node. The new node is then written to the new location on disk and the node pointer in the mapping table is updated to point to it. We discuss this process in more detail in "LLAMA and Mindful Stacking" on page 160, as the underlying log-structured storage is responsible for garbage collection, node consolidation, and relocation.

As soon as the node is consolidated, its old contents (the base node and all of the delta nodes) are no longer addressed from the mapping table. However, we cannot free the memory they occupy right away, because some of them might be still used by ongoing operations. Since there are no latches held by readers (readers did not have to pass through or register at any sort of barrier to access the node), we need to find other means to track live pages.

To separate threads that might have encountered a specific node from those that couldn't have possibly seen it, Bw-Trees use a technique known as *epoch-based reclamation*. If some nodes and deltas are removed from the mapping table due to consolidations that replaced them during some epoch, original nodes are preserved until every reader that started during the same epoch or the earlier one is finished. After that, they can be safely garbage collected, since later readers are guaranteed to have never seen those nodes, as they were not addressable by the time those readers started.

The Bw-Tree is an interesting B-Tree variant, making improvements on several important aspects: write amplification, nonblocking access, and cache friendliness. A modified version was implemented in Sled, an experimental storage engine. The CMU Database Group has developed an in-memory version of the Bw-Tree called OpenBw-Tree and released a practical implementation guide [WANG18].

We've only touched on higher-level Bw-Tree concepts related to B-Trees in this chapter, and we continue the discussion about them in "LLAMA and Mindful Stacking" on page 160, including the discussion about the underlying log-structured storage.

Cache-Oblivious B-Trees

Block size, node size, cache line alignments, and other configurable parameters influence B-Tree performance. A new class of data structures called *cache-oblivious structures* [DEMAINE02] give asymptotically optimal performance regardless of the underlying memory hierarchy and a need to tune these parameters. This means that the algorithm is not required to know the sizes of the cache lines, filesystem blocks, and disk pages. Cache-oblivious structures are designed to perform well without modification on multiple machines with different configurations.

So far, we've been mostly looking at B-Trees from a two-level memory hierarchy (with the exception of LMDB described in "Copy-on-Write" on page 112). B-Tree nodes are stored in disk-resident pages, and the page cache is used to allow efficient access to them in main memory.

The two levels of this hierarchy are *page cache* (which is faster, but is limited in space) and *disk* (which is generally slower, but has a larger capacity) [AGGARWAL88]. Here, we have only two parameters, which makes it rather easy to design algorithms as we only have to have two level-specific code modules that take care of all the details relevant to that level.

The disk is partitioned into blocks, and data is transferred between disk and cache in blocks: even when the algorithm has to locate a single item within the block, an entire block has to be loaded. This approach is *cache-aware*.

When developing performance-critical software, we often program for a more complex model, taking into consideration CPU caches, and sometimes even disk hierarchies (like hot/cold storage or build HDD/SSD/NVM hierarchies, and phase off data from one level to the other). Most of the time such efforts are difficult to generalize. In "Memory- Versus Disk-Based DBMS" on page 10, we talked about the fact that accessing disk is several orders of magnitude slower than accessing main memory, which has motivated database implementers to optimize for this difference.

Cache-oblivious algorithms allow reasoning about data structures in terms of a two-level memory model while providing the benefits of a multilevel hierarchy model. This approach allows having no platform-specific parameters, yet guarantees that the number of transfers between the two levels of the hierarchy is within a constant factor. If the data structure is optimized to perform optimally for any two levels of memory hierarchy, it also works optimally for the two *adjacent* hierarchy levels. This is achieved by working at the highest cache level as much as possible.

van Emde Boas Layout

A cache-oblivious B-Tree consists of a static B-Tree and a packed array structure [BENDER05]. A static B-Tree is built using the *van Emde Boas* layout. It splits the tree at the middle level of the edges. Then each subtree is split recursively in a similar manner, resulting in subtrees of `sqr(N)` size. The key idea of this layout is that any recursive tree is stored in a contiguous block of memory.

In Figure 6-8, you can see an example of a van Emde Boas layout. Nodes, logically grouped together, are placed closely together. On top, you can see a logical layout representation (i.e., how nodes form a tree), and on the bottom you can see how tree nodes are laid out in memory and on disk.

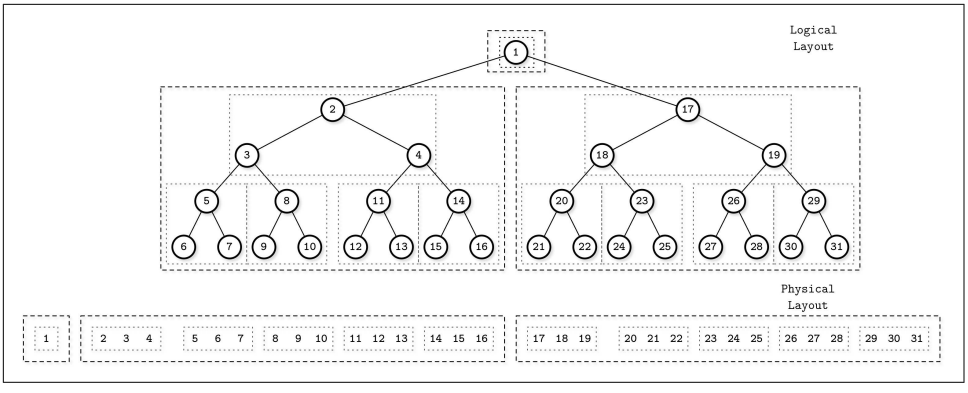

Figure 6-8. van Emde Boas layout

To make the data structure dynamic (i.e., allow inserts, updates, and deletes), cache-oblivious trees use a *packed array* data structure, which uses contiguous memory segments for storing elements, but contains gaps reserved for future inserted elements. Gaps are spaced based on the *density threshold*. Figure 6-9 shows a packed array structure, where elements are spaced to create gaps.

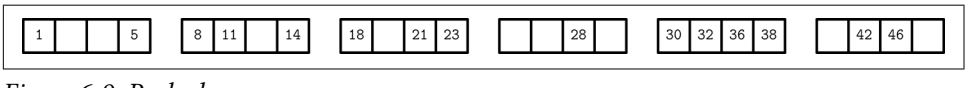

Figure 6-9. Packed array

This approach allows inserting items into the tree with fewer relocations. Items have to be relocated just to create a gap for the newly inserted element, if the gap is not already present. When the packed array becomes too densely or sparsely populated, the structure has to be rebuilt to grow or shrink the array.

The static tree is used as an index for the bottom-level packed array, and has to be updated in accordance with relocated elements to point to correct elements on the bottom level.

This is an interesting approach, and ideas from it can be used to build efficient B-Tree implementations. It allows constructing on-disk structures in ways that are very similar to how main memory ones are constructed. However, as of the date of writing, I'm not aware of any nonacademic cache-oblivious B-Tree implementations.

A possible reason for that is an assumption that when cache loading is abstracted away, while data is loaded and written back in blocks, paging and eviction still have a negative impact on the result. Another possible reason is that in terms of block transfers, the complexity of cache-oblivious B-Trees is the same as their cache-aware counterpart. This may change when more efficient nonvolatile byte-addressable storage devices become more widespread.

Summary

The original B-Tree design has several shortcomings that might have worked well on spinning disks, but make it less efficient when used on SSDs. B-Trees have high *write amplification* (caused by page rewrites) and high *space overhead* since B-Trees have to reserve space in nodes for future writes.

Write amplification can be reduced by using *buffering*. Lazy B-Trees, such as WiredTiger and LA-Trees, attach in-memory buffers to individual nodes or groups of nodes to reduce the number of required I/O operations by buffering subsequent updates to pages in memory.

To reduce space amplification, FD-Trees use *immutability*: data records are stored in the immutable sorted *runs*, and the size of a mutable B-Tree is limited.

Bw-Trees solve space amplification by using immutability, too. B-Tree nodes and updates to them are stored in separate on-disk locations and persisted in the log-structured store. Write amplification is reduced compared to the original B-Tree design, since reconciling contents that belong to a single logical node is relatively infrequent. Bw-Trees do not require latches for protecting pages from concurrent accesses, as the virtual pointers between the logical nodes are stored in memory.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Copy-on-Write B-Trees

Driscoll, J. R., N. Sarnak, D. D. Sleator, and R. E. Tarjan. 1986. "Making data structures persistent." In *Proceedings of the eighteenth annual ACM symposium on Theory of computing (STOC '86)*, 109-121. *https://dx.doi.org/10.1016/0022-0000(89)90034-2*.

Lazy-Adaptive Trees

Agrawal, Devesh, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. 2009. "Lazy-Adaptive Tree: an optimized index structure for flash devices." *Proceedings of the VLDB Endowment* 2, no. 1 (January): 361-372. *https://doi.org/10.14778/1687627.1687669*.

FD-Trees

Li, Yinan, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. "Tree Indexing on Solid State Drives." *Proceedings of the VLDB Endowment* 3, no. 1-2 (September): 1195-1206. *https://doi.org/10.14778/1920841.1920990*.

Bw-Trees

Wang, Ziqi, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. "Building a Bw-Tree Takes More Than Just Buzz Words." *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, 473–488. *https://doi.org/10.1145/3183713.3196895*

Levandoski, Justin J., David B. Lomet, and Sudipta Sengupta. 2013. "The Bw-Tree: A B-tree for new hardware platforms." In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*, 302-313. IEEE. *https://doi.org/10.1109/ICDE.2013.6544834*.

Cache-Oblivious B-Trees

Bender, Michael A., Erik D. Demaine, and Martin Farach-Colton. 2005. "Cache-Oblivious B-Trees." *SIAM Journal on Computing* 35, no. 2 (August): 341-358. *https://doi.org/10.1137/S0097539701389956*.

CHAPTER 7
Log-Structured Storage

> Accountants don't use erasers or they end up in jail.
>
> —Pat Helland

When accountants have to modify the record, instead of erasing the existing value, they create a new record with a correction. When the quarterly report is published, it may contain minor modifications, correcting the previous quarter results. To derive the bottom line, you have to go through the records and calculate a subtotal [HELLAND15].

Similarly, immutable storage structures do not allow modifications to the existing files: tables are written once and are never modified again. Instead, new records are appended to the new file and, to find the final value (or conclude its absence), records have to be reconstructed from multiple files. In contrast, mutable storage structures modify records on disk in place.

Immutable data structures are often used in functional programming languages and are getting more popular because of their safety characteristics: once created, an immutable structure doesn't change, all of its references can be accessed concurrently, and its integrity is guaranteed by the fact that it cannot be modified.

On a high level, there is a strict distinction between how data is treated inside a storage structure and outside of it. Internally, immutable files can hold multiple copies, more recent ones overwriting the older ones, while mutable files generally hold only the most recent value instead. When accessed, immutable files are processed, redundant copies are reconciled, and the most recent ones are returned to the client.

As do other books and papers on the subject, we use B-Trees as a typical example of mutable structure and *Log-Structured Merge Trees* (LSM Trees) as an example of an immutable structure. Immutable LSM Trees use append-only storage and merge

reconciliation, and B-Trees locate data records on disk and update pages at their original offsets in the file.

In-place update storage structures are optimized for read performance [GRAEFE04]: after locating data on disk, the record can be returned to the client. This comes at the expense of write performance: to update the data record in place, it first has to be located on disk. On the other hand, append-only storage is optimized for write performance. Writes do not have to locate records on disk to overwrite them. However, this is done at the expense of reads, which have to retrieve multiple data record versions and reconcile them.

So far we've mostly talked about mutable storage structures. We've touched on the subject of immutability while discussing copy-on-write B-Trees (see "Copy-on-Write" on page 112), FD-Trees (see "FD-Trees" on page 117), and Bw-Trees (see "Bw-Trees" on page 120). But there are more ways to implement immutable structures.

Because of the structure and construction approach taken by mutable B-Trees, most I/O operations during reads, writes, and maintenance are *random*. Each write operation first needs to locate a page that holds a data record and only then can modify it. B-Trees require node splits and merges that relocate already written records. After some time, B-Tree pages may require maintenance. Pages are fixed in size, and some free space is reserved for future writes. Another problem is that even when only one cell in the page is modified, an entire page has to be rewritten.

There are alternative approaches that can help to mitigate these problems, make some of the I/O operations sequential, and avoid page rewrites during modifications. One of the ways to do this is to use immutable structures. In this chapter, we'll focus on LSM Trees: how they're built, what their properties are, and how they are different from B-Trees.

LSM Trees

When talking about B-Trees, we concluded that space overhead and write amplification can be improved by using buffering. Generally, there are two ways buffering can be applied in different storage structures: to postpone propagating writes to disk-resident pages (as we've seen with "FD-Trees" on page 117 and "WiredTiger" on page 114), and to make write operations sequential.

One of the most popular immutable on-disk storage structures, LSM Tree uses buffering and append-only storage to achieve sequential writes. The LSM Tree is a variant of a disk-resident structure similar to a B-Tree, where nodes are fully occupied, optimized for sequential disk access. This concept was first introduced in a paper by Patrick O'Neil and Edward Cheng [ONEIL96]. Log-structured merge trees take their name from log-structured filesystems, which write all modifications on disk in a log-like file [ROSENBLUM92].

LSM Trees write immutable files and merge them together over time. These files usually contain an index of their own to help readers efficiently locate data. Even though LSM Trees are often presented as an alternative to B-Trees, it is common for B-Trees to be used as the internal indexing structure for an LSM Tree's immutable files.

The word "merge" in LSM Trees indicates that, due to their immutability, tree contents are merged using an approach similar to merge sort. This happens during maintenance to reclaim space occupied by the redundant copies, and during reads, before contents can be returned to the user.

LSM Trees defer data file writes and buffer changes in a memory-resident table. These changes are then propagated by writing their contents out to the immutable disk files. All data records remain accessible in memory until the files are fully persisted.

Keeping data files immutable favors sequential writes: data is written on the disk in a single pass and files are append-only. Mutable structures can pre-allocate blocks in a single pass (for example, indexed sequential access method (ISAM) [RAMAKRISHNAN03] [LARSON81]), but subsequent accesses still require random reads and writes. Immutable structures allow us to lay out data records sequentially to prevent fragmentation. Additionally, immutable files have higher *density*: we do not reserve any extra space for data records that are going to be written later, or for the cases when updated records require more space than the originally written ones.

Since files are immutable, insert, update, and delete operations do not need to locate data records on disk, which significantly improves write performance and throughput. Instead, duplicate contents are allowed, and conflicts are resolved during the read time. LSM Trees are particularly useful for applications where writes are far more common than reads, which is often the case in modern data-intensive systems, given ever-growing amounts of data and ingest rates.

Reads and writes do not intersect by design, so data on disk can be read and written without segment locking, which significantly simplifies concurrent access. In contrast, mutable structures employ hierarchical locks and latches (you can find more information about locks and latches in "Concurrency Control" on page 93) to ensure on-disk data structure integrity, and allow multiple concurrent readers but require exclusive subtree ownership for writers. LSM-based storage engines use linearizable in-memory views of data and index files, and only have to guard concurrent access to the structures managing them.

Both B-Trees and LSM Trees require some housekeeping to optimize performance, but for different reasons. Since the number of allocated files steadily grows, LSM Trees have to merge and rewrite files to make sure that the smallest possible number

of files is accessed during the read, as requested data records might be spread across multiple files. On the other hand, mutable files may have to be rewritten partially or wholly to decrease fragmentation and reclaim space occupied by updated or deleted records. Of course, the exact scope of work done by the housekeeping process heavily depends on the concrete implementation.

LSM Tree Structure

We start with ordered LSM Trees [ONEIL96], where files hold sorted data records. Later, in "Unordered LSM Storage" on page 152, we'll also discuss structures that allow storing data records in insertion order, which has some obvious advantages on the write path.

As we just discussed, LSM Trees consist of smaller memory-resident and larger disk-resident components. To write out immutable file contents on disk, it is necessary to first *buffer* them in memory and *sort* their contents.

A memory-resident component (often called a *memtable*) is mutable: it buffers data records and serves as a target for read and write operations. Memtable contents are persisted on disk when its size grows up to a configurable threshold. Memtable updates incur no disk access and have no associated I/O costs. A separate write-ahead log file, similar to what we discussed in "Recovery" on page 88, is required to guarantee durability of data records. Data records are appended to the log and committed in memory before the operation is acknowledged to the client.

Buffering is done in memory: all read and write operations are applied to a memory-resident table that maintains a sorted data structure allowing concurrent access, usually some form of an in-memory sorted tree, or any data structure that can give similar performance characteristics.

Disk-resident components are built by *flushing* contents buffered in memory to disk. Disk-resident components are used only for reads: buffered contents are persisted, and files are never modified. This allows us to think in terms of simple operations: writes against an in-memory table, and reads against disk and memory-based tables, merges, and file removals.

Throughout this chapter, we will be using the word *table* as a shortcut for *disk-resident table*. Since we're discussing semantics of a storage engine, this term is not ambiguous with a *table* concept in the wider context of a database management system.

Two-component LSM Tree

We distinguish between two- and multicomponent LSM Trees. *Two-component LSM Trees* have only one disk component, comprised of immutable segments. The disk

component here is organized as a B-Tree, with 100% node occupancy and read-only pages.

Memory-resident tree contents are flushed on disk in parts. During a flush, for each flushed in-memory subtree, we find a corresponding subtree on disk and write out the merged contents of a memory-resident segment and disk-resident subtree into the new segment on disk. Figure 7-1 shows in-memory and disk-resident trees before a merge.

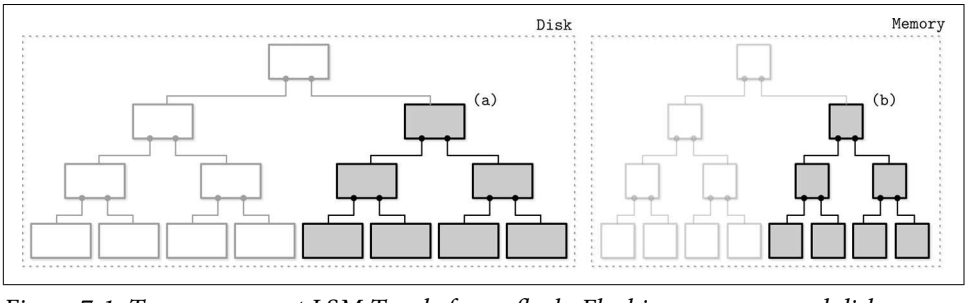

Figure 7-1. Two-component LSM Tree before a flush. Flushing memory- and disk-resident segments are shown in gray.

After the subtree is flushed, superseded memory-resident and disk-resident subtrees are discarded and replaced with the result of their merge, which becomes addressable from the preexisting sections of the disk-resident tree. Figure 7-2 shows the result of a merge process, already written to the new location on disk and attached to the rest of the tree.

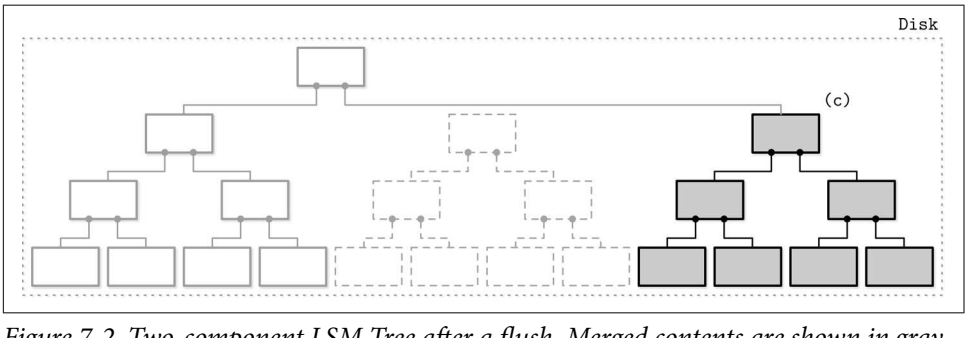

Figure 7-2. Two-component LSM Tree after a flush. Merged contents are shown in gray. Boxes with dashed lines depict discarded on-disk segments.

A merge can be implemented by advancing iterators reading the disk-resident leaf nodes and contents of the in-memory tree in lockstep. Since both sources are sorted, to produce a sorted merged result, we only need to know the *current* values of both iterators during each step of the merge process.

This approach is a logical extension and continuation of our conversation on immutable B-Trees. Copy-on-write B-Trees (see "Copy-on-Write" on page 112) use B-Tree structure, but their nodes are not fully occupied, and they require copying pages on the root-leaf path and creating a *parallel* tree structure. Here, we do something similar, but since we buffer writes in memory, we amortize the costs of the disk-resident tree update.

When implementing subtree merges and flushes, we have to make sure of three things:

1. *As soon as* the flush process starts, all new writes have to go to the new memtable.
2. *During* the subtree flush, both the disk-resident and flushing memory-resident subtree have to remain accessible for reads.
3. *After* the flush, publishing merged contents, and discarding unmerged disk- and memory-resident contents have to be performed atomically.

Even though two-component LSM Trees can be useful for maintaining index files, no implementations are known to the author as of time of writing. This can be explained by the write amplification characteristics of this approach: merges are relatively frequent, as they are triggered by memtable flushes.

Multicomponent LSM Trees

Let's consider an alternative design, multicomponent LSM Trees that have more than just one disk-resident table. In this case, entire memtable contents are flushed in a single run.

It quickly becomes evident that after multiple flushes we'll end up with multiple disk-resident tables, and their number will only grow over time. Since we do not always know exactly which tables are holding required data records, we might have to access multiple files to locate the searched data.

Having to read from multiple sources instead of just one might get expensive. To mitigate this problem and keep the number of tables to minimum, a periodic merge process called *compaction* (see "Maintenance in LSM Trees" on page 141) is triggered. Compaction picks several tables, reads their contents, merges them, and writes the merged result out to the new combined file. Old tables are discarded simultaneously with the appearance of the new merged table.

Figure 7-3 shows the multicomponent LSM Tree data life cycle. Data is first buffered in a memory-resident component. When it gets too large, its contents are flushed on disk to create disk-resident tables. Later, multiple tables are merged together to create larger tables.

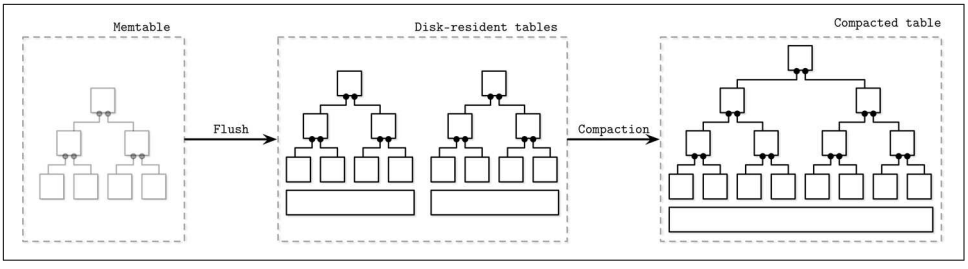

Figure 7-3. Multicomponent LSM Tree data life cycle

The rest of this chapter is dedicated to multicomponent LSM Trees, building blocks, and their maintenance processes.

In-memory tables

Memtable flushes can be triggered periodically, or by using a size threshold. Before it can be flushed, the memtable has to be *switched*: a new memtable is allocated, and it becomes a target for all new writes, while the old one moves to the flushing state. These two steps have to be performed atomically. The flushing memtable remains available for reads until its contents are fully flushed. After this, the old memtable is discarded in favor of a newly written disk-resident table, which becomes available for reads.

In Figure 7-4, you see the components of the LSM Tree, relationships between them, and operations that fulfill transitions between them:

Current memtable
: Receives writes and serves reads.

Flushing memtable
: Available for reads.

On-disk flush target
: Does not participate in reads, as its contents are incomplete.

Flushed tables
: Available for reads as soon as the flushed memtable is discarded.

Compacting tables
: Currently merging disk-resident tables.

Compacted tables
: Created from flushed or other compacted tables.

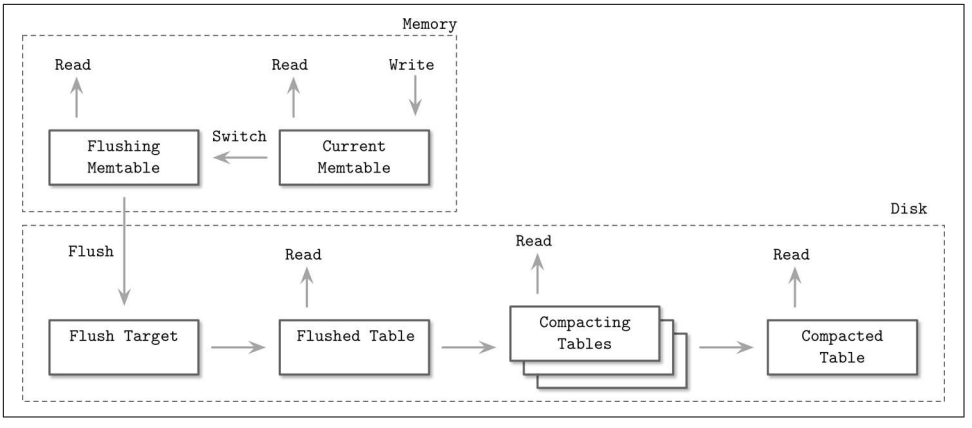

Figure 7-4. LSM component structure

Data is already sorted in memory, so a disk-resident table can be created by sequentially writing out memory-resident contents to disk. During a flush, both the flushing memtable and the current memtable are available for read.

Until the memtable is fully flushed, the only disk-resident version of its contents is stored in the write-ahead log. When memtable contents are fully flushed on disk, the log can be *trimmed*, and the log section, holding operations applied to the flushed memtable, can be discarded.

Updates and Deletes

In LSM Trees, insert, update, and delete operations do not require locating data records on disk. Instead, redundant records are reconciled during the read.

Removing data records from the memtable is not enough, since other disk or memory resident tables may hold data records for the same key. If we were to implement deletes by just removing items from the memtable, we would end up with deletes that either have no impact or would *resurrect* the previous values.

Let's consider an example. The flushed disk-resident table contains data record `v1` associated with a key `k1`, and the memtable holds its new value `v2`:

```
Disk Table        Memtable
| k1 | v1 |       | k1 | v2 |
```

If we just remove `v2` from the memtable and flush it, we effectively resurrect `v1`, since it becomes the only value associated with that key:

```
Disk Table        Memtable
| k1 | v1 |       ∅
```

Because of that, deletes need to be recorded *explicitly*. This can be done by inserting a special *delete entry* (sometimes called a *tombstone* or a *dormant certificate*), indicating removal of the data record associated with a specific key:

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

The reconciliation process picks up tombstones, and filters out the shadowed values.

Sometimes it might be useful to remove a consecutive range of keys rather than just a single key. This can be done using *predicate deletes*, which work by appending a delete entry with a predicate that sorts according to regular record-sorting rules. During reconciliation, data records matching the predicate are skipped and not returned to the client.

Predicates can take a form of `DELETE FROM table WHERE key ≥ "k2" AND key < "k4"` and can receive any range matchers. Apache Cassandra implements this approach and calls it *range tombstones*. A range tombstone covers a range of keys rather than just a single key.

When using range tombstones, resolution rules have to be carefully considered because of overlapping ranges and disk-resident table boundaries. For example, the following combination will hide data records associated with `k2` and `k3` from the final result:

```
Disk Table 1      Disk Table 2
| k1 | v1 |       | k2 | <start_tombstone_inclusive> |
| k2 | v2 |       | k4 | <end_tombstone_exclusive>   |
| k3 | v3 |
| k4 | v4 |
```

LSM Tree Lookups

LSM Trees consist of multiple components. During lookups, more than one component is usually accessed, so their contents have to be merged and reconciled before they can be returned to the client. To better understand the merge process, let's see how tables are iterated during the merge and how conflicting records are combined.

Merge-Iteration

Since contents of disk-resident tables are sorted, we can use a multiway merge-sort algorithm. For example, we have three sources: two disk-resident tables and one memtable. Usually, storage engines offer a *cursor* or an *iterator* to navigate through file contents. This cursor holds the offset of the last consumed data record, can be checked for whether or not iteration has finished, and can be used to retrieve the next data record.

A multiway merge-sort uses a *priority queue*, such as *min-heap* [SEDGEWICK11], that holds up to `N` elements (where `N` is the number of iterators), which sorts its contents and prepares the next-in-line smallest element to be returned. The head of each iterator is placed into the queue. An element in the head of the queue is then the minimum of all iterators.

A priority queue is a data structure used for maintaining an ordered queue of items. While a regular queue retains items in order of their addition (first in, first out), a priority queue re-sorts items on insertion and the item with the highest (or lowest) priority is placed in the head of the queue. This is particularly useful for merge-iteration, since we have to output elements in a sorted order.

When the smallest element is removed from the queue, the iterator associated with it is checked for the next value, which is then placed into the queue, which is re-sorted to preserve the order.

Since all iterator contents are sorted, reinserting a value from the iterator that held the previous smallest value of all iterator heads also preserves an invariant that the queue still holds the smallest elements from all iterators. Whenever one of the iterators is exhausted, the algorithm proceeds without reinserting the next iterator head. The algorithm continues until either query conditions are satisfied or all iterators are exhausted.

Figure 7-5 shows a schematic representation of the merge process just described: head elements (light gray items in source tables) are placed to the priority queue. Elements from the priority queue are returned to the output iterator. The resulting output is sorted.

It may happen that we encounter more than one data record for the same key during merge-iteration. From the priority queue and iterator invariants, we know that if each iterator only holds a single data record per key, and we end up with multiple records for the same key in the queue, these data records must have come from the different iterators.

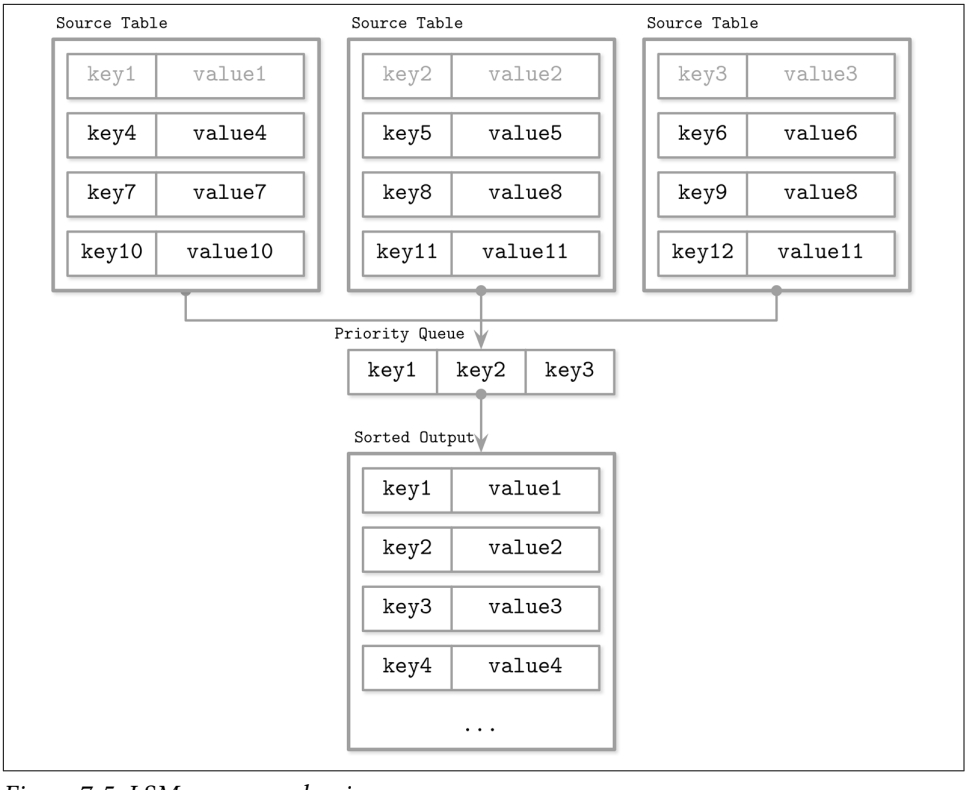

Figure 7-5. LSM merge mechanics

Let's follow through one example step-by-step. As input data, we have iterators over two disk-resident tables:

```
Iterator 1:         Iterator 2:
{k2: v1} {k4: v2}   {k1: v3} {k2: v4} {k3: v5}
```

The priority queue is filled from the iterator heads:

```
Iterator 1:         Iterator 2:         Priority queue:
{k4: v2}            {k2: v4} {k3: v5}   {k1: v3} {k2: v1}
```

Key `k1` is the smallest key in the queue and is appended to the result. Since it came from `Iterator 2`, we refill the queue from it:

```
Iterator 1:         Iterator 2:         Priority queue:      Merged Result:
{k4: v2}            {k3: v5}            {k2: v1} {k2: v4}    {k1: v3}
```

Now, we have *two* records for the `k2` key in the queue. We can be sure there are no other records with the same key in any iterator because of the aforementioned invariants. Same-key records are merged and appended to the merged result.

The queue is refilled with data from both iterators:

```
Iterator 1:         Iterator 2:         Priority queue:      Merged Result:
{}                  {}                  {k3: v5} {k4: v2}    {k1: v3} {k2: v4}
```

Since all iterators are now empty, we append the remaining queue contents to the output:

```
Merged Result:
  {k1: v3} {k2: v4} {k3: v5} {k4: v2}
```

In summary, the following steps have to be repeated to create a combined iterator:

1. Initially, fill the queue with the first items from each iterator.
2. Take the smallest element (head) from the queue.
3. Refill the queue from the corresponding iterator, unless this iterator is exhausted.

In terms of complexity, merging iterators is the same as merging sorted collections. It has `O(N)` memory overhead, where `N` is the number of iterators. A sorted collection of iterator heads is maintained with `O(log N)` (average case) [KNUTH98].

Reconciliation

Merge-iteration is just a single aspect of what has to be done to merge data from multiple sources. Another important aspect is *reconciliation* and *conflict resolution* of the data records associated with the same key.

Different tables might hold data records for the same key, such as updates and deletes, and their contents have to be reconciled. The priority queue implementation from the preceding example must be able to allow multiple values associated with the same key and trigger reconciliation.

An operation that inserts the record to the database if it does not exist, and updates an existing one otherwise, is called an *upsert*. In LSM Trees, insert and update operations are indistinguishable, since they do not attempt to locate data records previously associated with the key in all sources and reassign its value, so we can say that we *upsert* records by default.

To reconcile data records, we need to understand which one of them takes precedence. Data records hold metadata necessary for this, such as timestamps. To establish the order between the items coming from multiple sources and find out which one is more recent, we can compare their timestamps.

Records shadowed by the records with higher timestamps are not returned to the client or written during compaction.

Maintenance in LSM Trees

Similar to mutable B-Trees, LSM Trees require maintenance. The nature of these processes is heavily influenced by the invariants these algorithms preserve.

In B-Trees, the maintenance process collects unreferenced cells and defragments the pages, reclaiming the space occupied by removed and shadowed records. In LSM Trees, the number of disk-resident tables is constantly growing, but can be reduced by triggering periodic compaction.

Compaction picks multiple disk-resident tables, iterates over their entire contents using the aforementioned merge and reconciliation algorithms, and writes out the results into the newly created table.

Since disk-resident table contents are sorted, and because of the way merge-sort works, compaction has a theoretical memory usage upper bound, since it should only hold iterator heads in memory. All table contents are consumed sequentially, and the resulting merged data is also written out sequentially. These details may vary between implementations due to additional optimizations.

Compacting tables remain available for reads until the compaction process finishes, which means that for the duration of compaction, it is required to have enough free space available on disk for a compacted table to be written.

At any given time, multiple compactions can be executed in the system. However, these concurrent compactions usually work on nonintersecting sets of tables. A compaction writer can both merge several tables into one and partition one table into multiple tables.

Tombstones and Compaction

Tombstones represent an important piece of information required for correct reconciliation, as some other table might still hold an outdated data record shadowed by the tombstone.

During compaction, tombstones are not dropped right away. They are preserved until the storage engine can be certain that no data record for the same key with a smaller timestamp is present in any other table. RocksDB keeps tombstones until they reach the bottommost level. Apache Cassandra keeps tombstones until the GC (garbage collection) grace period is reached because of the eventually consistent nature of the database, ensuring that other nodes observe the tombstone. Preserving tombstones during compaction is important to avoid data resurrection.

Leveled compaction

Compaction opens up multiple opportunities for optimizations, and there are many different compaction strategies. One of the frequently implemented compaction strategies is called *leveled compaction*. For example, it is used by RocksDB.

Leveled compaction separates disk-resident tables into *levels*. Tables on each level have target sizes, and each level has a corresponding *index* number (identifier). Somewhat counterintuitively, the level with the highest index is called the *bottommost* level. For clarity, this section avoids using terms *higher* and *lower level* and uses the same qualifiers for *level index*. That is, since 2 is larger than 1, level 2 has a higher index than level 1. The terms *previous* and *next* have the same order semantics as level indexes.

Level-0 tables are created by flushing memtable contents. Tables in level 0 may contain overlapping key ranges. As soon as the number of tables on level 0 reaches a threshold, their contents are merged, creating new tables for level 1.

Key ranges for the tables on level 1 and all levels with a higher index do not overlap, so level-0 tables have to be partitioned during compaction, split into ranges, and merged with tables holding corresponding key ranges. Alternatively, compaction can include *all* level-0 and level-1 tables, and output partitioned level-1 tables.

Compactions on the levels with the higher indexes pick tables from two consecutive levels with overlapping ranges and produce a new table on a higher level. Figure 7-6 schematically shows how the compaction process migrates data between the levels. The process of compacting level-1 and level-2 tables will produce a new table on level 2. Depending on how tables are partitioned, multiple tables from one level can be picked for compaction.

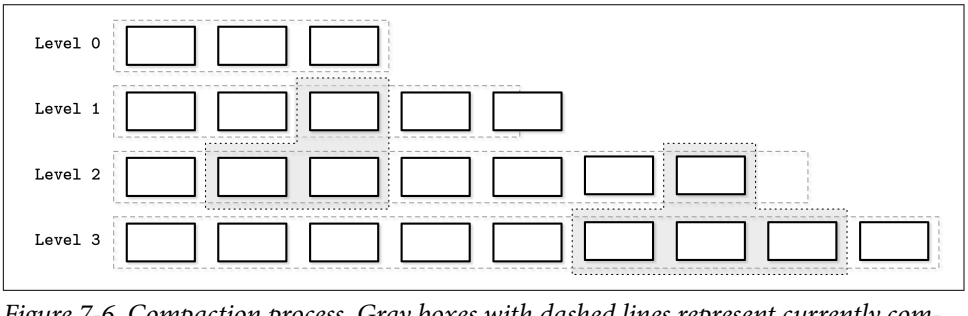

Figure 7-6. Compaction process. Gray boxes with dashed lines represent currently compacting tables. Level-wide boxes represent the target data size limit on the level. Level 1 is over the limit.

Keeping different key ranges in the distinct tables reduces the number of tables accessed during the read. This is done by inspecting the table metadata and filtering out the tables whose ranges do not contain a searched key.

Each level has a limit on the table size and the maximum number of tables. As soon as the number of tables on level 1 or any level with a higher index reaches a threshold, tables from the *current* level are merged with tables on the *next* level holding the overlapping key range.

Sizes grow exponentially between the levels: tables on each next level are exponentially larger than tables on the previous one. This way, the freshest data is always on the level with the lowest index, and older data gradually migrates to the higher ones.

Size-tiered compaction

Another popular compaction strategy is called *size-tiered compaction*. In size-tiered compaction, rather than grouping disk-resident tables based on their level, they're grouped by size: smaller tables are grouped with smaller ones, and bigger tables are grouped with bigger ones.

Level 0 holds the smallest tables that were either flushed from memtables or created by the compaction process. When the tables are compacted, the resulting merged table is written to the level holding tables with corresponding sizes. The process continues recursively incrementing levels, compacting and promoting larger tables to higher levels, and demoting smaller tables to lower levels.

One of the problems with size-tiered compaction is called *table starvation*: if compacted tables are still small enough after compaction (e.g., records were shadowed by the tombstones and did not make it to the merged table), higher levels may get starved of compaction and their tombstones will not be taken into consideration, increasing the cost of reads. In this case, compaction has to be forced for a level, even if it doesn't contain enough tables.

There are other commonly implemented compaction strategies that might optimize for different workloads. For example, Apache Cassandra also implements a *time window* compaction strategy, which is particularly useful for time-series workloads with records for which time-to-live is set (in other words, items have to be expired after a given time period).

The time window compaction strategy takes write timestamps into consideration and allows dropping entire files that hold data for an already expired time range without requiring us to compact and rewrite their contents.

Read, Write, and Space Amplification

When implementing an optimal compaction strategy, we have to take multiple factors into consideration. One approach is to reclaim space occupied by duplicate records and reduce space overhead, which results in higher write amplification caused by re-

writing tables continuously. The alternative is to avoid rewriting the data continuously, which increases read amplification (overhead from reconciling data records associated with the same key during the read), and space amplification (since redundant records are preserved for a longer time).

One of the big disputes in the database community is whether B-Trees or LSM Trees have lower write amplification. It is extremely important to understand the *source* of write amplification in both cases. In B-Trees, it comes from writeback operations and subsequent updates to the same node. In LSM Trees, write amplification is caused by migrating data from one file to the other during compaction. Comparing the two directly may lead to incorrect assumptions.

In summary, when storing data on disk in an immutable fashion, we face three problems:

Read amplification
: Resulting from a need to address multiple tables to retrieve data.

Write amplification
: Caused by continuous rewrites by the compaction process.

Space amplification
: Arising from storing multiple records associated with the same key.

We'll be addressing each one of these throughout the rest of the chapter.

RUM Conjecture

One of the popular cost models for storage structures takes three factors into consideration: *R*ead, *U*pdate, and *M*emory overheads. It is called RUM Conjecture [ATHANASSOULIS16].

RUM Conjecture states that reducing two of these overheads inevitably leads to change for the worse in the third one, and that optimizations can be done only at the expense of one of the three parameters. We can compare different storage engines in terms of these three parameters to understand which ones they optimize for, and which potential trade-offs this may imply.

An ideal solution would provide the lowest read cost while maintaining low memory and write overheads, but in reality, this is not achievable, and we are presented with a trade-off.

B-Trees are read-optimized. Writes to the B-Tree require locating a record on disk, and subsequent writes to the same page might have to update the page on disk multi-

ple times. Reserved extra space for future updates and deletes increases space overhead.

LSM Trees do not require locating the record on disk during write and do not reserve extra space for future writes. There is still some space overhead resulting from storing redundant records. In a default configuration, reads are more expensive, since multiple tables have to be accessed to return complete results. However, optimizations we discuss in this chapter help to mitigate this problem.

As we've seen in the chapters about B-Trees, and will see in this chapter, there are ways to improve these characteristics by applying different optimizations.

This cost model is not perfect, as it does not take into account other important metrics such as latency, access patterns, implementation complexity, maintenance overhead, and hardware-related specifics. Higher-level concepts important for distributed databases, such as consistency implications and replication overhead, are also not considered. However, this model can be used as a first approximation and a rule of thumb as it helps understand what the *storage engine* has to offer.

Implementation Details

We've covered the basic dynamics of LSM Trees: how data is read, written, and compacted. However, there are some other things that many LSM Tree implementations have in common that are worth discussing: how memory- and disk-resident tables are implemented, how secondary indexes work, how to reduce the number of disk-resident tables accessed during read and, finally, new ideas related to log-structured storage.

Sorted String Tables

So far we've discussed the hierarchical and logical structure of LSM Trees (that they consist of multiple memory- and disk-resident components), but have not yet discussed how disk-resident tables are implemented and how their design plays together with the rest of the system.

Disk-resident tables are often implemented using *Sorted String Tables* (SSTables). As the name suggests, data records in SSTables are sorted and laid out in key order. SSTables usually consist of two components: index files and data files. Index files are implemented using some structure allowing logarithmic lookups, such as B-Trees, or constant-time lookups, such as hashtables.

Since data files hold records in key order, using hashtables for indexing does not prevent us from implementing range scans, as a hashtable is only accessed to locate the first key in the range, and the range itself can be read from the data file sequentially while the range predicate still matches.

The index component holds keys and data entries (offsets in the data file where the actual data records are located). The data component consists of concatenated key-value pairs. The cell design and data record formats we discussed in Chapter 3 are largely applicable to SSTables. The main difference here is that cells are written sequentially and are not modified during the life cycle of the SSTable. Since the index files hold pointers to the data records stored in the data file, their offsets have to be known by the time the index is created.

During compaction, data files can be read sequentially without addressing the index component, as data records in them are already ordered. Since tables merged during compaction have the same order, and merge-iteration is order-preserving, the resulting merged table is also created by writing data records sequentially in a single run. As soon as the file is fully written, it is considered immutable, and its disk-resident contents are not modified.

SSTable-Attached Secondary Indexes

One of the interesting developments in the area of LSM Tree indexing is *SSTable-Attached Secondary Indexes* (SASI) implemented in Apache Cassandra. To allow indexing table contents not just by the primary key, but also by any other field, index structures and their life cycles are coupled with the SSTable life cycle, and an index is created per SSTable. When the memtable is flushed, its contents are written to disk, and secondary index files are created along with the SSTable primary key index.

Since LSM Trees buffer data in memory and indexes have to work for memory-resident contents as well as the disk-resident ones, SASI maintains a separate in-memory structure, indexing memtable contents.

During a read, primary keys of searched records are located by searching and merging index contents, and data records are merged and reconciled similar to how lookups usually work in LSM Trees.

One of the advantages of piggybacking the SSTable life cycle is that indexes can be created during memtable flush or compaction.

Bloom Filters

The source of read amplification in LSM Trees is that we have to address multiple disk-resident tables for the read operation to complete. This happens because we do not always know up front whether or not a disk-resident table contains a data record for the searched key.

One of the ways to prevent table lookup is to store its key range (smallest and largest keys stored in the given table) in metadata, and check if the searched key belongs to the range of that table. This information is imprecise and can only tell us if the data

record *can* be present in the table. To improve this situation, many implementations, including Apache Cassandra and RocksDB, use a data structure called a *Bloom filter*.

Probabilistic data structures are generally more space efficient than their "regular" counterparts. For example, to check set membership, cardinality (find out the number of distinct elements in a set), or frequency (find out how many times a certain element has been encountered), we would have to store all set elements and go through the entire dataset to find the result. Probabilistic structures allow us to store approximate information and perform queries that yield results with an element of uncertainty. Some commonly known examples of such data structures are a Bloom filter (for set membership), HyperLogLog (for cardinality estimation) [FLAJOLET12], and Count-Min Sketch (for frequency estimation) [CORMODE12].

A *Bloom filter*, conceived by Burton Howard Bloom in 1970 [BLOOM70], is a space-efficient probabilistic data structure that can be used to test whether the element is a member of the set or not. It can produce false-positive matches (say that the element is a member of the set, while it is not present there), but cannot produce false negatives (if a negative match is returned, the element is guaranteed not to be a member of the set).

In other words, a Bloom filter can be used to tell if the key *might be in the table* or *is definitely not in the table*. Files for which a Bloom filter returns a negative match are skipped during the query. The rest of the files are accessed to find out if the data record is actually present. Using Bloom filters associated with disk-resident tables helps to significantly reduce the number of tables accessed during a read.

A Bloom filter uses a large bit array and multiple hash functions. Hash functions are applied to keys of the records in the table to find indices in the bit array, bits for which are set to `1`. Bits set to `1` in all positions determined by the hash functions indicate a *presence* of the key in the set. During lookup, when checking for element presence in a Bloom filter, hash functions are calculated for the key again and, if bits determined by *all* hash functions are `1`, we return the positive result stating that item is a member of the set with a certain probability. If at least one of the bits is `0`, we can precisely say that element is not present in the set.

Hash functions applied to different keys can return the same bit position and result in a *hash collision*, and `1` bits only imply that *some* hash function has yielded this bit position for *some* key.

Probability of false positives is managed by configuring the size of the bit set and the number of hash functions: in a larger bit set, there's a smaller chance of collision; sim-

ilarly, having more hash functions, we can check more bits and have a more precise outcome.

The larger bit set occupies more memory, and computing results of more hash functions may have a negative performance impact, so we have to find a reasonable middle ground between acceptable probability and incurred overhead. Probability can be calculated from the expected set size. Since tables in LSM Trees are immutable, set size (number of keys in the table) is known up front.

Let's take a look at a simple example, shown in Figure 7-7. We have a 16-way bit array and 3 hash functions, which yield values `3`, `5`, and `10` for `key1`. We now set bits at these positions. The next key is added and hash functions yield values of `5`, `8`, and `14` for `key2`, for which we set bits, too.

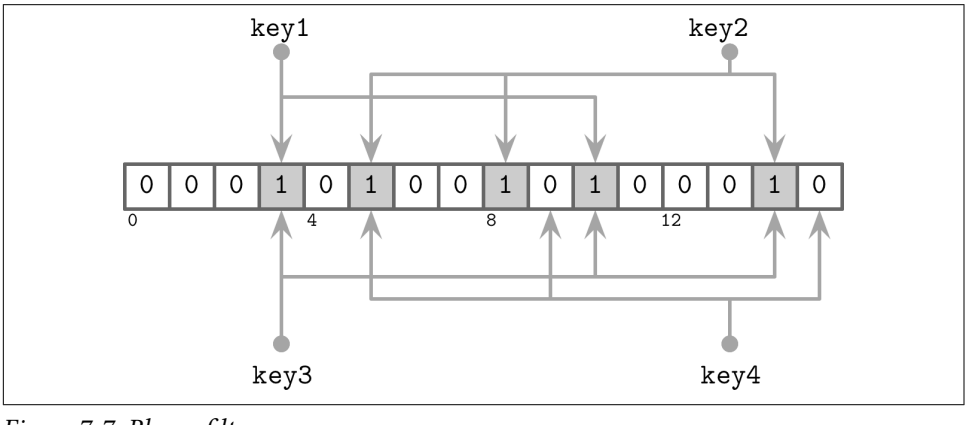

Figure 7-7. Bloom filter

Now, we're trying to check whether or not `key3` is present in the set, and hash functions yield `3`, `10`, and `14`. Since all three bits were set when adding `key1` and `key2`, we have a situation in which the Bloom filter returns a false positive: `key3` was never appended there, yet all of the calculated bits are set. However, since the Bloom filter only claims that element *might* be in the table, this result is acceptable.

If we try to perform a lookup for `key4` and receive values of `5`, `9`, and `15`, we find that only bit `5` is set, and the other two bits are unset. If even one of the bits is unset, we know for sure that the element was never appended to the filter.

Skiplist

There are many different data structures for keeping sorted data in memory, and one that has been getting more popular recently because of its simplicity is called a *skiplist* [PUGH90b]. Implementation-wise, a skiplist is not much more complex than a

singly-linked list, and its probabilistic complexity guarantees are close to those of search trees.

Skiplists do not require rotation or relocation for inserts and updates, and use probabilistic balancing instead. Skiplists are generally less cache-friendly than in-memory B-Trees, since skiplist nodes are small and randomly allocated in memory. Some implementations improve the situation by using unrolled linked lists.

A skiplist consists of a series of nodes of a different *height*, building linked hierarchies allowing to skip ranges of items. Each node holds a key, and, unlike the nodes in a linked list, some nodes have more than just one successor. A node of height `h` is linked *from* one or more predecessor nodes of a height *up to* `h`. Nodes on the lowest level can be linked from nodes of any height.

Node height is determined by a random function and is computed during insert. Nodes that have the same height form a *level*. The number of levels is capped to avoid infinite growth, and a maximum height is chosen based on how many items can be held by the structure. There are exponentially fewer nodes on each next level.

Lookups work by following the node pointers on the highest level. As soon as the search encounters the node that holds a key that is *greater than* the searched one, its predecessor's link to the node on the next level is followed. In other words, if the searched key is *greater than* the current node key, the search continues forward. If the searched key is *smaller than* the current node key, the search continues from the predecessor node on the next level. This process is repeated recursively until the searched key or its predecessor is located.

For example, searching for key 7 in the skiplist shown in Figure 7-8 can be done as follows:

1. Follow the pointer on the highest level, to the node that holds key `10`.
2. Since the searched key `7` is *smaller than* `10`, the next-level pointer from the head node is followed, locating a node holding key `5`.
3. The highest-level pointer on this node is followed, locating the node holding key `10` again.
4. The searched key `7` is *smaller than* `10`, and the next-level pointer from the node holding key `5` is followed, locating a node holding the searched key `7`.

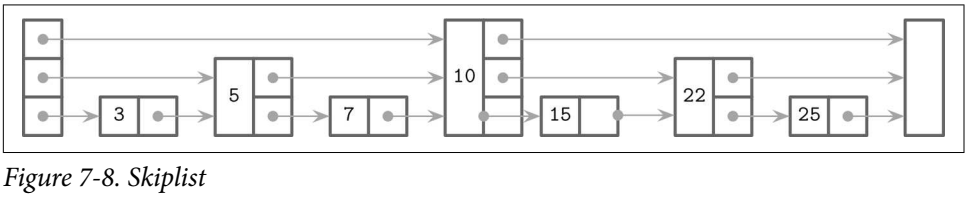

Figure 7-8. Skiplist

During insert, an insertion point (node holding a key or its predecessor) is found using the aforementioned algorithm, and a new node is created. To build a tree-like hierarchy and keep balance, the height of the node is determined using a random number, generated based on a probability distribution. Pointers in predecessor nodes holding keys *smaller than* the key in a newly created node are linked to point to that node. Their higher-level pointers remain intact. Pointers in the newly created node are linked to corresponding successors on each level.

During delete, forward pointers of the removed node are placed to predecessor nodes on corresponding levels.

We can create a concurrent version of a skiplist by implementing a linearizability scheme that uses an additional `fully_linked` flag that determines whether or not the node pointers are fully updated. This flag can be set using compare-and-swap [HERLIHY10]. This is required because the node pointers have to be updated on multiple levels to fully restore the skiplist structure.

In languages with an unmanaged memory model, reference counting or *hazard pointers* can be used to ensure that currently referenced nodes are not freed while they are accessed concurrently [RUSSEL12]. This algorithm is deadlock-free, since nodes are always accessed from higher levels.

Apache Cassandra uses skiplists for the secondary index memtable implementation. WiredTiger uses skiplists for some in-memory operations.

Disk Access

Since most of the table contents are disk-resident, and storage devices generally allow accessing data blockwise, many LSM Tree implementations rely on the page cache for disk accesses and intermediate caching. Many techniques described in "Buffer Management" on page 81, such as page eviction and pinning, still apply to log-structured storage.

The most notable difference is that in-memory contents are immutable and therefore require no additional locks or latches for concurrent access. Reference counting is applied to make sure that currently accessed pages are not evicted from memory, and in-flight requests complete before underlying files are removed during compaction.

Another difference is that data records in LSM Trees are not necessarily page aligned, and pointers can be implemented using absolute offsets rather than page IDs for addressing. In Figure 7-9, you can see records with contents that are not aligned with disk blocks. Some records cross the page boundaries and require loading several pages in memory.

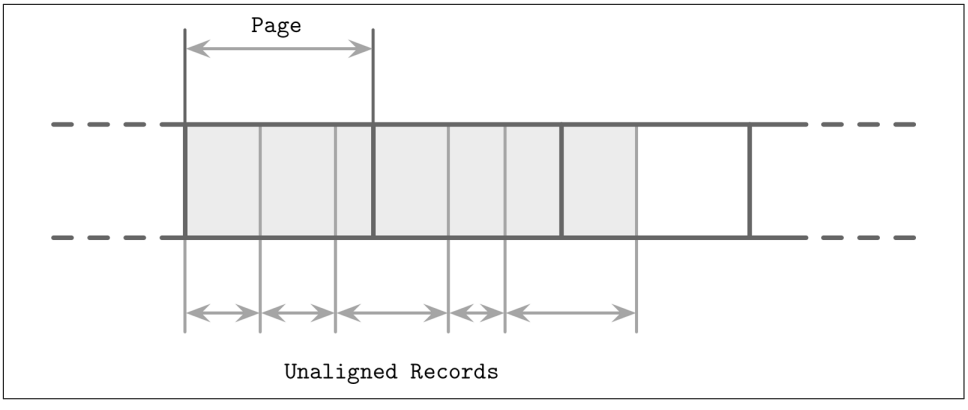

Figure 7-9. Unaligned data records

Compression

We've discussed compression already in context of B-Trees (see "Compression" on page 73). Similar ideas are also applicable to LSM Trees. The main difference here is that LSM Tree tables are immutable, and are generally written in a single pass. When compressing data page-wise, compressed pages are not page aligned, as their sizes are smaller than that of uncompressed ones.

To be able to address compressed pages, we need to keep track of the address boundaries when writing their contents. We could fill compressed pages with zeros, aligning them to the page size, but then we'd lose the benefits of compression.

To make compressed pages addressable, we need an indirection layer which stores offsets and sizes of compressed pages. Figure 7-10 shows the mapping between compressed and uncompressed blocks. Compressed pages are *always* smaller than the originals, since otherwise there's no point in compressing them.

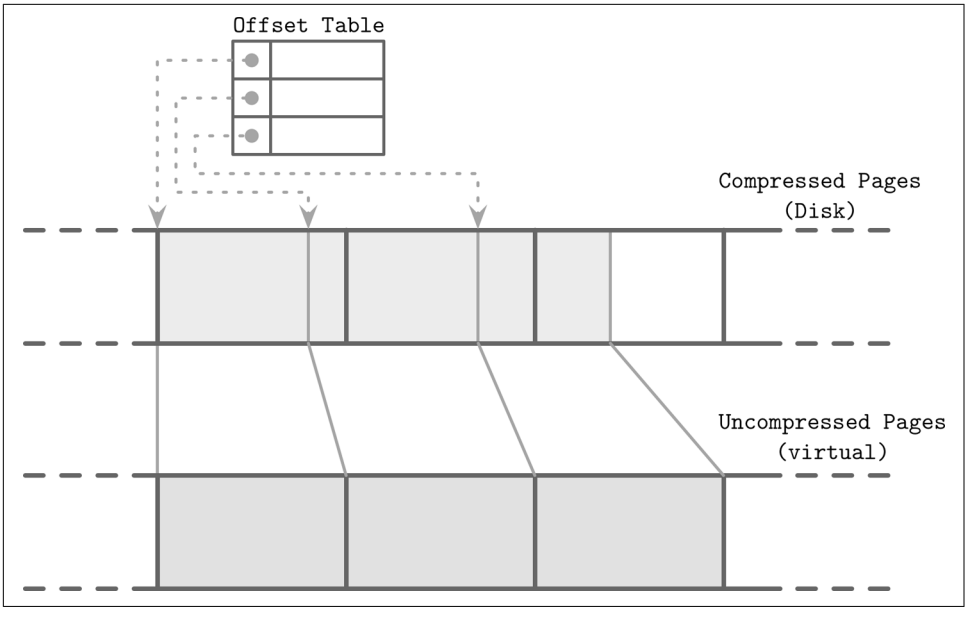

Figure 7-10. Reading compressed blocks. Dotted lines represent pointers from the mapping table to the offsets of compressed pages on disk. Uncompressed pages generally reside in the page cache.

During compaction and flush, compressed pages are appended sequentially, and compression information (the original uncompressed page offset and the actual compressed page offset) is stored in a separate file segment. During the read, the compressed page offset and its size are looked up, and the page can be uncompressed and materialized in memory.

Unordered LSM Storage

Most of the storage structures discussed so far store data *in order*. Mutable and immutable B-Tree pages, sorted runs in FD-Trees, and SSTables in LSM Trees store data records in key order. The order in these structures is preserved differently: B-Tree pages are updated in place, FD-Tree runs are created by merging contents of two runs, and SSTables are created by buffering and sorting data records in memory.

In this section, we discuss structures that store records in random order. Unordered stores generally do not require a separate log and allow us to reduce the cost of writes by storing data records in insertion order.

Bitcask

Bitcask, one of the storage engines used in Riak, is an unordered log-structured storage engine [SHEEHY10b]. Unlike the log-structured storage implementations discussed so far, it *does not* use memtables for buffering, and stores data records directly in logfiles.

To make values searchable, Bitcask uses a data structure called *keydir*, which holds references to the *latest* data records for the corresponding keys. Old data records may still be present on disk, but are not referenced from keydir, and are garbage-collected during compaction. Keydir is implemented as an in-memory hashmap and has to be rebuilt from the logfiles during startup.

During a *write*, a key and a data record are appended to the logfile sequentially, and the pointer to the newly written data record location is placed in keydir.

Reads check the keydir to locate the searched key and follow the associated pointer to the logfile, locating the data record. Since at any given moment there can be only one value associated with the key in the keydir, point queries do not have to merge data from multiple sources.

Figure 7-11 shows mapping between the keys and records in data files in Bitcask. Logfiles hold data records, and keydir points to the latest *live* data record associated with each key. Shadowed records in data files (ones that were superseded by later writes or deletes) are shown in gray.

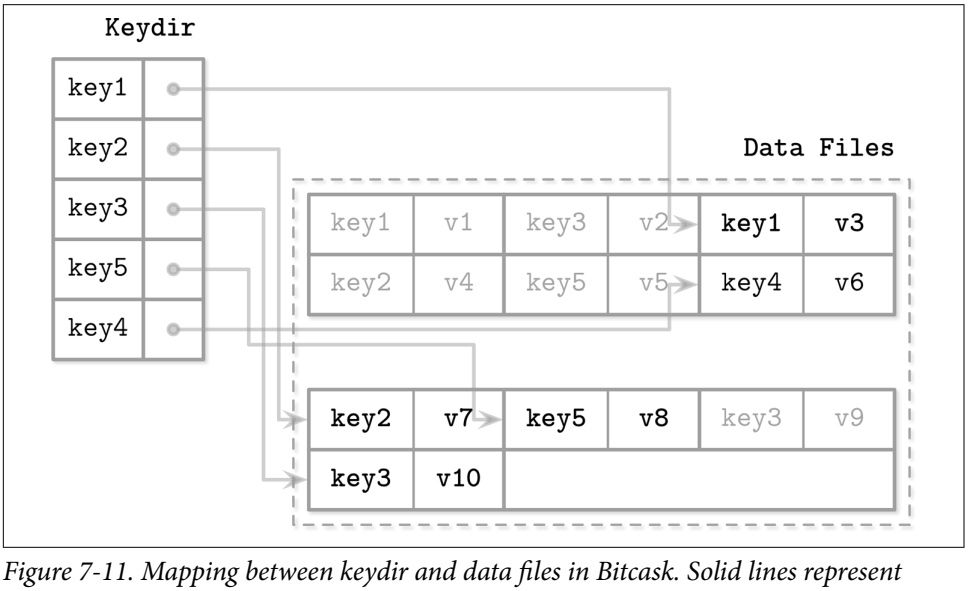

Figure 7-11. Mapping between keydir and data files in Bitcask. Solid lines represent pointers from the key to the latest value associated with it. Shadowed key/value pairs are shown in light gray.

During compaction, contents of all logfiles are read sequentially, merged, and written to a new location, preserving only *live* data records and discarding the shadowed ones. Keydir is updated with new pointers to relocated data records.

Data records are stored directly in logfiles, so a separate write-ahead log doesn't have to be maintained, which reduces both space overhead and write amplification. A downside of this approach is that it offers only point queries and doesn't allow range scans, since items are unordered both in keydir and in data files.

Advantages of this approach are simplicity and great point query performance. Even though multiple versions of data records exist, only the latest one is addressed by keydir. However, having to keep all keys in memory and rebuilding keydir on startup are limitations that might be a deal breaker for some use cases. While this approach is great for point queries, it does not offer any support for range queries.

WiscKey

Range queries are important for many applications, and it would be great to have a storage structure that could have the write and space advantages of unordered storage, while still allowing us to perform range scans.

WiscKey [LU16] decouples sorting from garbage collection by keeping the keys sorted in LSM Trees, and keeping data records in unordered append-only files called *vLogs* (value logs). This approach can solve two problems mentioned while discussing Bitcask: a need to keep all keys in memory and to rebuild a hashtable on startup.

Figure 7-12 shows key components of WiscKey, and mapping between keys and log files. vLog files hold unordered data records. Keys are stored in sorted LSM Trees, pointing to the latest data records in the logfiles.

Since keys are typically much smaller than the data records associated with them, compacting them is significantly more efficient. This approach can be particularly useful for use cases with a low rate of updates and deletes, where garbage collection won't free up as much disk space.

The main challenge here is that because vLog data is unsorted, range scans require random I/O. WiscKey uses internal SSD parallelism to prefetch blocks in parallel during range scans and reduce random I/O costs. In terms of block transfers, the costs are still high: to fetch a single data record during the range scan, the entire page where it is located has to be read.

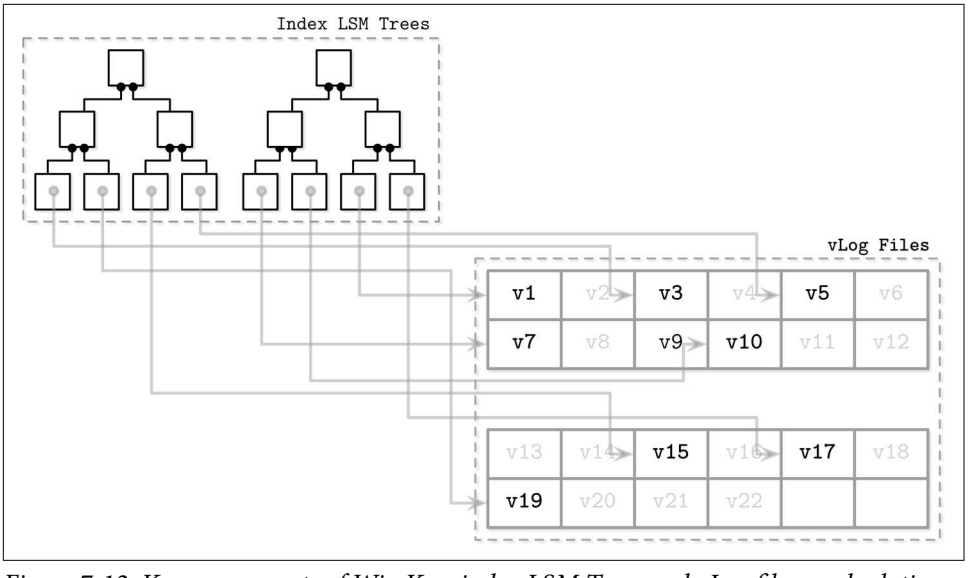

Figure 7-12. Key components of WiscKey: index LSM Trees and vLog files, and relationships between them. Shadowed records in data files (ones that were superseded by later writes or deletes) are shown in gray. Solid lines represent pointers from the key in the LSM tree to the latest value in the log file.

During compaction, vLog file contents are read sequentially, merged, and written to a new location. Pointers (values in a key LSM Tree) are updated to point to these new locations. To avoid scanning entire vLog contents, WiscKey uses `head` and `tail` pointers, holding information about vLog segments that hold live keys.

Since data in vLog is unsorted and contains no liveness information, the key tree has to be scanned to find which values are still live. Performing these checks during garbage collection introduces additional complexity: traditional LSM Trees can resolve file contents during compaction without addressing the key index.

Concurrency in LSM Trees

The main concurrency challenges in LSM Trees are related to switching *table views* (collections of memory- and disk-resident tables that change during flush and compaction) and log synchronization. Memtables are also generally accessed concurrently (except core-partitioned stores such as ScyllaDB), but concurrent in-memory data structures are out of the scope of this book.

During flush, the following rules have to be followed:

- The new memtable has to become available for reads and writes.
- The old (flushing) memtable has to remain visible for reads.
- The flushing memtable has to be written on disk.
- Discarding a flushed memtable and making a flushed disk-resident table have to be performed as an atomic operation.
- The write-ahead log segment, holding log entries of operations applied to the flushed memtable, has to be discarded.

For example, Apache Cassandra solves these problems by using operation order barriers: all operations that were accepted for write will be waited upon prior to the memtable flush. This way the flush process (serving as a consumer) knows which other processes (acting as producers) depend on it.

More generally, we have the following synchronization points:

Memtable switch
: After this, all writes go only to the new memtable, making it primary, while the old one is still available for reads.

Flush finalization
: Replaces the old memtable with a flushed disk-resident table in the table view.

Write-ahead log truncation
: Discards a log segment holding records associated with a flushed memtable.

These operations have severe correctness implications. Continuing writes to the old memtable might result in data loss; for example, if the write is made into a memtable section that was already flushed. Similarly, failing to leave the old memtable available for reads until its disk-resident counterpart is ready will result in incomplete results.

During compaction, the table view is also changed, but here the process is slightly more straightforward: old disk-resident tables are discarded, and the compacted version is added instead. Old tables have to remain accessible for reads until the new one is fully written and is ready to replace them for reads. Situations in which the same tables participate in multiple compactions running in parallel have to be avoided as well.

In B-Trees, log truncation has to be coordinated with flushing dirty pages from the page cache to guarantee durability. In LSM Trees, we have a similar requirement: writes are buffered in a memtable, and their contents are not durable until fully flushed, so log truncation has to be coordinated with memtable flushes. As soon as the flush is complete, the log manager is given the information about the latest flushed log segment, and its contents can be safely discarded.

Not synchronizing log truncations with flushes will also result in data loss: if a log segment is discarded before the flush is complete, and the node crashes, log contents will not be replayed, and data from this segment won't be restored.

Log Stacking

Many modern filesystems are log structured: they buffer writes in a memory segment and flush its contents on disk when it becomes full in an append-only manner. SSDs use log-structured storage, too, to deal with small random writes, minimize write overhead, improve wear leveling, and increase device lifetime.

Log-structured storage (LSS) systems started gaining popularity around the time SSDs were becoming more affordable. LSM Trees and SSDs are a good match, since sequential workloads and append-only writes help to reduce amplification from in-place updates, which negatively affect performance on SSDs.

If we stack multiple log-structured systems on top each other, we can run into several problems that we were trying to solve using LSS, including write amplification, fragmentation, and poor performance. At the very least, we need to keep the SSD flash translation layer and the filesystem in mind when developing our applications [YANG14].

Flash Translation Layer

Using a log-structuring mapping layer in SSDs is motivated by two factors: small random writes have to be batched together in a physical page, and the fact that SSDs work by using program/erase cycles. Writes can be done only into previously *erased* pages. This means that a page cannot be *programmed* (in other words, written) unless it is empty (in other words, was *erased*).

A *single* page cannot be erased, and only *groups* of pages in a *block* (typically holding 64 to 512 pages) can be erased together. Figure 7-13 shows a schematic representation of pages, grouped into blocks. The flash translation layer (FTL) translates logical page addresses to their physical locations and keeps track of page states (live, discarded, or empty). When FTL runs out of free pages, it has to perform garbage collection and erase discarded pages.

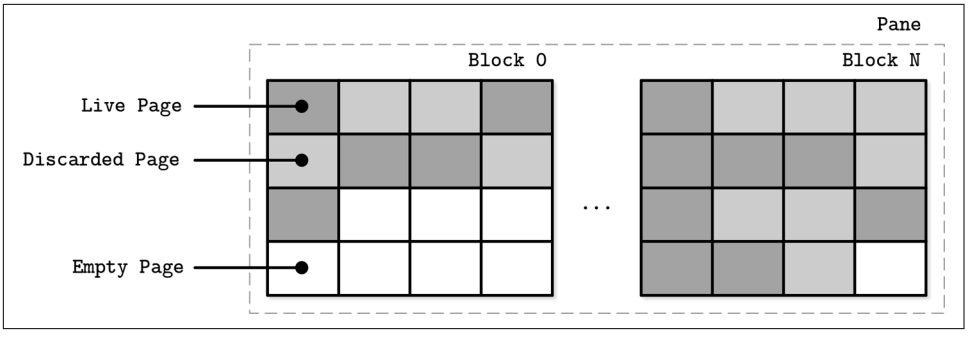

Figure 7-13. SSD pages, grouped into blocks

There are no guarantees that all pages in the block that is about to be erased are discarded. Before the block can be erased, FTL has to relocate its *live* pages to one of the blocks containing empty pages. Figure 7-14 shows the process of moving live pages from one block to new locations.

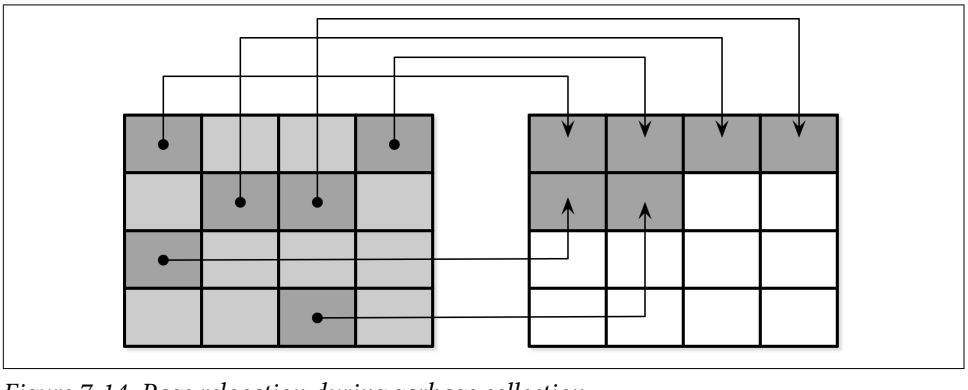

Figure 7-14. Page relocation during garbage collection

When all live pages are relocated, the block can be safely erased, and its empty pages become available for writes. Since FTL is aware of page states and state transitions and has all the necessary information, it is also responsible for SSD *wear leveling*.

Wear leveling distributes the load evenly across the medium, avoiding hotspots, where blocks fail prematurely because of a high number of program-erase cycles. It is required, since flash memory cells can go through only a limited number of program-erase cycles, and using memory cells evenly helps to extend the lifetime of the device.

In summary, the motivation for using log-structured storage on SSDs is to amortize I/O costs by batching small random writes together, which generally results in a

smaller number of operations and, subsequently, reduces the number of times the garbage collection is triggered.

Filesystem Logging

On top of that, we get filesystems, many of which also use logging techniques for write buffering to reduce write amplification and use the underlying hardware optimally.

Log stacking manifests in a few different ways. First, each layer has to perform its own bookkeeping, and most often the underlying log does not expose the information necessary to avoid duplicating the efforts.

Figure 7-15 shows a mapping between a higher-level log (for example, the application) and a lower-level log (for example, the filesystem) resulting in redundant logging and different garbage collection patterns [YANG14]. Misaligned segment writes can make the situation even worse, since discarding a higher-level log segment may cause fragmentation and relocation of the neighboring segments' parts.

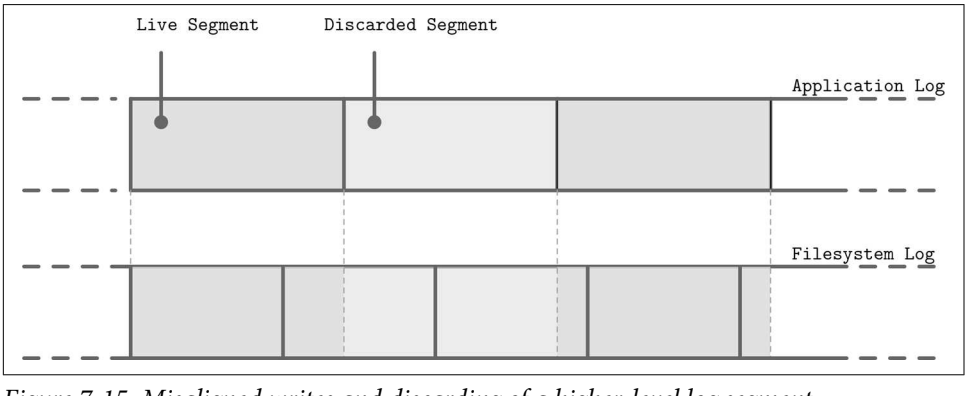

Figure 7-15. Misaligned writes and discarding of a higher-level log segment

Because layers do not communicate LSS-related scheduling (for example, discarding or relocating segments), lower-level subsystems might perform redundant operations on discarded data or the data that is about to be discarded. Similarly, because there's no single, standard segment size, it may happen that unaligned higher-level segments occupy multiple lower-level segments. All these overheads can be reduced or completely avoided.

Even though we say that log-structured storage is all about sequential I/O, we have to keep in mind that database systems may have multiple write streams (for example, log writes parallel to data record writes) [YANG14]. When considered on a hardware level, interleaved sequential write streams may not translate into the same sequential pattern: blocks are not necessarily going to be placed in write order. Figure 7-16

shows multiple streams overlapping in time, writing records that have sizes not aligned with the underlying hardware page size.

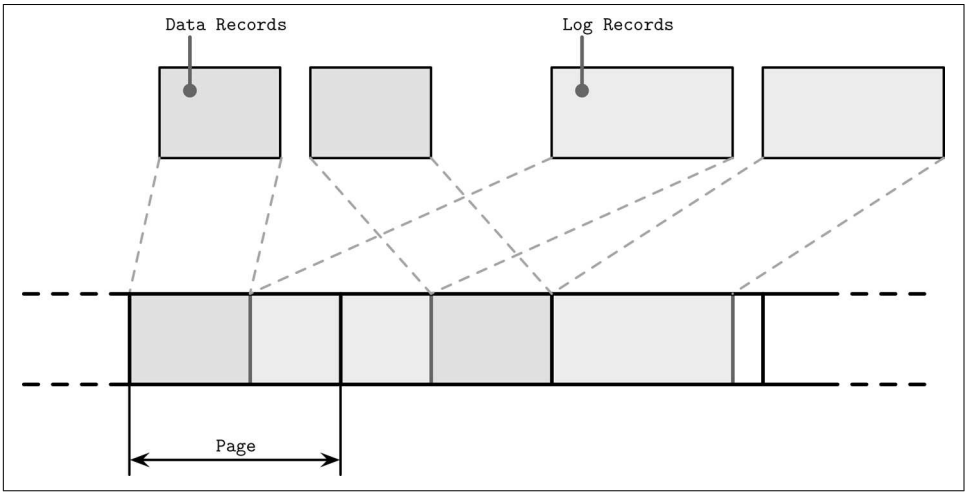

Figure 7-16. Unaligned multistream writes

This results in fragmentation that we tried to avoid. To reduce interleaving, some database vendors recommend keeping the log on a separate device to isolate workloads and be able to reason about their performance and access patterns independently. However, it is more important to keep partitions aligned to the underlying hardware [INTEL14] and keep writes aligned to page size [KIM12].

LLAMA and Mindful Stacking

> Well, you'll never believe this, but that llama you're looking at was once a human being. And not just any human being. That guy was an emperor. A rich, powerful ball of charisma.
>
> —Kuzco from *The Emperor's New Groove*

In "Bw-Trees" on page 120, we discussed an immutable B-Tree version called Bw-Tree. Bw-Tree is layered on top of a *latch-free, log-structured, access-method aware* (LLAMA) storage subsystem. This layering allows Bw-Trees to grow and shrink dynamically, while leaving garbage collection and page management transparent for the tree. Here, we're most interested in the *access-method aware* part, demonstrating the benefits of coordination between the software layers.

To recap, a *logical* Bw-Tree node consists of a linked list of *physical* delta nodes, a chain of updates from the newest one to the oldest one, ending in a base node. Logical nodes are linked using an in-memory mapping table, pointing to the location

of the latest update on disk. Keys and values are added to and removed from the logical nodes, but their physical representations remain immutable.

Log-structured storage buffers node updates (delta nodes) together in 4 Mb flush buffers. As soon as the page fills up, it's flushed on disk. Periodically, garbage collection reclaims space occupied by the unused delta and base nodes, and relocates the live ones to free up fragmented pages.

Without access-method awareness, interleaved delta nodes that belong to different logical nodes will be written in their insertion order. Bw-Tree awareness in LLAMA allows for the consolidation of several delta nodes into a single contiguous physical location. If two updates in delta nodes *cancel* each other (for example, an insert followed by delete), their *logical* consolidation can be performed as well, and only the latter delete can be persisted.

LSS garbage collection can also take care of consolidating the logical Bw-Tree node contents. This means that garbage collection will not only reclaim the free space, but also significantly reduce the physical node fragmentation. If garbage collection only rewrote several delta nodes contiguously, they would still take the same amount of space, and readers would need to perform the work of applying the delta updates to the base node. At the same time, if a higher-level system consolidated the nodes and wrote them contiguously to the new locations, LSS would *still* have to garbage-collect the old versions.

By being aware of Bw-Tree semantics, several deltas may be rewritten as a single base node with all deltas already applied *during* garbage collection. This reduces the total space used to represent this Bw-Tree node and the latency required to read the page while reclaiming the space occupied by discarded pages.

You can see that, when considered carefully, stacking can yield many benefits. It is not necessary to always build tightly coupled single-level structures. Good APIs and exposing the right information can significantly improve efficiency.

Open-Channel SSDs

An alternative to stacking software layers is to skip all indirection layers and use the hardware directly. For example, it is possible to avoid using a filesystem and flash translation layer by developing for Open-Channel SSDs. This way, we can avoid at least two layers of logs and have more control over wear-leveling, garbage collection, data placement, and scheduling. One of the implementations that uses this approach is LOCS (LSM Tree-based KV Store on Open-Channel SSD) [ZHANG13]. Another example using Open-Channel SSDs is LightNVM, implemented in the Linux kernel [BJØRLING17].

The flash translation layer usually handles data placement, garbage collection, and page relocation. Open-Channel SSDs expose their internals, drive management, and

I/O scheduling without needing to go through the FTL. While this certainly requires much more attention to detail from the developer's perspective, this approach may yield significant performance improvements. You can draw a parallel with using the `O_DIRECT` flag to bypass the kernel page cache, which gives better control, but requires manual page management.

Software Defined Flash (SDF) [OUYANG14], a hardware/software codesigned Open-Channel SSDs system, exposes an asymmetric I/O interface that takes SSD specifics into consideration. Sizes of read and write units are different, and write unit size corresponds to erase unit size (block), which greatly reduces write amplification. This setting is ideal for log-structured storage, since there's only one software layer that performs garbage collection and relocates pages. Additionally, developers have access to internal SSD parallelism, since every channel in SDF is exposed as a separate block device, which can be used to further improve performance.

Hiding complexity behind a simple API might sound compelling, but can cause complications in cases in which software layers have different semantics. Exposing *some* underlying system internals may be beneficial for better integration.

Summary

Log-structured storage is used everywhere: from the flash translation layer, to filesystems and database systems. It helps to reduce write amplification by batching small random writes together in memory. To reclaim space occupied by removed segments, LSS periodically triggers garbage collection.

LSM Trees take some ideas from LSS and help to build index structures managed in a log-structured manner: writes are batched in memory and flushed on disk; shadowed data records are cleaned up during compaction.

It is important to remember that many software layers use LSS, and make sure that layers are stacked optimally. Alternatively, we can skip the filesystem level altogether and access hardware directly.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Overview

Luo, Chen, and Michael J. Carey. 2019. "LSM-based Storage Techniques: A Survey." *The VLDB Journal* https://doi.org/10.1007/s00778-019-00555-y.

LSM Trees

O'Neil, Patrick, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. "The log-structured merge-tree (LSM-tree)." *Acta Informatica* 33, no. 4: 351-385. https://doi.org/10.1007/s002360050048.

Bitcask

Justin Sheehy, David Smith. "Bitcask: A Log-Structured Hash Table for Fast Key/Value Data." 2010.

WiscKey

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. "WiscKey: Separating Keys from Values in SSD-Conscious Storage." ACM Trans. Storage 13, 1, Article 5 (March 2017), 28 pages.

LOCS

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD." In Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14). ACM, New York, NY, USA, Article 16, 14 pages.

LLAMA

Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. "LLAMA: a cache/storage subsystem for modern hardware." Proc. VLDB Endow. 6, 10 (August 2013), 877-888.