Chapter 3 Storage and Retrieval

Two families of storage engines: log-structured storage engines, and page-oriented storage engines such as B-trees.

Data Structures That Power Your Database

- Many databases internally use a log, which is an append-only data file. Real databases have more issues to deal with (concurrency control, reclaiming disk space so that the log doesn't grow forever, and handling errors and partially written records).
- An index is an additional structure that is derived from the primary data. Many databases allow you to add and remove indexes, and this doesn't affect the contents of the database; it only affects the performance of queries.
- This is an important trade-off in storage systems: well-chosen indexes speed up read queries, but every index slows down writes.
  Databases don't usually index everything by default, it requires the application developer or database administrator to choose indexes manually, using application's query patterns.

Hash Indexes

- Key-value stores are similar to dictionaries in programming languages, implemented as a hash map. We have hash maps for in-memory data structures, we can use them to index data on disk.
- **If data storage is appending to a file, the indexing strategy is to keep an in-memory hash map, every key is mapped to a byte offset in the data file, where value can be found.**
  Insert and update: Whenever append a new key-value pair to the file, update the hash map to reflect the offset of the data just written.
  Look up a value: Use the hash map to find the offset in the data file, seek to that location, and read the value.

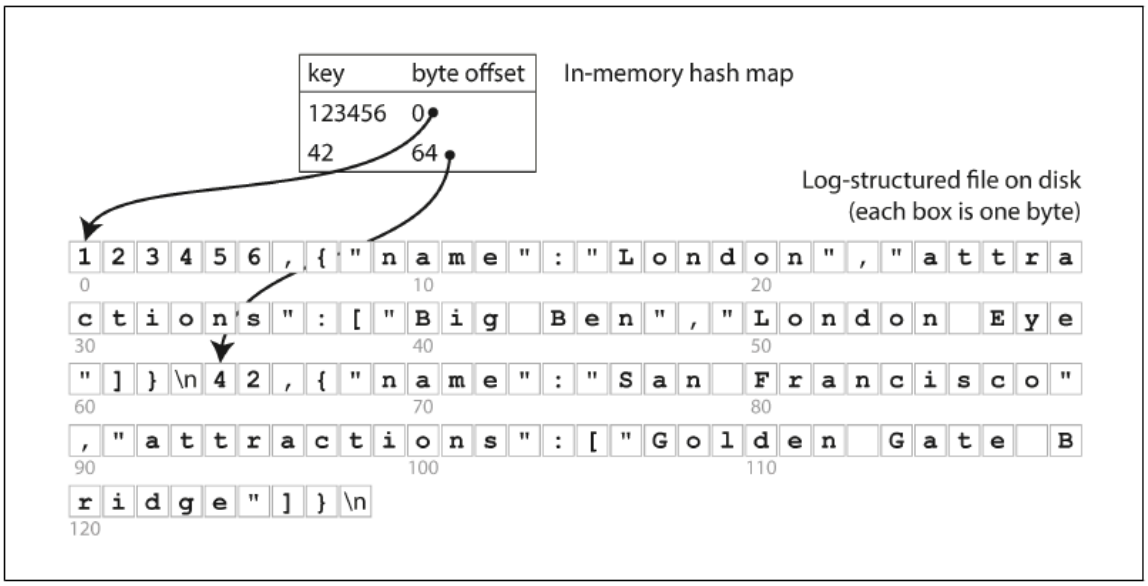

Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

- Bitcask offers high-performance reads and writes, all keys fit in RAM, since hash map is completely in memory. Values can be loaded from disk with just one disk seek. If that part of the data file is already in the filesystem cache, it doesn't require disk I/O.
- A storage engine like Bitcask is well suited to frequently updated value for each key. E.g, the key is the URL of a video, value is the number of times it is played. In this kind of workload, there are many writes per key, but not many distinct keys. It's feasible to keep all keys in memory.
- **While appending to a file, how to avoid running out of disk space?**
  Solution: break the log into segments of a certain size, close a segment file when it reaches a certain size, make writes to a new segment file. Then perform compaction on these segments.
  Compaction means throwing away duplicate keys in the log, and keeping the most recent update for each key.

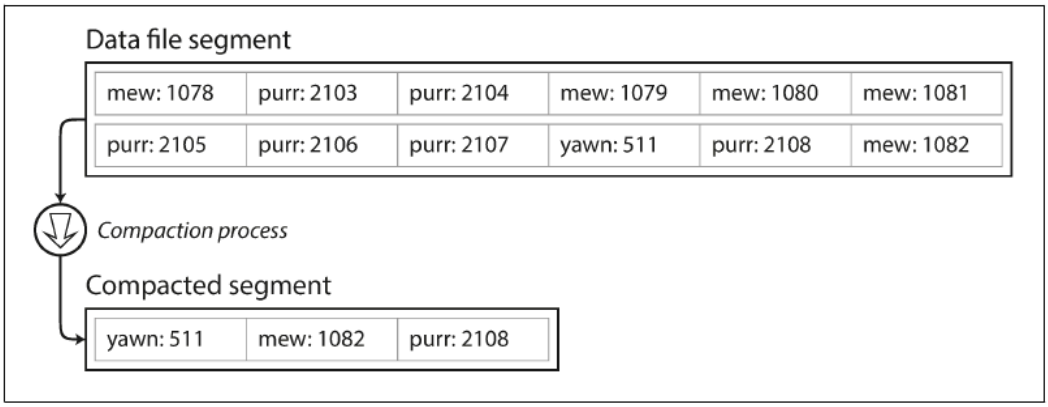

Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

- **We can merge several segments when performing compaction.**
  Segments are never modified after being written, the merged segment is written to a new file.
  Merging and compaction of frozen segments in background thread, continue to serve read and write requests using old segment files.
  After the merging completes, switch read requests to use the new merged segment, then old segment files can be deleted.

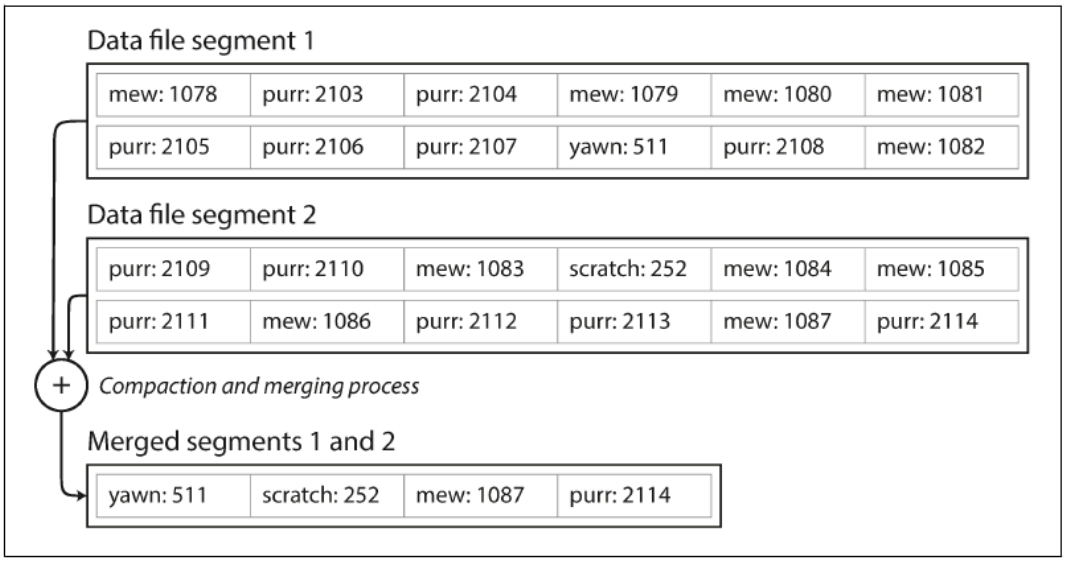

Figure 3-3. Performing compaction and segment merging simultaneously.

- Each segment has an in-memory hash table, mapping keys to file offsets.
  To find the value for a key, first check the most recent segment's hash map; if the key is not present we check the second-most-recent segment, and so on.
- **The merging process keeps the number of segments small, lookups need to check less hash maps.**

Issues in implementation:

- **File format**
  CSV is not the format for a log. **Binary format** is faster and simpler, first encodes the length of a string in bytes, followed by the raw string (no need for escaping).
- **Deleting records**
  To delete a key-value pair, **append a deletion record to the data file** (called a tombstone). When merging log segments, the tombstone tells the merging process to discard previous values for the deleted key.
- **Crash recovery**
  If the database is restarted, the in-memory hash maps are lost. Bitcask speeds up recovery by storing a snapshot of each segment's hashmap on disk, which can be loaded into memory more quickly.
- **Partially written records**
  The database may crash. Bitcask files include **checksums**, allowing corrupted parts of the log to be detected and ignored.
- **Concurrency control**

As writes are appended to log in a sequential order, have **one writer thread**.
Data file segments are append-only and immutable, and can be **read** concurrently by **multiple threads**.

Advantages of append-only design:

- Appending and segment merging are sequential write operations, much faster than random writes, both on hard drives and SSDs.
- Concurrency and crash recovery are much simpler if segment files are append-only or immutable.
- Merging old segments avoids the problem of data files getting fragmented over time.

Limitations of hash table index:

- **The hash table must fit in memory**. For a large number of keys, maintain a hashmap on disk, but an on-disk hash map performs badly. It requires many random access I/O, it is expensive to grow when it becomes full, and hash collisions require fiddly logic.
- **Range queries are not efficient**. Range queries need to look up each key individually in the hash maps.

SSTables and LSM-Trees

- **Sorted String Table -- SSTable**
  Change of format in segment files, make the sequence of key-value pairs sorted by key. Each key appears once within each merged segment file (the compaction process ensures this).

Advantages over log segments with hash indexes:

1. Merging segments is simple and efficient, even if the files are bigger than memory. Like a merge sort algorithm.

Read the input files side by side, look at the first key in each file, copy the lowest key (according to sort order) to the output file and repeat. Produces a new merged segment file, sorted by key.
When multiple segments contain the same key, keep the most recent segment and discard the values in older segments.

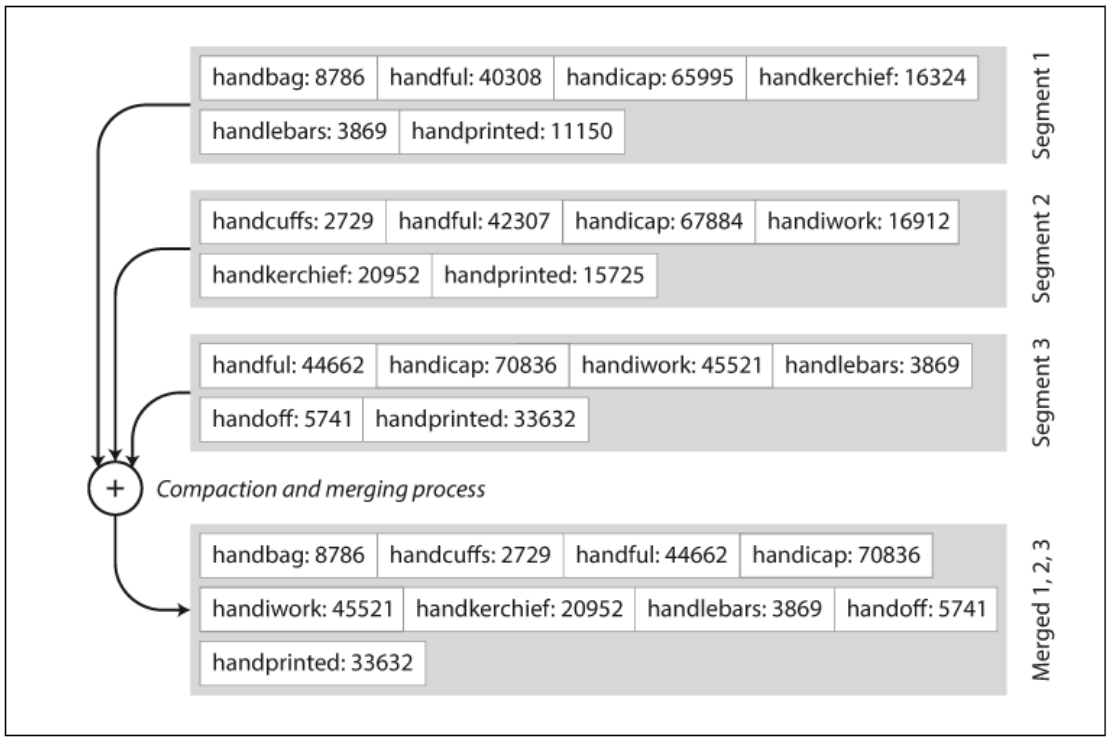

Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

2. To find a key in the file, no need to keep an index of all the keys in memory.
Use an **in-memory index to the offsets for some keys**, but it can be **sparse**: **one key for every few kilobytes of segment file**, because a few kilobytes can be scanned very quickly.

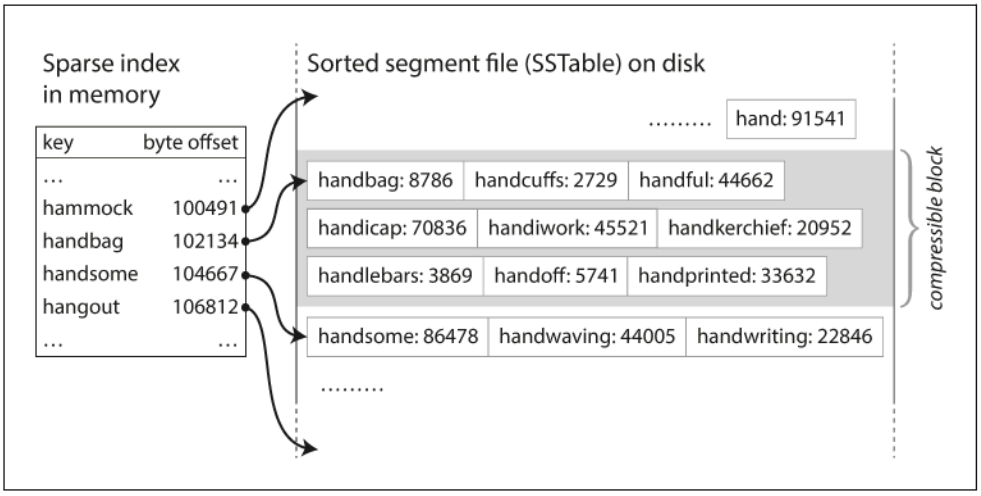

Figure 3-5. An SSTable with an in-memory index.

3. Read requests scan several key-value pairs in requested range, group records into a block and compress it before writing it to disk (shaded area in Figure 3-5). Each entry of the sparse in-memory index points at the start of a compressed block.
Compression saving disk space, reduces I/O bandwidth.

Constructing and maintaining SSTables

- Maintaining a sorted structure in memory is easier than maintaining it on disk.
  Use tree data structures, like red-black trees or AVL trees, insert keys in any order and read them back in sorted order.

Storage engine work as follows:

- When writing, add it to in-memory balanced tree data structure (red-black tree). The in-memory tree is called a **memtable**.
- When the memtable is bigger than a threshold(a few megabytes), write it out to disk as an SSTable file. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.
- To serve a read request, first find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

Problem and solution:

- If the database crashes, the most recent writes (in the memtable) are lost.
- To avoid the problem, keep a separate log on disk when write is appended.
  The log is not sorted, its purpose is to restore the memtable after a crash. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

Making an LSM-tree out of SSTables

- **Key-value storage engines: Cassandra and HBase**. Both were inspired by Google's Bigtable paper(introduced the terms SSTable and memtable).

- **Log-Structured Merge-Tree (or LSM-Tree):**
  Originally from **log-structured filesystems**. Storage engines based on this **principle of merging and compacting sorted files** are called **LSM storage engines**.

- Lucene, an **indexing engine for full-text search** used by Elasticsearch and Solr, uses a similar method for storing its **term dictionary**.
  A full-text index is more complex than a key-value index but is based on a similar idea: given a word in a search query, find all documents mentioning the word.
  Implemented with a key-value structure, the key is a word (a term) and the value is the list of IDs of all the documents that contain the word (the postings list).
  In Lucene, this mapping from term to postings list is kept in SSTable-like sorted files, which are merged in the background as needed.

Performance optimizations

- **If a key does not exist in DB, LSM-tree can be slow. Because first check memtable, then all the segments from disk. Can optimize this with additional Bloom filters.**

- **Bloom filter: a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.**
- Various strategies to determine the order and timing of how SSTables are compacted and merged. The most common options are **size-tiered** and **leveled compaction**. HBase uses size-tiered, Cassandra supports both.
- In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows the compaction to proceed more incrementally and use less disk space.
- Overall, the basic idea of LSM-trees is simple and effective – keeping a cascade of SSTables that are merged in the background:
    - Works well for big datasets much bigger than available memory
    - Perform range queries
    - High write throughput

B-Trees

The most widely used indexing structure is B-tree.
Similarity with SSTables: key-value pairs sorted by key, efficient key/value lookups and range queries.
Log-structured indexes: break the database down into variable-size segments, few megabytes in size, and always write a segment sequentially.
B-trees break the database down into fixed-size blocks or pages, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time. This design is more close to the hardware, as disks are also arranged in fixed-size blocks.
Each page is identified using an address or location, which allows one page to refer to another, like a pointer on disk instead of in memory. We can use page references to construct a tree of pages.

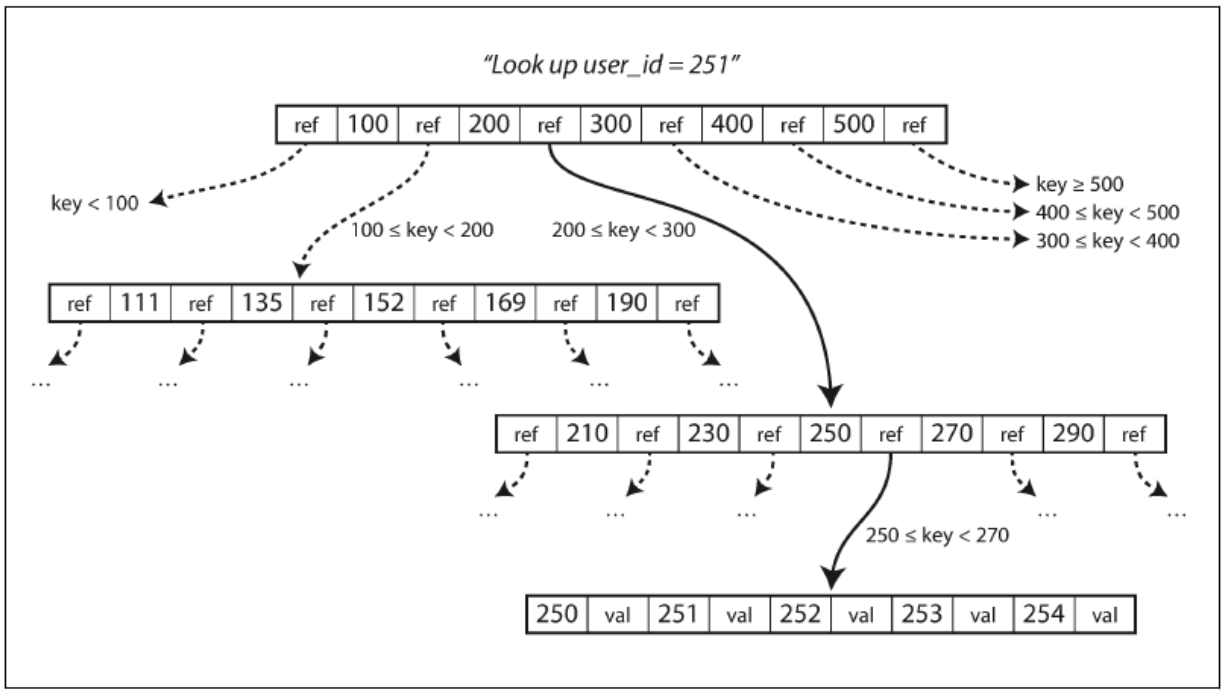

Figure 3-6. Looking up a key using a B-tree index.

Root of B-tree: One designated page to start with when looking up a key in the index. It contains several keys and references to child pages.
Each child is responsible for a continuous range of keys, and the keys between the references indicate the boundaries between those ranges.
Leaf page: contains the value for the key or contains references to pages of values.
Branching factor: The number of references to child pages in one page of the B-tree.

Updates:

- If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, change the value in that page, and write the page back to disk (any references to that page remain valid).
- If you want to add a new key, you need to find the page whose range encompasses the new key and add it to that page.
- If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.

This algorithm ensures the tree remains **balanced**: a B-tree with n keys always has a depth of O(log n). Most databases can fit into a B-tree that is three or four levels deep, so you don't need to follow many page references to find the page you are looking for. (A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB.)
4KB * 500^4 == 250 * 10^12
KB 10^3
MB 10^6
GB 10^9

TB 10^12
PB 10^15

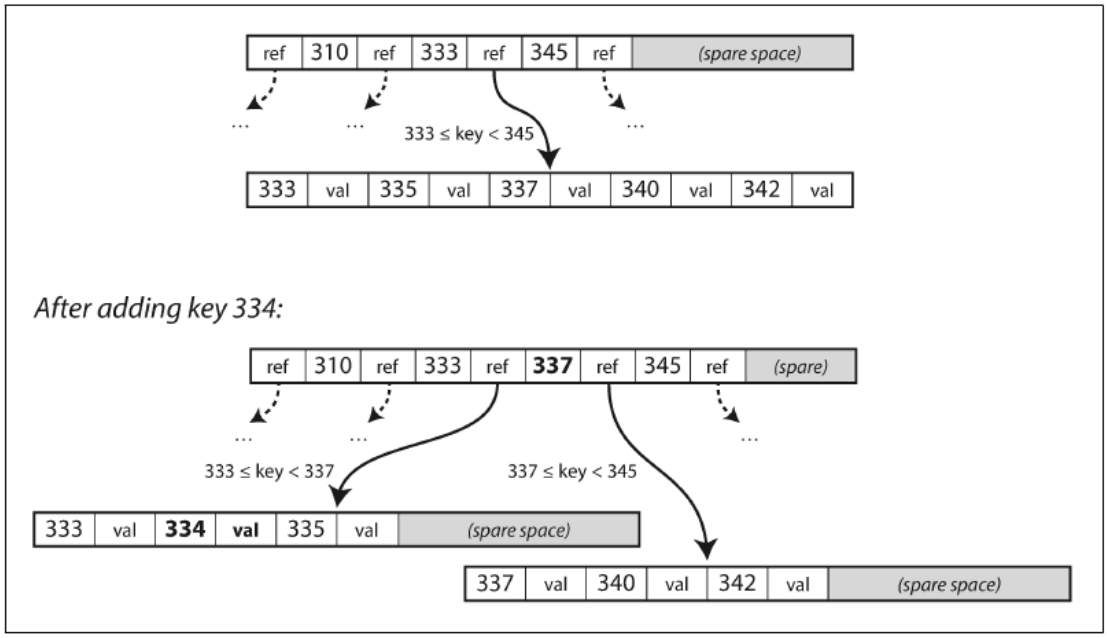

Figure 3-7. Growing a B-tree by splitting a page.

Making B-trees reliable

- Make the database resilient to crashes: **Write-ahead log / WAL / redo log**: an append-only file, every B-tree modification must be written before it can be applied to the pages of the tree itself. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state.
- When updating pages in place, Concurrency control is required if multiple threads access the B-tree at the same time. Typically protect tree's data structures with latches (lightweight locks).
- No need for Logstructured approaches since they do merging in the background without interfering with incoming queries and atomically swap old segments for new segments from time to time.

B-tree optimizations

- Instead of overwriting pages and maintaining a WAL for crash recovery, some databases use a copy-on-write scheme. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. This is also useful for concurrency control("Snapshot Isolation and Repeatable Read").
- Abbreviated keys, only provide boundaries/key ranges. Packing more keys into a page allows a higher branching factor, and fewer levels.
- Pages can be positioned anywhere on disk. If a query scan over a large part of the key range in sorted order, that page-by-page layout can be inefficient. Many implementations

try to lay out the tree so that leaf pages appear in sequential order on disk. However, it's difficult to maintain that order as the tree grows. By contrast, it's easier for LSM-trees to keep sequential keys close to each other on disk since they rewrite large segments of the storage in one go during merging.
- Additional pointers added. E.g, each leaf page has references to its sibling pages to the left and right, thus allows scanning keys in order without jumping back to parent pages.
- B-tree variants such as fractal trees borrow some log-structured ideas to reduce disk seeks.

Comparing B-Trees and LSM-Trees

- **LSM-trees**: **faster for writes** (Reads are slower because they have to check several different data structures and SSTables at different stages of compaction).
- **B-trees: faster for reads**.

Advantages of LSM-trees

- B-tree index must write data at least twice: tree page and write-ahead log. Maybe more when page split, or avoid partially updated page in power failure.
  Overhead when writing an entire page at a time, even if only a few bytes in that page changed.
- Log-structured indexes also rewrite data multiple times due to repeated compaction and merging of SSTables.
- Write amplification: one write to the database resulting in multiple writes to the disk over the course of the database's lifetime. Particularly on SSDs which can only overwrite blocks a limited number of times before wearing out.
- In write-heavy applications, write amplification has performance cost: the more that a storage engine writes to disk, the fewer writes per second it can handle within the available disk bandwidth.
- LSM-trees can handle higher write throughput than B-trees, since they might have lower write amplification, and they sequentially write compact SSTable files rather than overwrite several pages in the tree(random writes).
- LSM-trees can be compressed better, and often produce smaller files on disk than B-trees. B-tree storage engines leave some disk space unused due to fragmentation: when a page is split or when a row cannot fit into an existing page, some space in a page remains unused.
- LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, have lower storage overheads, especially when leveled compaction.

Downsides of LSM-trees

- The compaction process can impact the performance of ongoing reads and writes. Throughput and average response time is OK, but at higher percentiles the response time can be unpredictably high.

- Issues during high write throughput: the disk write bandwidth needs to be shared between the initial write (logging and flushing memtable to disk) and the compaction threads running in the background.
- If compaction cannot keep up with the rate of incoming writes, the number of unmerged segments on disk keeps growing until run out of disk space. Reads slow down because they need to check more segment files.
- An advantage of B-trees is each key exists in exactly one place in the index, but log-structured storage engines have multiple copies of the same key in different segments. B-trees offer strong transactional semantics: in relational databases, transaction isolation is implemented using locks on ranges of keys; in B-tree index, locks can be directly attached to the tree.

Other Indexing Structures

Storing values within the index

Multi-column indexes

Full-text search and fuzzy indexes

Keeping everything in memory

Transaction Processing or Analytics?

- A transaction does not need to have ACID properties. Transaction processing means clients make low-latency reads and writes, as opposed to batch processing jobs, which only run periodically.
- **Access pattern: online transaction processing (OLTP)**.

Access patterns are similar for transactions. An application looks up a small number of records by some key, using an index. Records are inserted or updated based on the user's input.

- **Access pattern: online analytic processing (OLAP)**
  Data analytics has different access patterns. Analytic query scans over a huge number of records, only reading a few columns per record, and calculates aggregate statistics rather than returning the raw data to the user. Feed into reports that help the management of a company make better decisions (**business intelligence**).

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Data Warehousing

- A data warehouse is a **separate database**, analysts can query without affecting OLTP operations. The data warehouse contains a **read-only copy** of the data in **various OLTP systems**.
- The process of **getting data** to **warehouse** is **Extract-Transform-Load (ETL)**:
  Data is extracted from OLTP databases (a periodic data dump or a continuous stream of updates), transformed into an analysis schema, cleaned up, and loaded into a data warehouse.

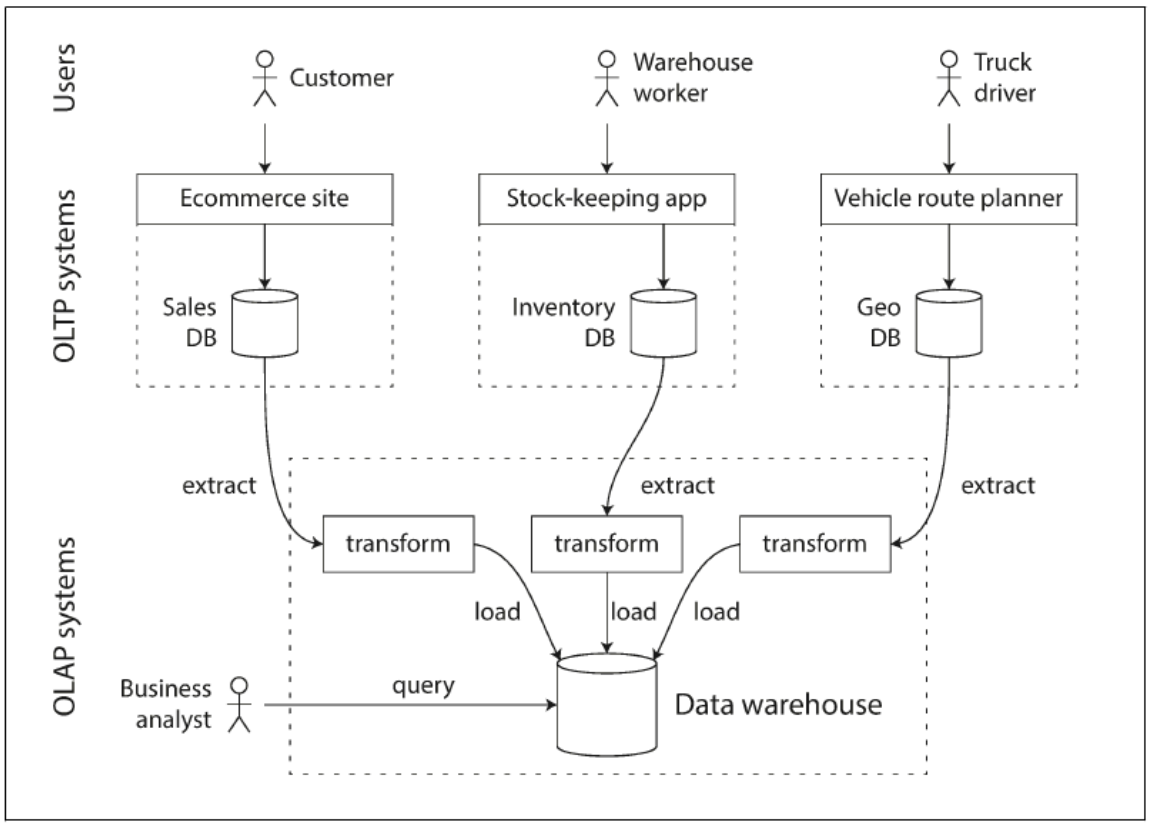

Figure 3-8. Simplified outline of ETL into a data warehouse.

An **advantage** of using a separate data warehouse: data warehouse can be **optimized** for analytic access patterns. It turns out that the indexing algorithms work well for OLTP, but not good at answering analytic queries.

The divergence between OLTP databases and data warehouses

- The **data model** of a data warehouse is **relational**, SQL is a good fit for analytic queries.
- A data warehouse and a relational OLTP database look similar because they have SQL query interface. But the internals of the systems are different, because they are optimized for different query patterns.
- Microsoft SQL Server support for transaction processing and data warehousing in the same product. However, they are two separate storage and query engines, which are accessible through a common SQL interface.

Stars and Snowflakes: Schemas for Analytics

- Different data models are used in transaction processing, depending on the needs of the application.
  In analytics, there is less diversity of data models. Data warehouses are used in a fairly formulaic style: a **star schema (dimensional modeling)**.

At the center of the schema is a fact table, each row of the fact table represents an event that occurred at a particular time.

- Usually, facts are captured as individual events, because this allows maximum flexibility of analysis later. However, the fact table can become super large.
  Big enterprise may have **tens of petabytes** of transaction history in its data warehouse, most in fact tables.
- Some of the columns in the fact table are **attributes**. Other columns are **foreign key** references to other tables, called **dimension tables**.
  As each row in the fact table represents an event, the dimensions represent the who, what, where, when, how, and why of the event.
- A variation of this template is the snowflake schema, where dimensions are further broken down into subdimensions.
  Snowflake schemas are more normalized than star schemas, but star schemas are preferred because they are simpler for analysts to work with.
- In a typical data warehouse, tables are often very wide: fact tables often have over 100 columns, or several hundred. Dimension tables can also be wide, they include all the metadata relevant for analysis.

Column-Oriented Storage

- If you have trillions of rows and petabytes of data in fact tables, storing and querying is a challenging problem. Dimension tables are smaller (millions of rows).
  A typical data warehouse query only accesses a few columns at one time.
- In OLTP databases, storage is row-oriented: values from one row of a table are stored next to each other. Document databases are similar: an entire document is stored as one contiguous sequence of bytes.
- The idea behind **column-oriented storage**: store the values from each column together. Each column is stored in a separate file, a query only reads and parses columns used in that query.
- Column storage is easy to understand in a relational data model, but it applies to non-relational data. E.g, **Parquet** is a **columnar storage format** that supports a **document data model**.
- Column-oriented storage layout relies on each column file containing rows in the same order. To reassemble an entire row, take the 23rd entry from each individual column file and put them together to form the 23rd row of the table.

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 3-10. Storing relational data by column, rather than by row.

Column Compression

- We can **compress data to reduce disk demands**. Column-oriented storage lends itself well to compression.
- The sequences of values for each column are repetitive, a good sign for compression. Depending on data in the column, use different compression techniques.
- An effective technique in data warehouses is **bitmap encoding**.

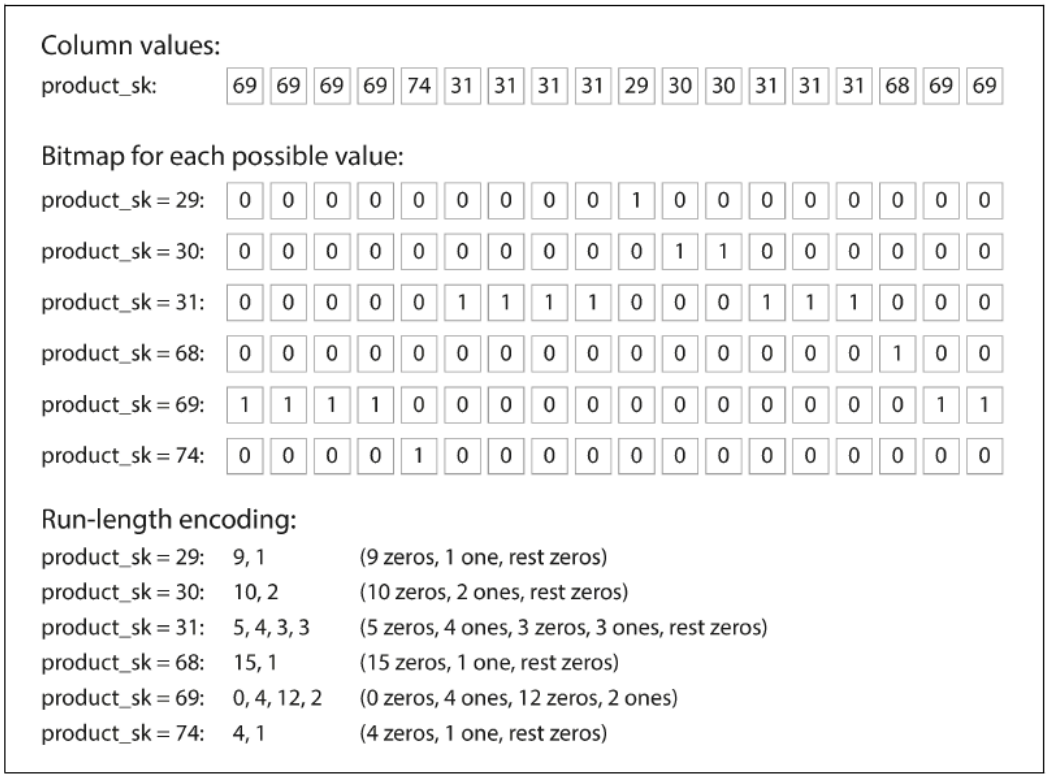

Figure 3-11. Compressed, bitmap-indexed storage of a single column.

- The number of distinct values in a column is small compared to the number of rows.
- Take a column with n distinct values, turn it to n separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.
- If n is small, bitmaps can be stored with **one bit per row**.
  If n is big, there will be many zeros in most of the bitmaps (**sparse**). The bitmaps can be **run-length encoded.** Making the encoding of a column remarkably compact.

Bitmap indexes are good for queries in a data warehouse. Example:

1. WHERE product_sk IN (30, 68, 69):
   Load the three bitmaps for product_sk = 30, product_sk = 68, product_sk = 69, calculate the bitwise OR of the three bitmaps, which can be done efficiently.
2. WHERE product_sk = 31 AND store_sk = 3:
   Load the bitmaps for product_sk = 31 and store_sk = 3, calculate the bitwise AND.
   This works because columns contain rows in the same order, the kth bit in one column's bitmap corresponds to the same row as the kth bit in another column's bitmap.

There are also various other compression schemes for different kinds of data.

Memory bandwidth and vectorized processing

- For data warehouse queries scan over millions of rows, one of the bottlenecks is bandwidth to get data from disk into memory.

- Developers of analytical databases worry about efficiently using the bandwidth from main memory into the CPU cache, avoiding branch mispredictions and bubbles in the CPU instruction processing pipeline, and making use of single-instruction-multi-data (SIMD) instructions in modern CPUs.
- Besides reducing the volume of data to be loaded from disk, **column oriented storage layouts are good for making efficient use of CPU cycles**.
  E.g, the query engine takes a chunk of compressed column data that fits in the CPU's L1 cache and iterates through it in a tight loop (no function calls). CPU executes such a loop faster than code requires function calls and conditions for each record processed.
- Column compression allows rows from a column to fit in the same amount of L1 cache. Operators like bitwise AND and OR can be designed to operate on such chunks of compressed column data directly. This technique is **vectorized processing.**

Sort Order in Column Storage

- In a column store, it doesn't matter what order rows are stored. Easy to store in the order of insertion since inserting a row is appending to each of the column files.
  We can impose an order, like SSTables, and use it as an indexing mechanism.
- The administrator of the database can choose the columns to be sorted, using their knowledge of common queries.
  E.g, if queries often target date ranges, it makes sense to make date_key the first sort key. The query optimizer can scan only the rows of that time period.
- A second column can determine the sort order of any rows with the same value in the first column.
- E.g, if date_key is the first sort key, it makes sense for product_sk to be the second sort key; sales for the same product on the same day can be grouped together, and easily filtered.
- Another advantage of sorted order: it helps column compression.
  If the primary sort column has fewer values, after sorting, it will have long sequences where the same value is repeated many times in a row.
  Run-length encoding can compress that column down to a few kilobytes.
- That compression effect is strongest on the first sort key.
  The second and third sort keys do not have such long runs of repeated values.
  Columns further down the sorting priority in random order, they won't compress.

Several different sort orders

- Store redundant data sorted in different ways, different queries benefit from different sort orders. When processing a query, use the version that best fits the query pattern.
- Having multiple sort orders in a column-oriented store is similar to having multiple secondary indexes in a row-oriented store.
  Difference: The row-oriented store keeps every row in one place (in heap file or a clustered index), secondary indexes just contain pointers to the rows.
  In a column store, no pointers to data, only columns containing values.

Writing to Column-Oriented Storage

- In data warehouses, Most of the load consists of large read-only queries run by analysts. These optimizations, column-oriented storage, compression, and sorting make read queries faster. However, there are downsides of making writes difficult.
- An update-in-place approach, like B-trees, is not possible with compressed columns. Insert a row in the middle of a sorted table, you have to rewrite all column files.
  As rows are identified by position in a column, the insertion has to update all columns consistently.
- **LSM-trees is a good solution:**
  Writes go to in-memory store, added to sorted structure, and prepared for writing to disk.
  When enough writes have accumulated, they are merged with the column files on disk and written to new files in bulk.
- Queries need to examine both the column data on disk and the recent writes in memory, and combine the two.
  The query optimizer hides this distinction from the user.
  From an analyst's point of view, data modified with inserts, updates, or deletes is immediately reflected in subsequent queries.

Aggregation: Data Cubes and Materialized Views

- Not every data warehouse is a column store: row-oriented databases and few other architectures are used. Columnar storage is significantly faster for ad hoc analytical queries.
- Another aspect of data warehouses is **materialized aggregates**.
  We can cache some counts or sums that queries often use, one way to create a cache is a materialized view.
- In a relational data model, it is defined like a standard (virtual) view: a table-like object contains the results of a query.
  Difference:
  A materialized view is an actual copy of the query results, written to disk.
  A virtual view is a shortcut for writing queries. When read from a virtual view, the SQL engine expands it into the view's underlying query on the fly and processes the expanded query.
- When the underlying data changes, a materialized view needs to be updated, because it is a **denormalized** copy of the data.
  The database can update automatically, but it makes writes more expensive, it's not used in OLTP databases.
  It makes more sense in read-heavy data warehouses, read performance improves or not also depends on individual case).
- A special case of a materialized view is a **data cube** or **OLAP cube**: **a grid of aggregates grouped by different dimensions.**

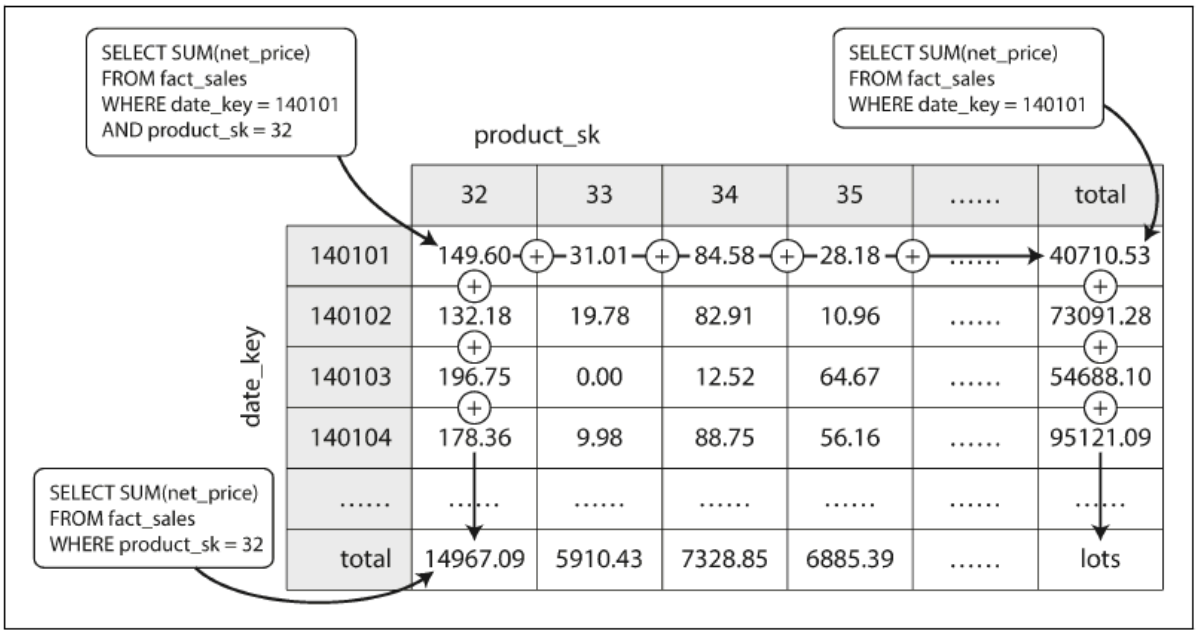

Figure 3-12. Two dimensions of a data cube, aggregating data by summing.

- Each fact has foreign keys to two dimension tables (date and product).
- Draw a two-dimensional table, with dates along one axis and products along the other.
- Each cell contains the aggregate (e.g., SUM) of an attribute (e.g., net_price) of all facts with date-product combination.
- Apply the same aggregate along each row or column, get a summary that has been reduced by one dimension (the sales by product regardless of date, or the sales by date regardless of product).
- Facts often have more than two dimensions. But the principle remains the same: each cell contains the sales for a particular date-product-store-promotion-customer-xxx-xxx combination. These values can repeatedly be summarized along each dimension.
- **Advantage** of a materialized data cube: certain queries become fast because they are effectively precomputed.
  E.g, to know the total sales per store yesterday, you just need to look at the totals along the appropriate dimension, no need to scan millions of rows.
- **Disadvantage:** Data cubes have less flexibility than querying raw data.
  E.g, You cannot calculate which proportion of sales comes from items that cost more than $100, because the price isn't one of the dimensions.
  Most data warehouses try to keep as much raw data as possible, and use aggregates like data cubes only as a performance boost for certain queries.