Chapter 3: Storage and Retrieval

Indexes are additional data structures used in databases to efficiently locate data for specific keys. They act as metadata to speed up read queries. While multiple indexes can be created for different search patterns, **they are derived from the primary data and only impact query performance,** not the data itself. However, **maintaining indexes adds overhead, particularly slowing down write operations** as indexes need to be updated. Therefore, choosing indexes wisely based on typical query patterns is crucial to balance the **trade-off between faster reads and slower writes**. Databases usually require manual selection of indexes by developers or administrators.

Hash Indexes:

Hash indexes for key-value data function similarly to in-memory hash maps. A simple approach involves maintaining an in-memory hash map where each key points to the byte offset of its value in an append-only data file. When new data is written, the hash map is updated with the offset. Lookups involve using the hash map to find the offset and then reading the value from the file at that location.

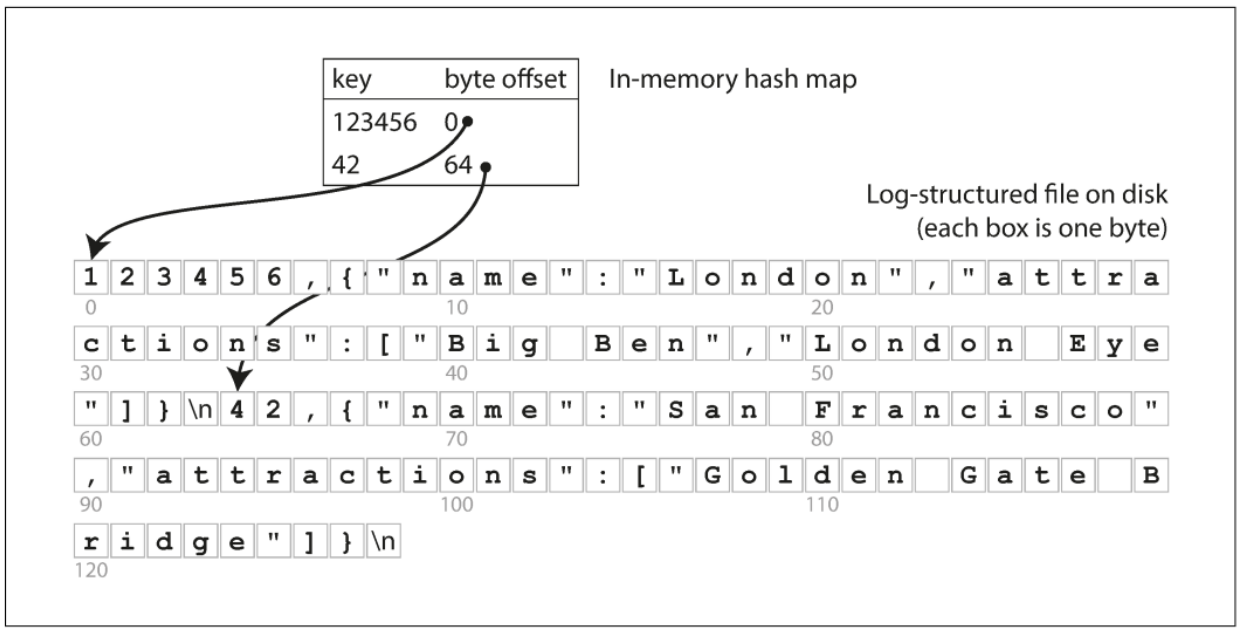
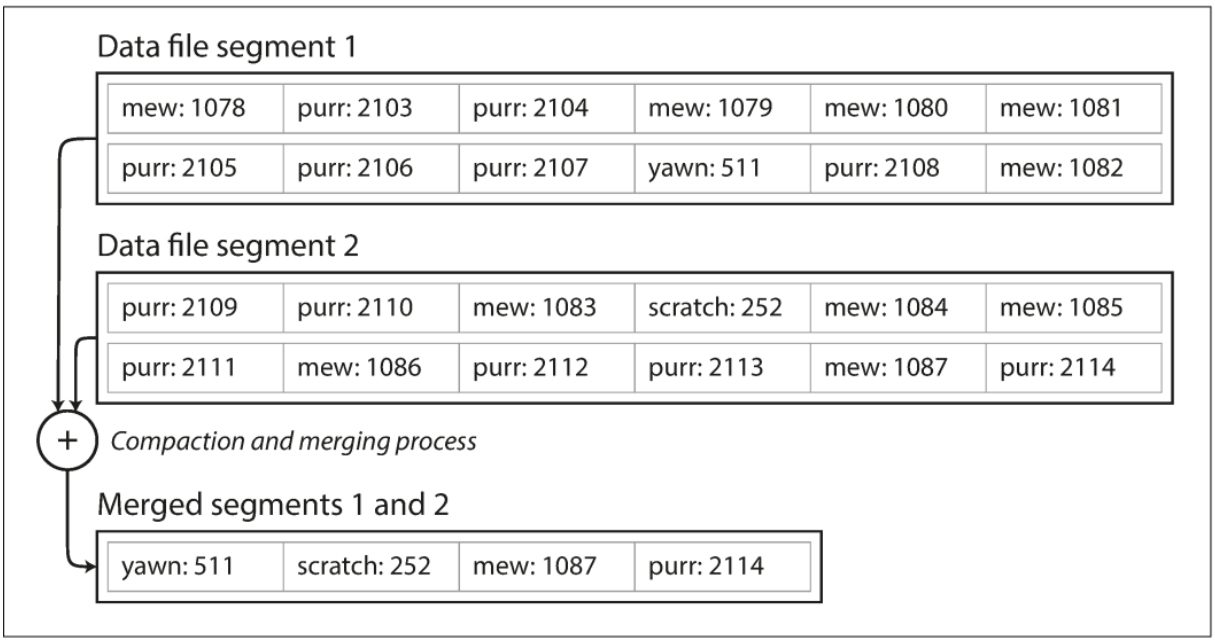


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.



Performing compaction and segment merging simultaneously.

To manage an ever-growing append-only log, it can be **divided into segments.**

Once a segment reaches a certain size, a **new one is created, and compaction is performed to discard duplicate keys**, keeping only the latest update.

Since compaction often makes segments much smaller (assuming that a key is overwritten several times on average within one segment), **we can also merge several segments together at the same time as performing the compaction.**

The **merging and compaction of frozen segments can be done in a background thread,** and while it is going on, we can continue to serve read and write requests as normal, using the old segment files. **After the merging process is complete, we switch read requests to using the new merged segment instead of the old segments**—and then the old segment files can simply be deleted.

Each segment has its own in-memory hash table mapping keys to file offsets, and lookups involve checking these tables sequentially from the newest segment.

Practical implementations need to consider file format (binary is preferred over CSV), **handling deletions with tombstone records**, crash recovery (e.g., by storing hash map snapshots), **dealing with partially written records using checksums,** and managing concurrency with a single writer thread and concurrent readers.

The append-only design offers advantages like faster sequential writes, simpler concurrency and crash recovery, and prevention of data fragmentation.

However, hash table indexes are limited by the **need for the hash table to fit in memory and the inefficiency of range queries.**

SSTables and LSM-Trees

Initially, log-structured storage segments contain key-value pairs in the order they were written, with later values overriding earlier ones for the same key. A modification introduces the requirement that these segments are sorted by key, leading to the format called Sorted String Table (SSTable). In an SSTable, each key appears only once per merged segment. This new format provides several benefits compared to log segments using hash indexes.

Here's a summary of the advantages of SSTables:

1. **Efficient Merging:** Merging SSTable segments is straightforward and efficient, even for files larger than memory, using a merge sort like process. When the same key exists in multiple segments during a merge, the value from the most recent segment is retained.
2. **Efficient Key Lookup:** Due to the sorted nature of SSTables, it's no longer necessary to keep an index of all keys in memory. Instead, a sparse index with offsets to certain keys can be used. To find a specific key, you can jump to the offset of the nearest preceding key in the index and then scan sequentially.
3. **Better Compression and I/O Efficiency:** Since range queries often involve scanning multiple key-value pairs, SSTables can group these into compressed blocks before writing to disk. The sparse in-memory index then points to the start of these compressed blocks. This not only saves disk space but also reduces the amount of I/O bandwidth required.

Constructing and maintaining SSTables

Incoming writes are first added to an **in-memory sorted data structure called a memtable**. When the memtable exceeds a threshold, **it's written to disk as a sorted SSTable file**, becoming the newest segment.
Read requests are served **by checking the memtable first, followed by the most recent SSTable segments.**

A background process merges and compacts SSTable segments over time.
To prevent data loss in case of a crash, **every write is also immediately appended to a separate on-disk log,** which can be used to restore the memtable upon restart.
Once the memtable is successfully written to an SSTable, its corresponding log can be discarded.
Performance optimizations

To optimize lookups for non-existent keys in LSM-trees, **Bloom filters** are often used to quickly determine if a key is definitely not present, **avoiding unnecessary disk reads**.
There are different strategies for compacting and merging SSTables: **size-tiered**, where smaller, newer tables are merged into larger, older ones, and **leveled**, where the key range is **split into levels**, allowing more incremental compaction and better disk space usage.
Despite these complexities, the fundamental LSM-tree concept of cascading SSTables merged in the background is effective even for large datasets. Furthermore, the sorted nature of data enables efficient range queries, and **the sequential disk writes** support high write throughput.

B-Trees

B-trees are another indexing structure that divides the database into fixed-size pages **(typically 4 KB)**, reading and writing one page at a time, **which aligns well with disk hardware.**
These pages form a **tree structure using on-disk pointers**. The search for a key starts at the **root page, which contains keys and references to child pages** representing key ranges.

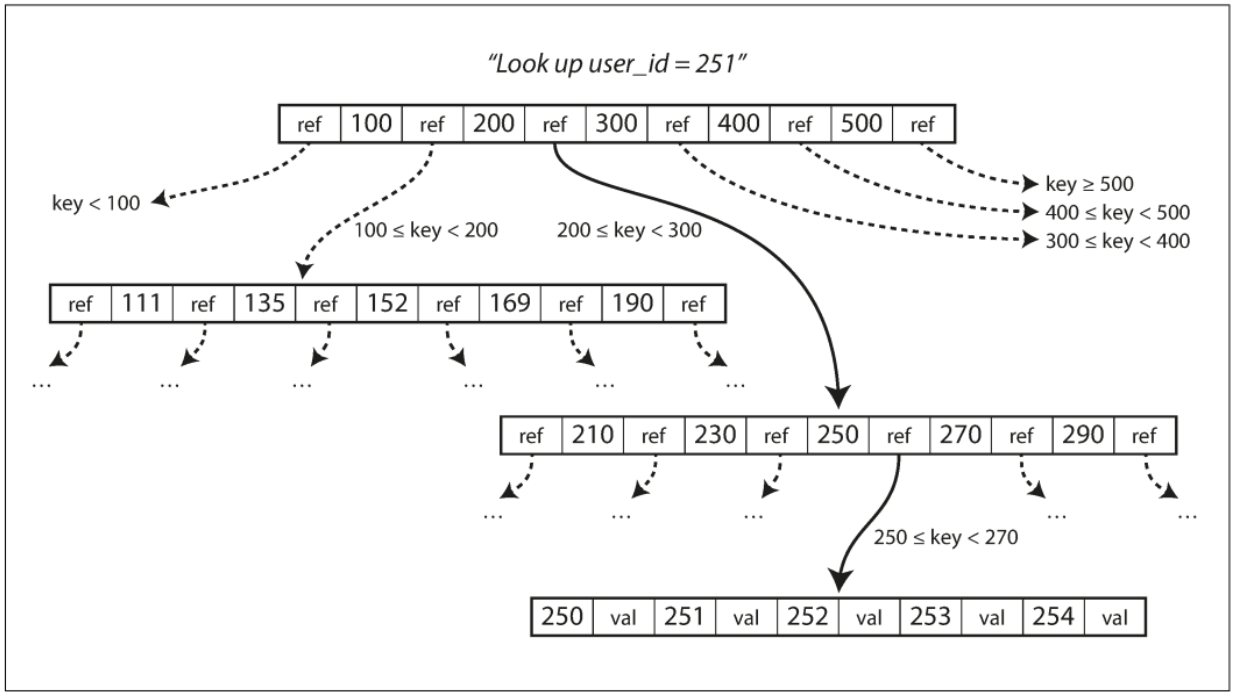


Figure 3-6. Looking up a key using a B-tree index.

This process continues down the tree until the leaf page containing the key is found. **Leaf pages either hold the value or pointers to it**. The branching factor is the number of child references per page.

Updating a key involves finding its leaf page, modifying the value, and writing the page back.

Adding a new key might require **splitting a full page and updating its parent**.

B-trees maintain balance, resulting in a logarithmic search depth, making lookups efficient even for large databases.

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.)**

Making B-trees reliable

Unlike log-structured indexes, B-trees rely on overwriting pages in place on disk. This can be complex at the hardware level and poses challenges for reliability, especially when operations like page splits require **overwriting multiple pages, potentially leading to corruption on crashes.**

To address this, B-tree implementations commonly use a **Write-Ahead Log (WAL)** or redo log. This is an append-only file where all **modifications to the B-tree are first recorded**

before being applied to the actual tree pages, ensuring consistency and enabling recovery after a crash.
Furthermore, in-place updates in B-trees **necessitate careful concurrency control** using latches to manage simultaneous access from multiple threads.
Log-structured approaches simplify concurrency by performing **merging in the background** and **atomically swapping segments**.

B-tree optimizations

One approach is **copy-on-write,** where **modified pages are written to new locations**, and **parent pages are updated, aiding in crash recovery and concurrency**.

Copy-on-write (COW) B-trees work on the principle of immutability. Instead of directly modifying existing pages on disk, **any operation that would change a page (like inserting, deleting, or updating a key) results in the creation of a new version of that page.** Here's a more detailed breakdown:

1. **Immutable Pages:** The core idea is that once a page in a COW B-tree is written, it is never modified in place. This immutability is key to the benefits of this approach.
2. **Write Operation:** When a write operation needs to occur:
    a. **Identify the Leaf Page:** The B-tree is traversed to find the leaf page where the change needs to be made.
    b. **Create a New Leaf Page:** Instead of modifying the existing leaf page, a completely new leaf page is created at a different location on disk.
    c. **Copy and Modify:** The content of the original leaf page is copied to the new page. The necessary modification (insertion, deletion, or update) is then performed on this new page.
3. **Updating Parent Pointers:** Since the leaf page has moved to a new location, any parent page that previously pointed to the old leaf page now needs to point to the new one. This update is also done using the copy-on-write approach:
    a. **Identify the Parent Page:** The parent page of the modified leaf page is located.
    b. **Create a New Parent Page:** A new version of the parent page is created.
    c. **Copy and Update Pointer:** The content of the original parent page is copied to the new page. The pointer that previously referenced the old leaf page is updated to point to the new leaf page.
4. **Propagating Changes Up the Tree:** This process of creating new versions of pages and updating pointers continues up the tree. If the parent page was modified, its

parent (the grandparent of the original leaf) will also need to be updated in a new copy, and so on, potentially all the way up to the root of the B-tree.

5. **Updating the Root Pointer:** Eventually, the chain of new page creations might reach the root of the B-tree. The system then needs to update a single, stable pointer that always points to the current, valid root of the tree. **This root pointer is often the only mutable part of the COW B-tree structure.**
6. **Benefits of Copy-on-Write:**
    a. **Crash Recovery:** If the system crashes during a write operation, the old version of the tree remains intact and consistent. **Upon restart, the database can simply revert to the last known good root pointer.** There's no need for a separate Write-Ahead Log (WAL) in many COW implementations, simplifying recovery.
    b. **Concurrency Control (Snapshot Isolation):** The immutability of pages allows for efficient support of concurrent readers and writers without the need for complex locking mechanisms in some cases. **Readers can continue to use an older version of the tree (pointed to by an older root pointer) while writers are making changes in a new version.** This provides a form of snapshot isolation.
    c. **Simplified Transactions and Rollback:** Keeping older versions of pages can make it easier to implement transactional semantics and support rollback operations, **as reverting to a previous state simply involves switching back to an older root pointer.**
7. **Garbage Collection:** Over time, **many old versions of pages will no longer be referenced by the current root or any ongoing transactions**. A garbage collection process is necessary to identify and reclaim the disk space occupied by these obsolete pages. This is a crucial aspect of managing a COW B-tree to **prevent storage from filling up with outdated data.**

In essence, copy-on-write B-trees trade the cost of writing out new pages for increased reliability and simplified concurrency control. Every change creates a new lineage of pages from the modified leaf up to a new root (or a new pointer to an existing ancestor), leaving the old versions untouched.

Another optimization involves abbreviating keys in interior pages, using only enough information to define key range boundaries, thus increasing the branching factor and reducing tree depth.

While ideally leaf pages with sequential key ranges should be laid out sequentially on disk for efficient range scans, this is difficult to maintain as the tree grows; LSM-trees handle this better.

Adding **sibling pointers** between leaf pages allows for efficient sequential scanning without needing to traverse back up the tree.

Finally, some B-tree variants, like fractal trees, incorporate log-structured ideas to reduce disk seeks.

Advantages of LSM-trees

Generally, LSM-trees are faster for writes, while B-trees are considered faster for reads, although this can depend on the workload.

Reads in LSM-trees might be slower as they need to check multiple data structures.

B-trees **write data at least twice (WAL and page),** with potential overhead and sometimes double overwrites for safety.

LSM-trees also **rewrite data during compaction, leading to write amplification**, a concern for SSD lifespan.

In write heavy applications, B-tree write amplification can limit throughput. LSM-trees often achieve **higher write throughput due to potentially lower write amplification and sequential writes of SSTables**, especially beneficial on magnetic drives.

LSM-trees also tend to have better compression and smaller on-disk sizes due to **less fragmentation compared to B-trees**, which can have **unused space in pages.**

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.

Since LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, **they have lower storage overheads,** especially when using leveled compaction.

While SSDs internally optimize writes, lower write amplification and fragmentation remain advantageous for I/O bandwidth.

Downsides of LSM-trees

Log-structured storage has downsides related to its compaction process, which can interfere with ongoing read and write performance due to **limited disk resources, potentially causing high latency at higher percentiles.**

Additionally, high write throughput can l**ead to disk bandwidth contention between initial writes and background compaction**. If compaction can't keep up, **unmerged segments accumulate**, leading to disk space exhaustion and slower reads.

Unlike B-trees where each key exists once, log-structured storage can have multiple copies, making **B-trees more suitable for databases needing strong transactional semantics** with range locks.
While B-trees are well-established and perform consistently, log-structured indexes are gaining popularity in new datastores, and the best choice depends on the specific use case, requiring empirical testing.

Storing values within the index

In database indexes, the value associated with a key can either be the actual row or a reference to it. Using a **heap file** stores rows separately in no particular order, with indexes referencing their location. This avoids data duplication across multiple secondary indexes. Updating values in a heap file is efficient if the new value isn't larger, but larger updates might require moving the row and updating index references or using forwarding pointers. However, the extra step to the heap file can cause read performance penalties.

To avoid this penalty, a **clustered index** stores the entire row data directly within the index. This is the case with the primary key in MySQL's InnoDB.

A compromise is a **covering index** (or index with included columns), which stores some of a table's columns within the index. This allows certain queries to be answered **directly from the index** itself, without needing to access the full row.

Keeping everything in memory

- Disks are durable and cheaper per GB than RAM but have performance limitations due to the need for careful data layout.
- As RAM costs decrease, keeping entire datasets in memory becomes feasible, leading to the development of in-memory databases.
- Some in-memory databases are for caching (data loss on restart is acceptable), while others prioritize durability.
- Durability in in-memory databases can be achieved through battery-powered RAM, logging changes to disk, periodic snapshots, or replicating to other machines.
- Even with disk usage for durability, reads in in-memory databases are primarily served from memory.
- In-memory databases can offer significant performance improvements by eliminating overhead associated with managing on-disk data structures.

- Examples of in-memory databases include VoltDB, MemSQL, Oracle TimesTen, RAMCloud, Redis, and Couchbase.
- The performance advantage of in-memory databases comes from avoiding the overhead of encoding data for disk storage, not just from avoiding disk reads (as OS caching can also achieve that).
- In-memory databases can easily implement complex data models that are difficult with disk-based indexes (e.g., Redis with priority queues and sets).
- Research is exploring "anti-caching" approaches to allow in-memory databases to handle datasets larger than available memory by efficiently swapping less recently used data to disk at a record level.
- The anti-caching approach still typically requires indexes to fit entirely in memory.
- The emergence of non-volatile memory (NVM) technologies is a future area that could further impact storage engine design.

Transaction Processing or Analytics?

- Initially, database writes corresponded to business transactions, but the term "transaction" now refers to a logical unit of reads and writes.
- **Transaction processing (OLTP)** involves low-latency reads and writes, typically accessing a small number of records by key for interactive applications.
- **Data analytics (OLAP)** involves scanning large numbers of records, reading specific columns, and calculating aggregate statistics for business intelligence.
- Examples of OLAP queries include revenue per store, sales comparisons during promotions, and product co-purchase analysis.
- Initially, the same databases were used for both OLTP and OLAP.
- SQL is versatile enough to handle both types of queries.
- Later, companies started using separate databases for OLTP and OLAP, with the analytical database being called a **data warehouse**.

Data Warehousing

- Enterprises have numerous independent OLTP systems for various business functions.
- OLTP systems require high availability and low latency, making database administrators hesitant to allow resource-intensive analytical queries on them.
- A **data warehouse** is a separate, read-only database specifically for analytical queries, preventing impact on OLTP operations.

- Data from various OLTP systems is extracted, transformed, and loaded (ETL) into the data warehouse.
- Data warehouses are common in large enterprises but less so in small ones with fewer systems and smaller data volumes.
- A key advantage of a data warehouse is that it can be optimized for analytical access patterns, which differ from OLTP access patterns.

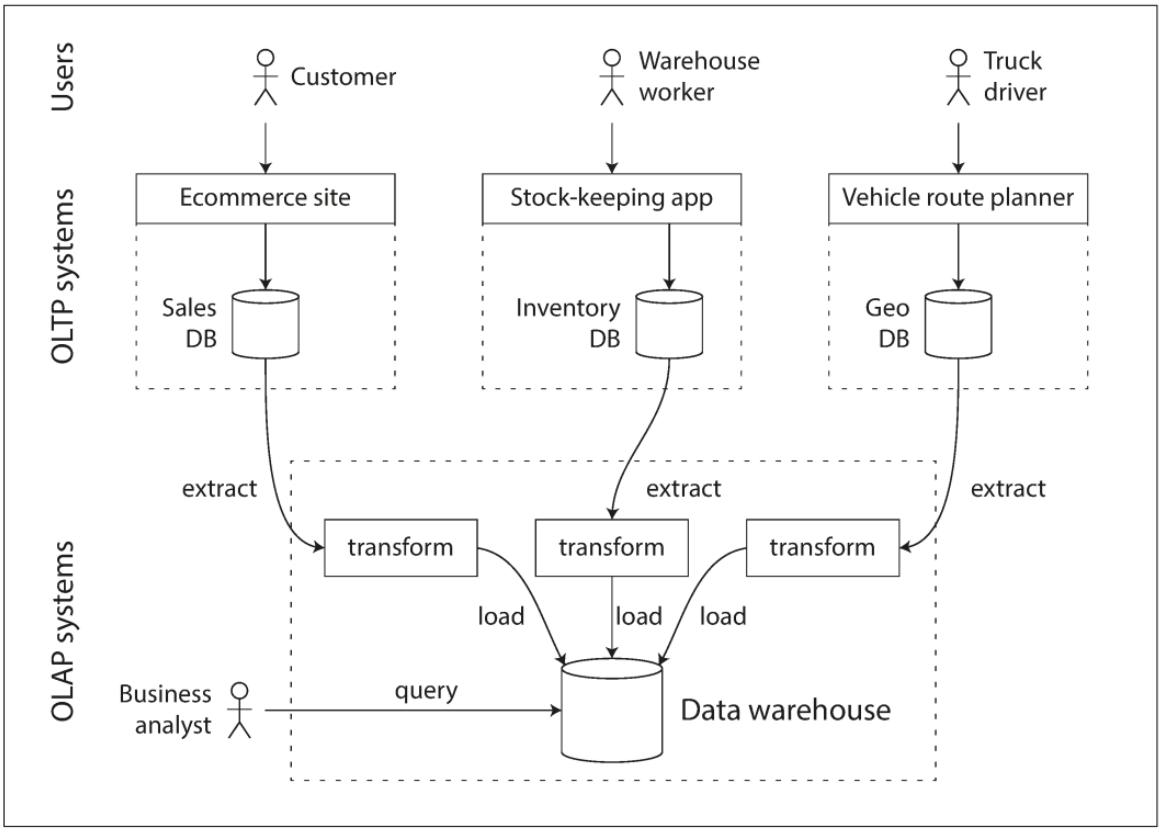


Figure 3-8. Simplified outline of ETL into a data warehouse.

Column-Oriented Storage

The idea behind column-oriented storage is simple: don't store all the values from one row together but store all the values from each column together instead. If each col- umn is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work.

Memory bandwidth and vectorized processing

For data warehouse queries scanning millions of rows, a major bottleneck is the speed of transferring data from disk to memory. However, optimizing the use of bandwidth from

main memory to the CPU cache, minimizing CPU pipeline stalls, and leveraging SIMD instructions are also critical concerns for analytical database performance.

Column-oriented storage helps address these bottlenecks by:

- Reducing the amount of data that needs to be loaded from disk.
- Enabling efficient use of CPU cycles by allowing the query engine to process chunks of compressed column data within the fast L1 cache using optimized loops.
- Column compression further increases the amount of relevant data that can fit in the CPU cache.

Column compression allows more rows from a column to fit in the same amount of L1 cache. Operators, such as the bitwise AND and OR described previously, can be designed to operate on such chunks of compressed column data directly. This technique is known as vectorized processing.

Writing to Column-Oriented Storage

- Optimizations for column-oriented storage in data warehouses, while improving read performance, make writing more challenging.
- In-place updates are difficult with compressed columns; inserting into a sorted column requires rewriting column files.
- The **LSM-tree** approach offers a solution for efficient writes to column-oriented storage.
- New writes are first added to an in-memory store (can be row or column-oriented).
- When a sufficient number of writes accumulate, they are merged with the existing column files on disk and written as new files in bulk.
- This is the approach used by systems like Vertica.
- Queries need to examine both the column data on disk and the recent writes in memory.
- The query optimizer handles this transparently, ensuring users see a consistent and up-to-date view of the data.