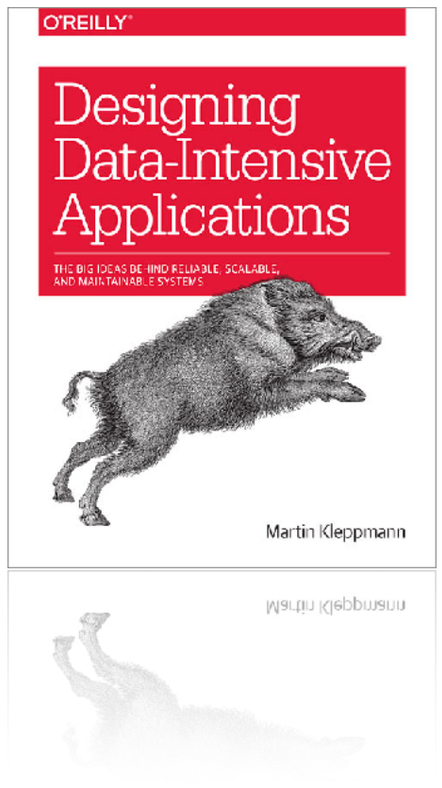
Data Models,Representation and Storage



Disclaimer: all pictures are taken from this book for teaching purposes

Relation Model

- Proposed by Edgar Codd in 1970
- Data is organized into *relations* (*tables* in SQL), where each relation is an unordered collection of *tuples* (*rows* in SQL)
- Born as a theoretical proposal, in mid-1980s became the standard model for relational database management systems (RDMS) and SQL
- The goal was to hide the implementation details behind a cleaner interface
- Different alternatives proposed (network model, hierarchical model, object model, XML model) but never lasted

Impedance Mismatch

If data is stored in relational tables, an awkward translation layer is required between the objects in application code and the data model of tables, rows and columns.

http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

one-to-one relationships

one-to-many relationships

To rebuild the original data, we need to perform multiple queries or messy multiway joins

Document Model


```
{
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```


They store nested records (one-to-many relationships) within their parent record rather than in a separate table

Document vs. Relational

- Document models have poor support for joins
- Document models cannot refer directly to a nested item within a document
- Document models good for one-to-many relationships
- Relational models good for many-to-many relationships
- *Schema Flexibility*
    - Document: schema-on-read (data schema is implicit and interpreted when read)
    - Relational: schema-on-write (data schema is explicit and enforced when written)
- Query Performance

Graph-like Models

- A graph consists of two kinds of object: vertices (also known as *nodes* or *entities*) and edges (also known as *relationships* or *arcs*).

- Examples:
  - Social graphs: vertices are people, edges indicate which people know each other.
  - The Web graph: vertices are web pages, edges indicate HTML links to other pages.
  - Road or rail networks: vertices are junctions, and edges represent the roads or railway lines between them.

Property Graph Models

- Each **vertex** consists of: a unique identifier, a set of outgoing edges, a set of incoming edges, and a collection of properties (key-value pairs).

- Each **edge** consists of: a unique identifier, the vertex at which the edge starts (the tail vertex), the vertex at which the edge ends (the head vertex), a label to describe the kind of relationship between the two vertices, and a collection of properties (key-value pairs).

- Property graphs have a great deal of flexibility for data modeling:

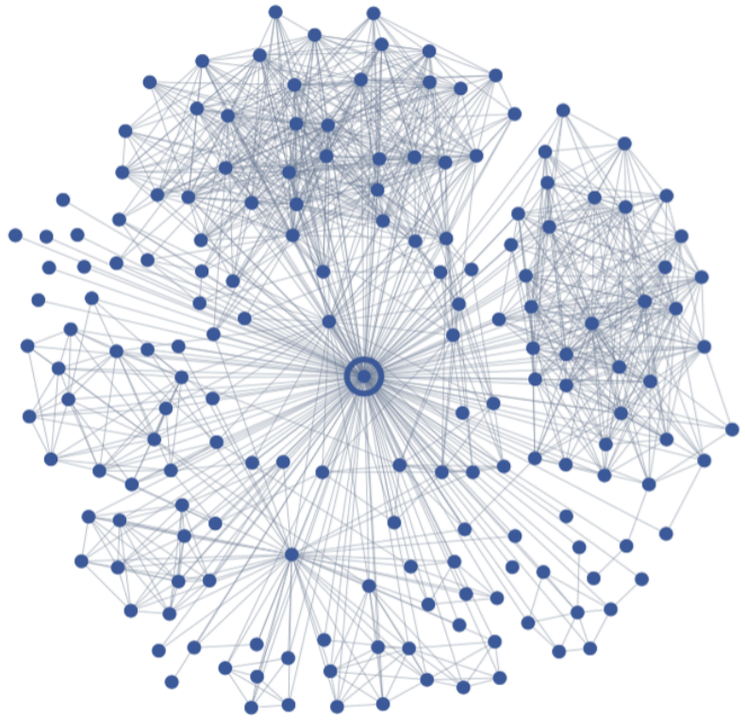
    1. Any vertex can have an edge connecting it with any other vertex. There is no schema that restricts which kinds of things can or cannot be associated.

    2. Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus traverse the graph.

    3. By using different labels for different kinds of relationship, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

Example: Facebook

https://research.facebook.com/publications/unicorn-a-system-for-searching-the-social-graph/



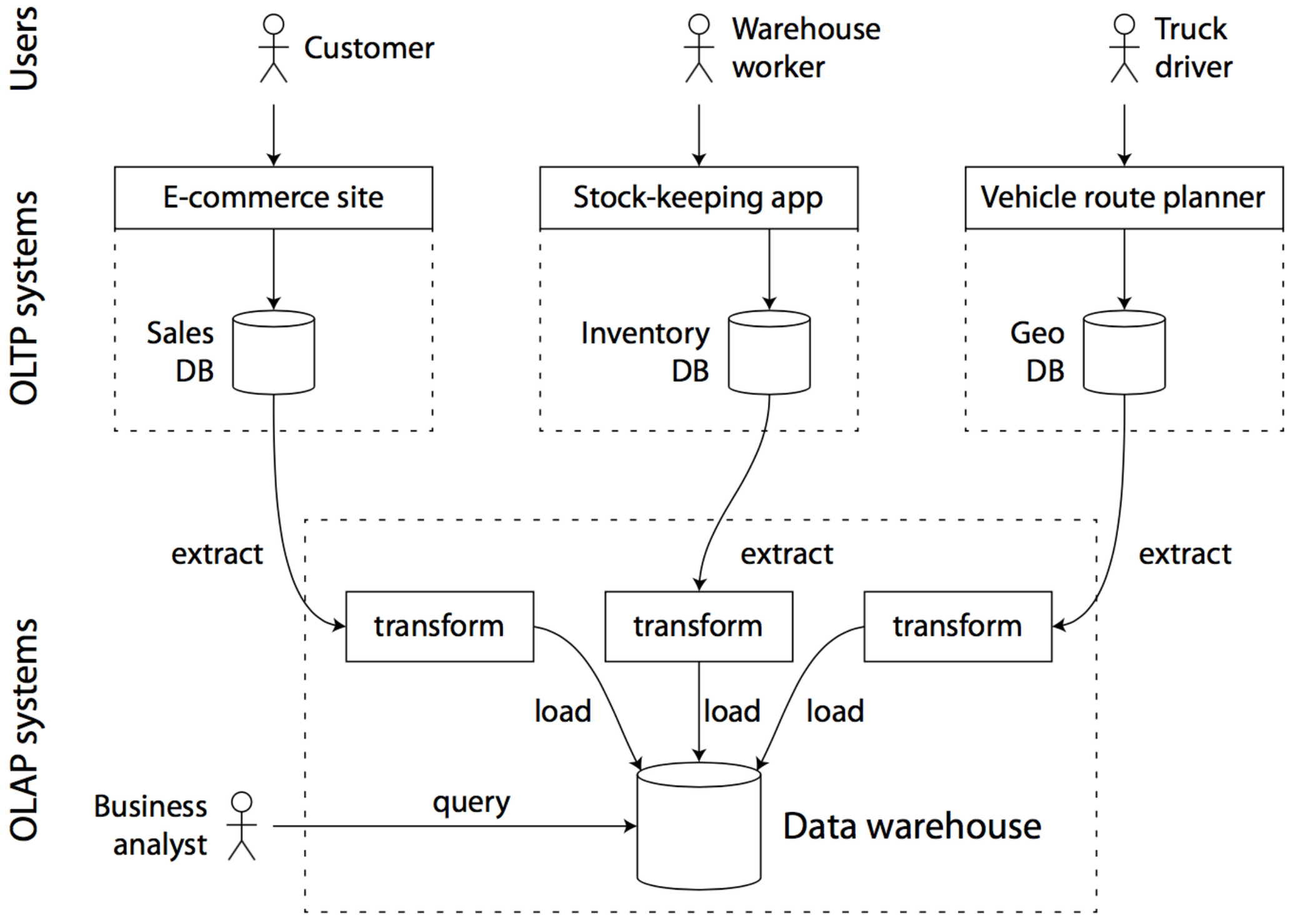
Edge-Type	#-out	in-id-type	out-id-type	Description
`friend`	*hundreds*	USER	USER	Two users are friends (symmetric)
`likes`	*a few*	USER	PAGE	pages (movies, businesses, cities, etc.) liked by a user
`likers`	10–10M	PAGE	USER	users who have liked a page
`live-in`	1000–10M	PAGE	USER	users who live in a city
`page-in`	*thousands*	PAGE	PAGE	pages for businesses that are based in a city
`tagged`	*hundreds*	USER	PHOTO	photos in which a user is tagged
`tagged-in`	*a few*	PHOTO	USER	users tagged in a photo (see Section 7.2)
`attended`	*a few*	USER	PAGE	schools and universities a user attended

Table 1: A sampling of some of the most common edge-types in the social graph. The second column gives the typical number of hits of the output type that will be yielded per term.

Database 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



OLTP

Indexing

- In order to efficiently find the value for a particular key in the database, we need a different data structure: an *index*.

- The general idea behind an index is to keep some additional metadata on the side, which acts as a signpost and helps you to locate the data you want.

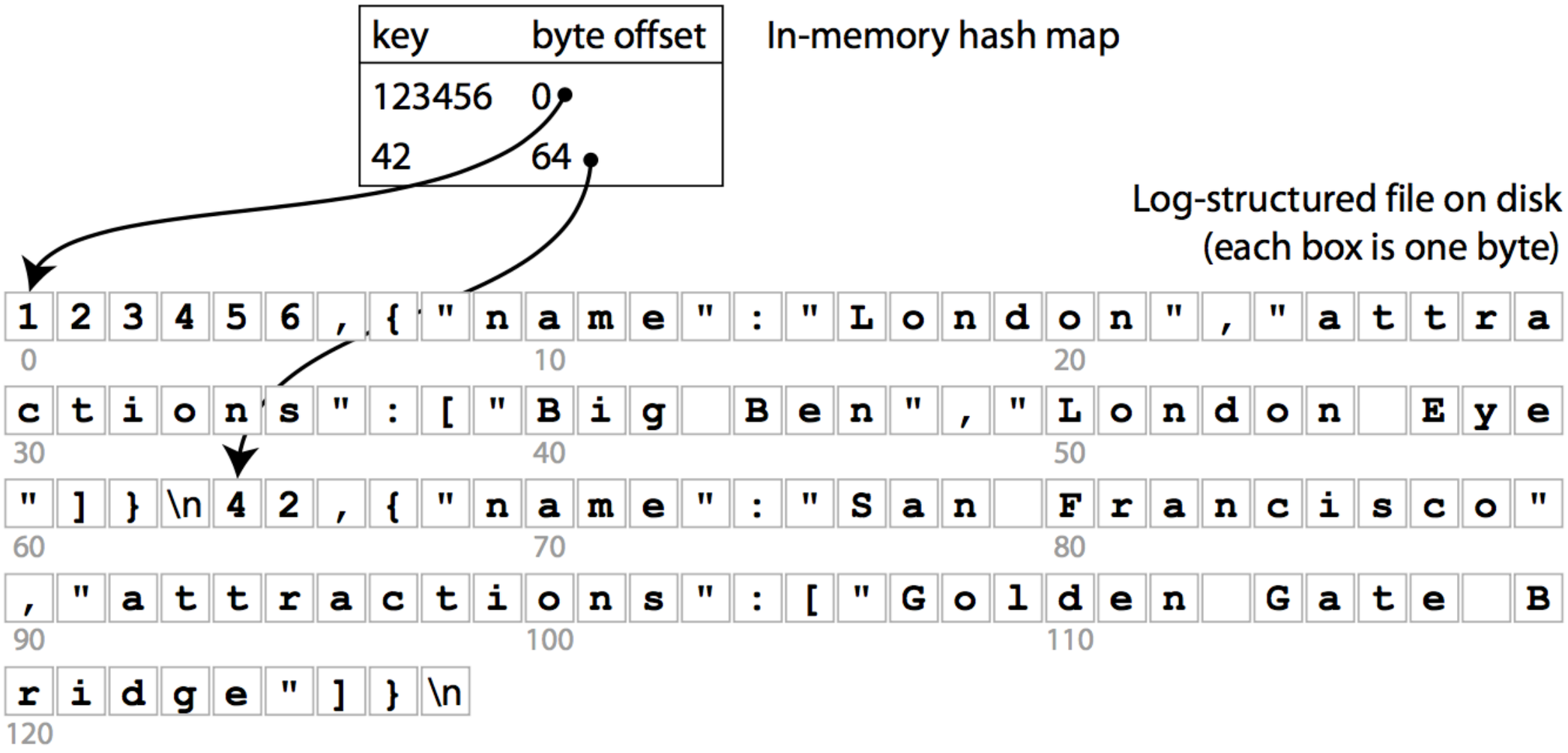
- If you want to search the same data in several different ways, you may need several different indexes on different parts of the data.

- An index is an additional structure that is derived from the primary data, and their maintenance is overhead, especially on writes.

- Well-chosen indexes speed up read queries, but every index slows down writes. For this reason, databases don't usually index everything by default, but require you — the application developer or database administrator — to choose indexes manually, using your knowledge of the application's typical query patterns.

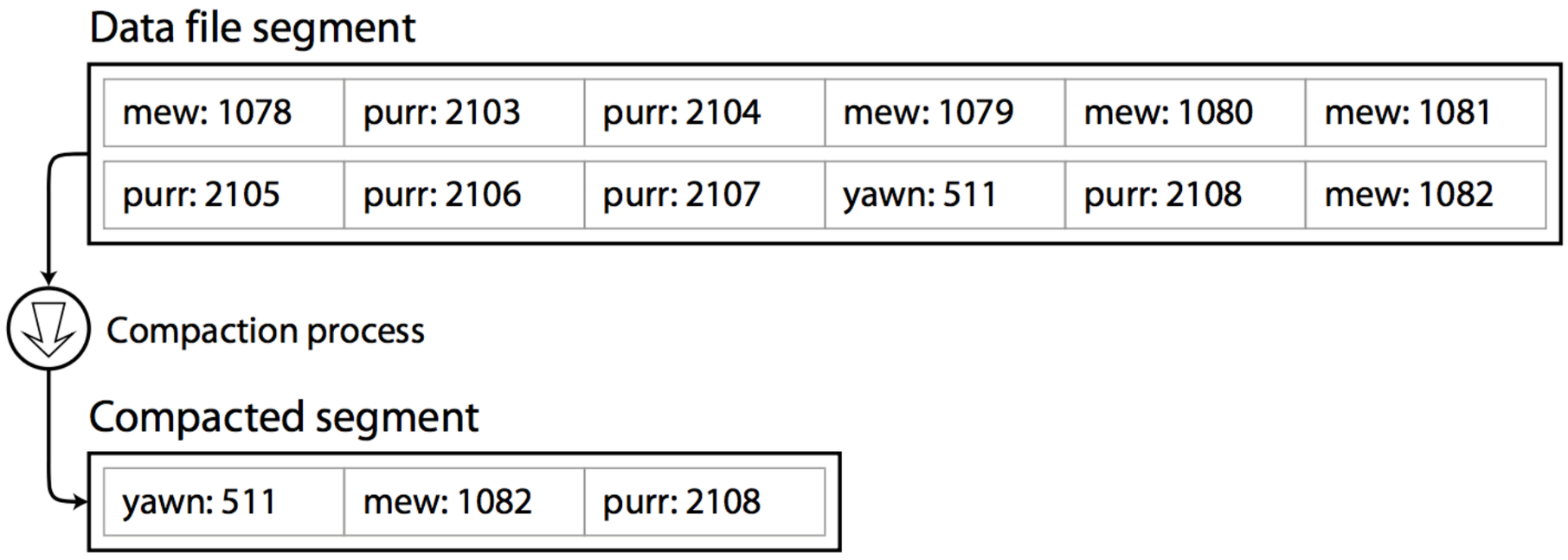
Hash Indexes

- Focus on **append-only key-value data**
- Typically stored in **main memory**
- When you want to look up a value, use the hash map to find the offset in the data file, seek to that location, and read the value.
- These pairs appear **in the order that they were written**, and later values take precedence over early values for the same key.



Segments

- How do we avoid eventually running out of disk space?
- Break the data file into segments of a certain size, and to perform compaction on these segments



- 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

Details and limitations

- **File format**: use a binary format which first encodes the length of a string in bytes, followed by the raw string.

- **Deleting records**: if you want to delete a key and its associated value, you have to append a special deletion record to the data file (sometimes called a tombstone). When log segments are merged, the tombstone tells the merging process to discard any previous values for the deleted key.

- **Crash recovery**: if the database is restarted, the in-memory hash maps are lost. We can rebuilt them by scanning the whole segments, or snapshooting each segment's hash map on disk.

- **Partially written records**: the database may crash at any time, including halfway through appending a record to the log. Corrupted parts can be detected and ignored through checksums.

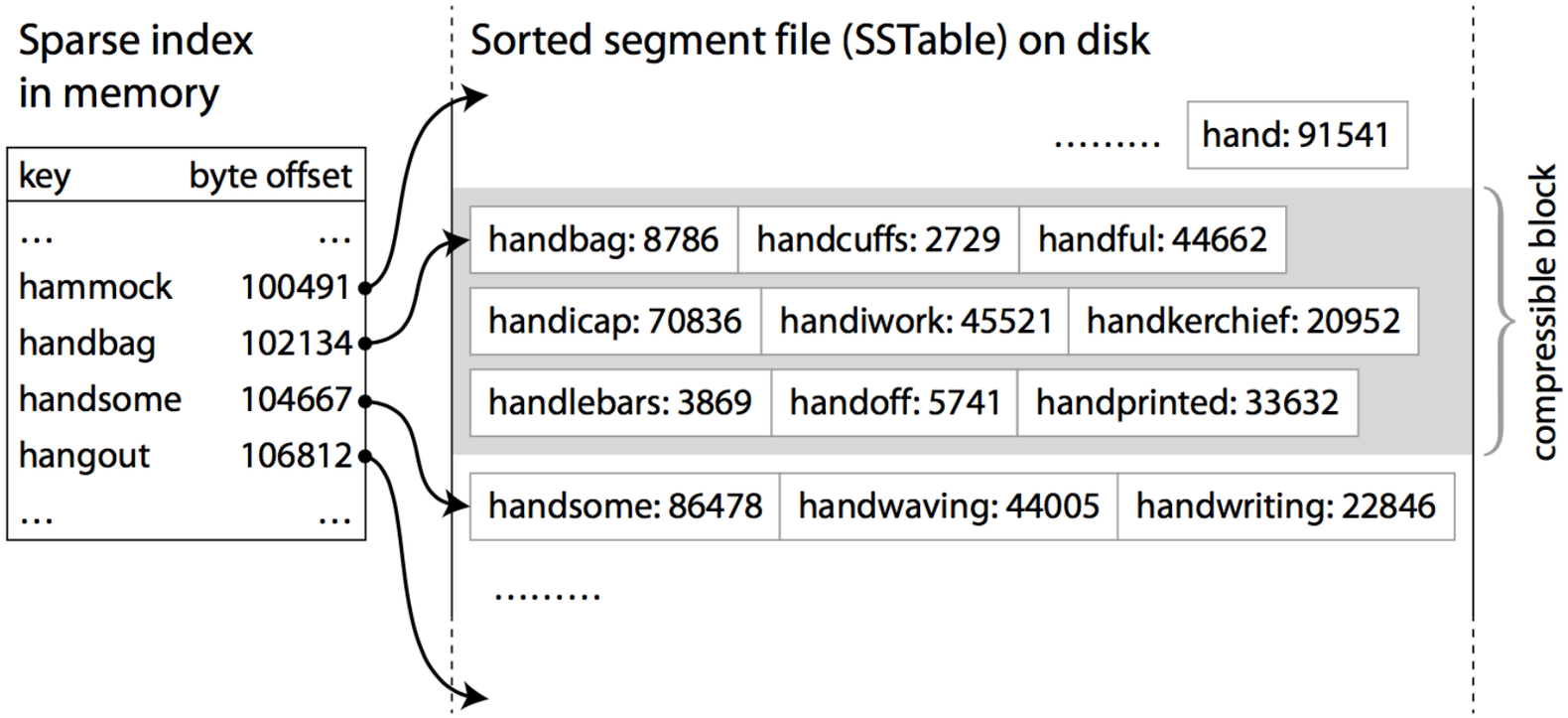
- **Concurrency control**: As writes are appended to the log in a strictly sequential order, a common implementation choice is to have only one writer thread. Data file segments are append-only and otherwise immutable, so they can be concurrently read by multiple threads.

- The **hash table** must fit in memory, so if you have a very large number of keys, you're out of luck. In principle, you could maintain a hash map on disk, but unfortunately it is difficult to make an on-disk hash map perform well.

- **Range queries** are not efficient.

Sorted String Tables

- Sorted String Table (SSTable): the sequence of key-value pairs is sorted by key (no insertion time)
- We also require that each key only appears once within each merged segment file (the merging process already ensures that).
- SSTables have several big advantages over segments with hash indexes:
    - Merging segments is simple and efficient, even if the files are bigger than the available memory. When multiple segments contain the same key, we can keep the value from the most recent segment, and discard the values in older segments.
    - No longer need to keep an index of all the keys in memory, just use an in-memory sparse index for the segments boundaries, then linear scan of a segment.
- Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk.



Storage Engine Workflow

- When a write comes in, add it to an in-memory balanced tree data structure (e.g., a Red-Black tree). This in-memory tree is sometimes called a memtable.

- When the memtable gets bigger than some threshold — typically a few megabytes — write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. When the new SSTable is ready, the memtable can be emptied.

- In order to serve a read request, first try to 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.

- If the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost. In order to avoid that problem, we can keep a separate hash indexed log on disk to which every write is immediately appended. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

Notes

- Originally this indexing structure was names Log-Structured Merge-Tree (LSM-Tree)
  - P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil: "*The Log- Structured Merge-Tree (LSM-Tree)*", Acta Informatica, volume 33, number 4, pages 351–385, June 1996.
- BitCask implements hash indexes.
- LevelDB, RocksDB, Cassandra, Hadoop's HBase, Google's BigTable use similar approach.

B-Trees

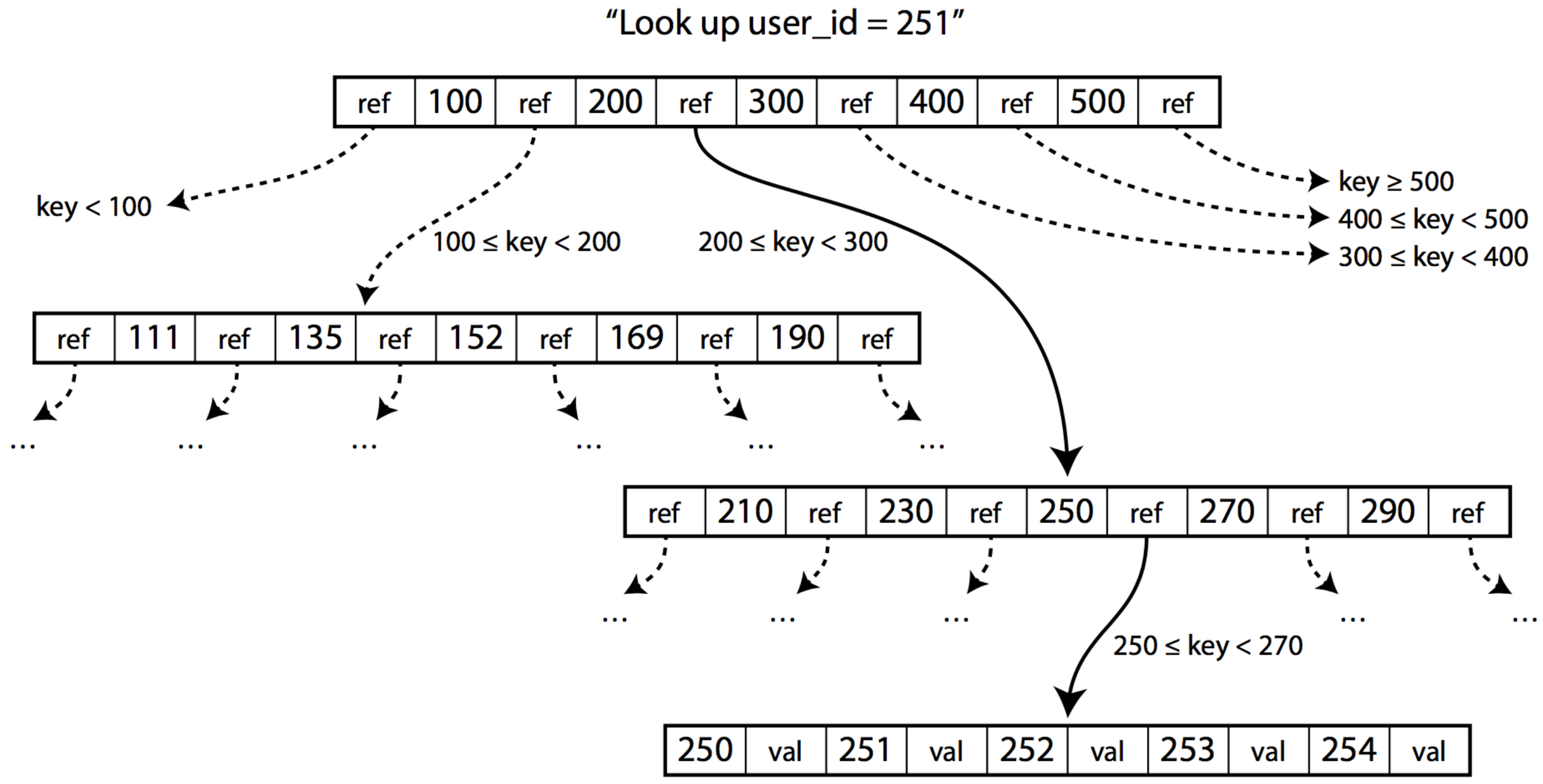
- The most widely-used indexing structure is quite different: the B-tree.

- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.

- SSTables break the database down into variable-size segments, typically several megabytes or more 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, and read or write one page at a time. This corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

B-Trees


"Look up user_id = 251"
ref
100
ref
200
ref
300
ref
400
ref
500
ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
300 ≤ key < 400
400 ≤ key < 500
key ≥ 500
ref
111
ref
135
ref
152
ref
169
ref
190
ref
ref
210
ref
230
ref
250
ref
270
ref
290
ref
250 ≤ key < 270
250
val
251
val
252
val
253
val
254
val

B-Trees

- **Update a value for an existing key**: search for the leaf page containing that key, and change the value that page, and write the page back to disk (any references to that page remain valid).

- **Add a new key**: 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 for key additions ensures that the tree remains balanced: a B-tree with *n* keys always has a height of *O(log n)*.

- B-tree implementations normally include a write-ahead log (WAL a.k.a. redo log). This is an append-only file to which 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.

- In-place updates required locks to avoid that a thread sees the tree in an inconsistent state.