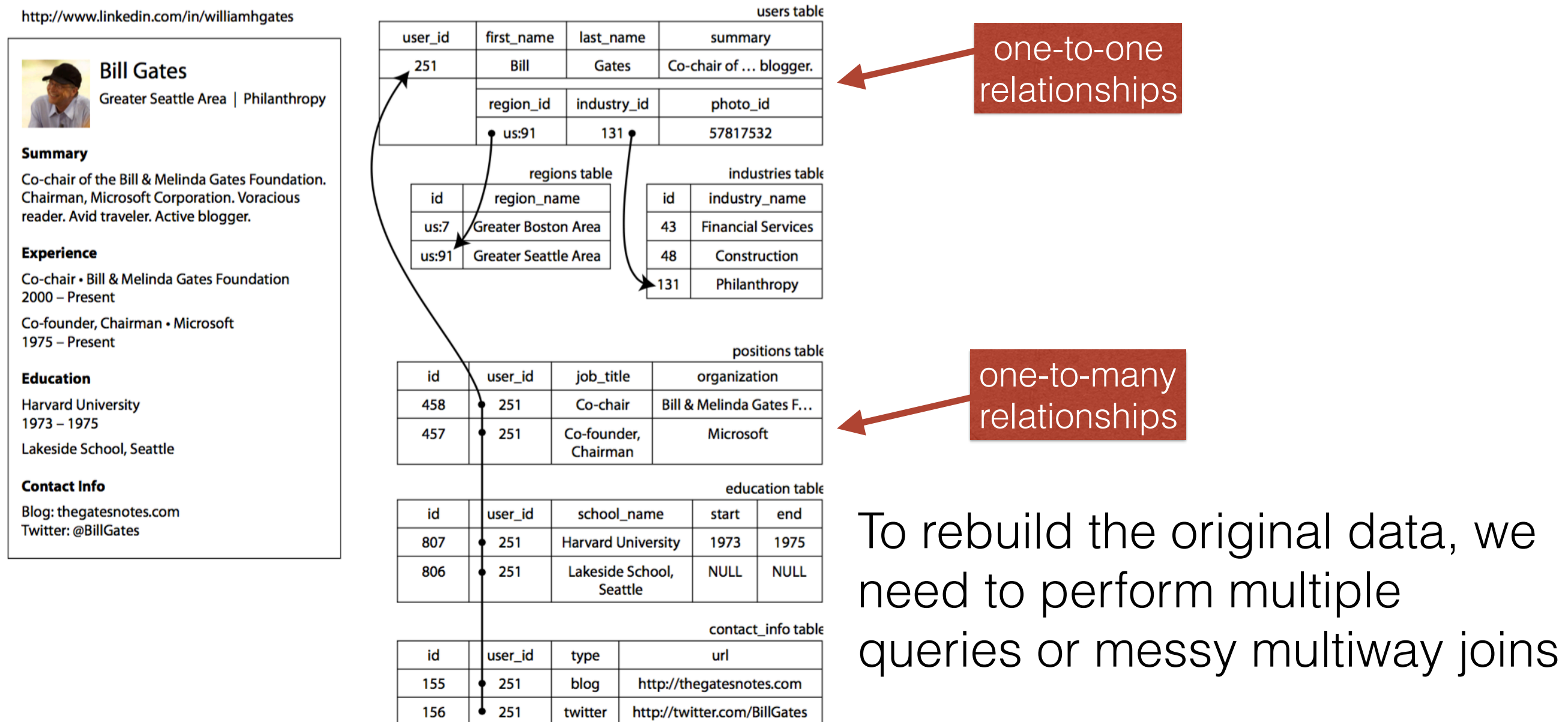


Relation Model

- 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.



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

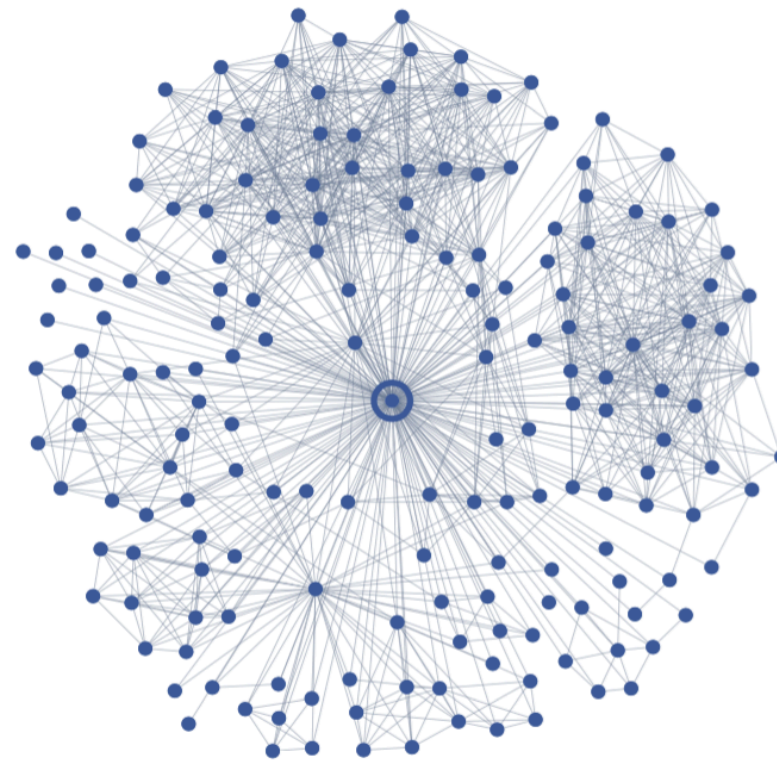
- A **graph** consists of two kinds of object: **vertices** (also known as *nodes* or *entities*) and **edges** (also known as *relationships* or *arcs*).
- Examples:
 - Social graphs: vertices are people, edges indicate which people know each other.
 - The Web graph: vertices are web pages, edges indicate 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/>



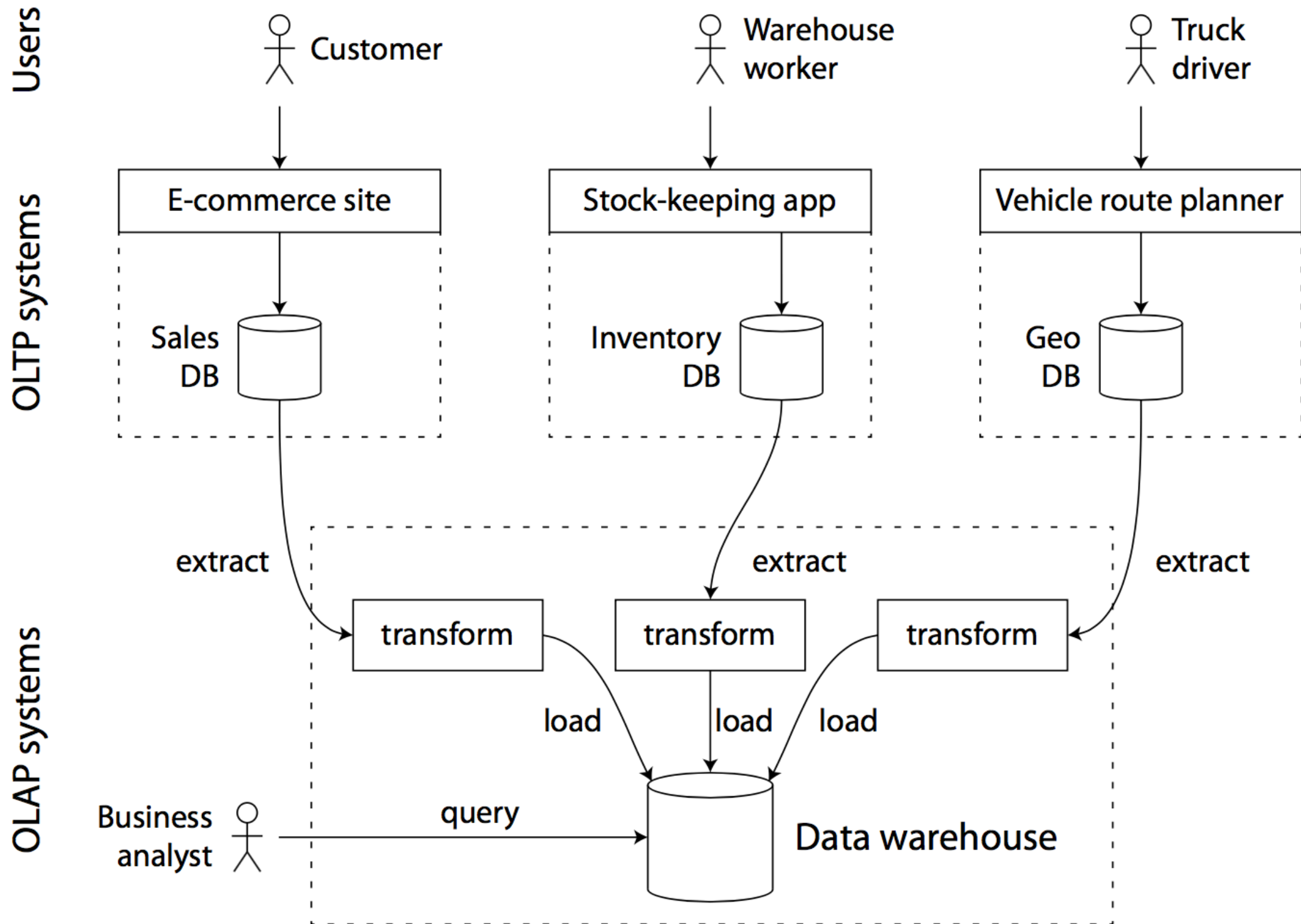
Edge-Type	#-out	in-id-type	out-id-type	Description
friend	<i>hundreds</i>	USER	USER	Two users are friends (symmetric)
likes	<i>a few</i>	USER	PAGE	pages (movies, businesses, cities, etc.) liked by a user
likers	10–10M	PAGE	USER	users who have liked a page
live-in	1000–10M	PAGE	USER	users who live in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



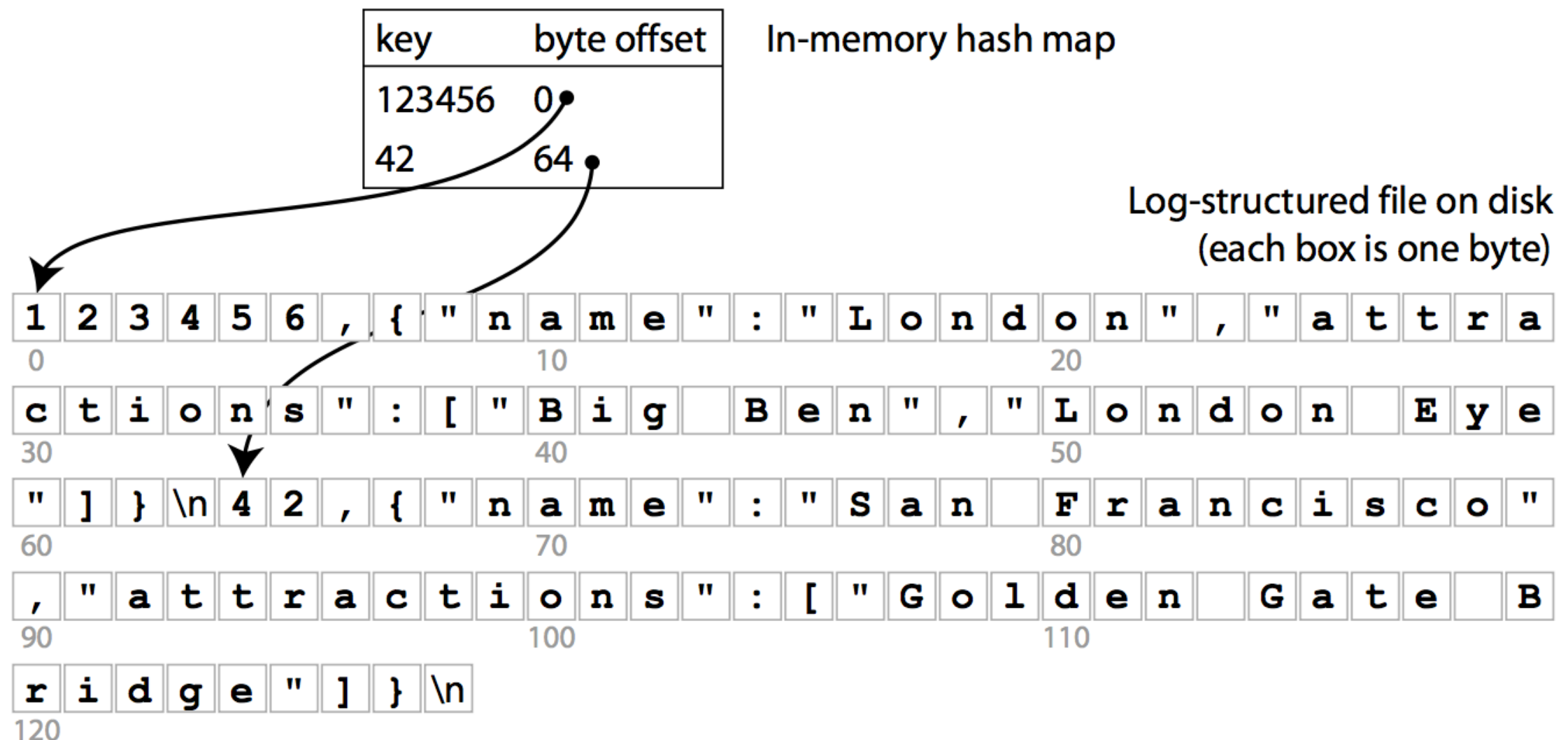
OLTP

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.

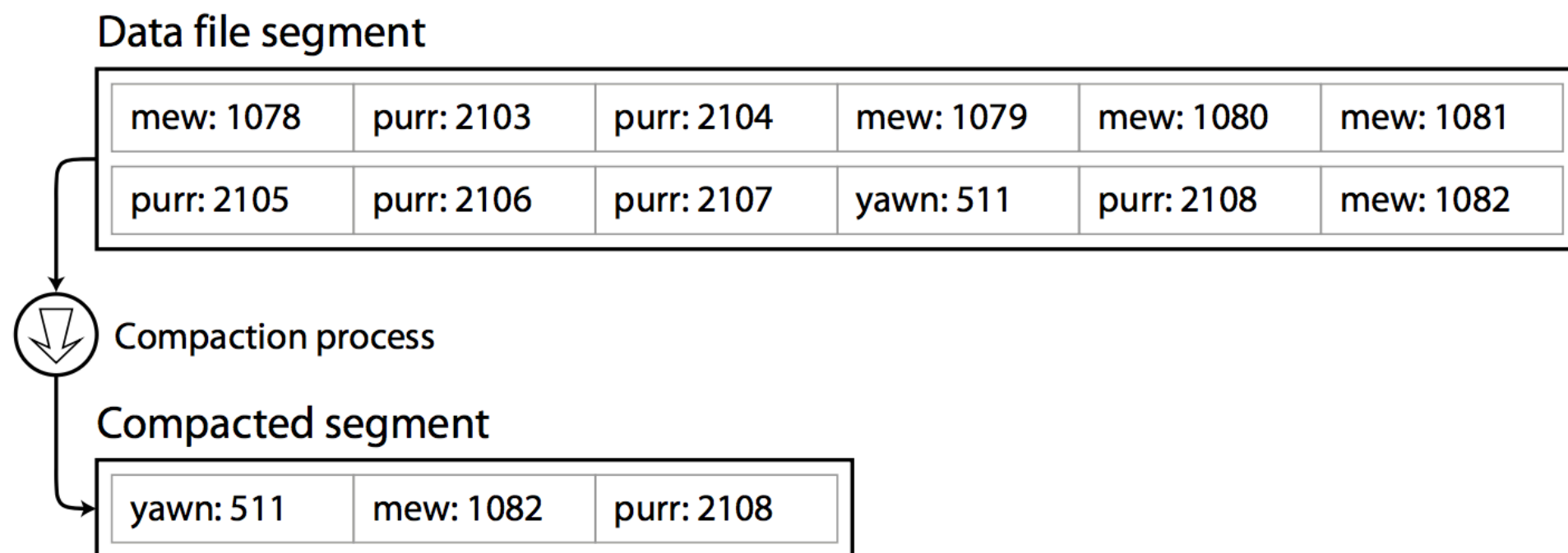
Hash Indexes

- 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 rebuild them by scanning the whole segments, or **snapshotting** 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.