Designing Data-Intensive Applications

Part 1: Storage and Retrieval

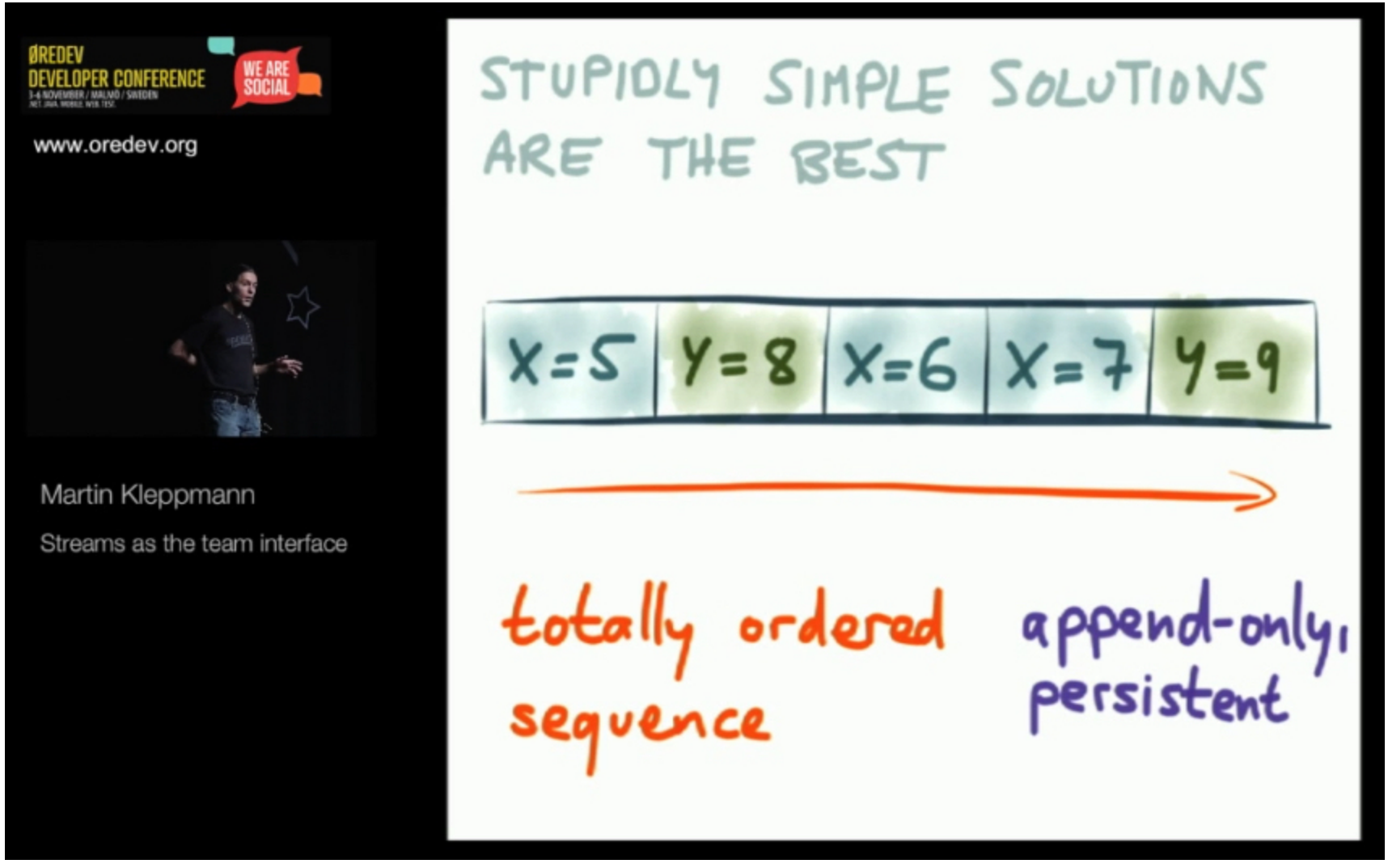
(Which really is chapter 3)

> Technology is a powerful force in our society. Data, software, and communication can be used for bad: to entrench unfair power structures, to undermine human rights, and to protect vested interests. But they can also be used for good: to make underrepresented people's voices heard, to create opportunities for everyone, and to avert disasters. This book is dedicated to everyone working toward the good.

This book has over 800 references to articles, blog posts, talks, documentation, and more...

Turning the database inside-out

https://martin.kleppmann.com/2015/11/05/database-inside-out-at-oredev.html



CAP is broken, and it's time to replace it

A Critique of the CAP Theorem

Martin Kleppmann

Abstract

The *CAP Theorem* is a frequently cited impossibility result in distributed systems, especially among *NoSQL* distributed databases. In this paper we survey some of the confusion about the meaning of CAP, including inconsistencies and ambiguities in its definitions, and we highlight some problems in its formalization. CAP is often interpreted as proof that eventually consistent databases have better availability properties than strongly consistent databases; although there is some truth in this, we show that more careful reasoning is required. These problems cast doubt on the utility of CAP as a tool for reasoning about trade-offs in practical systems. As alternative to CAP, we propose a *delay-sensitivity* framework, which analyzes the sensitivity of operation latency to network delay, and which may help practitioners reason about the trade-offs between consistency guarantees and tolerance of network faults.

1 Background

Replicated databases maintain copies of the same data on multiple nodes, potentially in disparate geographical locations, in order to tolerate faults (failures of nodes or communication links) and to provide lower latency to users (requests can be served by a nearby site). However, implementing reliable, fault-tolerant applications in a distributed system is difficult: if there are multiple copies of the data on different nodes, they may be inconsistent with each other, and an application that is not designed to handle such inconsistencies may produce incorrect results.

In order to provide a simpler programming model to application developers, the designers of distributed data systems have explored various consistency guarantees that can be implemented by the database infrastructure, such as linearizability [30], sequential consistency [38], causal consistency [4] and pipelined RAM (PRAM) [42]. When multiple processes execute operations on a shared storage abstraction such as a database, a consistency model describes what values are allowed to be returned by operations accessing the storage, depending on other operations executed previously or concurrently, and the return values of those operations.

Similar concerns arise in the design of multiprocessor computers, which are not geographically distributed, but nevertheless present inconsistent views of memory to different threads, due to the various caches and buffers employed by modern CPU architectures. For example, x86 microprocessors provide a level of consistency that is weaker than sequential, but stronger than causal consistency [48]. However, in this paper we focus our attention on distributed systems that must tolerate partial failures and unreliable network links.

A strong consistency model like linearizability provides an easy-to-understand guarantee: informally, all operations behave as if they executed atomically on a *single copy* of the data. However, this guarantee comes at the cost of reduced performance [6] and fault tolerance [22] compared to weaker consistency models. In particular, as we discuss in this paper, algorithms that ensure stronger consistency properties among replicas are more sensitive to message delays and faults in the network. Many real computer networks are prone to unbounded delays and lost messages [10], making the fault tolerance of distributed consistency algorithms an important issue in practice.

A *network partition* is a particular kind of communication fault that splits the network into subsets of nodes such that nodes in one subset cannot communicate with nodes in another. As long as the partition exists, any data modifications made in one subset of nodes cannot be visible to nodes in another subset, since all messages between them are lost. Thus, an algorithm that maintains the illusion of a single copy may have to delay operations until the partition is healed, to avoid the risk of introducing inconsistent data in different subsets of nodes.

This trade-off was already known in the 1970s [22, 23, 32, 40], but it was rediscovered in the early 2000s, when the web's growing commercial popularity made

Data-intensive = data is its primary challenge

- Quantity of data
- Complexity of data
- Speed of change

Opposed to compute-intensive where CPU cycles are the bottleneck

> [...]the term “Big Data” is so overused and underdefined that it is not useful in a serious engineering discussion.

Direction

- Companies need to handle huge volumes of data traffic
- CPU clock speeds are barely increasing
- Multi-core processors are standard
- Networks are getting faster
- Services are expected to be highly available

Chapter 3: Storage and Retrieval

A database needs to do two things

- when you give it some data, it should store the data
- when you ask it again later, it should give the data back to you

World’s simplest database, implemented as two Bash functions

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

```
> source db.sh; db_set hello world
> cat database
hello,world
> source db.sh; db_set hello foo
> cat database
hello,world
hello,foo
> source db.sh; db_get hello
foo
```

Good

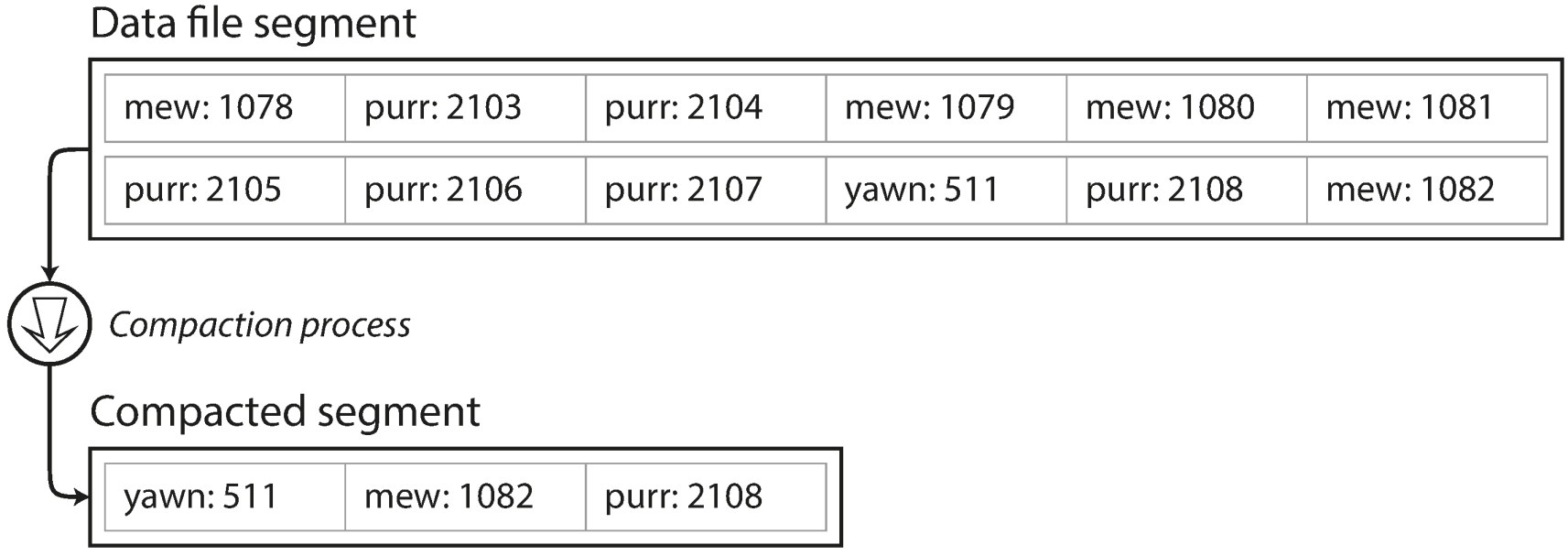
- Performance - appending to file is very efficient
    - Using a *log* (append-only) internally is common

Bad

- Update doesn't remove old data
- Read scans entire database
    - Double the number of records, twice as slow

How do we avoid running out of disk space?

- Break the log into segments of a certain size
- Make subsequent writes to a new segment file
- Perform *compaction* on the segments



Compaction

- Often makes segments much smaller (key is overwritten)
- Can be done on multiple segments at once
- Segments are never modified
- Merging and compactation can run on a background thread
- After merging, point to new segment, delete olds segments

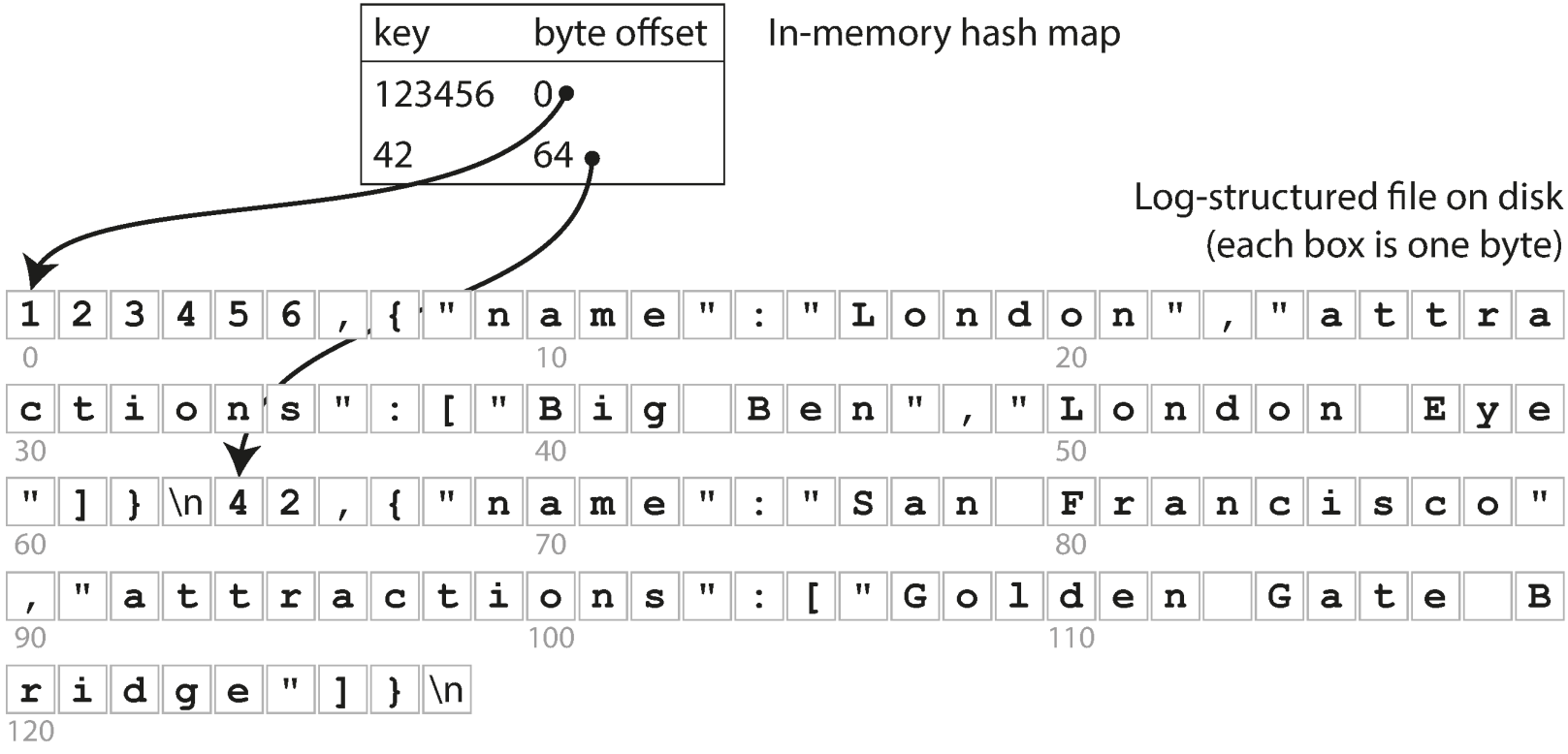
Speed up reads: Index

Index

- Additional structure derived from the primary data
- Add / remove indexes doesn't affect the content
- Only affects the performance of queries
- Well-chosen indexes speed up read quries
- Usually slows down writes - why not enabled by default
- Requires knowledge about application's typical query patterns
- Cost / benefit

Hash Indexes

- Keep an in-memory hash map where every key is mapped to a byte offset in the data file



Hash index

- Sounds simplistic but is a viable approach
- Essentially what Bitcask (default Riak storage engine) is doing
- Offers high-performance read and writes
- Suited when the value for each key is updated frequently
- Requires that keys fit into the available RAM

Index & compaction

- Each segment has its own in-memory hash map
  - Mapping key to offset
- On lookup, check the most recent segment's hash map
  - If not present, pick the second most recent (and so on)
- Merging process keeps the number of segments small
- Lookup doesn't need to check many hash maps

Improvements

- File format
  - Binary format is faster and simpler
- Deletions
  - Special "tombstone" record
- Crash recovery (in-memory hash map is lost)
  - Re-building is possible but slow
  - Bitcask store a snapshot on disk
- Partially written records
  - Bitcask include checksum
- Concurrency control
  - Common to have one writer thread

Good things

- Appending and segment merging are sequential write operations
  - Much faster than random writes, especially on spinning-disk
  - To some extent preferable on SSD (see book)
- Merging old segments avoid data files getting fragmented over time
- Immutable is good - no worries about crash during writes

Problems

- Hash map must fit into memory
  - Hash map on disk is difficult to make performant
- Range queries are not efficient
  - Can't find people between `age > 20 and < 50`
  - Every key must be looked up in hash map

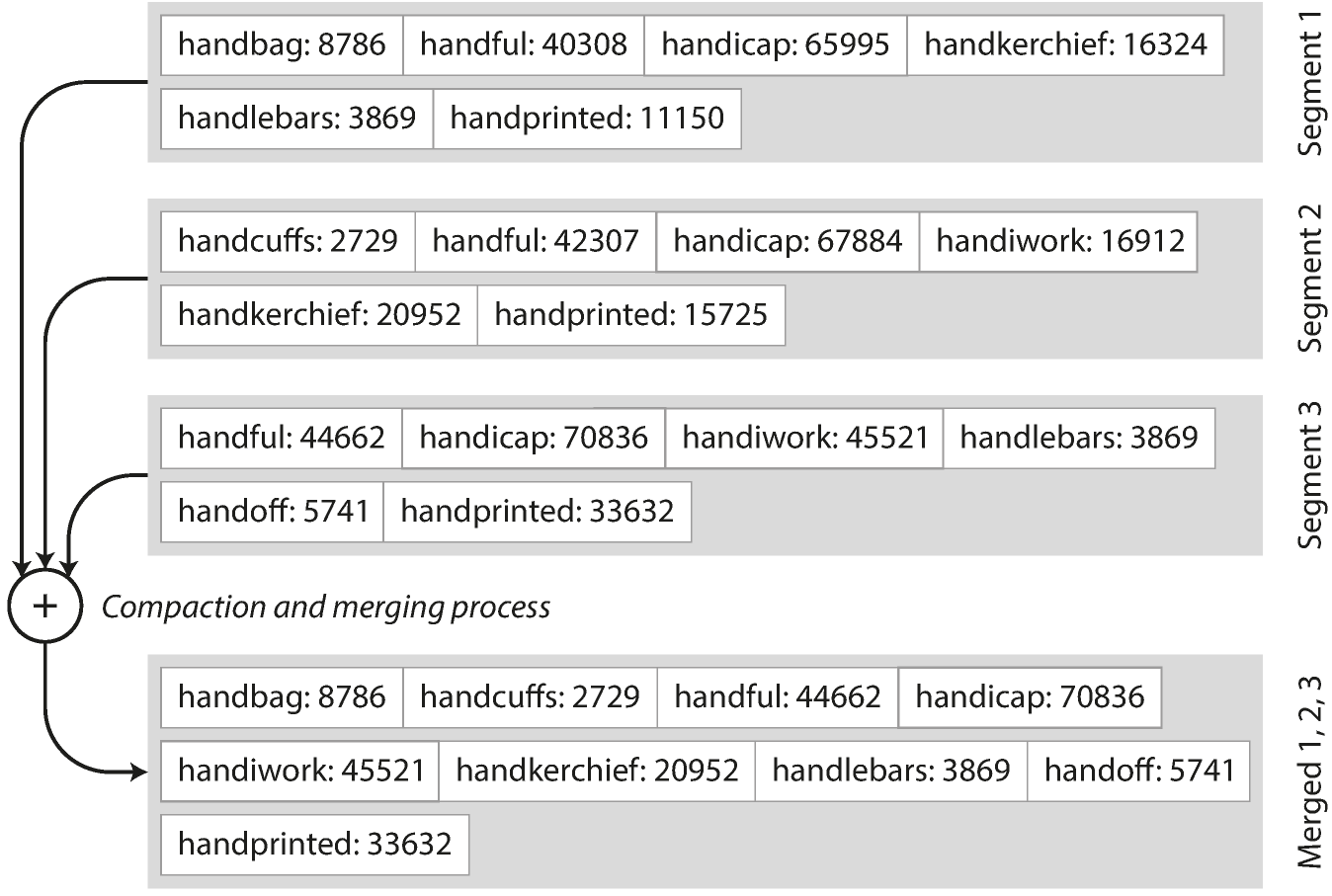
Sorted String Tables & LSM-Trees

(SSTables)

Simple change

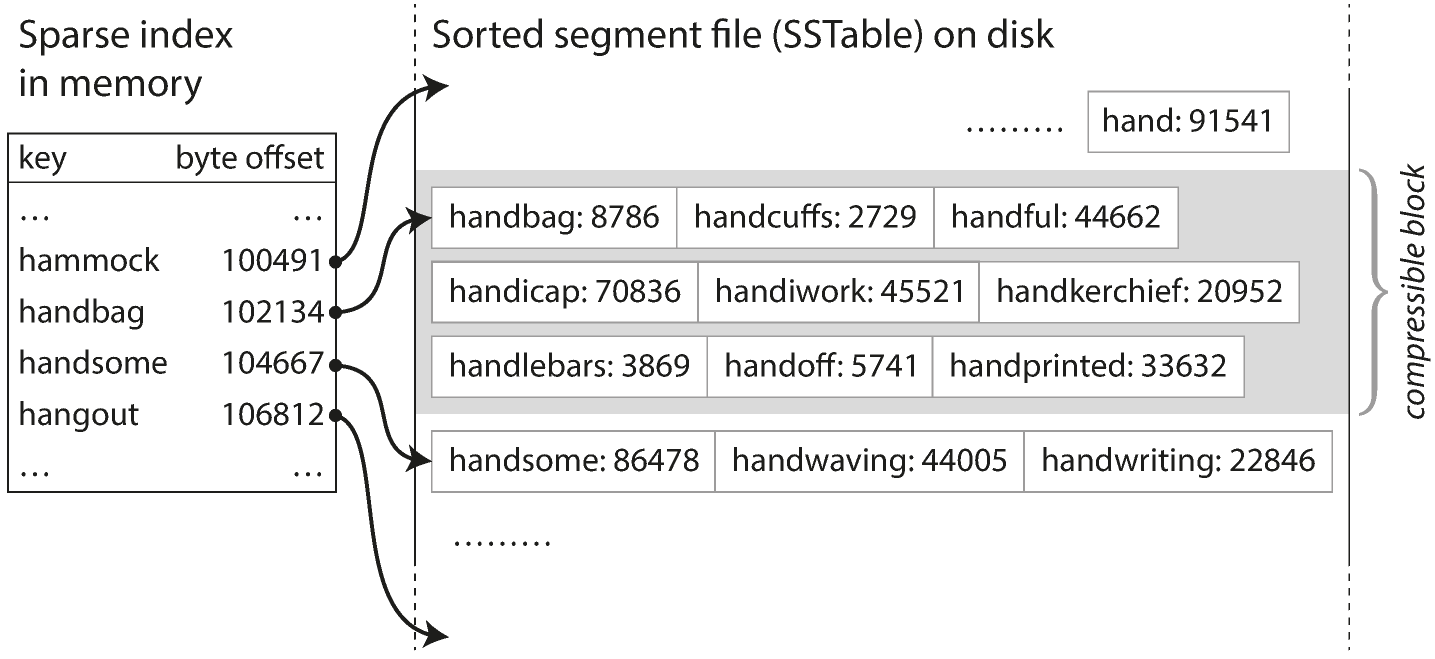
Require that the sequence of key-value pairs is sorted by key

Mergesort



- Copy the "lowest" key
- If identical - keep the value from the most recent segment

Find by key



- `handiwork` has unknown exact offset, but must be between `handbag` and `handsome`
- Jump to `handbag` and scan until found (or not)
- Still need in-memory index but it can be small & sparse
  - One key for every few kb (scanned quickly)

Improve scan on read request

- Group into a block and compress it before writing to disk
- Each entry of the sparse in-memory points to the start of a compressed block
- Saves disk space and reduces I/O bandwidth use

Constructing and maintaining SSTables

(Sorted segments)

- Maintaining on disk is possible (B-Trees later)
- Maintaining in memory is much easier
  - Weel known data structures: red-black trees / AVL trees
  - Insert key, get them back ordered
  - Sometimes called "memtable"
- Memtable gets bigger than some threshold (few mb)
  - Write it to disk as an SSTable file
  - Efficient - already sorted!
  - While writing to disk, start maintaining a new in memory

Running

- To serve read request
  - check memtable
  - most recent on-disk segment
  - next segment, and so on
- From time to time, run compaction in the background
- Data is sorted - efficient range queries
- Disk writes are sequential - high write throughput
- Stratergies for compaction & merge (size-tierd / leveled)

TL;DR

> 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.

Problem1: on crash the memory is wiped

- Maintain separate append-only log written immediately
- Not sorted, only used on restore
- When memtable is written to an SSTable, discard log

Problem 2: looking up non-existing keys

- Check memtable and ALL segment files
  - Use *Bloom filters* to approximate content of set

B-Trees

Log-structured indexes are gaining acceptance - but the most widely used is the B-tree

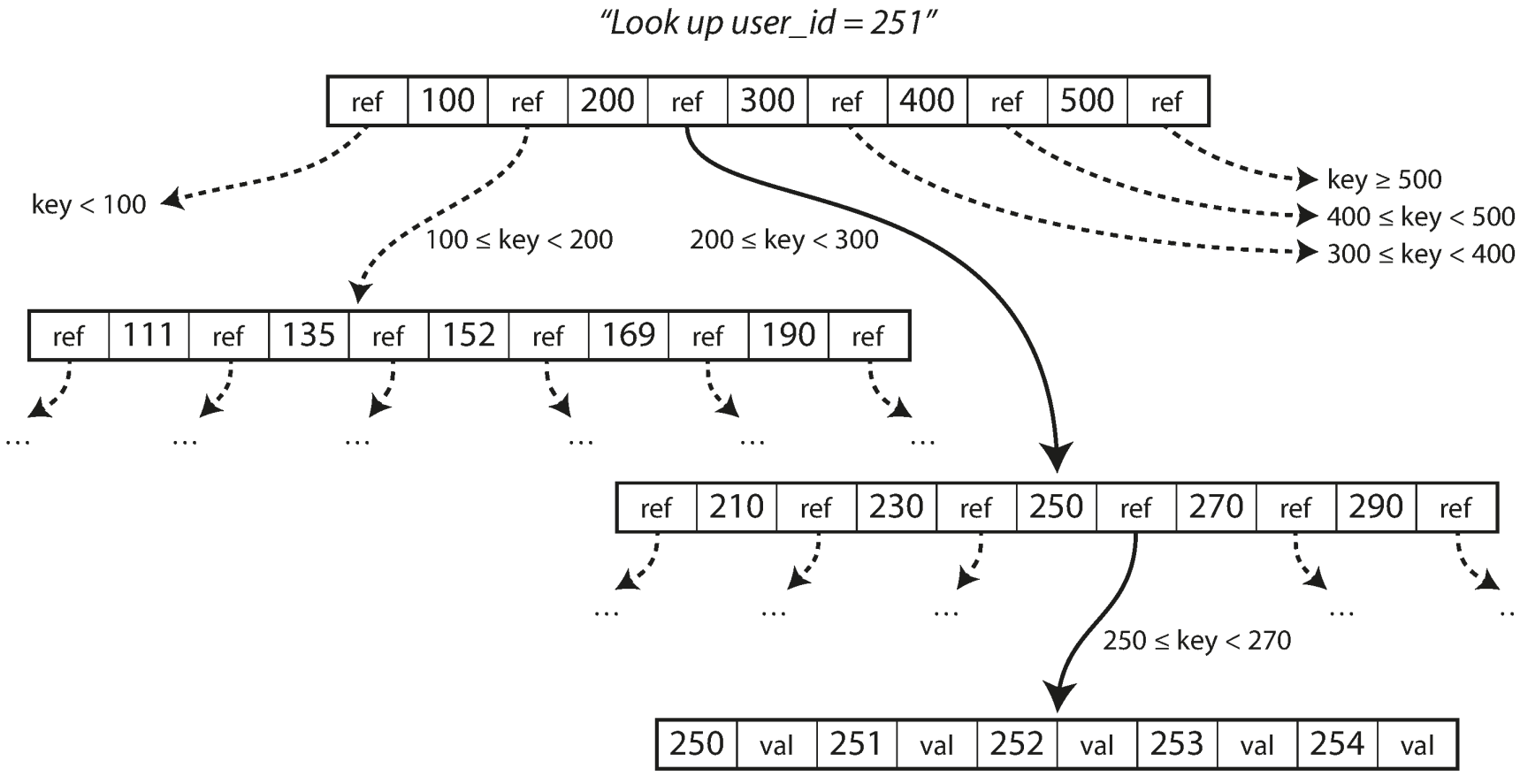
Similarities with SSTables

- Keep key-value pairs sorted by key
- Efficient lookup and range queries
- Otherwise very different

B-trees

- Break down database into fixed-size "blocks" or "pages"
- Read or write one page at a time
- Design corresponds to underlying hardware
    - Disks are also arranged in fixed size blocks
- Each page can be identified using an address or location
    - One page can refer to another (similar to pointers)

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
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
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

- One page is designated the root (lookup starts here)
- Page contains everal keys and references to child pages
- Each child is responsible for a continuous range of keys
- Keys between references indicate the boundaries

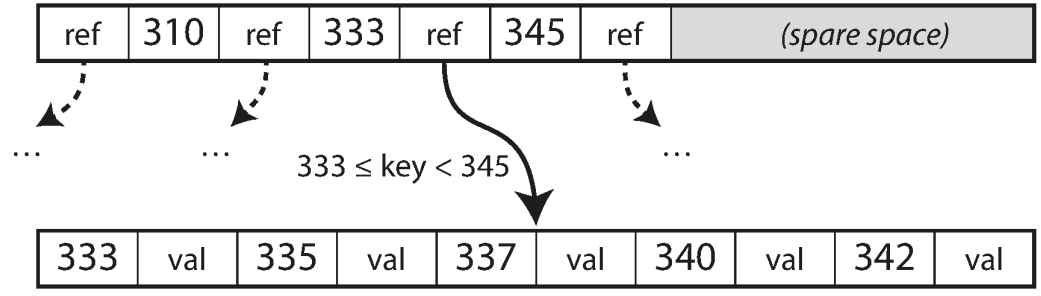
Updates

- Find the leaf page
- Change the value in that page and write page back to disk
- References remain valid

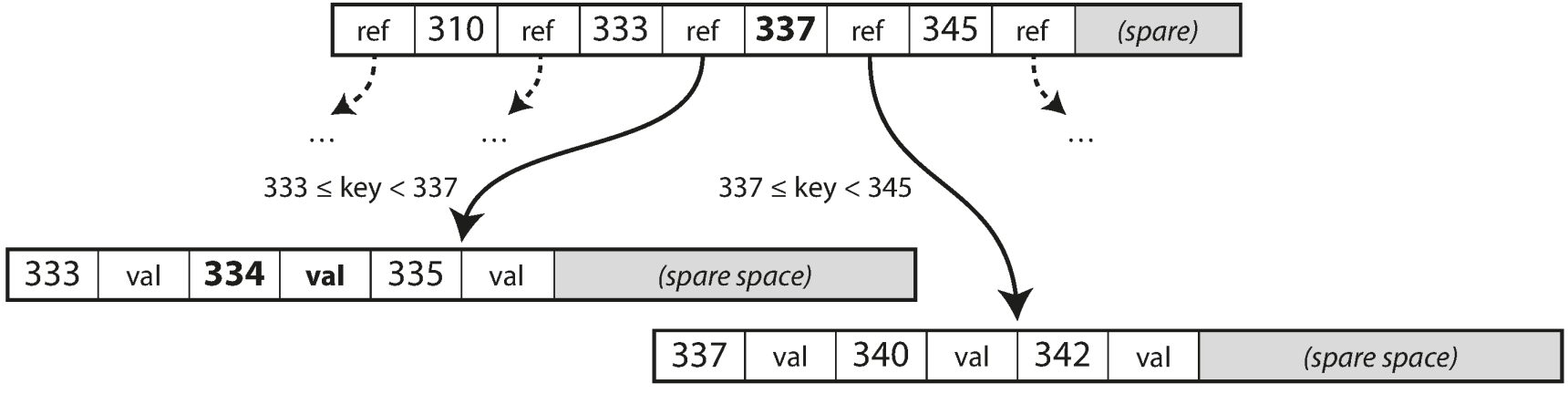
Add new key

- Find page whose ranges encompasses the new key and add it
- If there isn't enough free space
    - Split into two half-full pages
    - Update parent page
- Ensures the tree is balanced

Adding key 334



After adding key 334:



- 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
- A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB

Making B-trees reliable

- Basic underlying write operation is to overwrite a page with new data
- Assumed that the write does not change location of the page
- Some operation requires several pages be overwritten - dangerous on crash
    - Use *write-ahead log* (WAL, a.k.a. *redo log*)
    - Append only structure, written to first before te tree itself
    - Restore B-Tree with it after crash
- Concurrency control (update in place)
    - Use *latches* (lightweight locks)
    - More complicated than logs

B-tree optimizations

- Copy-on-write (instead of WAL for crash recovery)
- Save space by not storing the entire key
  - Only need to provide enough information to act as boundries between key ranges
  - Packing more keys into a page - high branching factor, fewer levels
- Lay out tree so leaf appear in sequential order on disk
  - Difficult to maintain when the tree grows
- Additional pointer (sibling references) for faster scan
- B-tree variants such as *fractal trees* borrow some log-structure ideas to readuce disk seeks

Comparing B-Trees & LSM-trees

Advantages of LSM-trees

- B-tree index must write all data at least twice
  - Write-ahead log (WAL)
  - Actaul tree page (and perhaps again if pages are split)
- B-tree has overhead for writing an entire page at a time
  - Some even overwrite twice to avoid partially updated pages on power failure
- (Although log also rewrite several times, *write amplification*)
- Typically higher write throughput (lower write amplification)
  - Mostly on magnetic drives where sequential writes are fast
- Better compression (smaller files on disk)
- Less fragmentation (B-tree split, page space remain unused)

Downsides of LSM-trees

- Compaction can interfear with performance (read & write)
- Response time of queries can be high (B-trees are more predicatable)
- Disk's finite bandwidth is shared between compaction and write
- On high throughput, compaction won't keep up
- LSM-trees store multiple copies of the same key
- B-trees has "built in" support for transaction isolation because locks can be attached to the tree

Secondary indexes

Same thing, but keys are not unique

- Store a list of matching row identifiers
- Make key unique by appending row identifier

Storing values with the index

- Value can be either:
  - the actual row (document, vertex)
  - a reference to the row stored elsewhere (heap file)

Heap file

- Heap file is common - no duplicate data on secondary indexes
- On update (larger value)
  - Move to a new location in the heap file
  - all indexes need to be updated
  - or, forward pointer is added to the heap file

If extra hop to heap file is too expensive

- Use *clustered index*, store row with index (MySQL's InnoDB)
- Primary key is always a clustered index
- Secondary indexes refer to the primary key (not heap file)
- Why Uber switched from Postgres to MySQL
  https://eng.uber.com/mysql-migration/

Covering index

- Stores *some* of a table's columns with the index
- Allows *some* queries to be answered using the index alone

Limitations

Problem 1: Multi-column indexes

```
SELECT * FROM restaurants
WHERE latitude  > 51.4946 AND latitude  < 51.5079
  AND longitude > -0.1162 AND longitude < -0.1004;
```

- B-tree or LSM-tree can't answer this query efficiently
- Only all resturants in a longitude range (but anywhere between the North and South pole)

Multi-column indexes

- Single number space-filling curve (and then regular B-tree)
- Specialized spatial indexes (R-trees)
- 2D index

Problem 2: full-text search, fuzzy index

See book (references)

Or - just keep everything in memory

- RAM becoms cheapter (cost-per-gigaby argument eroded)
- Restart?
  - Special hardware: battery-powered RAM
  - Write a log of changes / periodic snapshot to disk
- Performance not due to not reading from disk!
  - Even disk based may never read from disk (cached blocks)
  - Real reason: overhead of encoding in-memory data structures to disk format
- Other possibility - *anti-caching*, evict least recently used to disk
- Non-volatile memory (NVM) - keep an eye in the future

Transaction Processing vs Analytics

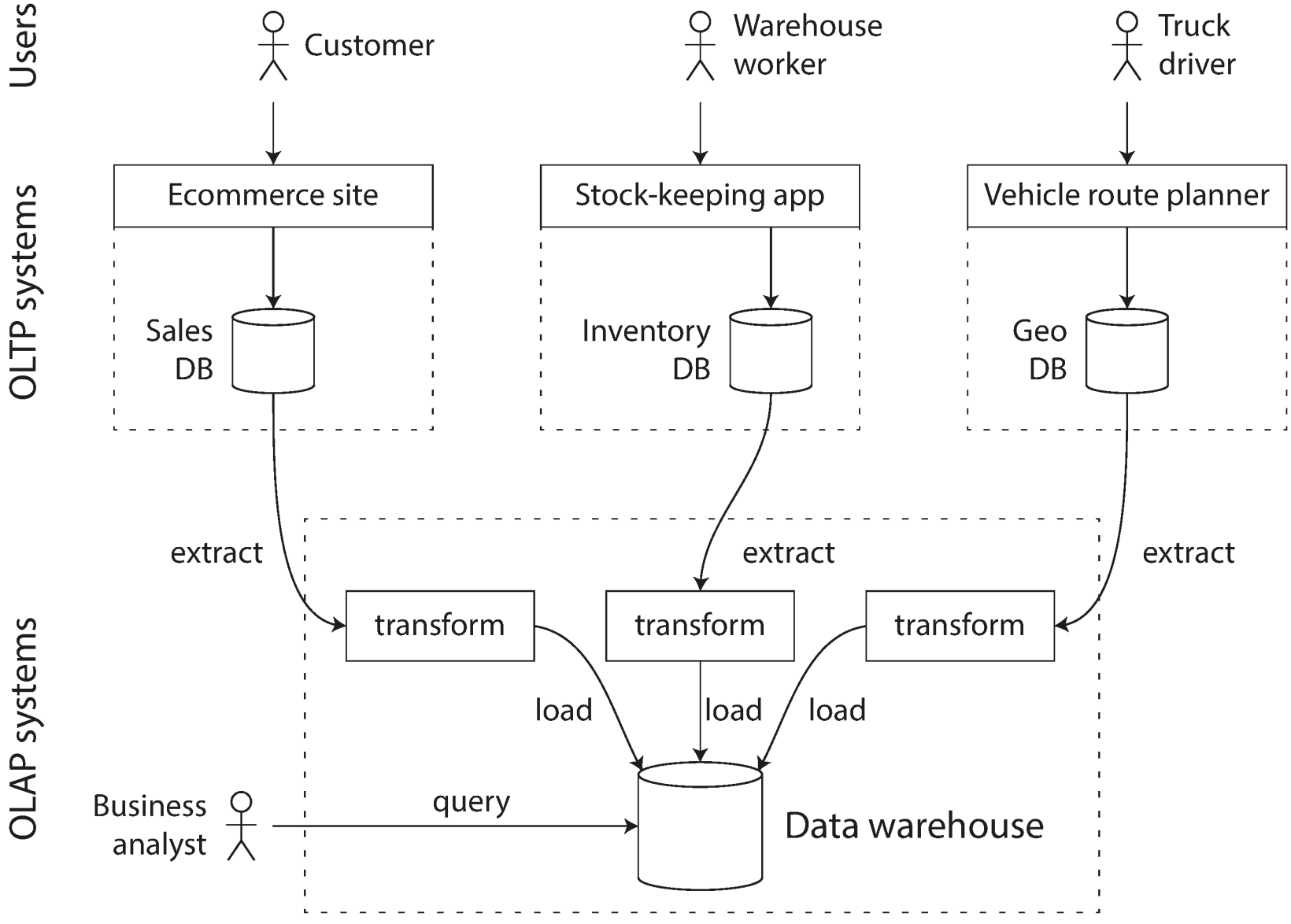
Typical application

- Looks up small number of records by key (indexed)
- Records are inserted or updated based on the user's input
- Applications are interactive
- For end user / customer (via web application)
- Latest state of data (current point in time)
- Gigabytes to terabytes
- Highly available / low latency
- OnLine Transaction Processing (OLTP)

Data analytics

- Query needs to scan over many records
- Only reading a few columns per record
- Calculates aggregate statistics (`count` , `sum` , `average` , ...)
- Bulk import (Extract Transform Load - "ETL") or event stream
- For analyst / decision support / report to management
- History of events that happened over time
- Terabytes to petabytes
- Read only copy
- OnLine Analytic Processing (OLAP)

ELT - Extract Transform Load



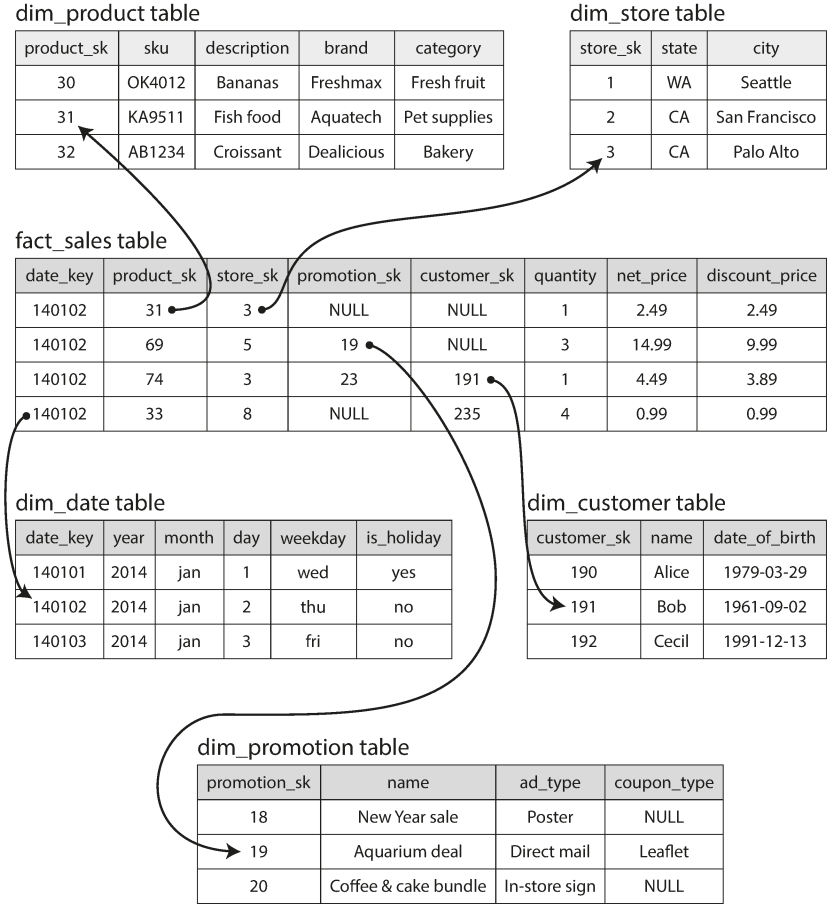
Data warehousing

- Historically used the same database
- Separate database "data warehouse"
- Commonly relational databases
- SQL quite flexible for OLTP and OLAP
    - Graphical query generating tools
    - "Drill-down" & "slicing and dicing"
- Index algorithm (previously discussed) not very good
- Need storage engines optimized for analytics instead

Stars & Snowflakes

Star schema (dimensional modeling)

Fact table at the center



dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Fact tables

- Each row represents an event
- The dimensions represent the *who*, *what*, *where*, *when*, *how*, *and why* of the event
- Each row represent an event at a particular time (e.g. purchase)
- Maximum flexibility for analysis
- Table can become extremely large
- Some columns are attributes (e.g. price)
- Other columns are foregin key references (dimension tables)
- Typically very wide

Snowflake schema

- Dimensions are futher broken down into subdimensions
- More normalized (than star schemas)
- Harder to work with

Column-Oriented Storage

- Fact tables are often 100 columns wide
- A typical data warehouse query only accesses 4 or 5

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date
  ON fact_sales.date_key   = dim_date.date_key
  JOIN dim_product
  ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

Column-Oriented Storage

- OLTP databases usually store in row-oriented fashion
- All values from one row are stored next to each other
- Document databases are similar
- Problematic with previous query - all data needs to be scanned
- Solution: store each column together instead!
- Also works in nonrelational data models
- Reassemble a row is problematic (but rarely needed)

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

Compression

- Column-oriented storage often good for compression
- Many repeating column values (previous image)
- Save disk space
- Efficient use of CPU cycles
- Retail typically have billions of sale but only 100,000 products
- Bitmap encoding

Sort order in column storage

- Doesn't necessarily matter - easiest to store in insert order
- Can't sort each column independently (reconstruct row)
  - Must sort entire row
- Queries often target date ranges - make it the sort key
- Several sort keys can be used (telephone book)
- Also helps compression (mostly on first sort key)
- Store same data in several different ways
  - Data needs replication anyway

Writing to column-oriented storage

- Writes are more difficult
- Update-in-place (B-trees) is not possible
- LSM-treees works
    - Writes first go in-memory store
    - Added to a sorted structured, prepared for disk write
    - Doesn't matter if it's row-oriented or column-oriented

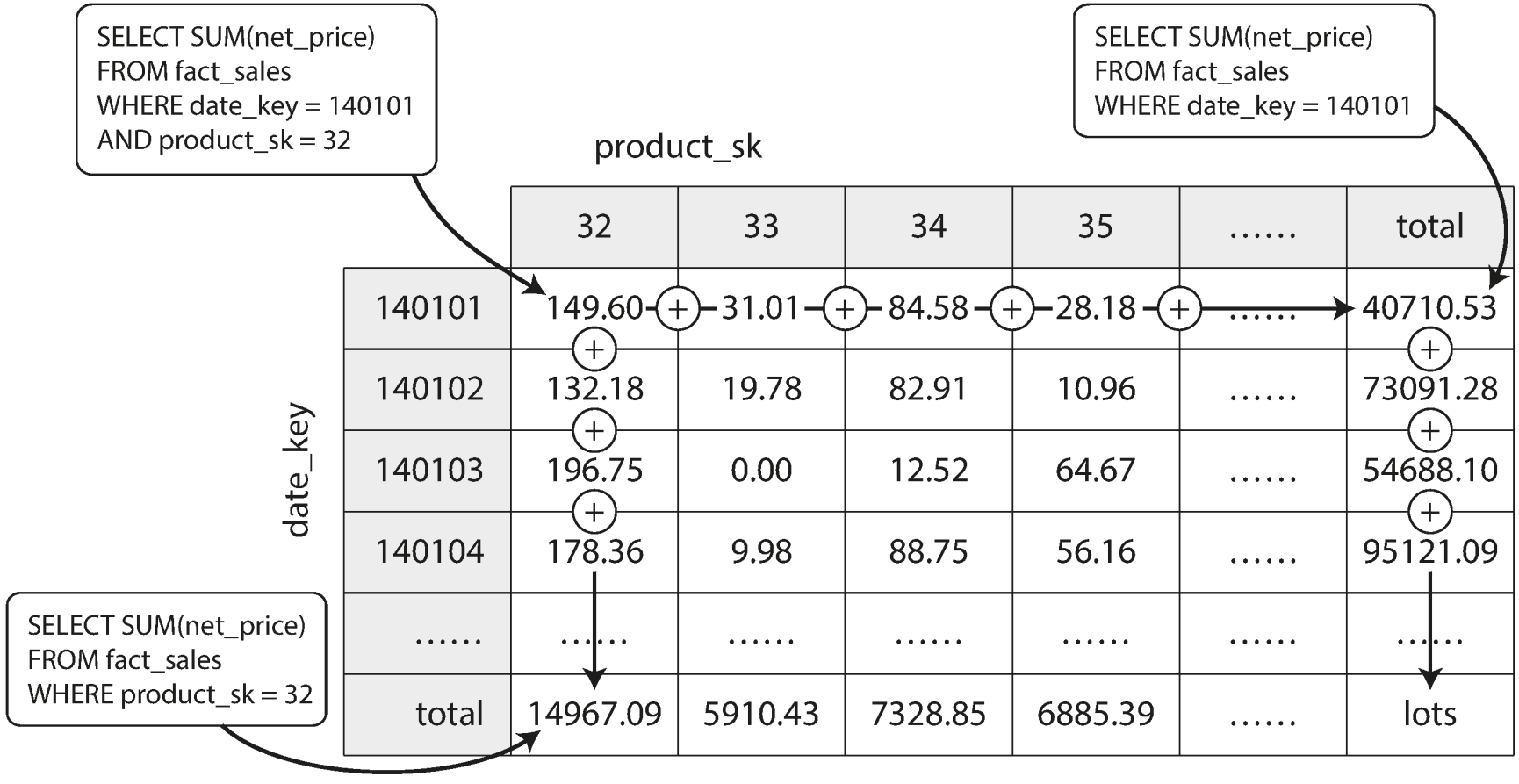
Data cubes & materialized views

Materialized aggregates

- Queries often use `count`, `sum`, `avg`, `min` or `max`
- Wasteful to crunch if used in many queries
- Materialized views - copy of query results written to disk
  - Needs to be updated (not a "virtual view")
  - Makes sense in OLAP (not in OLTP)

Data cube / OLAP cube

Grid of aggregates group by different dimensions



Data cube / OLAP cube

- Can have many dimensions (e.g. five-dimensional hypercube)
- Hard to imagine but same principle

Advantages

- Certain queries become very fast (precomputed)

Disadvantages

- Not same flexibility as raw data
  - E.g. sales from items which cost more than $100 (price isn't a dimension)

So...

- Data warehouses typically keep as much raw data as possible
- Aggregates (data cubes) only as performance boost

Summary

OLTP

- Typically user-facing
- Hughe volume of requests
- Touch a small number of records in each query
- Requests records using some kind of key
- Storage engine uses and index to find data
- Disk seek time is the often the bottleneck

Data warehouse

- Analytics systems are less well known
- Primary used by business analysts - not end users
- Lower volume of queries
- Queries are typically very demanding
- Column-oriented storage is increasingly popular solution
- Disk bandwidth (not seek time) is bottleneck
- Indexes irrelevant
- Important to encode data very compactly
    - Minimize data needed to be read from disk
    - Column-oriented storage helps this

Seek time: time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written

Bandwidth: the bit-rate of available or consumed information capacity expressed typically in metric multiples of bits per second

Log-structures

- Only permits
  - Appending to files
  - Deleting obsolete files
- Never updates a file (files are immutable)
- Bitcask, SSTables, LSM-trees, LevelDB, Cassandra, HBase, Luecene, ...
- Comparatively recent development
- Turn random-access writes to sequential writes on disk
  - Higher throughput

Update-in-place

- Treats the disk as a set of fixed-size pages
- Pages can be overwritten
- B-trees is the most common
- Used in "all" major relational databases and many non-relational

Also...

- More complicated index structures
- Databases optimized for keeping all data in memory

With this knowlege

- You know the internals of stage engines
- Which tool is best suited for your application
- Adjust the tuning of a database
- A vocabulary to make sense of the documentation

THE END