Reliability, Scalability, and Maintainability

Challenges

- **Reliability**: The system should continue to work *correctly* in response to hardware, software, or user errors

- **Scalability**: As the system grows (data volume, traffic, or complexity) there should be reasonable ways of dealing with that growth. The design should be *elastic.*

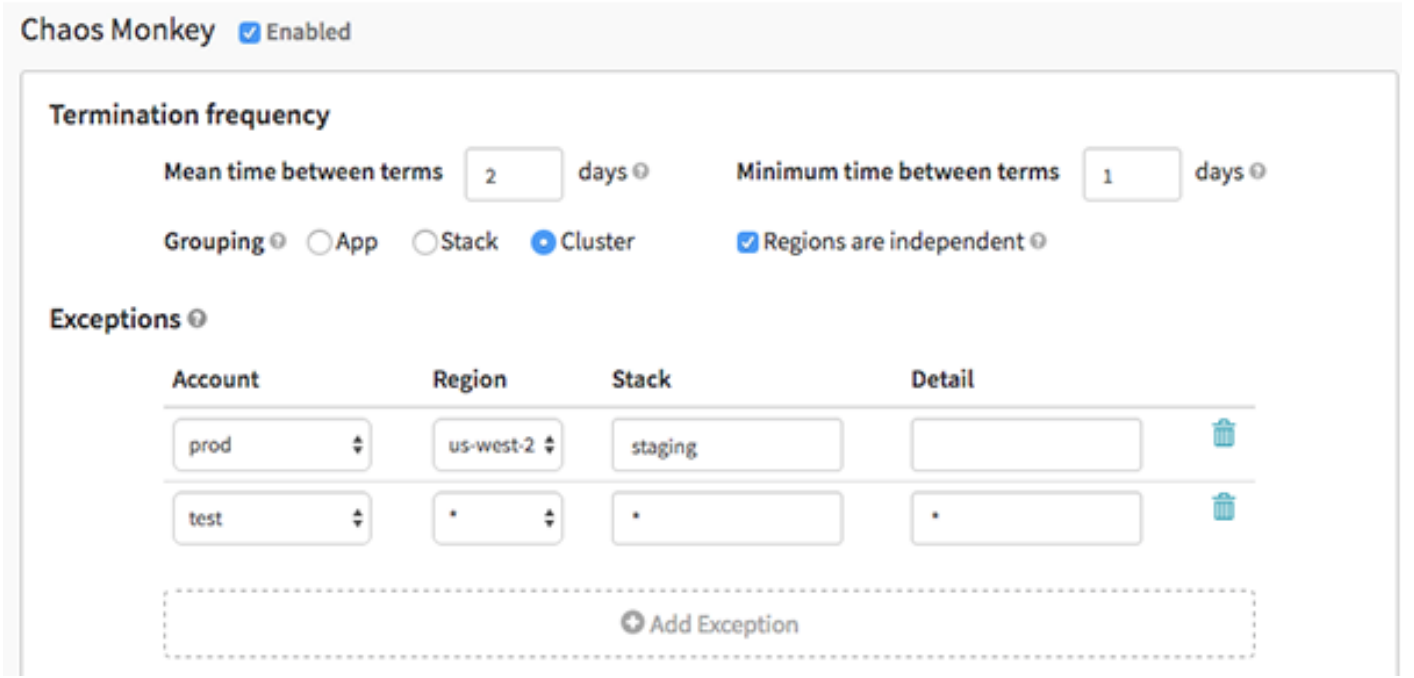
- **Maintainability**: Productively work on the system as people come and go. Add new features to adapt to changing requirements.

What is reliability?

- **Predictable**: Application performs as expected
- **Fault-tolerant:** Application copes with human error & hardware failures.
- **Performant**: Performance is adequate under expected load and data volume
- **Secure**: System prevents unauthorized access

Reliability

- **Faults** are a deviation in the spec of one particular component of the system. **Failures** cause the whole system to stop, requiring user intervention.
- A system is *fault-tolerant* if it anticipates and copes with certain kind of faults.
- One way to design fault-tolerant systems is to induce them deliberately! (e.g., submit bad input by the user, shut down a process, etc.)

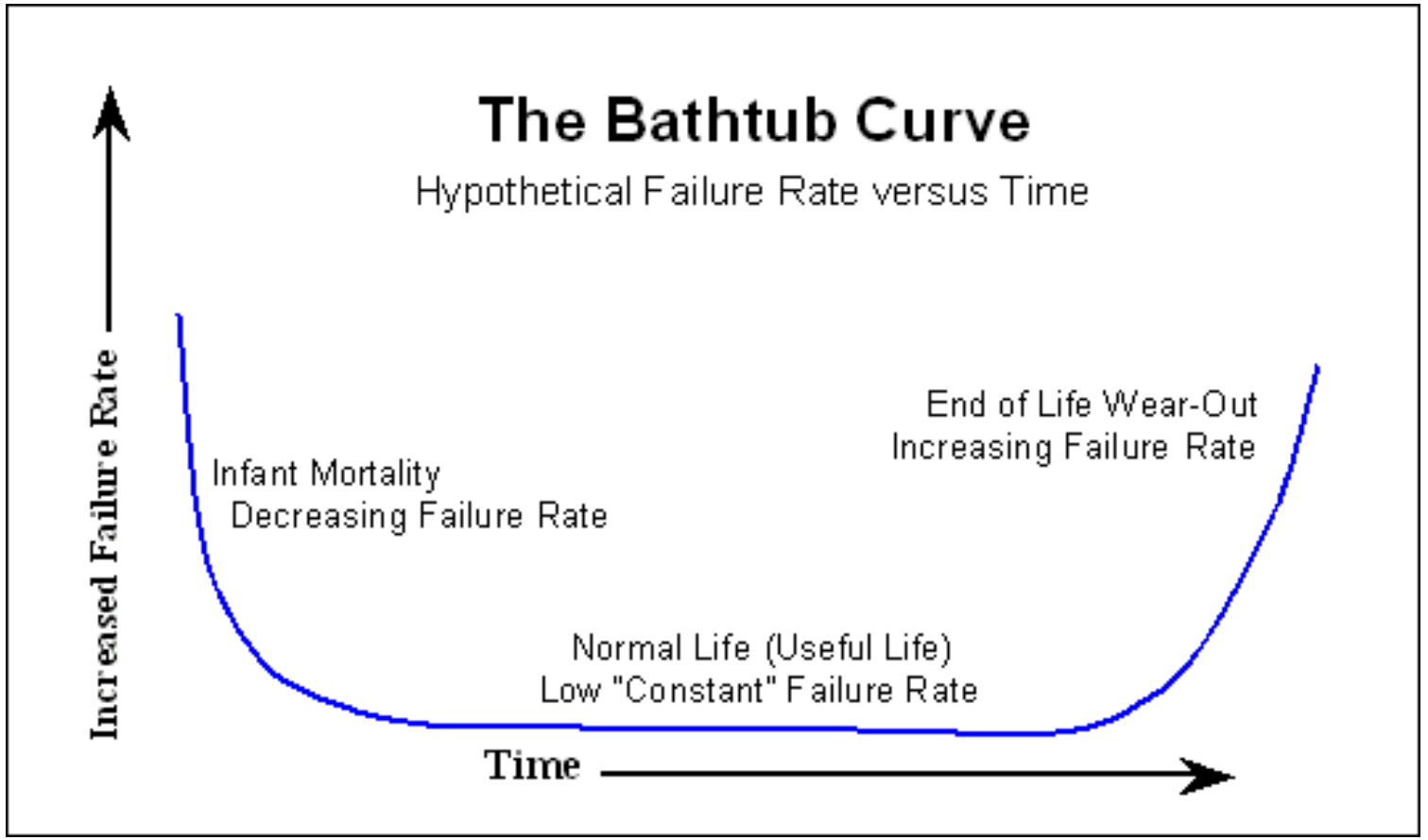


Netflix Chaos Monkey

Reliability: Hardware failures

MTTF: Mean Time to Failure (irreparable)

MTBF: Mean Time Between Failures (repair and return to service)



Reliability: Hard-disks

Hard Drive Annualized Failure Rates for 2016

Reporting period 1/1/2016 - 12/31/2016 inclusive

MFG	Model	Drive Size	Drive Count	Avg Age (months)	Drive Days	Drive Failures	Failure Rate
HGST	HUH728080ALE600	8 TB	45	22.99	16,155	-	0.00%
Seagate	ST8000DM002	8 TB	8,660	4.72	1,075,720	48	1.63%
Seagate	ST8000NM0055	8 TB	60	1.44	1,560	-	0.00%
Seagate	ST6000DX000	6 TB	1,889	21.48	684,840	16	0.85%
WDC	WD60EFRX	6 TB	446	24.14	166,152	25	5.49%
Toshiba	MD04ABA500V	5 TB	45	22.15	16,425	1	2.22%
HGST	HDS5C4040ALE630	4 TB	2,625	45.35	987,011	14	0.52%
HGST	HMS5C4040ALE640	4 TB	7,014	29.48	2,579,698	28	0.40%
HGST	HMS5C4040BLE640	4 TB	9,407	15.51	2,436,130	34	0.51%
Seagate	ST4000DM000	4 TB	34,738	21.73	12,359,750	938	2.77%
Seagate	ST4000DX000	4 TB	184	38.54	72,615	27	13.57%
Toshiba	MD04ABA400V	4 TB	146	20.61	52,983	-	0.00%
WDC	WD40EFRX	4 TB	75	17.16	16,790	1	2.17%
HGST	HDS5C3030ALA	3 TB	4,476	55.87	1,647,137	34	0.75%
HGST	HDS723030ALA	3 TB	978	61.21	361,937	22	2.22%
Toshiba	DT01ACA300	3 TB	46	44.12	16,900	2	4.32%
WDC	WD30EFRX	3 TB	1,105	30.39	390,379	35	3.27%
		Totals	**71,939**		**22,882,182**	**1,225**	**1.95%**

BACKBLAZE

AFR =
(Drive Failures / Drive Days) X 365 X 100%

Hard-disk MTTF: 10-50 years
(approx. one disk failure per day with 10,000 disks)

Reliability: Hard Drives

AFR =

(Drive Failures / Drive Days) X 365 X 100%

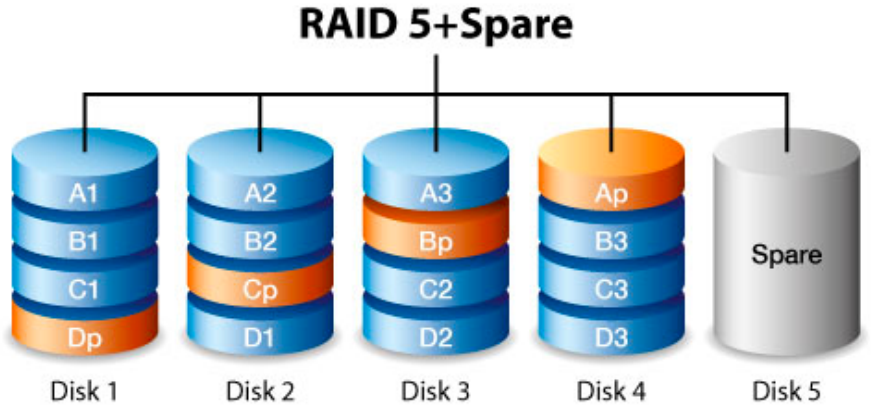
Backblaze Hard Drive Failure Rates for 2020

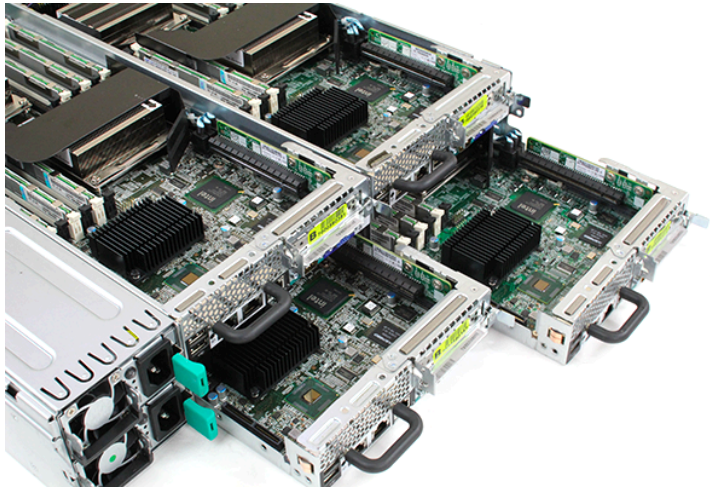
Reporting period 1/1/2020 - 12/31/2020 inclusive

MFG	Model	Drive Size	Drive Count	Avg Age (months)	Drive Days	Drive Failures	AFR
HGST	HMS5C4040ALE640	4TB	3,100	56.65	1,083,774	8	0.27%
HGST	HMS5C4040BLE640	4TB	12,744	50.43	4,663,049	34	0.27%
HGST	HUH728080ALE600	8TB	1,075	34.85	372,000	3	0.29%
HGST	HUH721212ALE600	12TB	2,600	15.04	820,272	7	0.31%
HGST	HUH721212ALE604	12TB	2,506	3.78	275,779	9	1.19%
HGST	HUH721212ALN604	12TB	10,830	21.01	3,968,475	50	0.46%
Seagate	ST4000DM000	4TB	18,939	62.35	6,983,470	269	1.41%
Seagate	ST6000DX000	6TB	886	68.84	324,275	2	0.23%
Seagate	ST8000DM002	8TB	9,772	51.07	3,584,788	91	0.93%
Seagate	ST8000NM0055	8TB	14,406	41.34	5,286,790	177	1.22%
Seagate	ST10000NM0086	10TB	1,201	38.73	439,247	16	1.33%
Seagate	ST12000NM0007	12TB	23,036	29.78	11,947,303	339	1.04%
Seagate	ST12000NM0008	12TB	19,287	9.76	5,329,149	148	1.01%
Seagate	ST12000NM001G	12TB	7,130	6.08	1,296,149	30	0.84%
Seagate	ST14000NM001G	14TB	5,987	2.89	454,090	13	1.04%
Seagate	ST14000NM0138	14TB	360	1.56	5,784	0	0.00%
Seagate	ST16000NM001G	16TB	59	12.93	21,323	1	1.71%
Seagate	ST18000NM000J	18TB	60	3.27	5,820	2	12.54%
Toshiba	MD04ABA400V	4TB	99	67.29	36,234	0	0.00%
Toshiba	MG07ACA14TA	14TB	21,046	7.65	4,103,823	102	0.91%
Toshiba	MG07ACA14TEY	14TB	160	1.22	2,562	0	0.00%
Toshiba	MG08ACA16TEY	16TB	1,014	2.14	33,774	0	0.00%
WDC	WUH721414ALE6L4	14TB	6,002	1.68	229,861	1	0.16%
		Totals	**162,299**		**51,267,791**	**1,302**	**0.93%**

Machine-Level Redundancy

- The most basic approach to increasing reliability
- A system can tolerate the unplanned shutdown of an entire node or machine
- Enables *rolling upgrades*
- AWS: whole instances can become unavailable without warning (platforms prioritize flexibility and elasticity over single-machine reliability.)





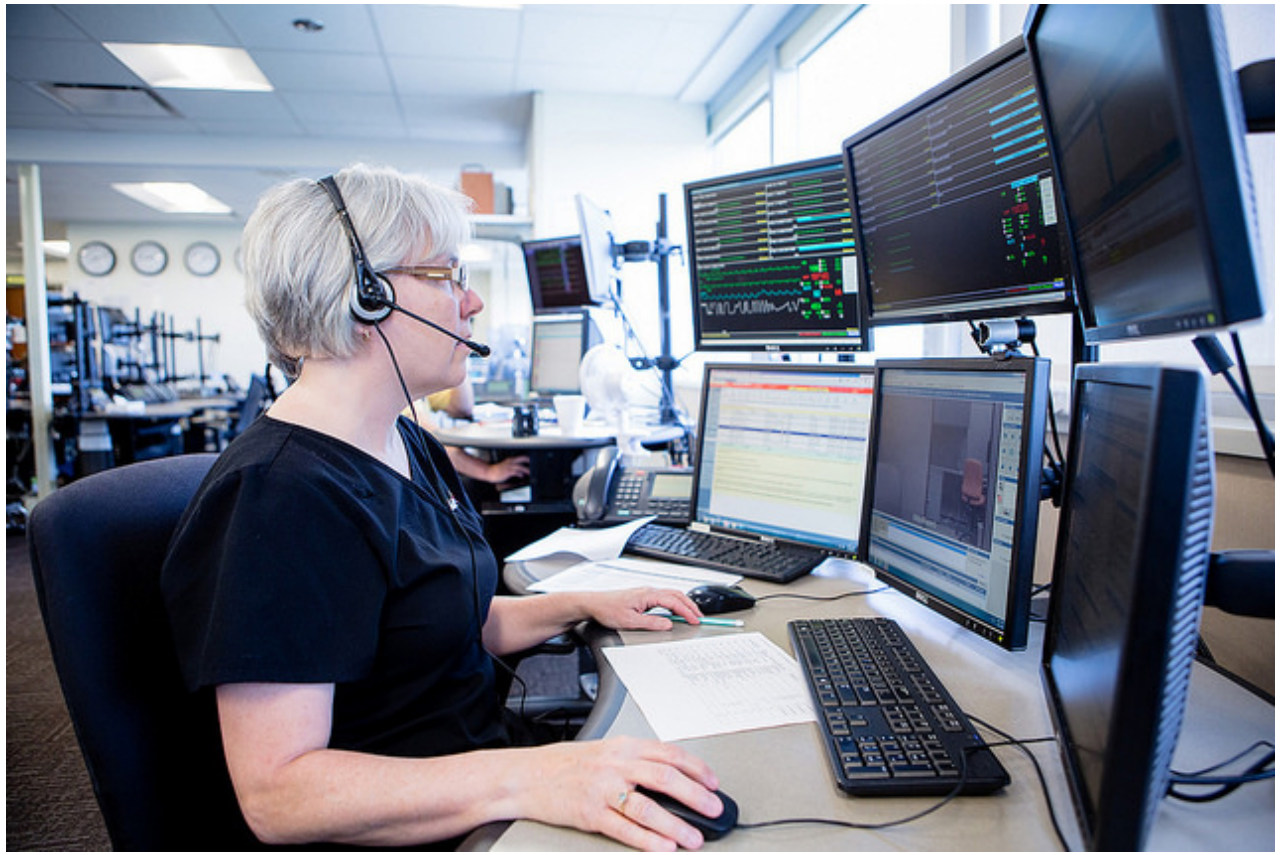
Reliability: Software Errors

- Unlike random hardware faults, they may be correlated across nodes. (Linux leap-second bug in June 2012, runaway process using up shared resources, unresponsive shared service, etc.)

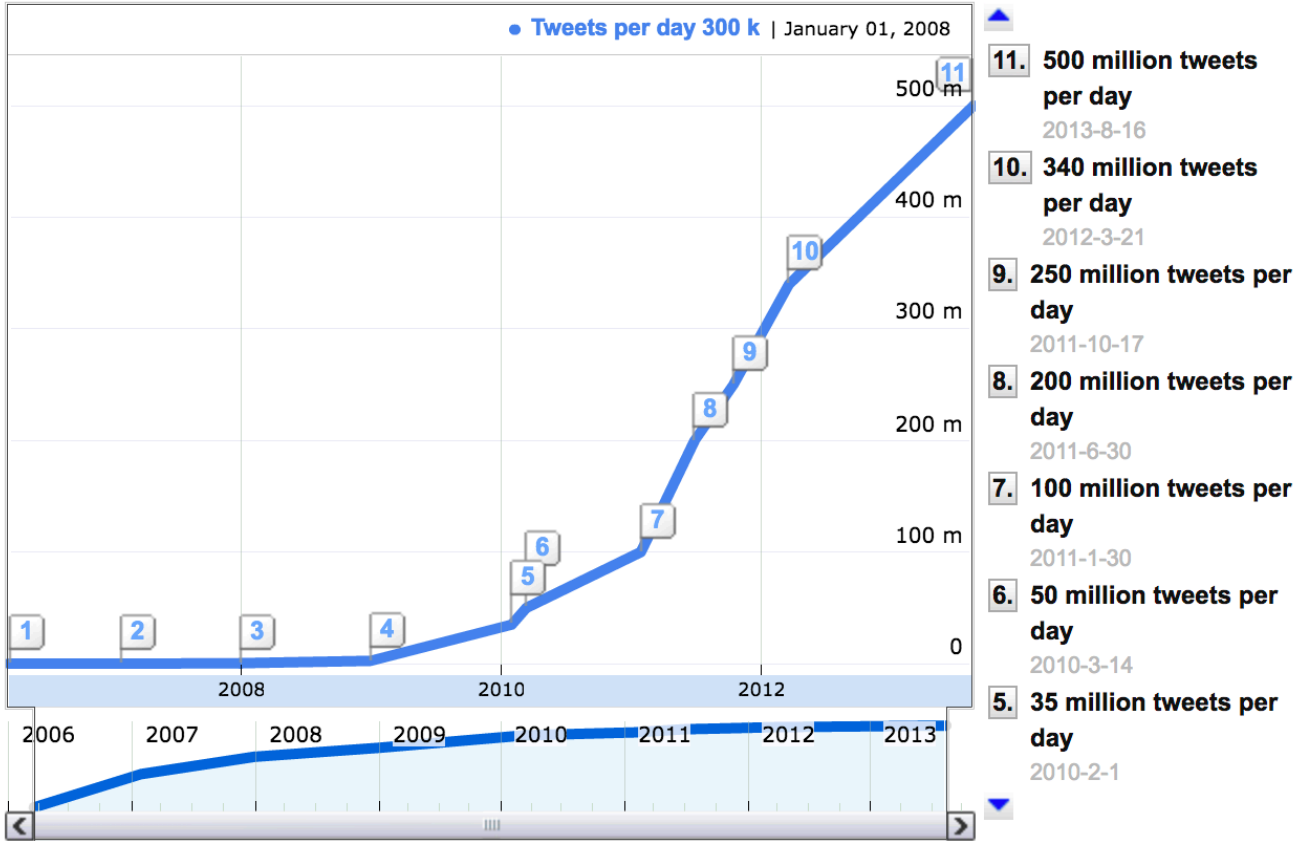
- Resolution: Extensive testing and alert mechanisms.

Reliability: Human Errors

- Application / service / network configuration errors
- Balance flexibility with restrictions aimed at encouraging users to do the right thing
- Sandbox testing/ Automated unit testing to identify edge cases
- Automated rollout / rollback of code changes (DevOps)
- Monitoring / Dashboards / Telemetry for error detection, performance tracking



Scalability: Describing “Load” (A twitter use-case)



Load Parameters

Post tweet: A user can publish a new message to their followers (6-10 thousand tweets/second)

Home timeline: A user can view tweets posted by the people they follow (300k requests/second)

Following tweets: An RDB approach (early twitter)

```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

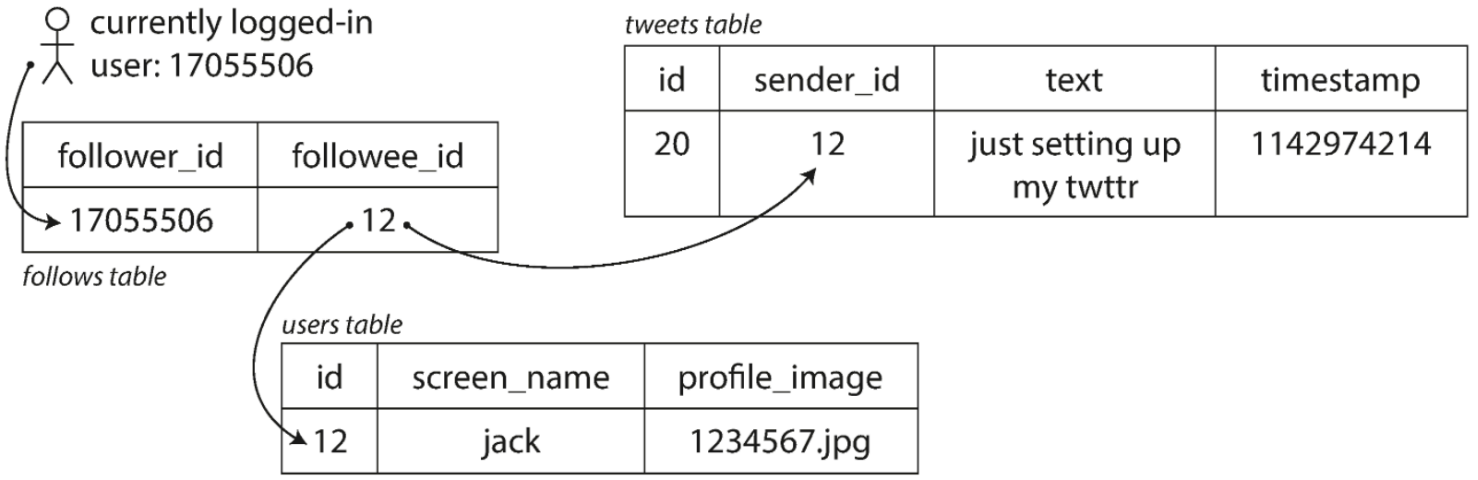


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

Following tweets: A scalable approach

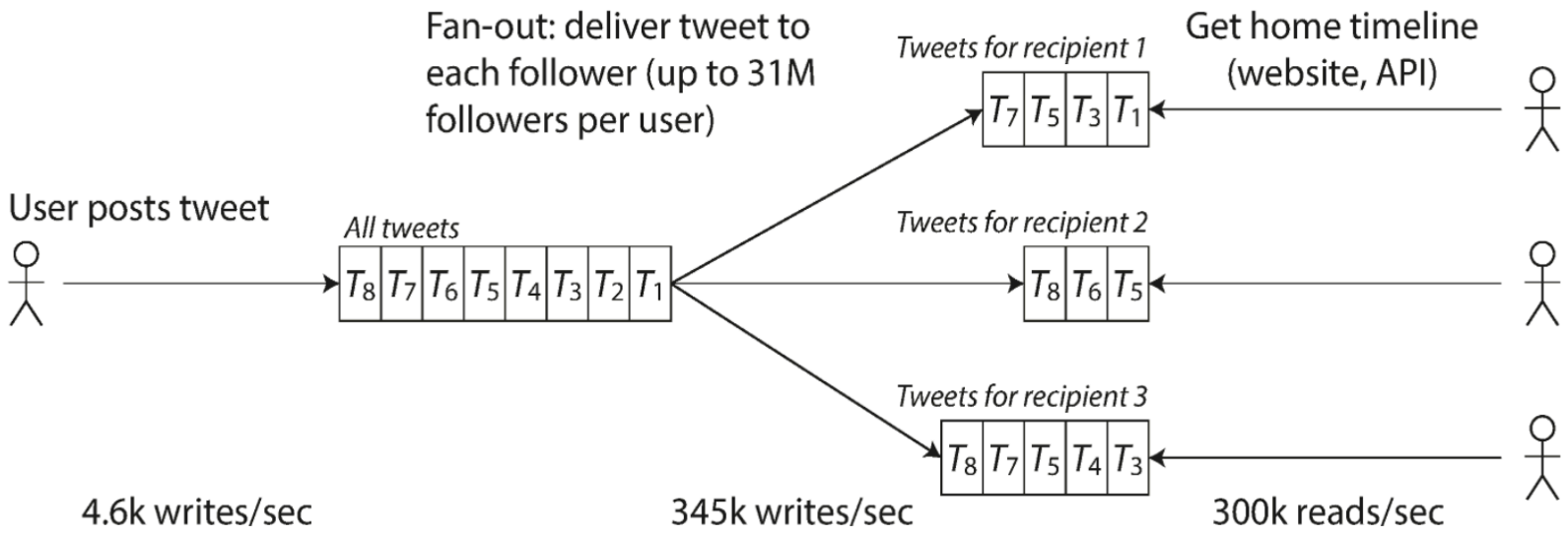


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

On average a tweet is delivered to 75 followers – and some users have millions of followers!

Performance objective: Deliver a tweet within 5 seconds.

Hybrid approach: pre-fetch tweets of celebrities separately and merge them into user's timelines.

Scalability: Describing “Performance”

Key Questions:

When you increase a load parameter, keeping resources constant, how is system performance affected?

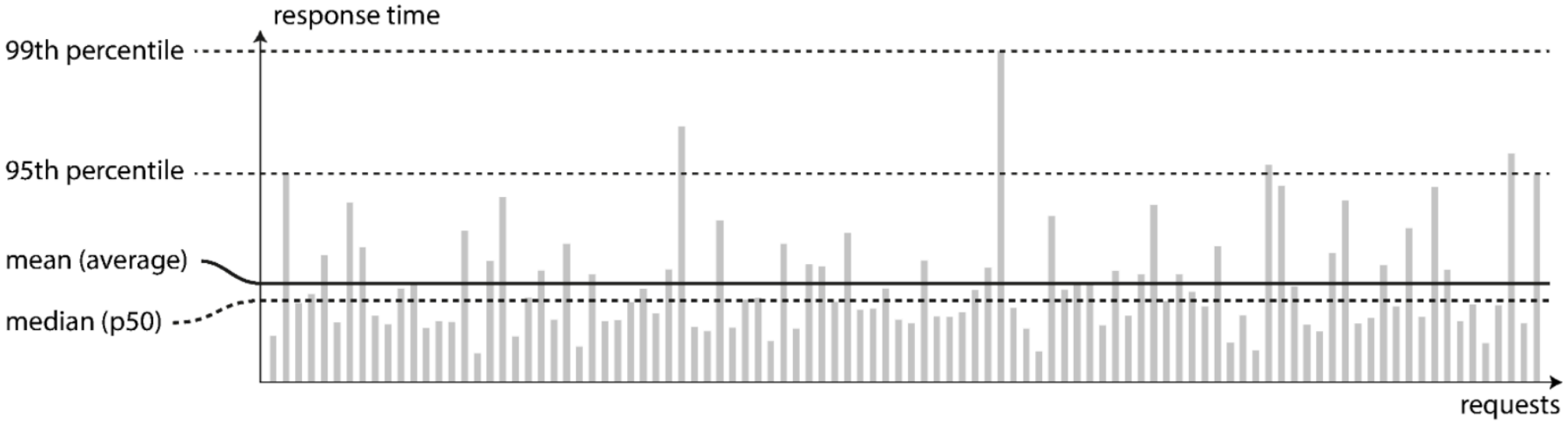
When you increase a load parameter, how much do you do you need to increase resources to keep performance unchanged?

Common Performance Measures:

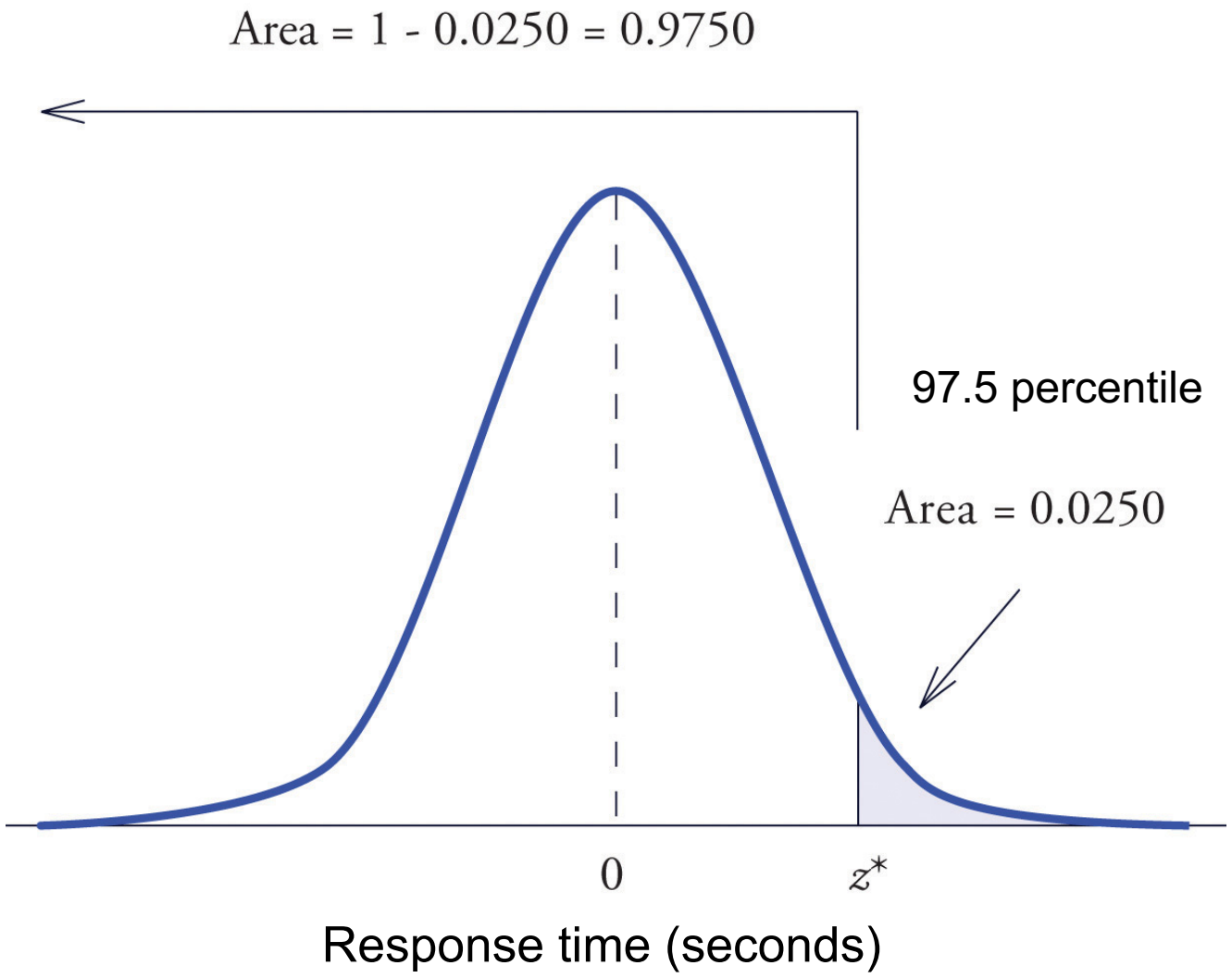
Throughput: Records/sec or run time for jobs of a certain size. (batch processing systems like Hadoop)

Response time: How long it takes for the user to see an answer when they submit a request. a.k.a. *service time*

Latency: The duration that a request is waiting to be handled.



Response time percentiles



Amazon:

99.9th percentile:

100 ms increase → 1% lower sales

1 sec increase → 16% reduction in customer satisfaction

Service Level Agreements (SLAs)

A service must be up and available (i.e., be responsive) 99.9% of the time.

Available *means*:

50^{th} percentile response time < 200 msec

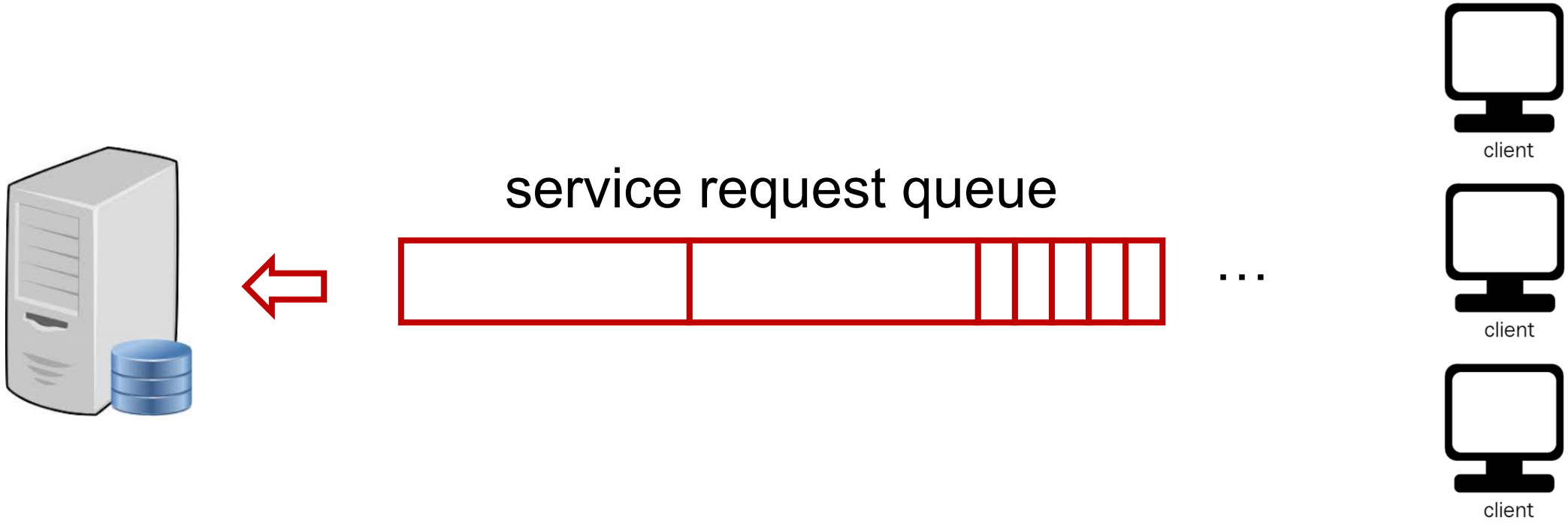
99^{th} percentile response time < 1 sec

<u>Some vendors take a different approach!</u>

8. AVAILABILITY OF WEBSITE/SERVICES

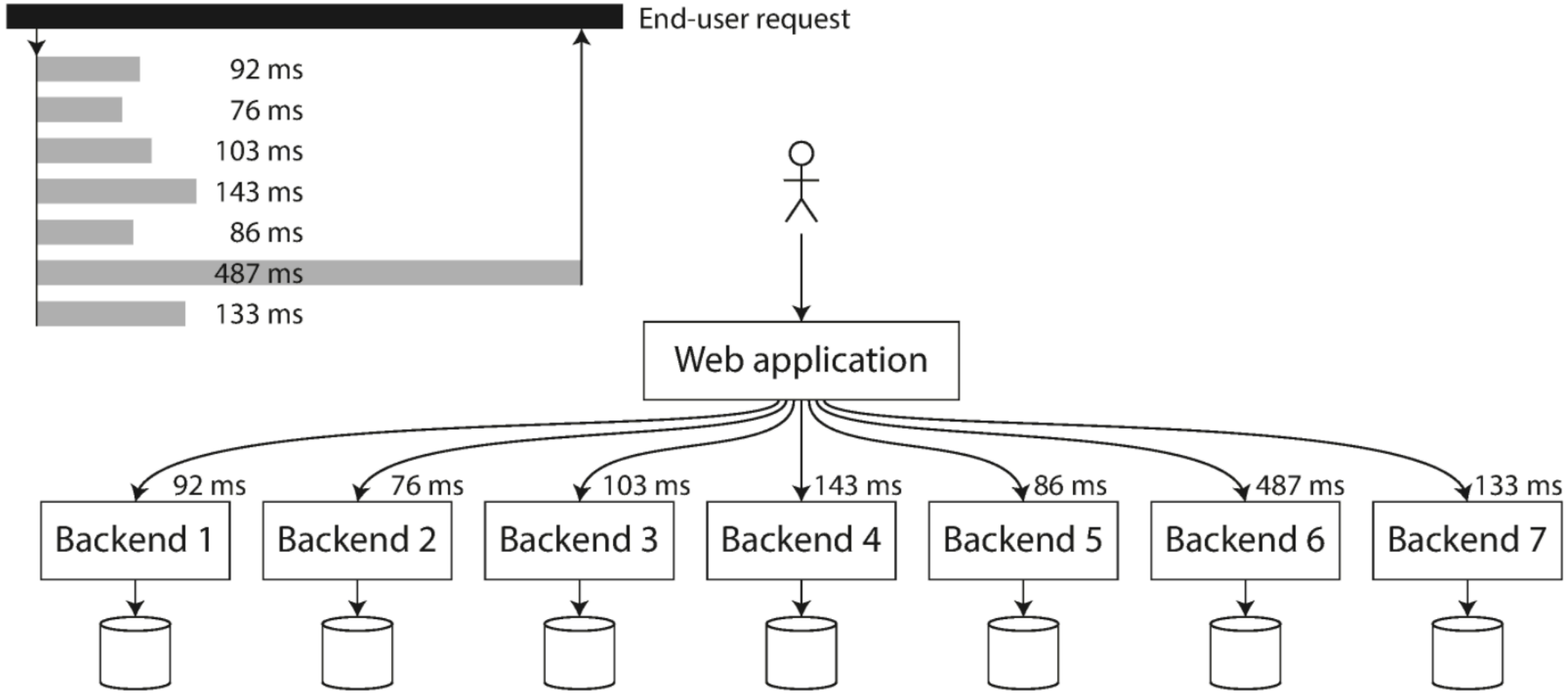
Subject to the terms and conditions of this Agreement and our other policies and procedures, we shall use commercially reasonable efforts to attempt to provide this Site and the Serviceson a twenty–four (24) hours a day, seven (7) days a week basis. You acknowledge and agree that from time to time this Site may be inaccessible or inoperable for any reason including, but not limited to, equipment malfunctions; periodic maintenance, repairs or replacements that we undertake from time to time; or causes beyond our reasonable control or that are not reasonably foreseeable including, but not limited to, interruption or failure of telecommunication or digital transmission links, hostile network attacks, network congestion or other failures. You acknowledge and agree that we have no control over the availability of this Site or the Service on a continuous or uninterrupted basis, and that we assume no liability to you or any other party with regard thereto.

Response time: head-of-line blocking

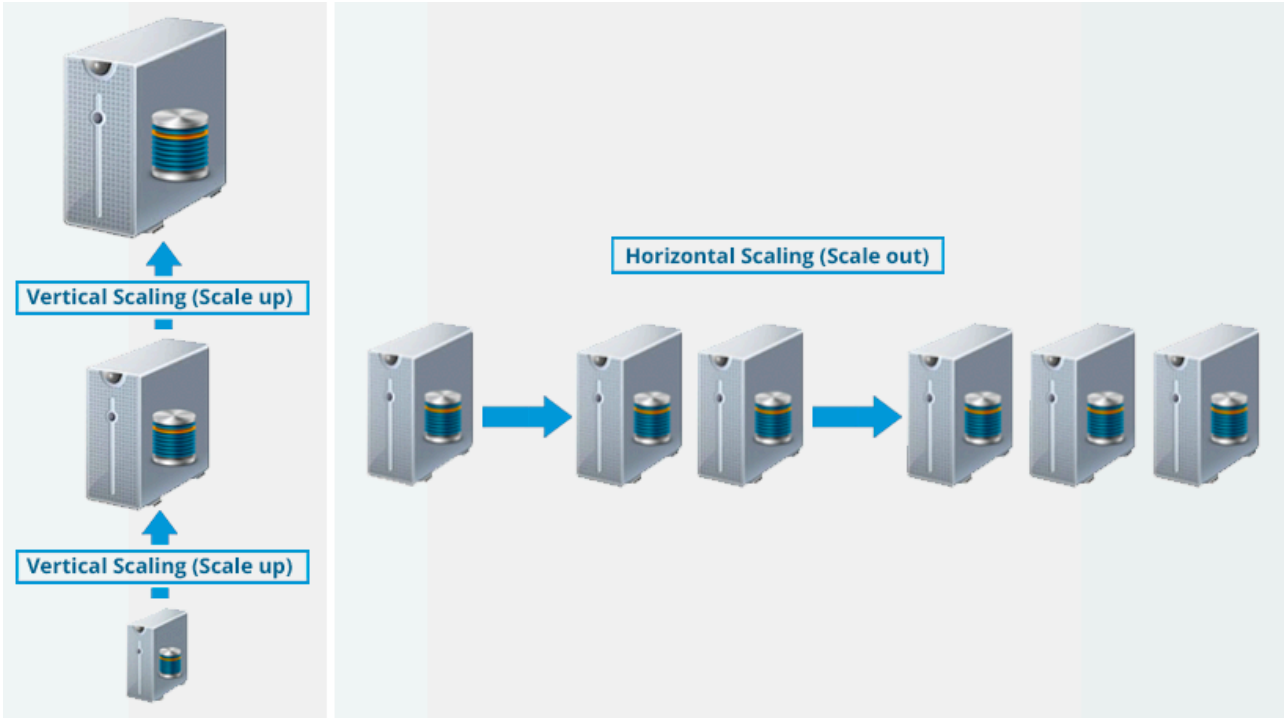


head-of-line blocking: Slow requests block shorter waiting requests, impacting overall response time.

Response time: Tail latency amplification



Scaling up vs. Scaling out



Conventional wisdom: Scale vertically (up) until the demands of high-availability make it necessary to scale out with a distributed computing model.

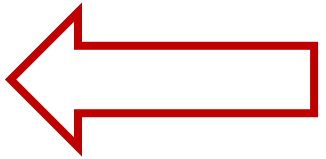
Why? Scaling up is easier – you don't need to modify your computing model or your application. But there are practical and financial limits.

BUT: Frameworks like Apache Spark and EC2 are making it easier to scale out.

Maintainability

Software cost breakdown:

Initial development: ~10-20%
Error correction: ~10-20%
On-going Maintenance: ~60-80%



Operability: Make it easy for operations teams to keep system running

Simplicity: Make it easy for engineers to understand

Evolvability: Make it easy for engineers to modify in response to changing requirements *(extensibility / plasticity)*

Storage and Retrieval

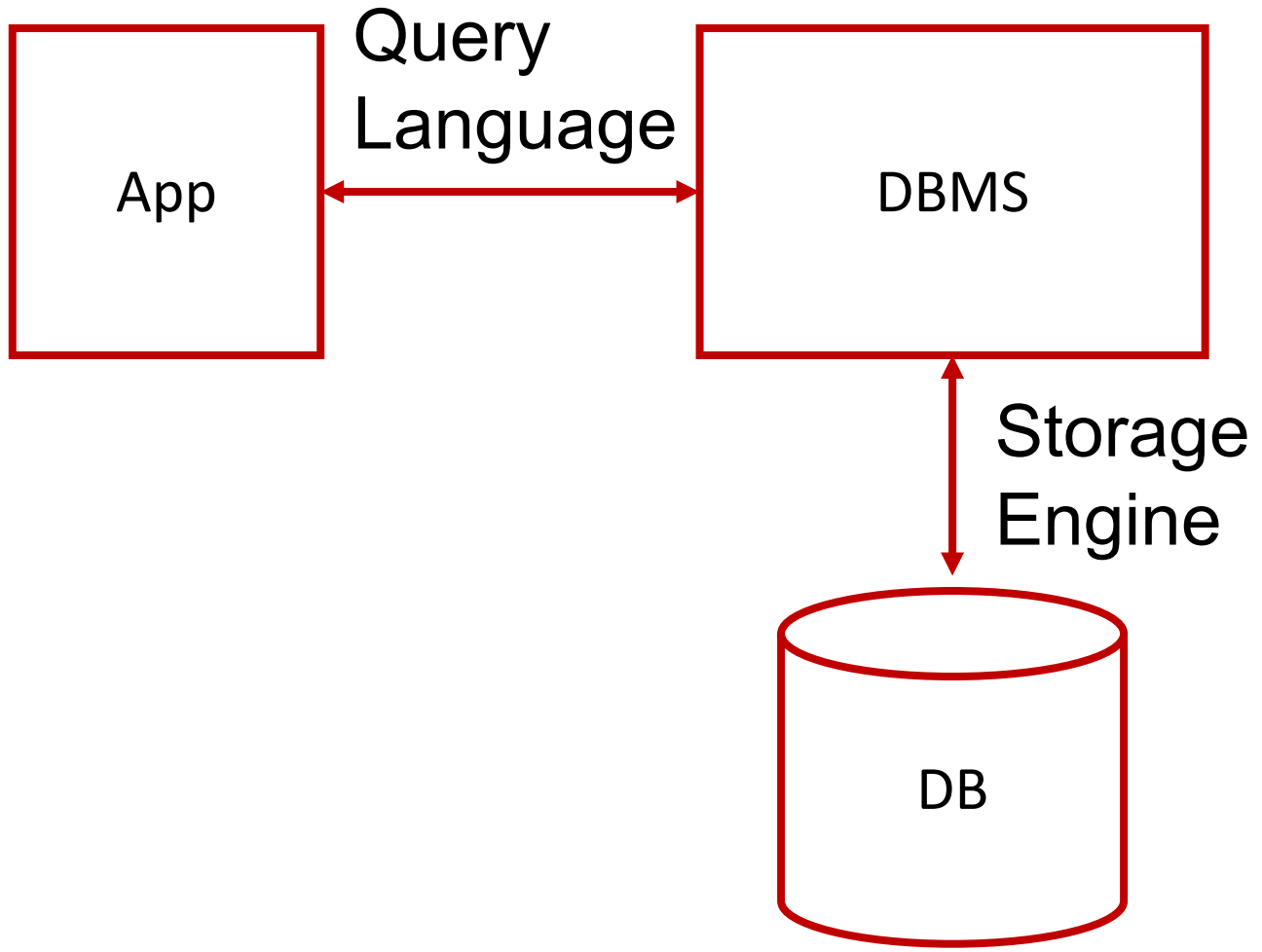
An Introduction to Key-Value Stores

Storage and Retrieval

Fundamental purpose of a database:

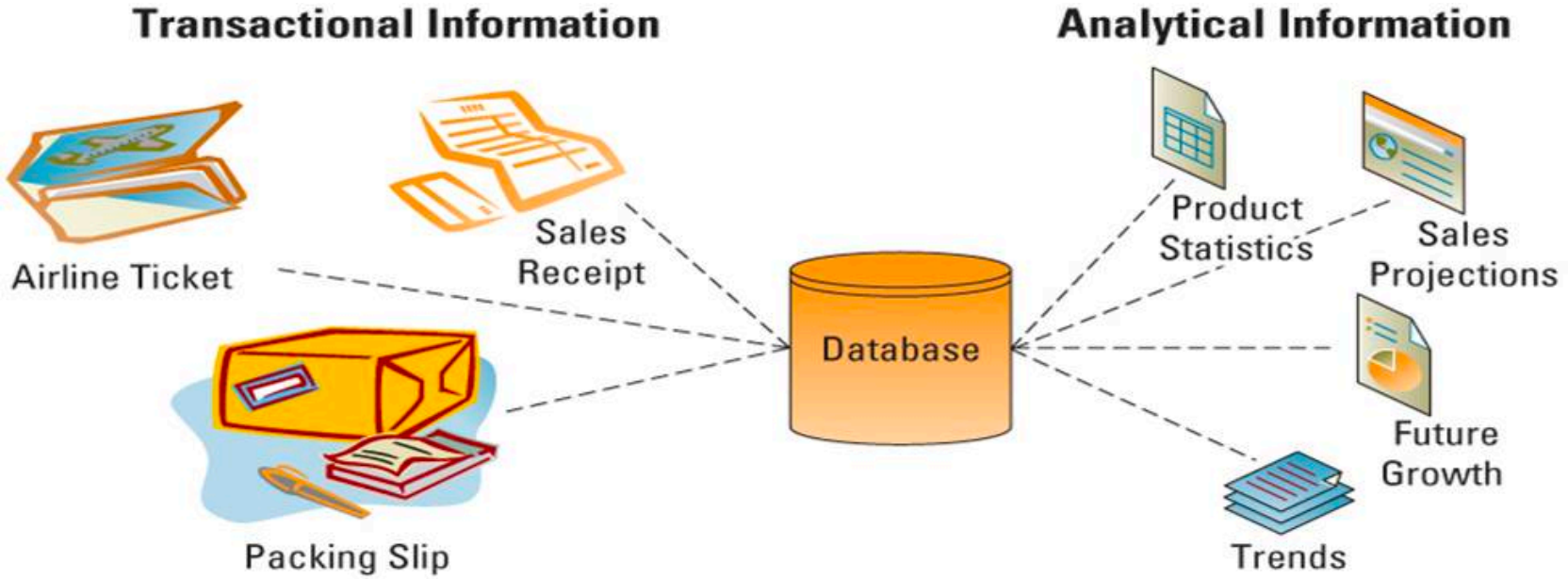
- Store data
- Retrieve data

Storage engines are the algorithms and technologies used to achieve these goals.



Why care about storage engines?

- Different storage engines are optimized for different types of work loads.
- Performance tuning



The world's simplest key-value store

```
#!/bin/bash

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

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

What is our performance?

Simple key-value store: Tradeoffs

- We use the term *log* to refer to an append-only data file, as in the previous example. Inserts are very fast, **O**(1).
- Lookups are slow, **O**(n) because we scan the entire database each time. We need an *index* of some kind in order to do lookups more quickly.
- There is a **downside:** Indexes use space, and make inserts slower. Usually, we don't index everything – only what we need to make the common lookups run faster.

Indexing

Advantage:

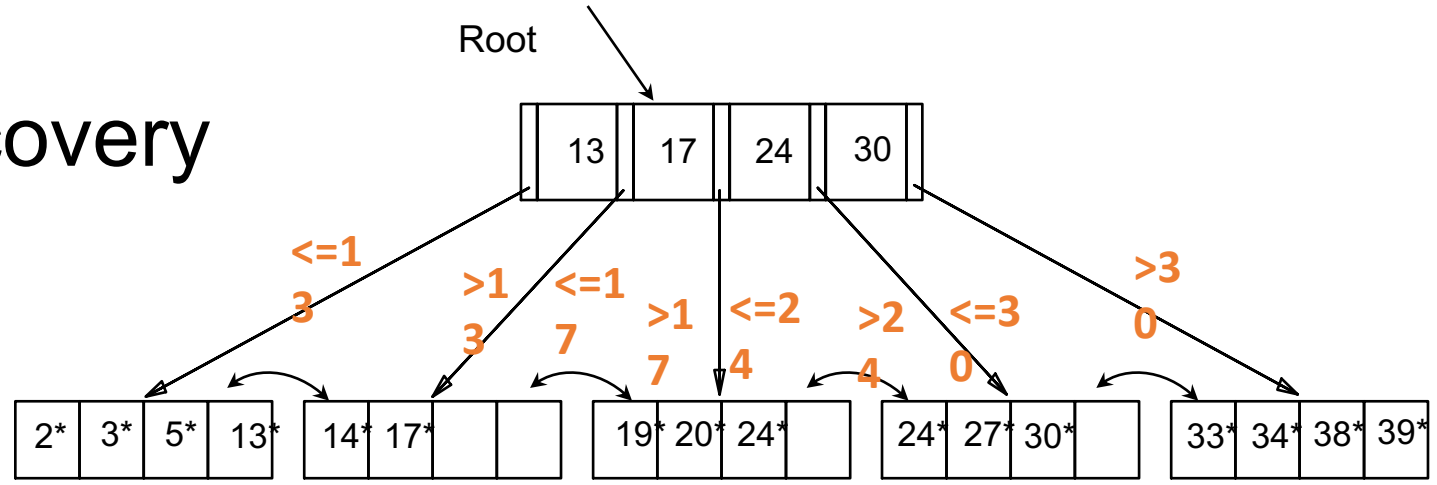
Faster read access

Disadvantage:

Slower inserts / updates

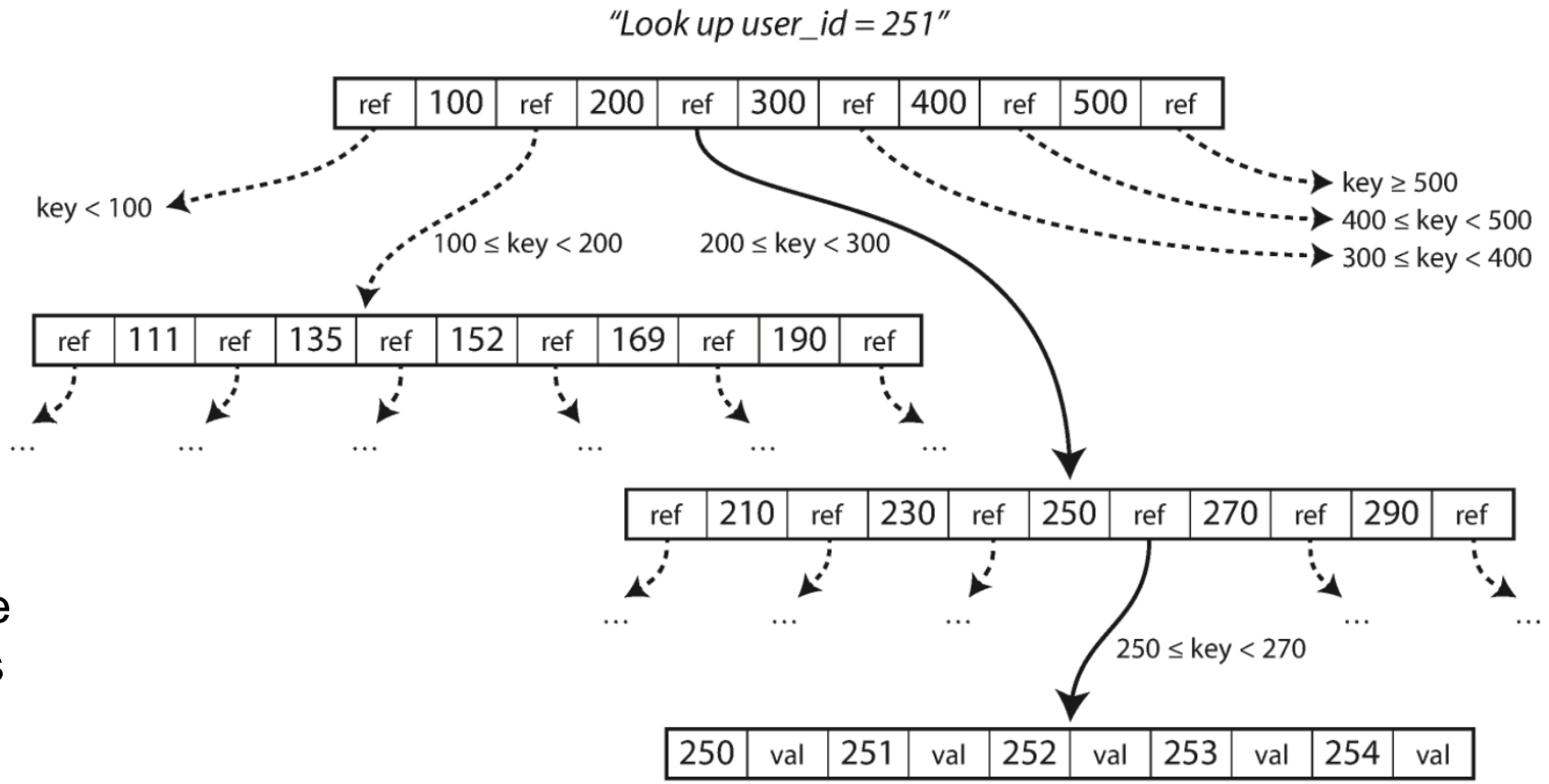
Storage overhead

More complicated recovery



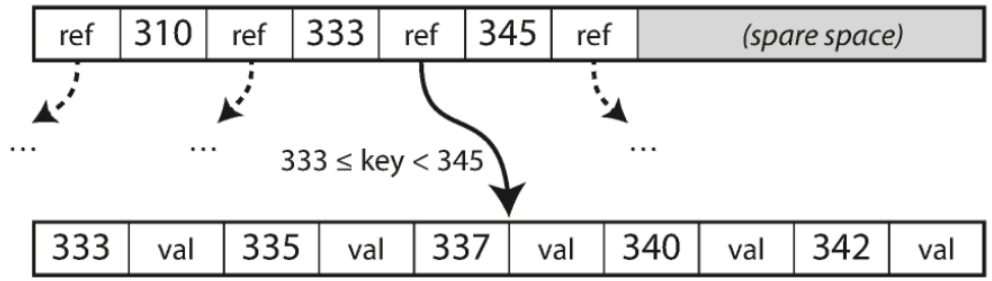
B-Trees

- Most commonly used since their introduction in the 1970s
- Keys are sorted, enabling efficient range queries
- B-Trees organize the DB into fixed sized blocks (pages) typically 4k or more that live on disk.
- Each child node is responsible for a contiguous range of keys
- Branching factors of several hundred are common.
  (4 levels, 4 KB page, 500 bf => 256 TB storage)

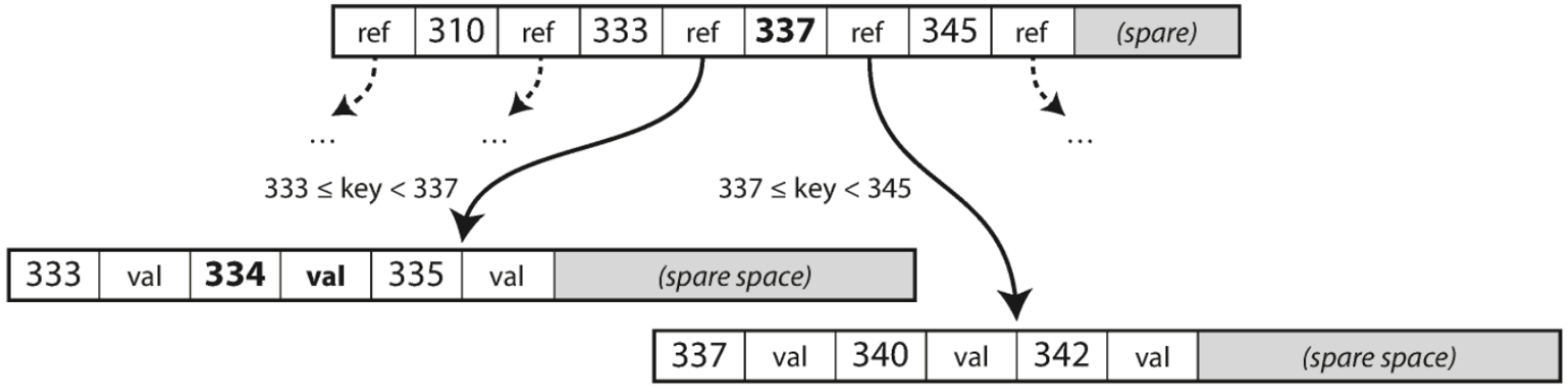


B-Trees: Key addition with page split

- We want to add a value for key 334 but there is no more room in the page
- The page is split and references to the parent page are updated

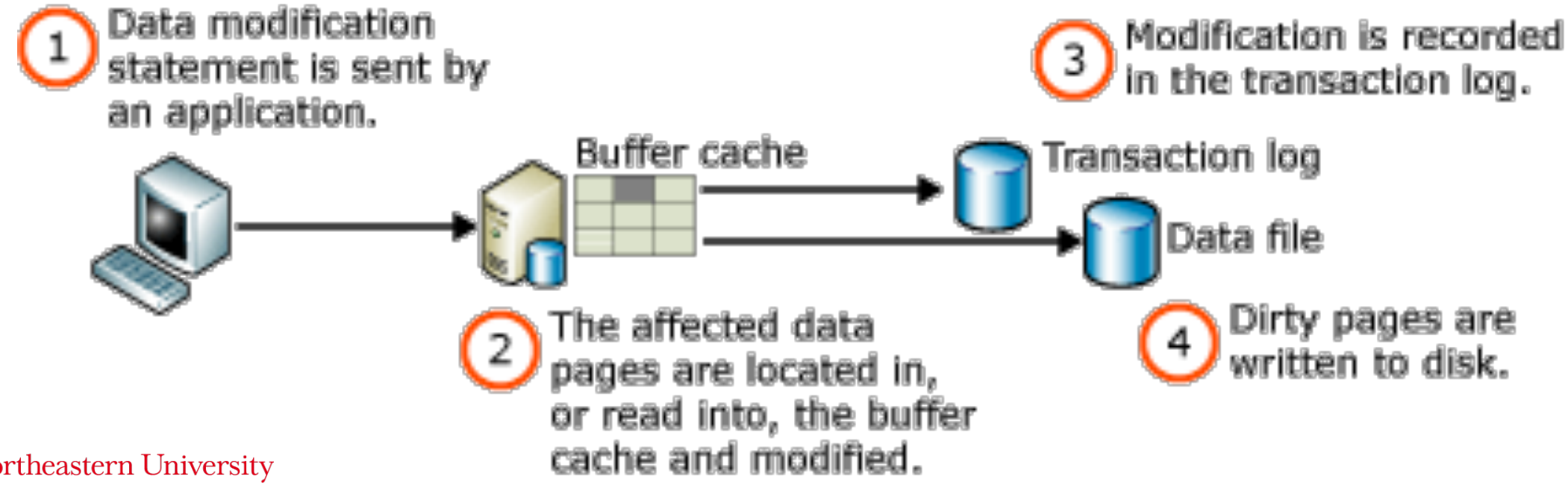


After adding key 334:



WAL: Write-Ahead Logs

- A system failure could leave the index in a corrupted state. Why?
- Data pages cannot be written to disk until the log records describing those changes are written to disk FIRST.
- Use WAL for crash recovery



B-Trees: Advantages and Disadvantages

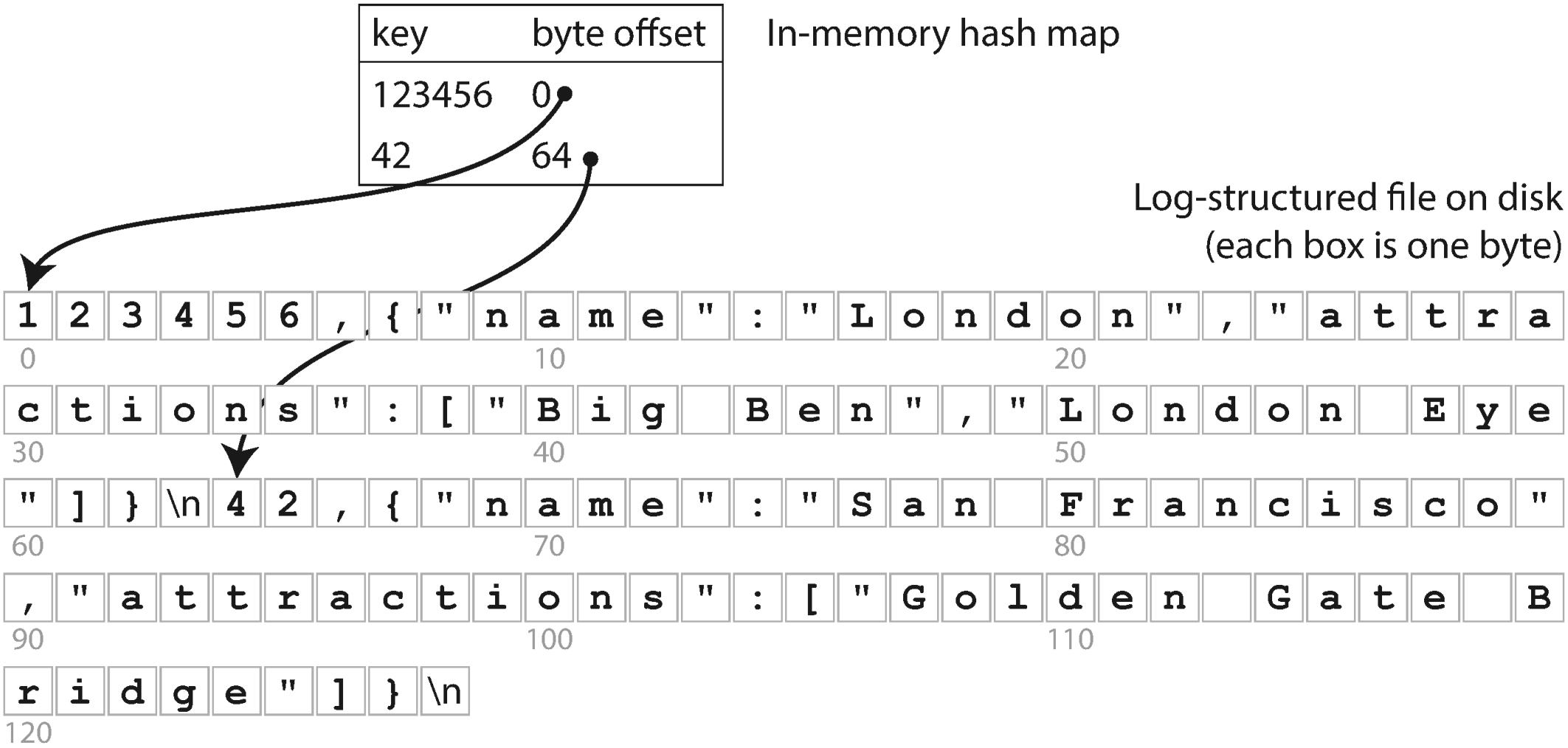
Advantages

- Very mature technology (developed in the 1970's). Standard in relational and some non-relational DBs
- Consistent performance over a wide range of read / write workloads
- Range query support

Disadvantages

- Page fragmentation
- In-place updates requires additional write to WAL.
- Pages are written as a block creating some I/O overhead
- Concurrency support requires locking mechanisms

Hash Indexing (single sequential log file)



Hash Indexing Use-cases

- Index is stored entirely in memory → Not too many distinct keys
- Many updates per key
- Video likes

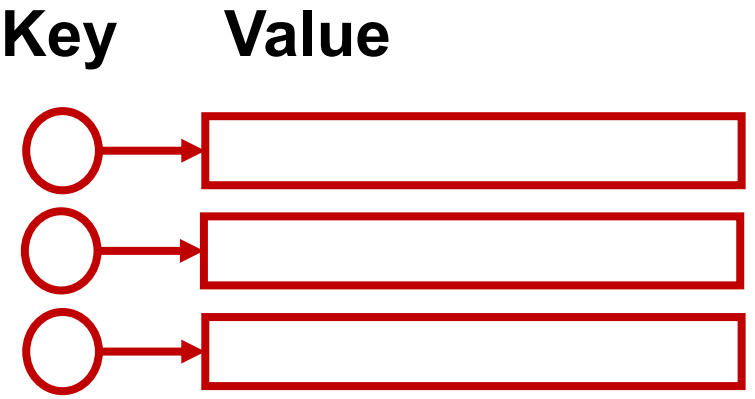
Video name	Uploader	Likes[* 1]	Upload date	Date achieved	Days held	Refs	Notes
"Despacito"[6]	Luis Fonsi	16,010,000	January 12, 2017	July 25, 2017	568[† 1]	[60]	[M]
"See You Again"[8]	Wiz Khalifa	11,210,000	April 6, 2015	August 27, 2016	332	[61]	
"Gangnam Style"[10]	Psy	1,570,000	July 15, 2012	September 13, 2012	**1,444**	[62]	[N]
"Party Rock Anthem"[65]	LMFAO	1,390,000	March 8, 2011	April 4, 2012	162	[66]	
"KONY 2012"[67]	Invisible Children	1,350,000	March 5, 2012	March 17, 2012	18	[68]	[O]
"Party Rock Anthem"[65]	LMFAO	950,000	March 8, 2011	November 3, 2011	135	[69]	[P]
"Baby"[18]	Justin Bieber	760,000	February 19, 2010	July 6, 2011	120	[73]	
"Evolution of Dance"[74]	Judson Laipply	620,000	April 6, 2006	March 31, 2010	**462**	[75]	[Q]

1. ^ As of February 13, 2019

Heap files

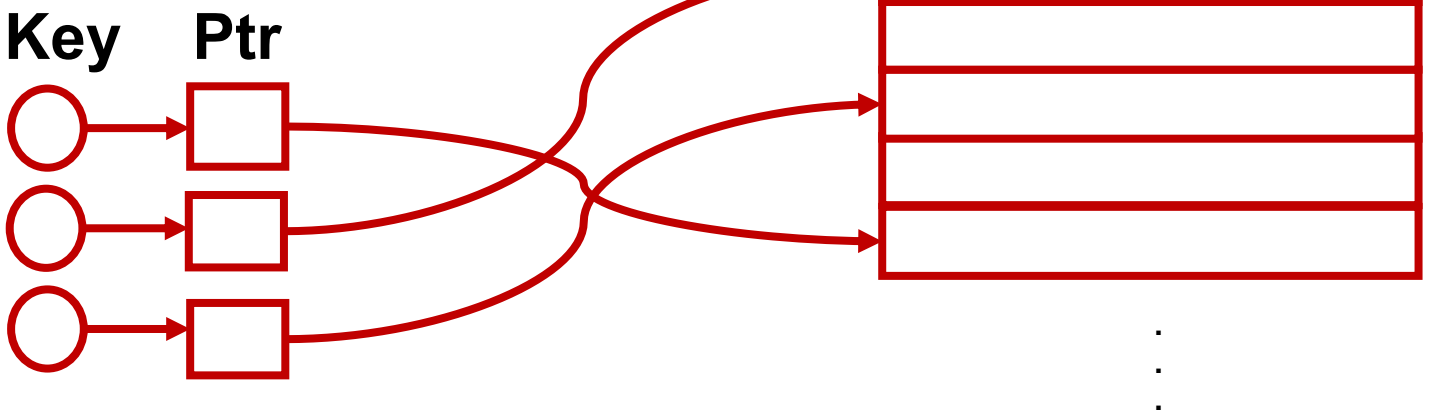
We could store both keys *and* values in the index…

Clustered Index:



… or put values on disk in a *heap file*….

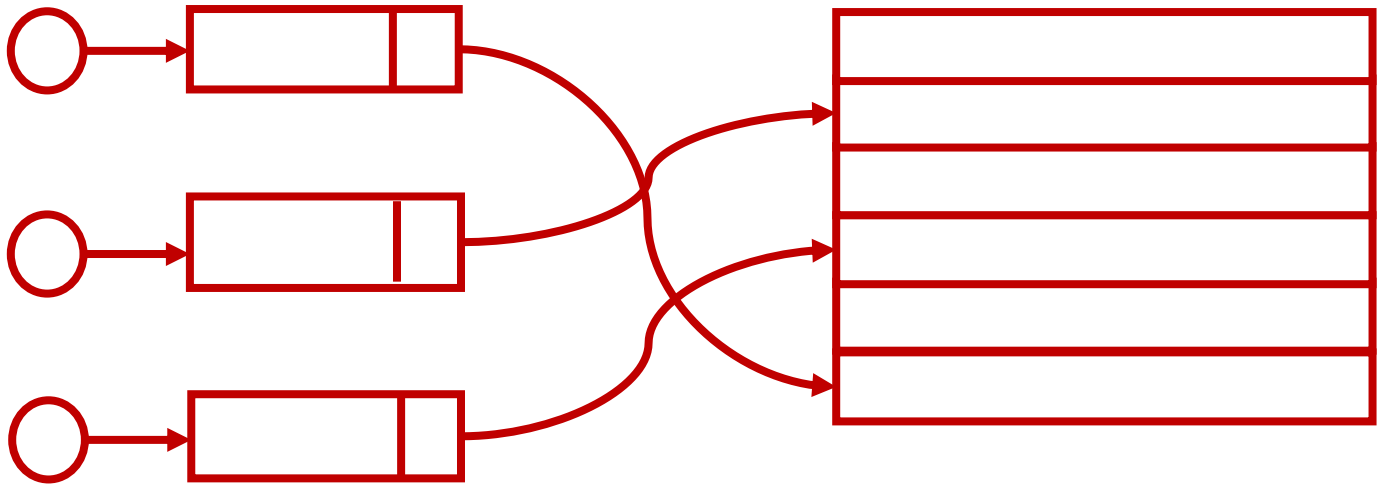
Non-clustered Index:



Heap files

… or both with some columns in index, others in heap!

Covering Index:



Secondary Indexes (Non-unique rows)

Sales transactions

0 → { amount : 20.32, state : VT }
1 → { amount : 34.99, state : MA }
2 → { amount : 12.98, state : VT }
3 → { amount : 93.23, state : VT }
4 → { amount : 50.17, state : CA }

What if we want to find transactions in Vermont?

Secondary Index on State

VT → { transactions : [0, 2, 3] }
MA → { transactions : [1] }
CA → { transactions : [4] }

Compaction: Avoiding unlimited growth

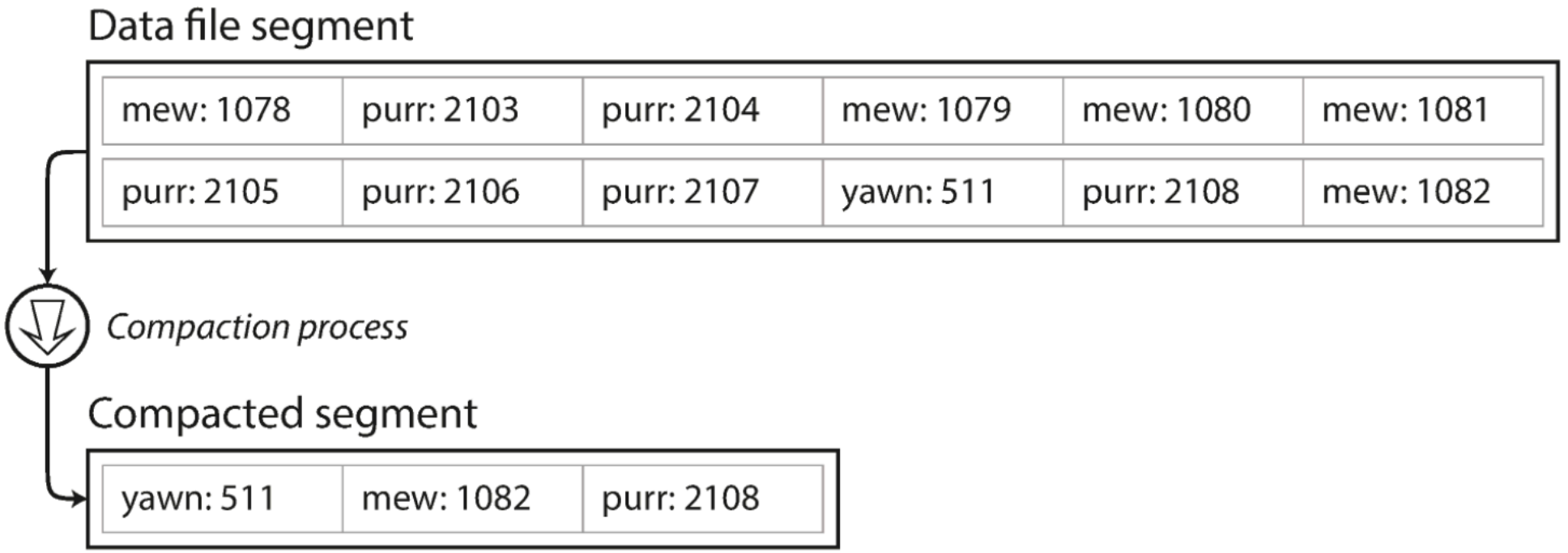


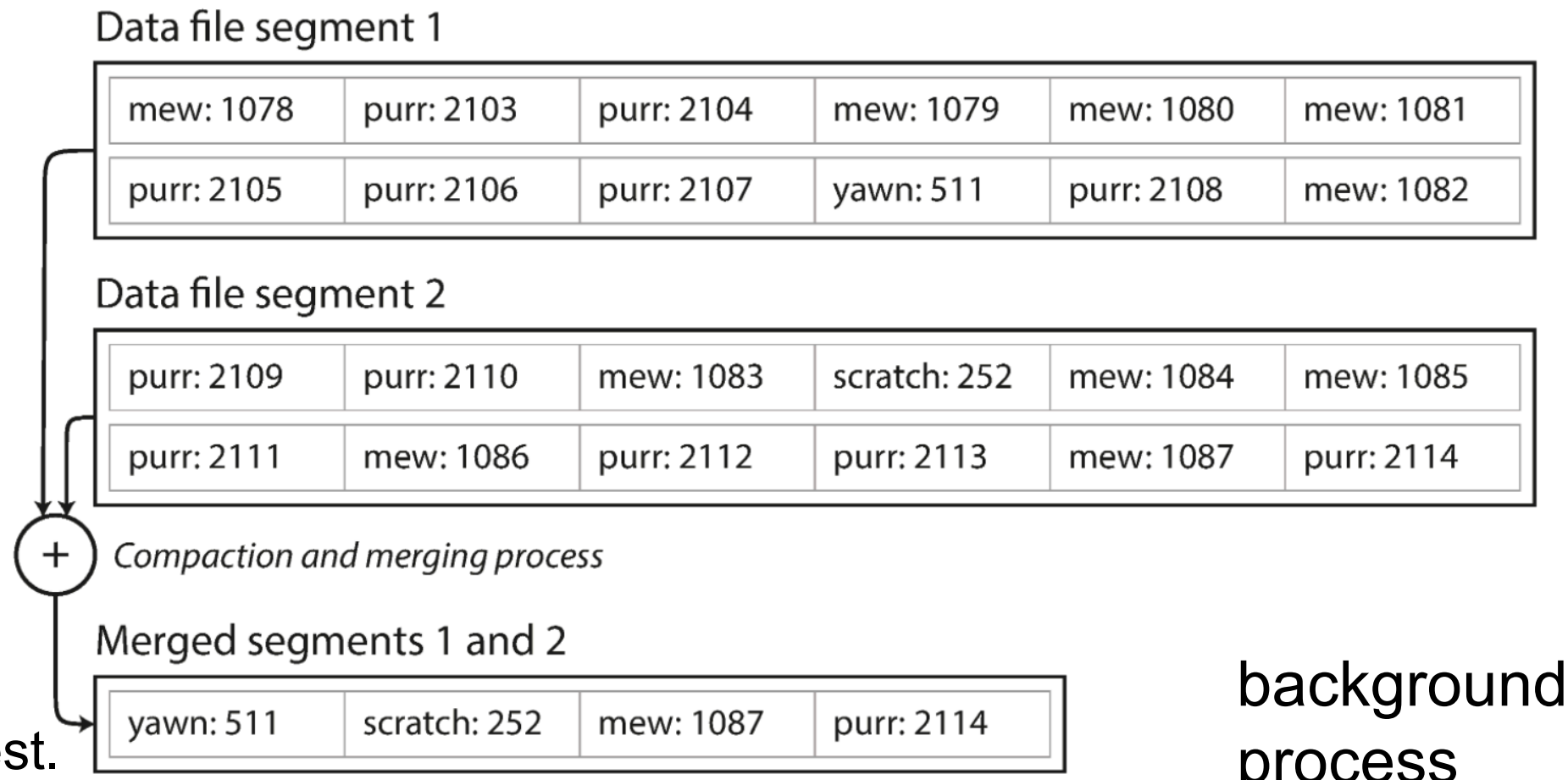
Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

merging

In-memory Hash Tables

Key	Offset

Key	Offset



One hash table per data segment. Check newest to oldest.

background process

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

Some considerations

- **File format**: Raw bytes are fast and flexible. One byte per character. CSV lacks the flexibility to store different types of values.

- **Deleted values:** Append a special byte / character called a **tombstone.** On merge, all older values are discarded.

- **Crash recovery**: Recover in-memory hash tables from data segment or save hash table snapshots to disk. Recovery simpler if data is always appended (rather than updated in place in the middle of the data segment.)

- **Concurrency**: One active data segment, one writer thread. Multiple concurrent threads can read from immutable (older) data segments.

Disadvantages / Limitations of hashed-based indexes

- Hash tables should all fit in RAM
  (Disk-based hash tables are I/O intensive and slow due to all the random-access requests.)

- You can't do range queries

  WHERE x < 100
  WHERE x BETWEEN 1000 AND 2000

- But this works:

  WHERE x IN (100, 205, 300, 320,…)

Riak Key-Value Store / Bitcask Storage Engine

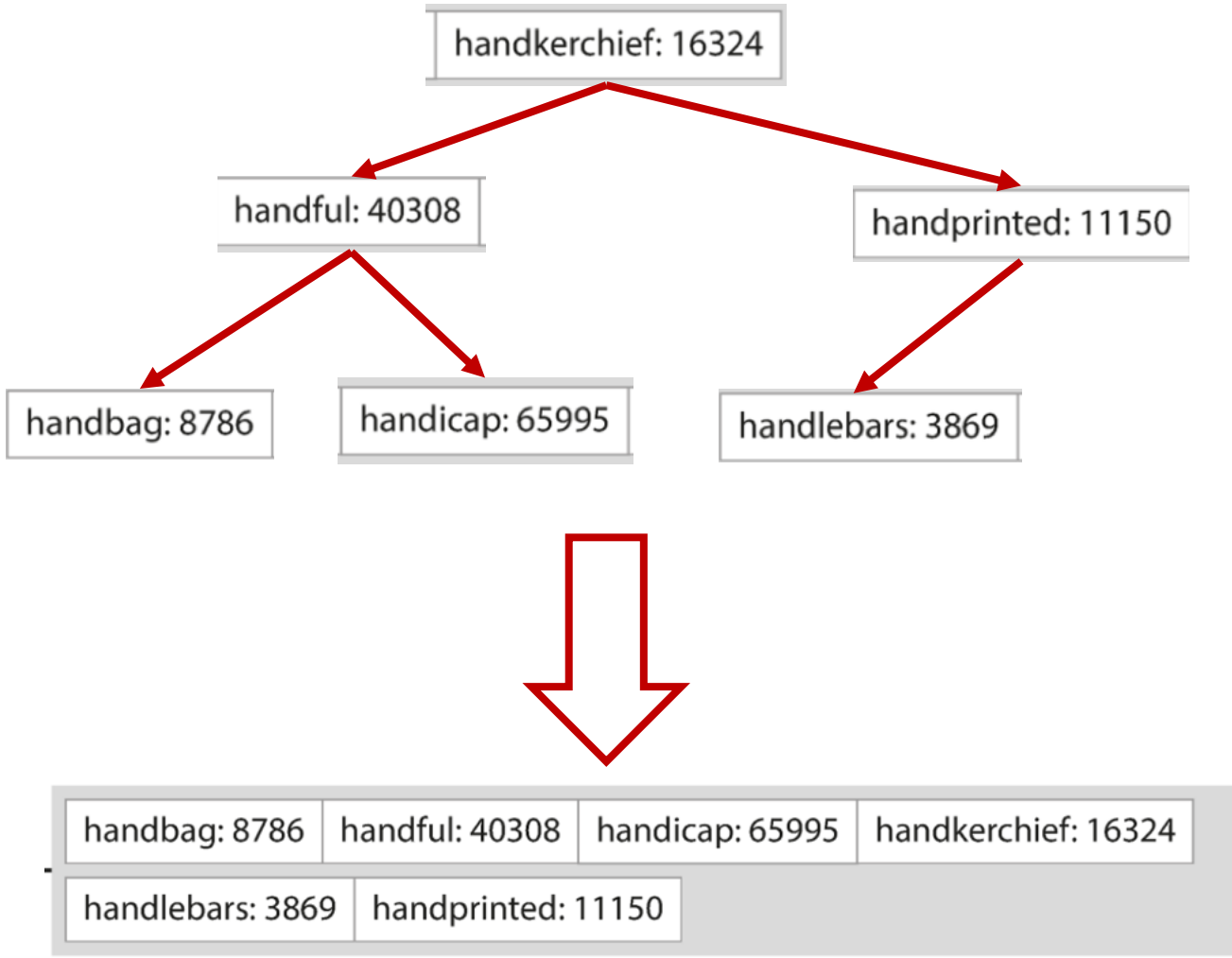
- In memory hash index: keys in memory, values on disk
- High performance reads *and* writes
- Use case: many operations across many different keys
- Subject to the restriction that the *keys* all fit in RAM

Advantages of Hash Indexing + Logs

- Append-only operations are faster than random-access writes
- Concurrency and crash recovery is simpler with immutable data segments
- Merging prevents segments from getting too large or fragmented

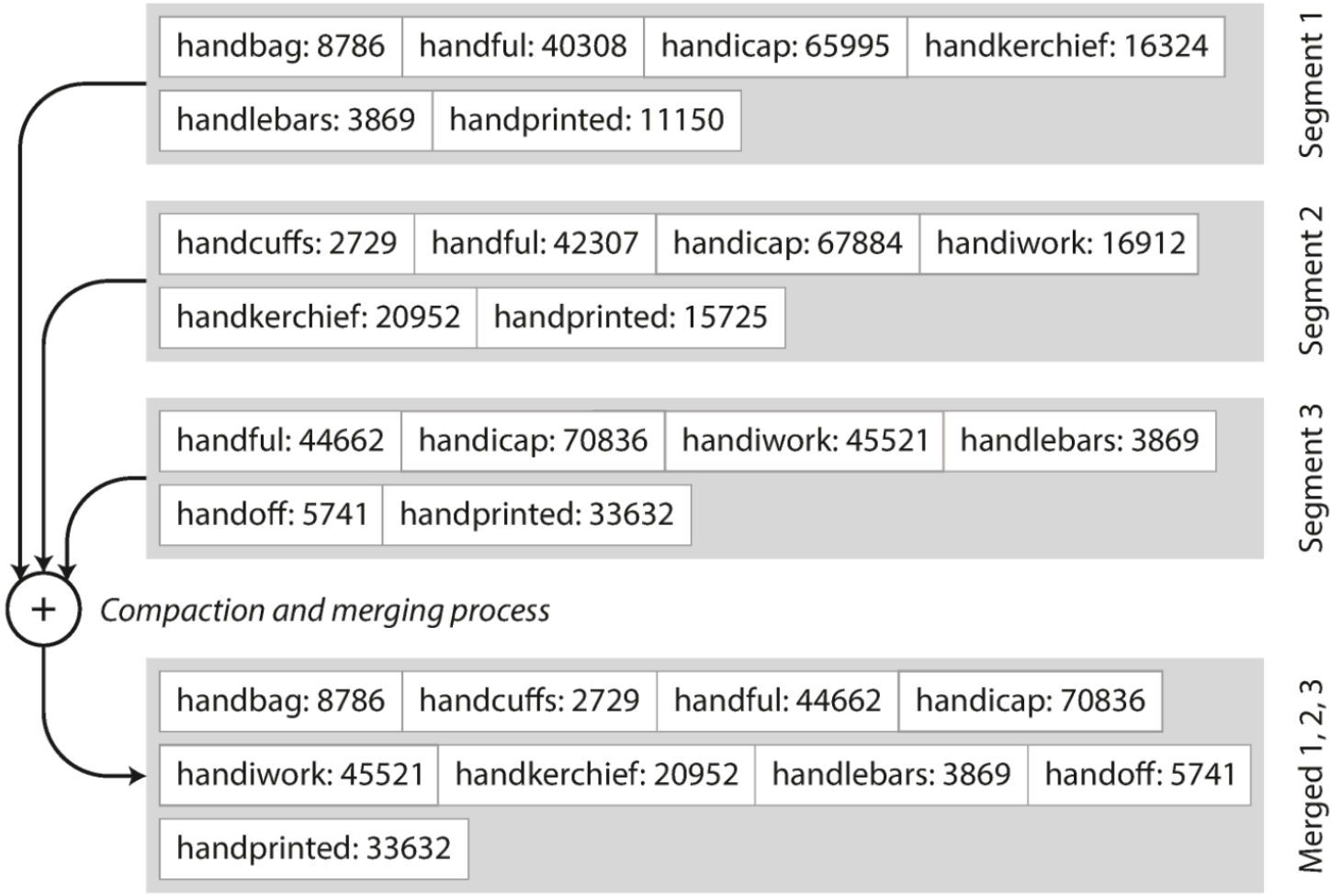
Log-Structured Merge-Trees (LSM-Trees)

- The active segment is now an in-memory balanced tree called a *memtable*
- Periodically, we persist the memtable to a segment file, walking through the keys in sorted order. These segments are called String-Sorted Tables (*SSTables*)
- Often, each segment is compressed to save space



Compaction and Merging with LSM-Trees

- Compaction and merging is very efficient because each segment is in sorted order
- As we merge, we take the next key in order. On takes, take the key/value from the most recent segment.



Sparse Index: No longer need every key in RAM!

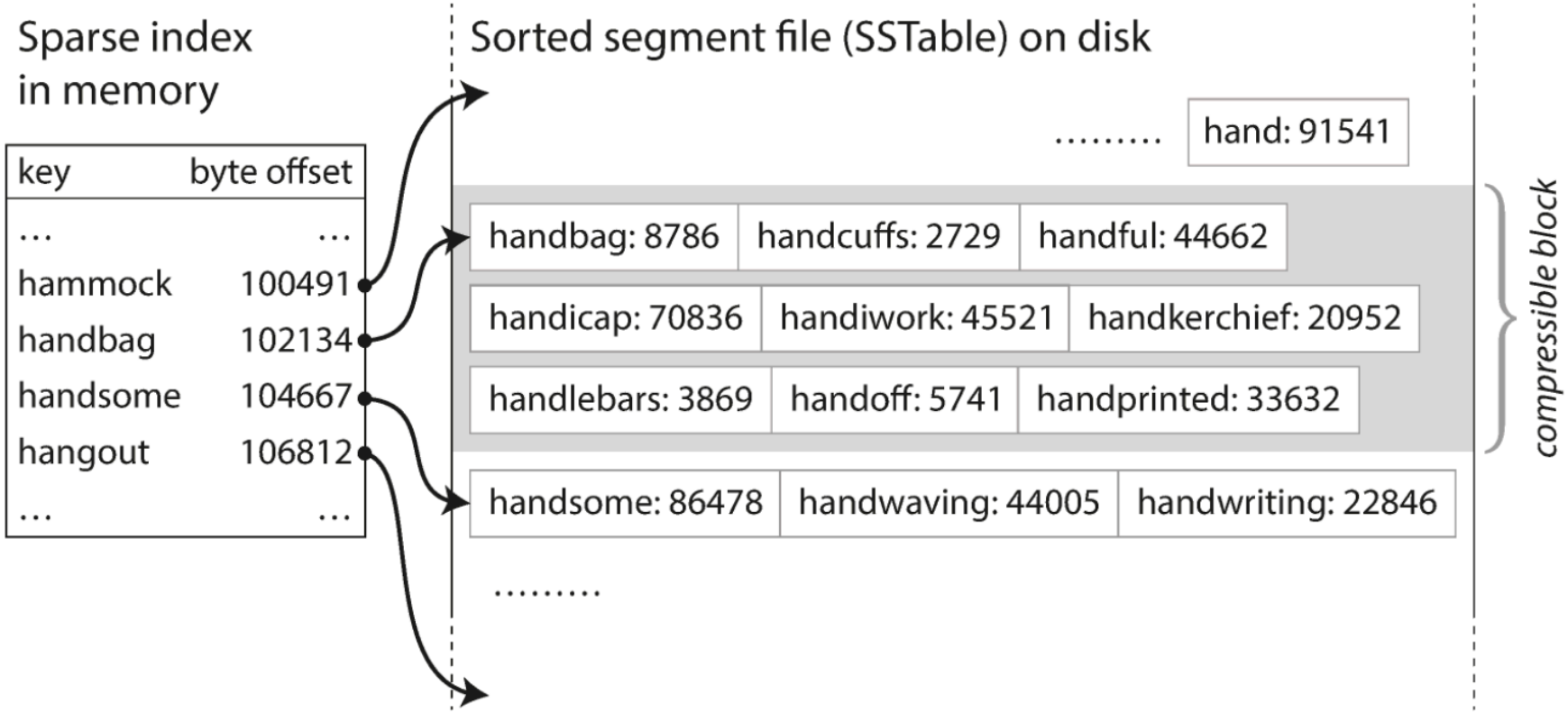
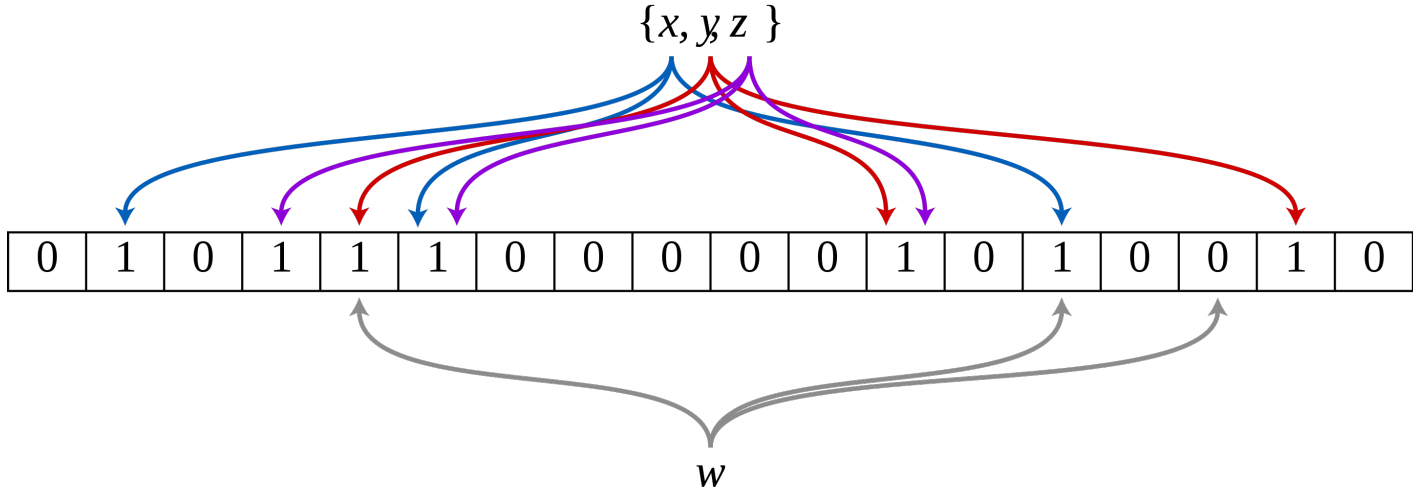


Figure 3-5. An SSTable with an in-memory index.

Using Bloom Filters to speed up LSM Trees

A bloom filter can test whether an element is a member of a set.
It returns “possibly in set” or “definitely not in set”

We can avoid costly searches across memtable plus multiple data when searching segments for items not in the database.

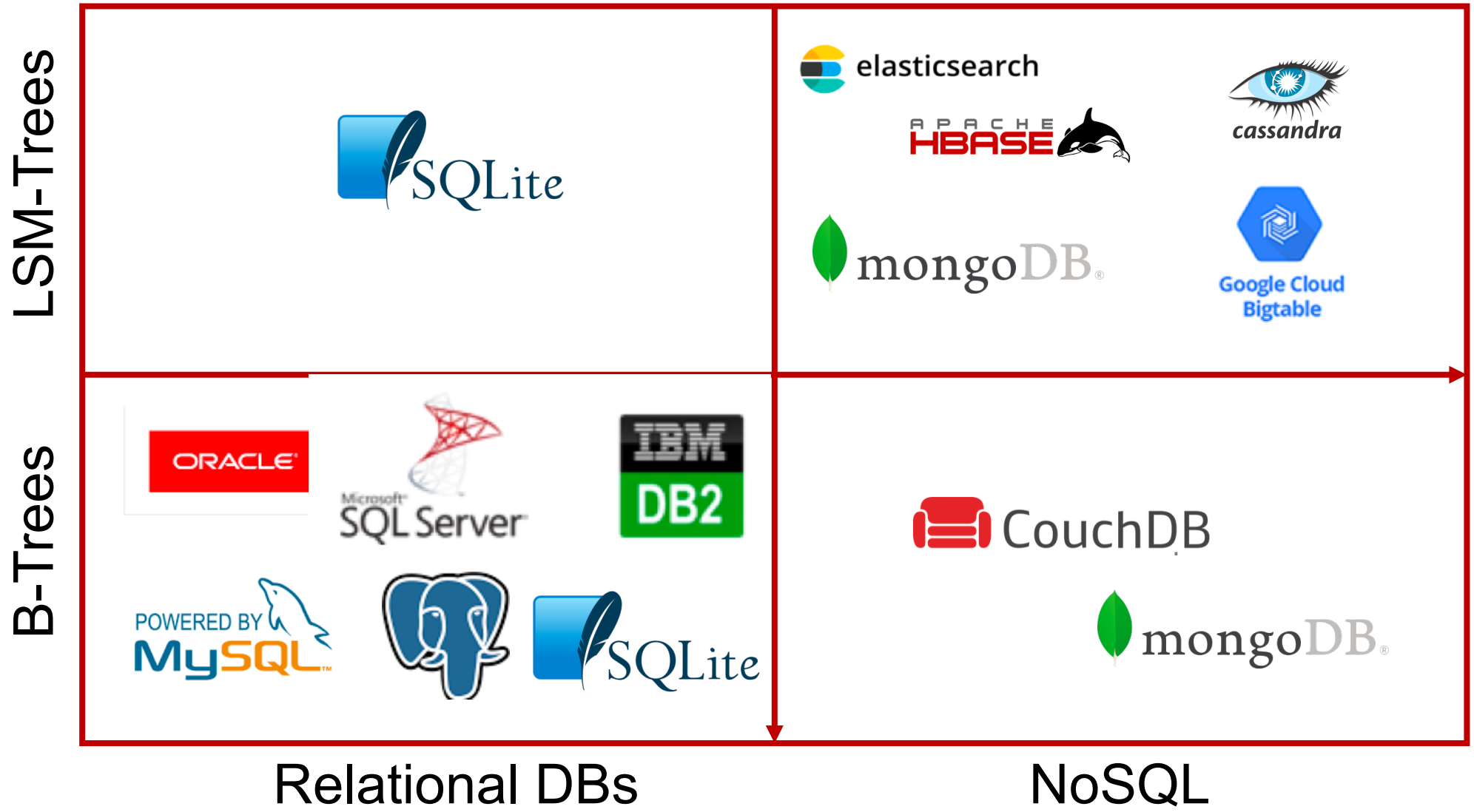


LSM-Trees vs. B-Trees

	LSM-Tree	B-Tree	
Faster Read		X *	LSM-Trees might have to access multiple data segments searching for latest key value (and still return null). (Bloom filters can mitigate this for non-existing keys.)
Faster Write	X *		B-Trees write whole pages 2 or more times due to WAL and possible parent page updates
Storage Efficiency	X		B-Tree pages may be fragmented due to fixed page size
Consistent performance		X	LSM-Tree performance may depend on state of compaction.
Technical maturity		X	B-Trees have been around since the 1970s
Overall popularity		X	Most if not all major DBMS's use B-Trees
Simplicity	X		LSM-Trees are relatively easy to implement and maintain

* But opinions and conclusions vary

LSM-Trees vs. B-Trees



Riak and Bitcask

- Riak is an early key-value store
- Basho Technologies founded in 2008
- Once considered a rising star in the NoSQL world. Raised 66 million (2008-2015)
- Now in receivership (Jul 2017)

Bitcask

A Log-Structured Hash Table for Fast Key/Value Data

Justin Sheehy David Smith
with inspiration from Eric Brewer

Basho Technologies

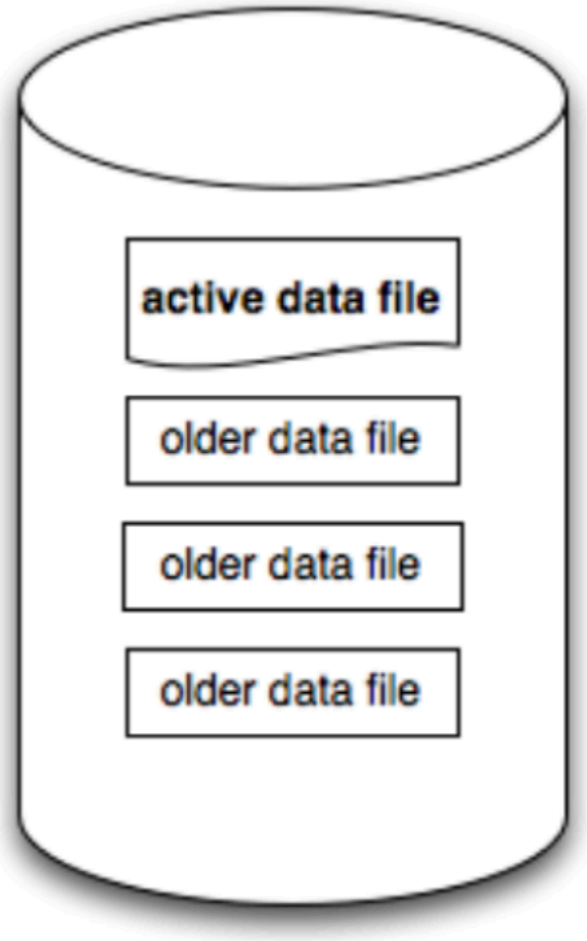
justin@basho.com,dizzyd@basho.com

Riak Design Motivation

- Low latency per item read or written
- High throughput for handling an incoming stream of random items
- Size(Datasets) >> Size(RAM)
- Crash recovery
- Backup & Restore support
- Simplicity & Maintainability
- A license that allowed for “easy default use”

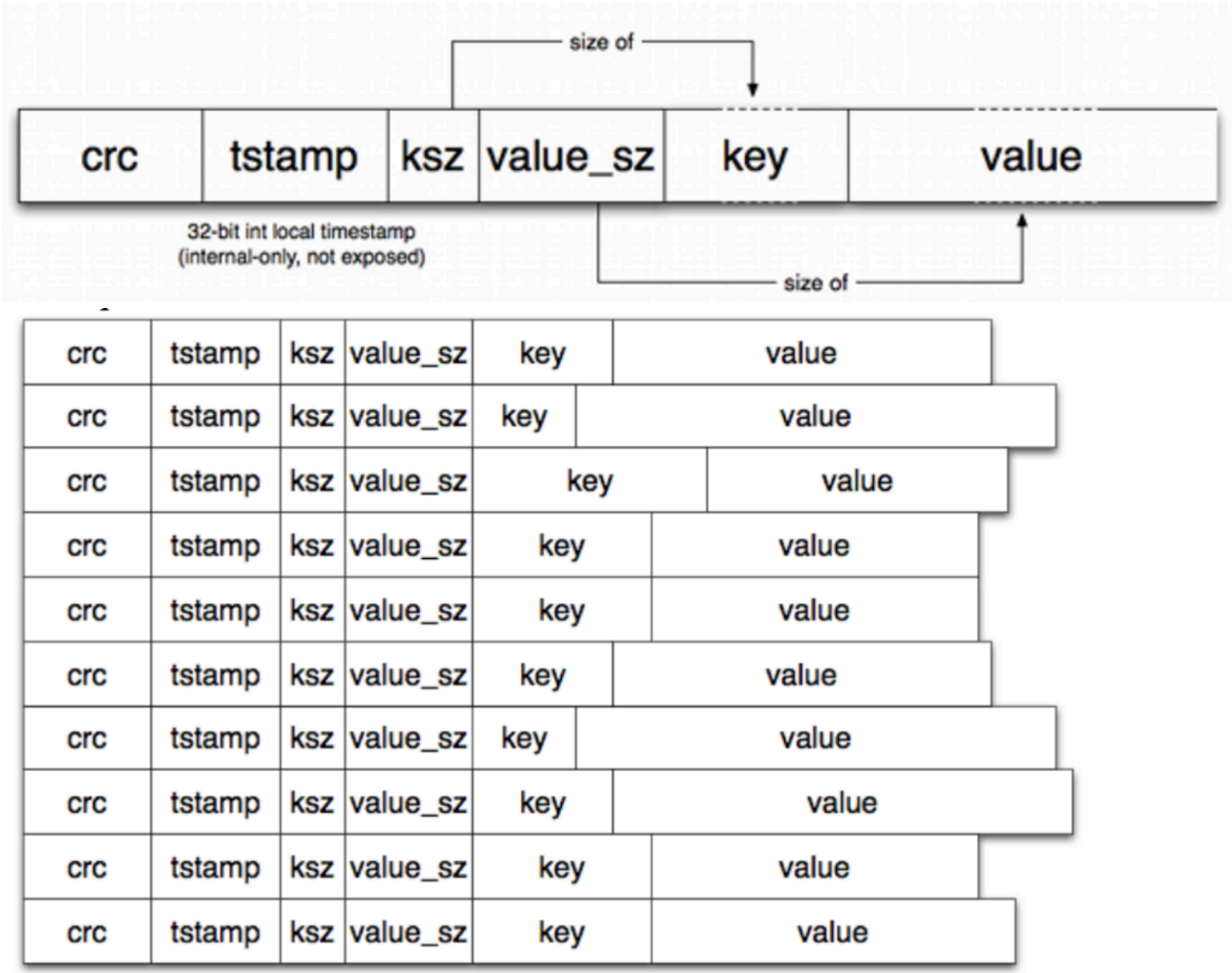
Appending to a log file

- Each database *instance* is stored in its own directory
- Each instance has only one active file / log. When it reaches a size threshold it becomes immutable and a new log file is started.



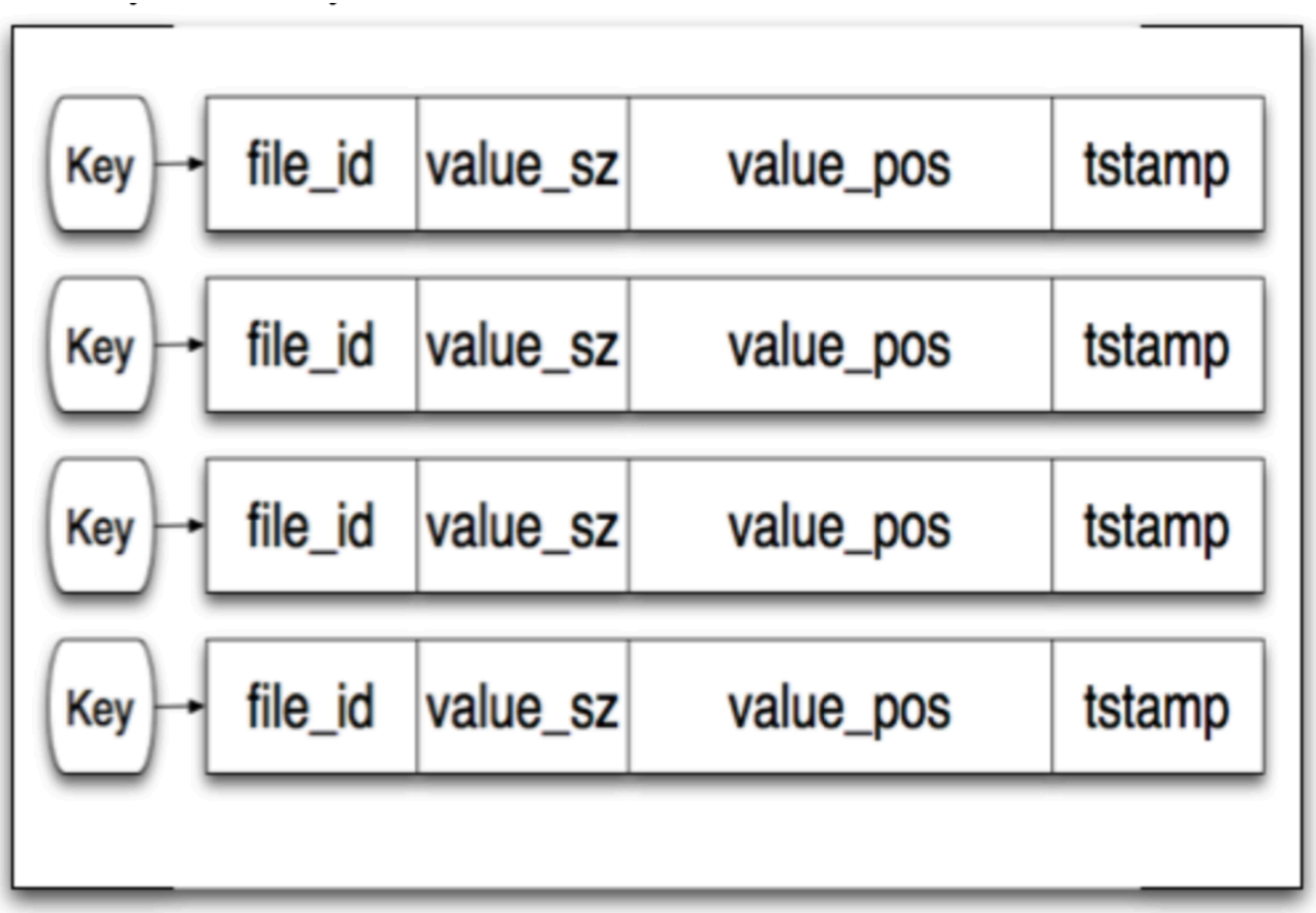
Sequential log files

- Appends to the active file are the **only** operation, whether you are inserting, updating, or deleting!
- Deletes use a special t*ombstone value* indicating that an item has been deleted.
- Two problems need addressing:
  a) Tracking locale of latest key value
  b) log files can grow unbounded



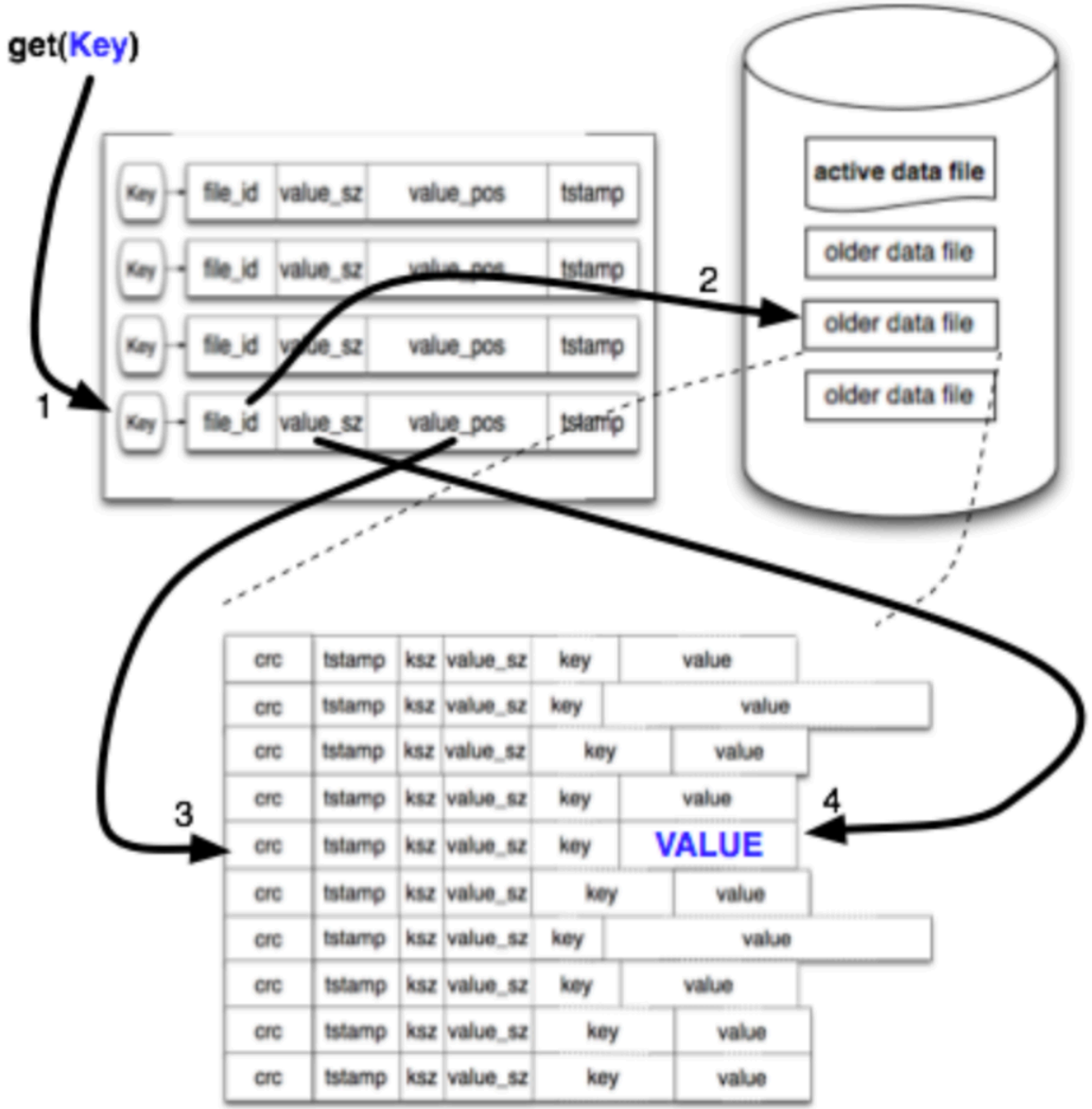
KeyDir (multiple sequential log files)

- On insert or update, the key is atomically updated with the value location of the latest value and a new record is written to the one active file.
- Old data still resides on disk but gets removed later.



Locating data using the KeyDir

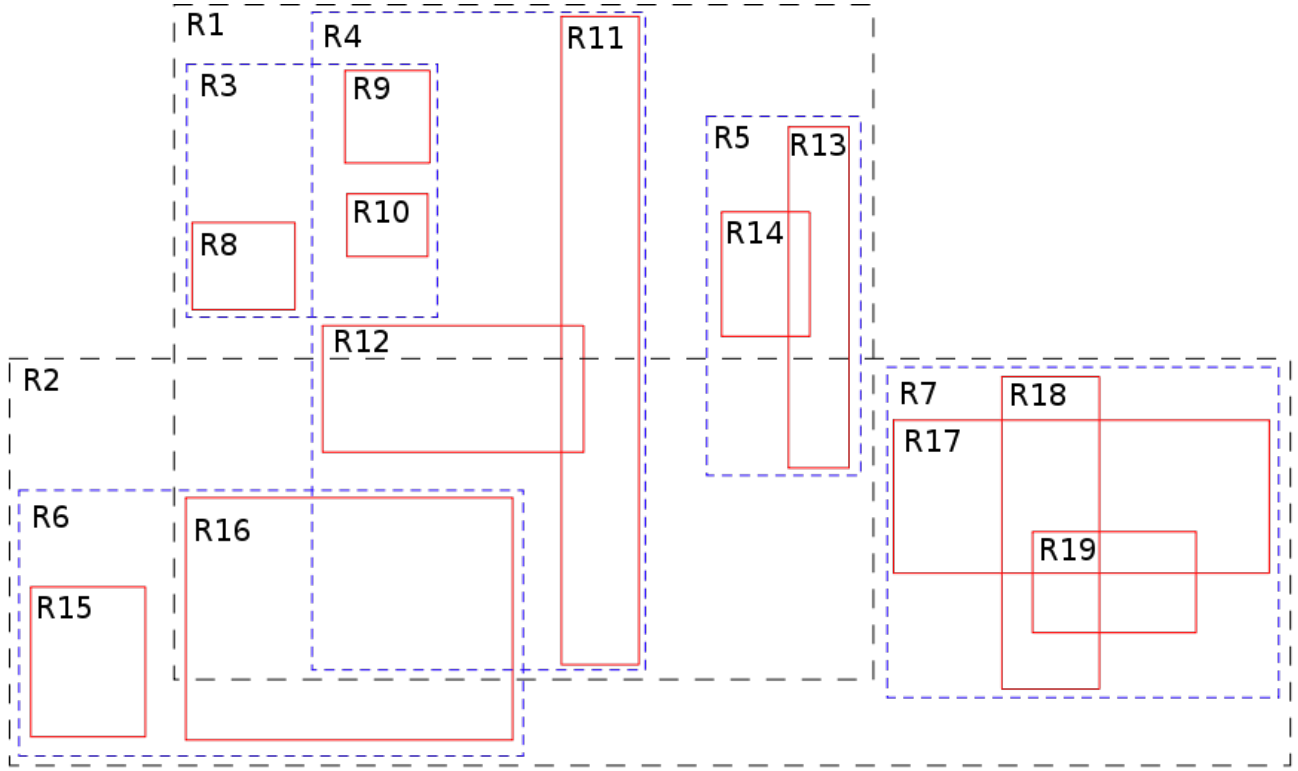
1. Look up key in keydir
2. Use KeyDir record to identify the correct file
3. Use the **value_pos** field to identify the start location of the value record
4. Use **value_sz** to fetch the bytes of the value.

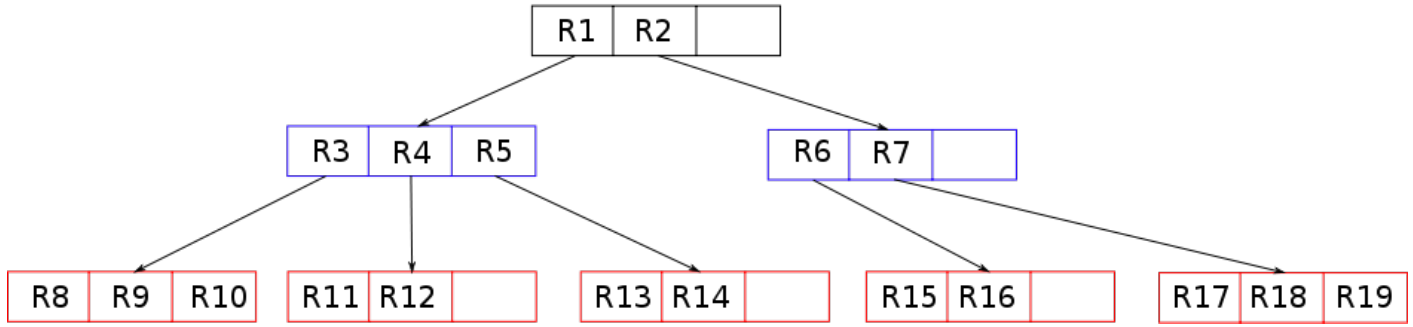


R-Trees: Multi-dimensional indexing

R-trees are tree data structures used for indexing multi-dimensional data

- Geospatial search
- Navigation
- Color similarity (r,g,b)

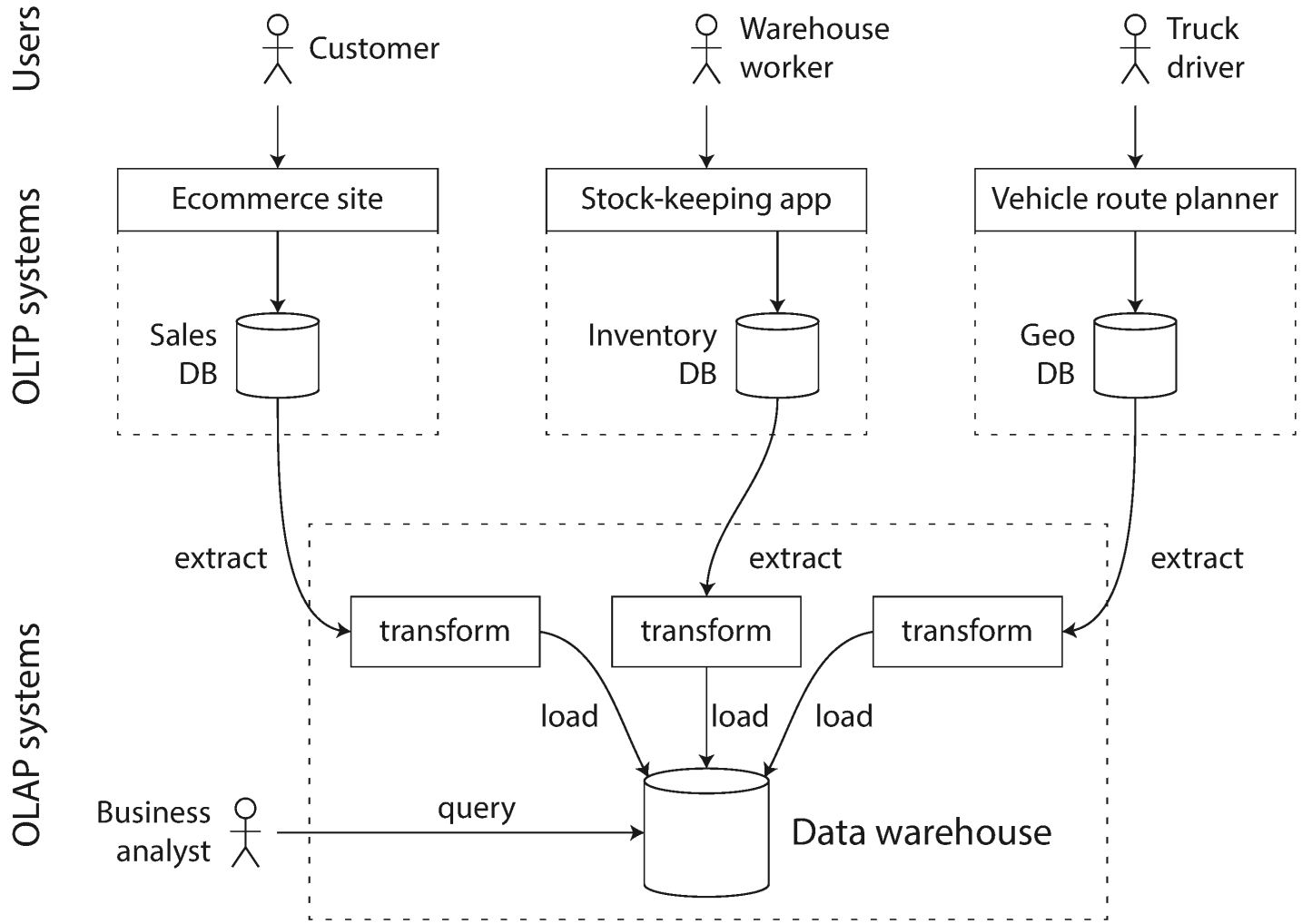




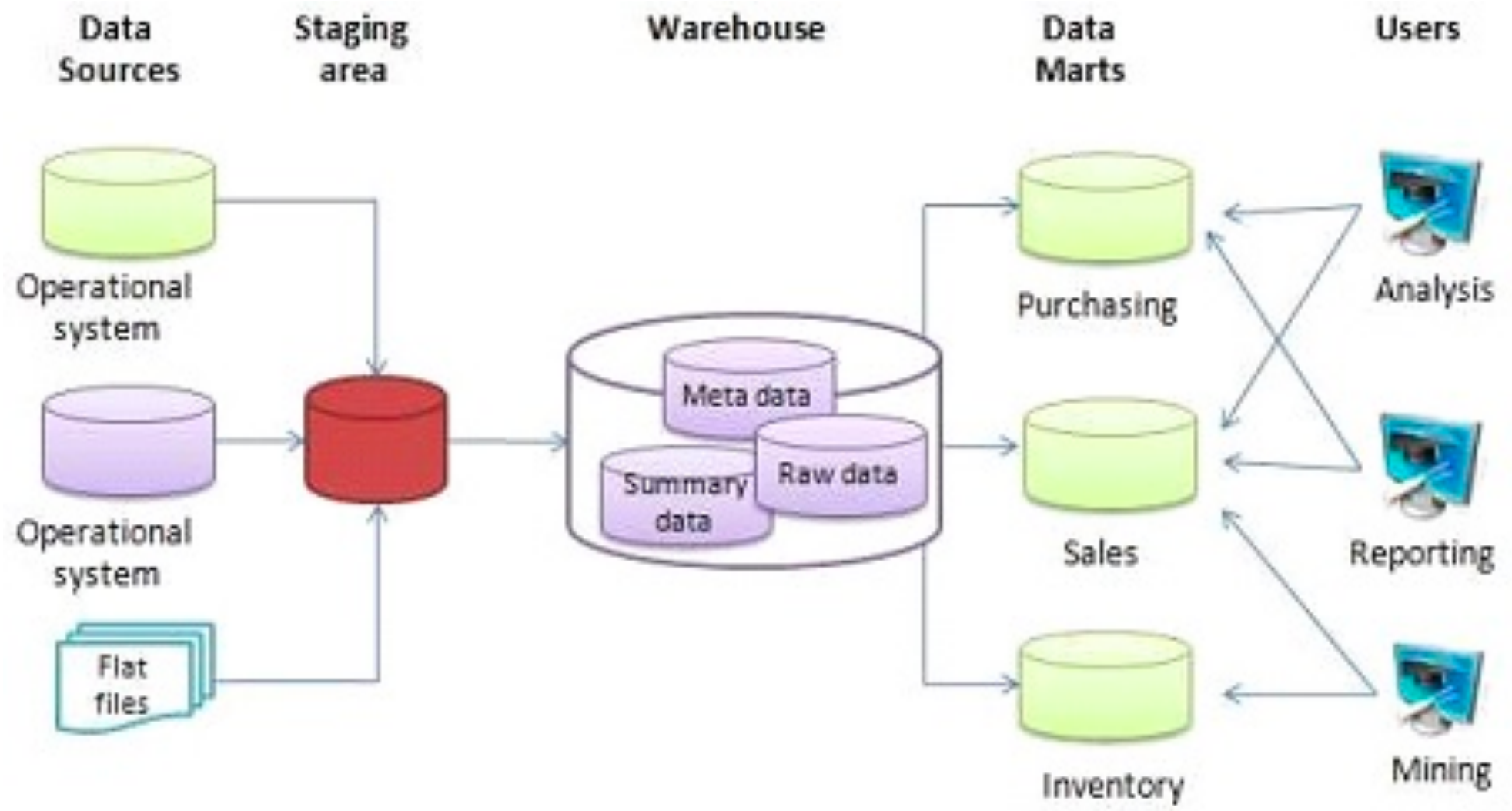
Data Warehousing

A *data warehouse* provides a separate resource for data analytics.

Why is this useful?



Data Warehouse → Data Marts



OLTP vs. OLAP

OLTP

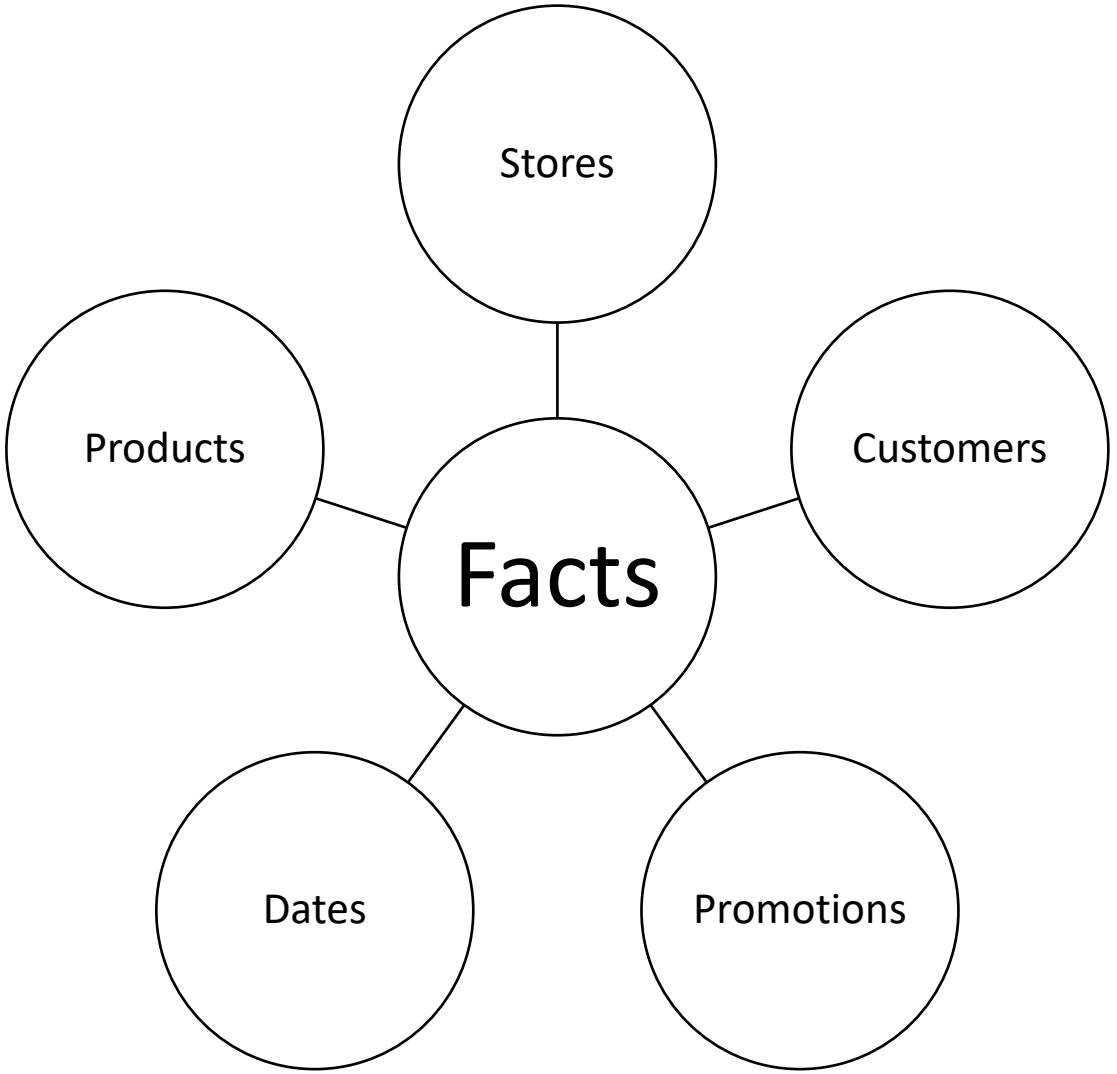
- Interactive
- Low latency
- Row-level operations

OLAP

- Batch-oriented
- Table-level operations
- ETL
- Aggregate Analytics
- Data warehousing

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

Star Schema



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

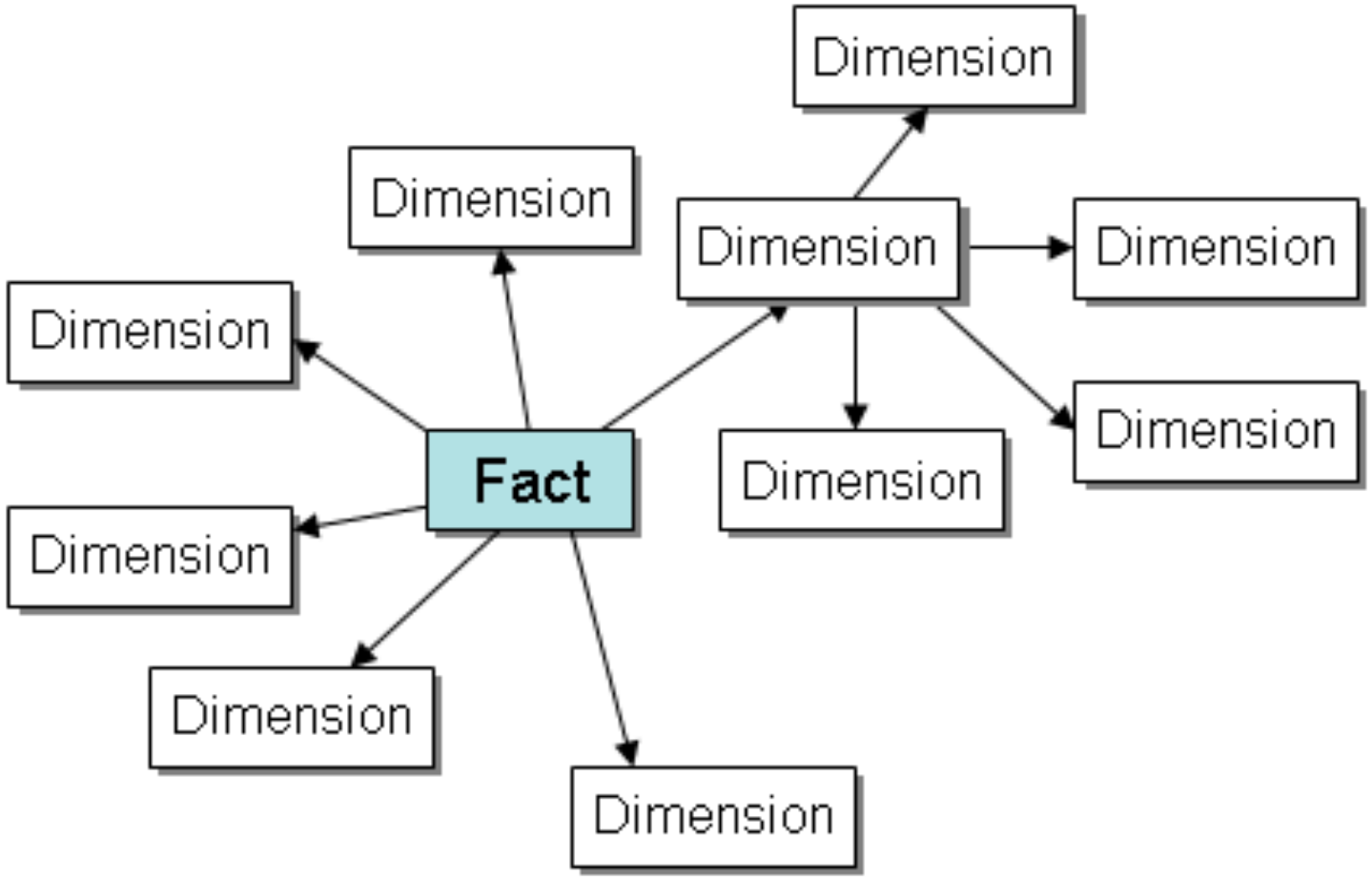
promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Snowflake Schema

Greater normalization increases complexity, and requires more joins

BUT:

Normalization reduces redundancy and (therefore) storage requirements.



Columnar Storage

Data from each column is stored contiguously on disk. Aggregates on particular columns require less disk I/O.

For example: parquet

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

Column Compression using Bitmaps

Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)