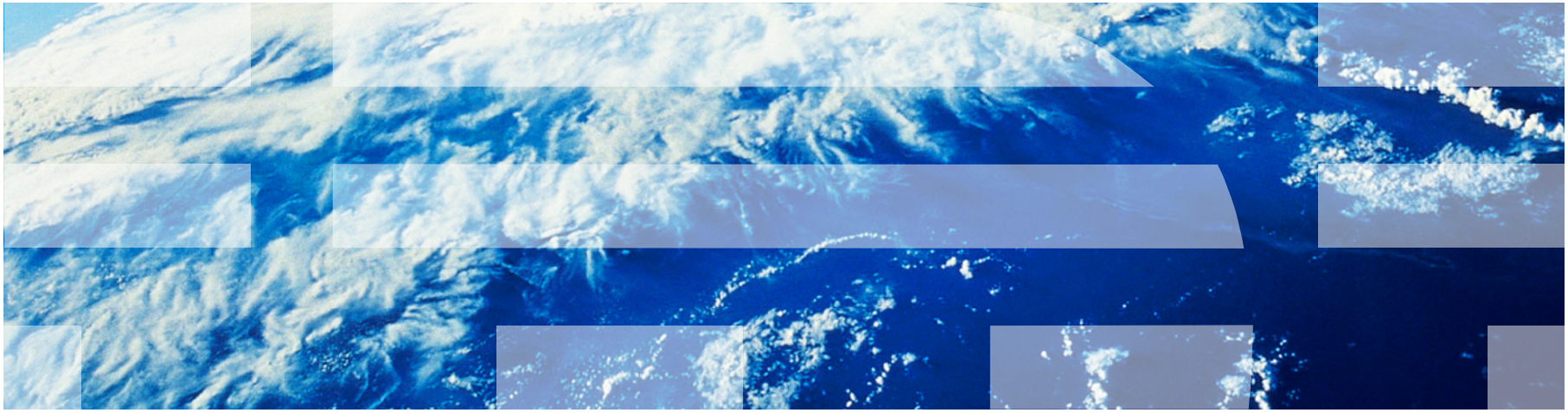
Borrowed from Shiva Shivakumar

Computer Systems for Data Science
Topic 3

Transactions and OLTP Databases



DBMS techniques and algorithms

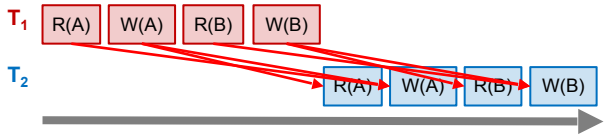
- Write-ahead logging — Provides A and D in ACID
- Serializability — Whether I happens in ACID
- 2-Phase Locking — Help C and I in ACID
- Indexing — Faster access
- Bloom filters — Reduce unnecessary access attempts
- Caching — Reduce work duplication

Anomaly terminologies
All of these are not allowed by serializability

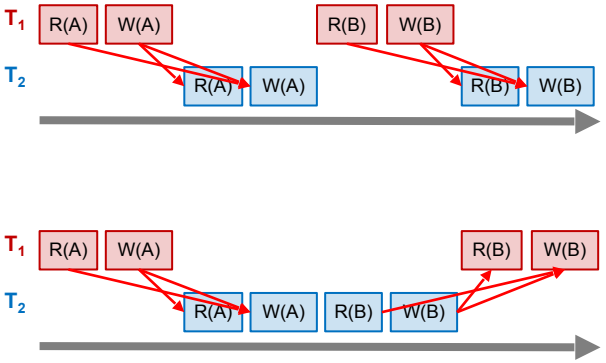
- **Lost updates** (write-write conflict)
  - Two users update the same row at the same time, and one of their updates is forever lost without being committed
- **Dirty reads** (write-read conflict)
  - User reads an update that was never committed
    - E.g., Tx updates but then aborts, Ty reads the update even though it wasn't committed
- **Unrepeatable reads** (read-write conflict)
  - User reads two different values for same row within the same transaction
    - E.g., Tx reads value A=0, Ty updates value A=1 and commits, Tx reads value A again, reads A=1
- **Phantom reads** (read-write conflict)
  - User reads same set of rows twice, but new rows are inserted/removed to/from that set by another committed transaction in between

What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



Interleaved Schedules:



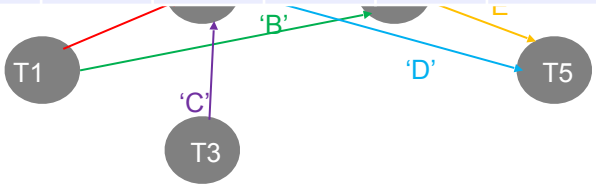
Example with 5 transactions

Schedule S1

Good or Bad schedule?
Conflict serializable?

	w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)
T1	w1(A)		w1(B)							
T2		r2(A)			R2(C)		w2(D)			
T3				w3(C)						
T4						r4(B)		w4(E)		
T5									r5(D)	w5(E)

Step1
Find conflicts
(RW, WW, WR)



Step2
Build Conflict graph
Acyclic?

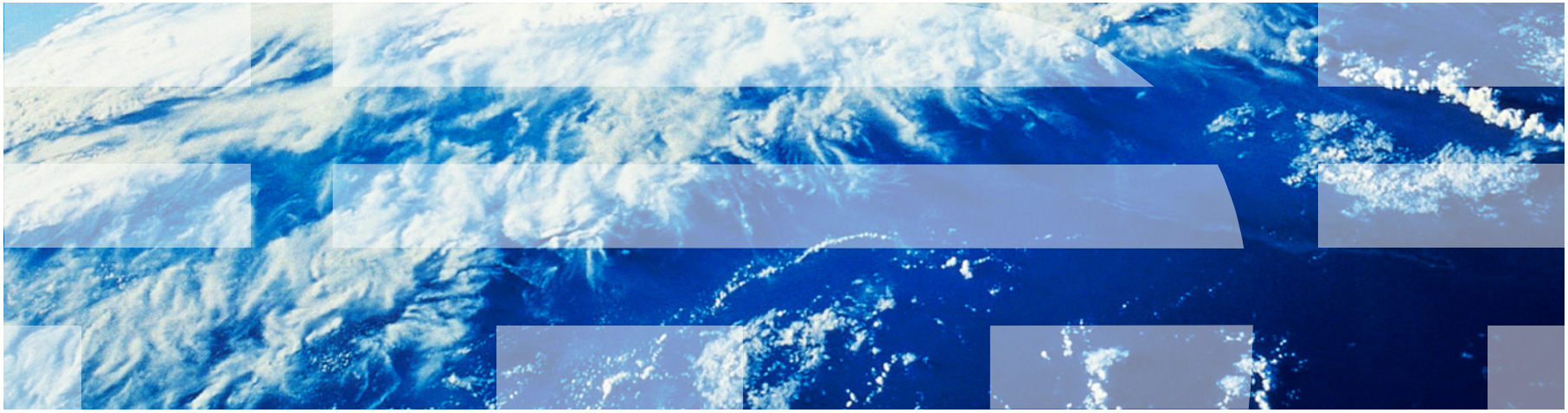
Acyclic
⇒ Conflict serializable!
⇒ Serializable

S2

T3	T1	T1	T4	T4	T2	T2	T2	T5	T5
w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)	r2(C)	w2(D)	r5(D)	w5(E)

Step3
Example serial schedule
Conflict Equiv to S1

2PL: A Simple Locking Algorithm



Strict Two-Phase Locking (2PL)

- Algorithm: *strict two-phase locking* - as a way to deal with concurrency
  - Guarantees conflict serializability
  - (if it completes- see upcoming…)

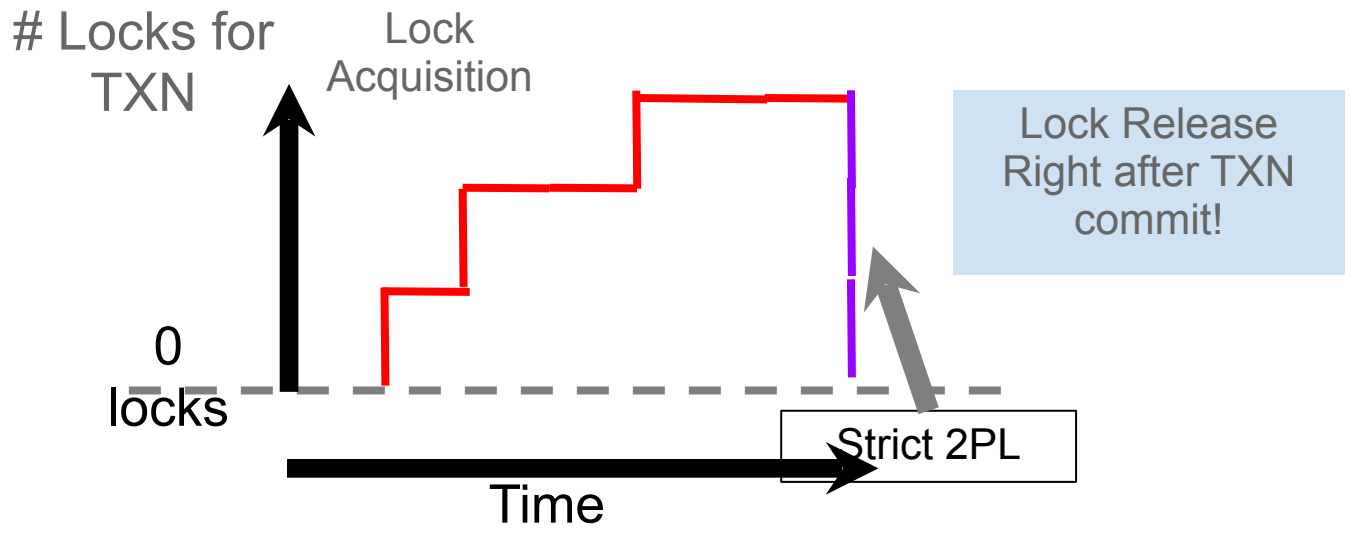
- Also (*conceptually*) straightforward to implement, and transparent to the user!

Strict Two-phase Locking (2PL) Protocol

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Picture of 2-Phase Locking (2PL)



2PL: A transaction can not request additional locks once it releases any locks. Thus, there is a “growing phase” followed by a “shrinking phase”.

Strict 2PL: Release locks right after COMMIT (COMMIT Record flushed) or ABORT

Strict 2PL

If a schedule follows strict 2PL, it is **conflict serializable**…

- …and thus serializable
- …and we get isolation & consistency!

Popular implementation

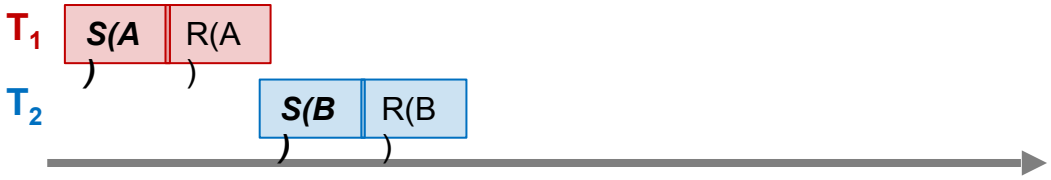
- Simple !
- Produces subset of *all* conflict serializable schedules
- There are MANY more complex LOCKING schemes

- One key, subtle problem (next)

Deadlock Detection

First, T_1 requests a shared lock on A to read from it

Deadlock Detection



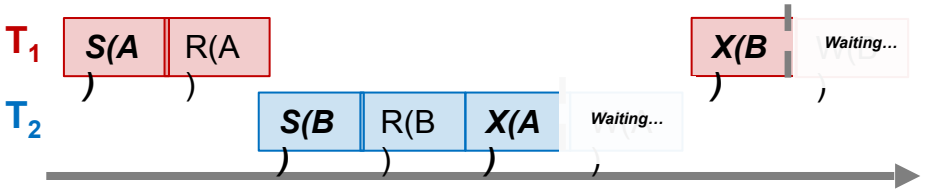
Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

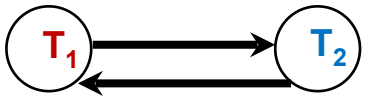
T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Waits-For graph: Track which Transactions are waiting
IMPORTANT: WAITS-FOR graph different than CONFLICT graph we learnt earlier !

Deadlock Detection: Example



Waits-for graph:



Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- Deadlock prevention
- Deadlock detection

Deadlock Prevention: C2PL

Conservative 2 Phase Locking (C2PL)

- Obtains all locks before the transaction begins
- If cannot obtain ALL locks, release and try again

- Ensures that no deadlocks occurs
- BUT: can degrade performance

Deadlock Prevention: Wait-Die / Wound-Wait

- Transactions assigned an ID (e.g., based on their start time)
  - T1, T2, …
  - ID determines priority (priority of T1 > T2 > T3, …)
  - If Tx wants a lock that Ty holds, we have two deadlock avoidance algorithms

- Scheme 1: Wait-Die (non-preemptive):
  - If Tx has higher priority, Tx waits for Ty, else Tx aborts
- Scheme 2: Wound-Wait (preemptive):
  - If Tx has higher priority, Ty aborts, else Tx waits

- In both cases aborted transactions start again some random later with the same time stamp (priority)
- What is the shape of the wait graph under both of these schemes?
- What are the pros and cons of each?

Alternative approach: deadlock detection

Create the **waits-for graph**:

- Nodes are transactions
- There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*

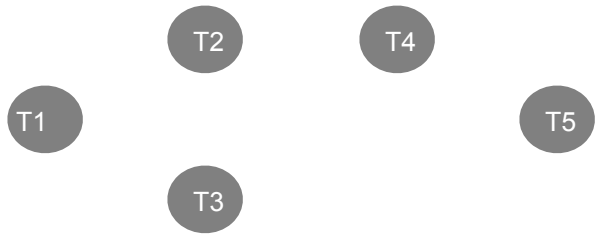
Periodically check for (***and break***) cycles in the waits-for graph

Example with 5 Transactions (2PL)

Operation Arrival

Execute with 2PL

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)



Waits- For Graph

Example with 5 Transactions (2PL)

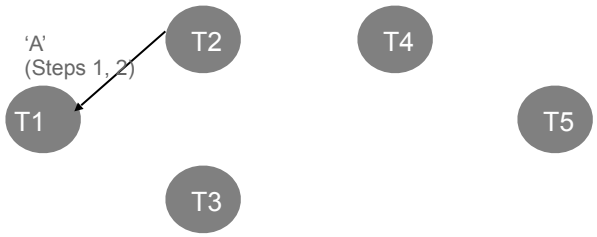
Operation Arrival

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Execute with 2PL

T1 **T2** **T3** **T4** **T5**

	T1	T2	T3	T4	T5
Step 0	X (A) w1(A)				
Step 1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3		Get S(A) r2(A)			
Step 4			X (C) w3(C) Unl C		
Step 5		S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S(D) r5(D)
Step 10					X (E) w5(E) Unl D, E



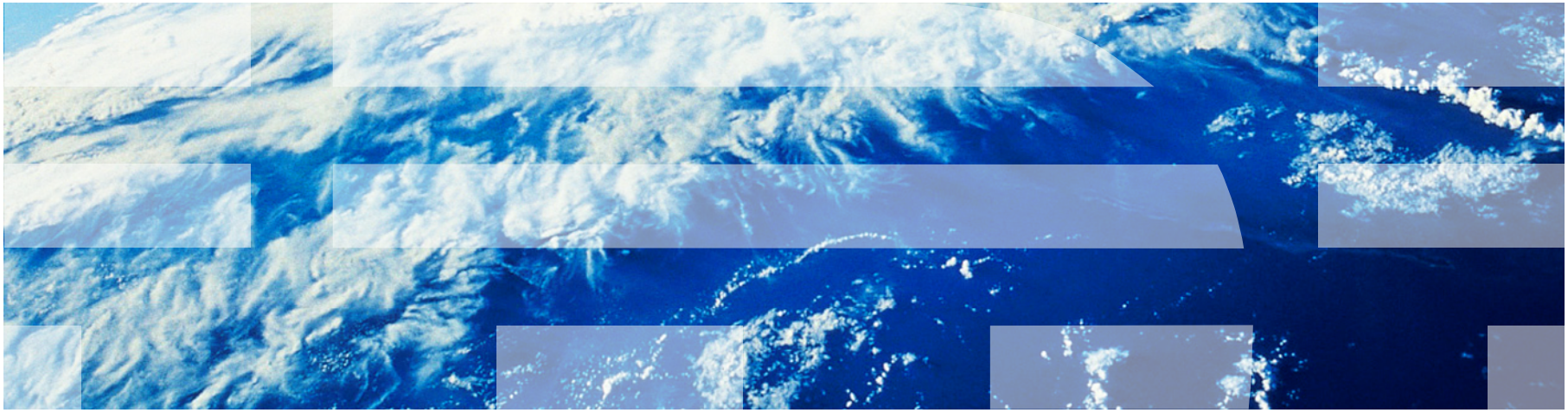
Waits- For Graph

Summary

Locking allows only conflict serializable schedules

- If the schedule completes… (it may deadlock!)
- → No guarantee that the schedule will finish!

Other Isolation Levels Beyond Serializability



Real-world Isolation Guarantees

- Serializability is unfortunately rare!
- Some databases claim to be serializable, but actually are not!

- **Read committed**: very common in SQL OLTP
  - Any data that was read was committed
  - Disallows:
    - Dirty reads
  - Allows:
    - Lost updates
    - Unrepeatable reads
    - Phantom reads

- **Repeatable reads**: stronger than read committed, by still allows lost updates and phantom reads

Strict serializability, the gold standard

- Limitations of serializability:
  - No constraints on equivalent order of transactions
  - No relationship between final serial order and when the transactions were submitted
  - Two databases given identical set of transactions in the same order at the same time may produce very different results

- Strict serializability is even stronger than conflict serializability
- Definition: If transaction Y starts after transaction X commits (X and Y are not concurrent)
  1. Final state is equivalent to serial order
  2. X must have been executed before Y in that serial order

2PL produces **strictly serializable** schedules

Why?

An Empirical Evaluation of In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

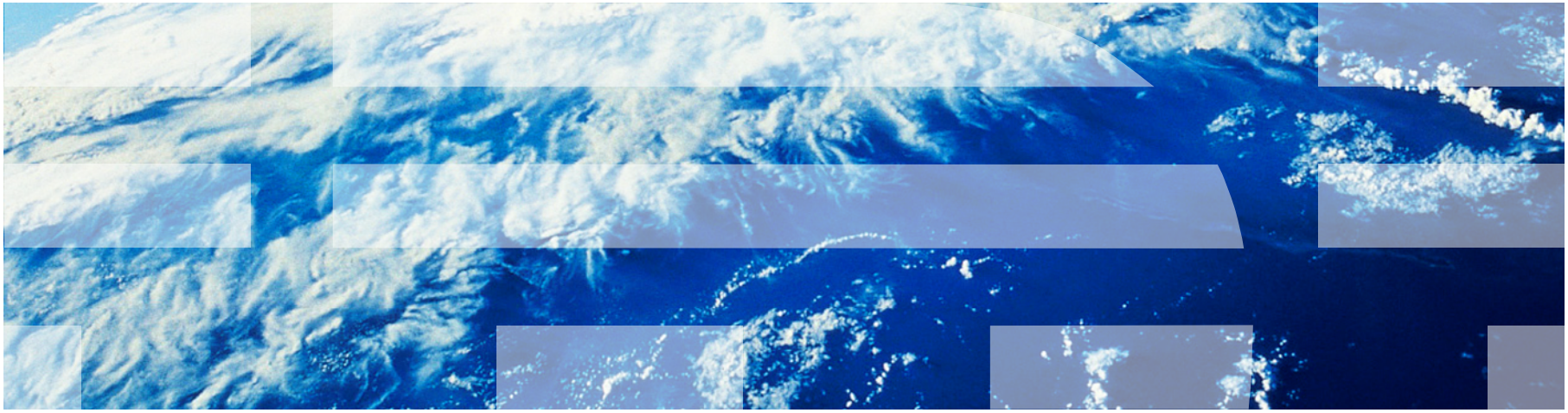
To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory

https://www.vldb.org/pvldb/vol10/p781-Wu.pdf

Indexing



Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key**: attribute used to look up records in a file.
- An **index** consists of records (called **index entries**) of the form

- The index is typically much smaller than the original data
  - E.g., 100X smaller
- How big does the pointer need to be?
  - Example: A pointer to a specific byte within 1GB of data
    - 1GB can be represented as an array with 30 bits (0's or 1's)
    - 000…00 points to the start of 1GB
    - 000…01 points to the first byte
  - General idea: take the size of the data, and figure out how many bits need to represent it
- Where would you store the index?
  - Usually in memory
  - → Index has to be small, since memory capacity is limited

Index Evaluation Metrics

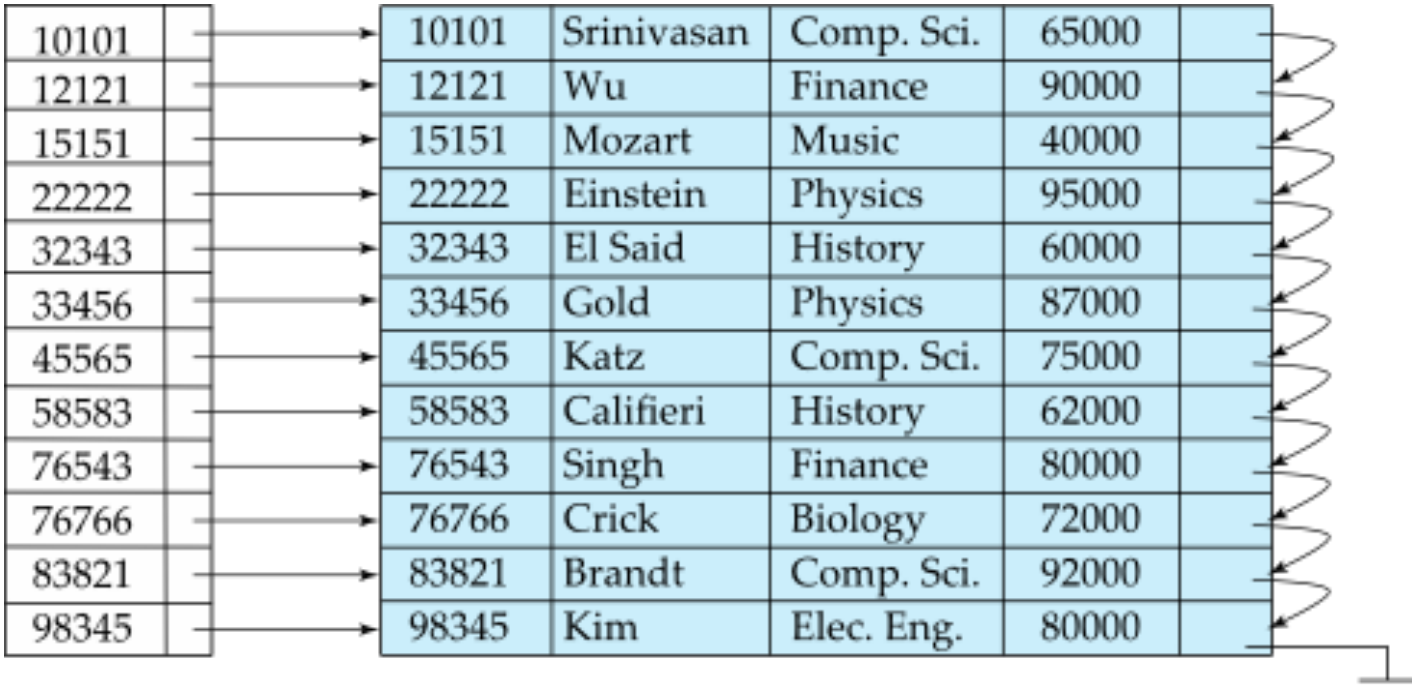
- Access types supported efficiently:
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values
- Access time
- Insertion time
- Update time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index,** index entries sorted on the search key
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file
  - We'll show what this means in a bit

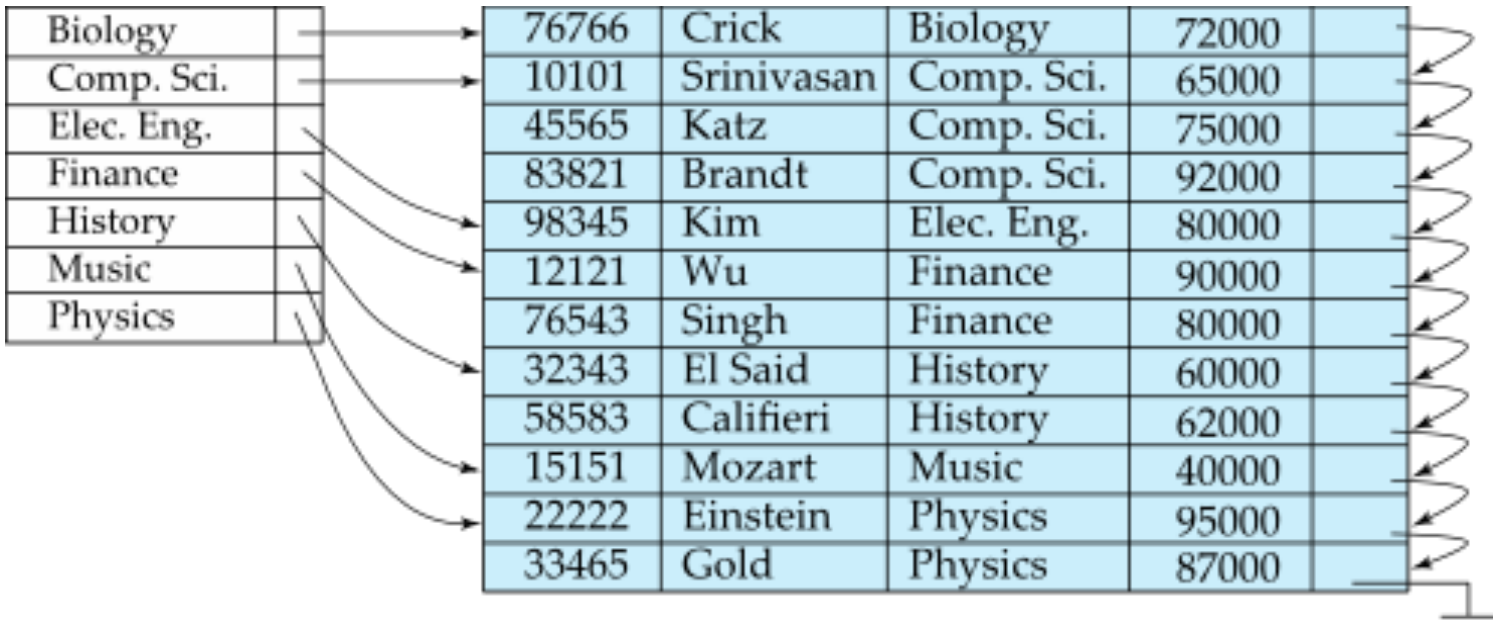
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the database.
- E.g. index on *ID* attribute of *instructor* relation



Dense Index Files: non-unique attribute as search key

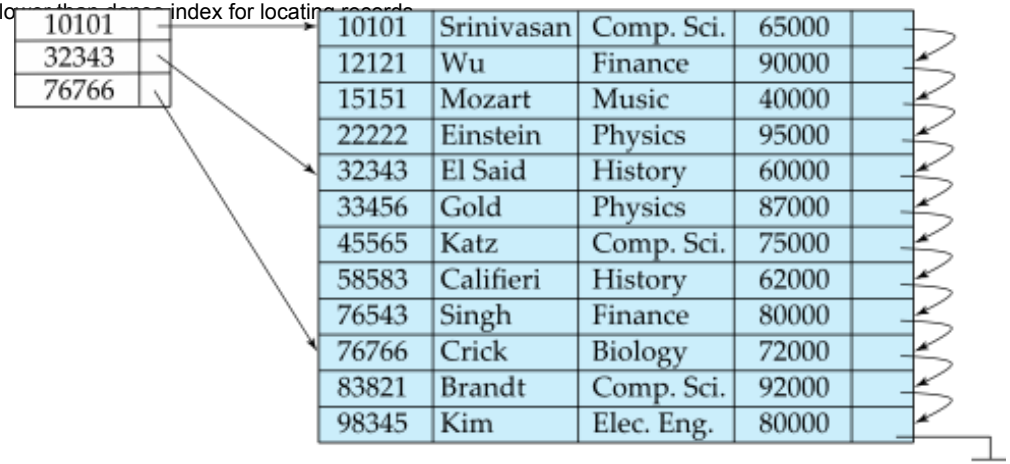
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records

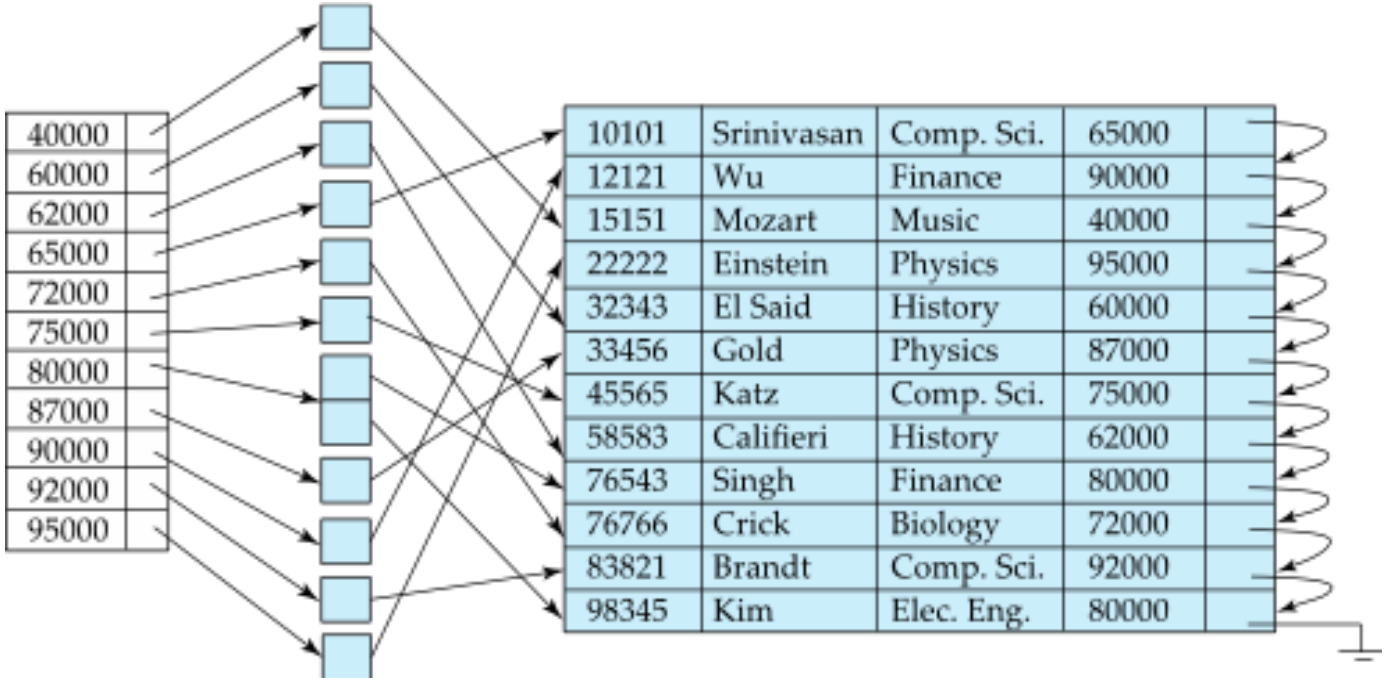


10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Secondary Index Example

- Secondary index on salary field of instructor

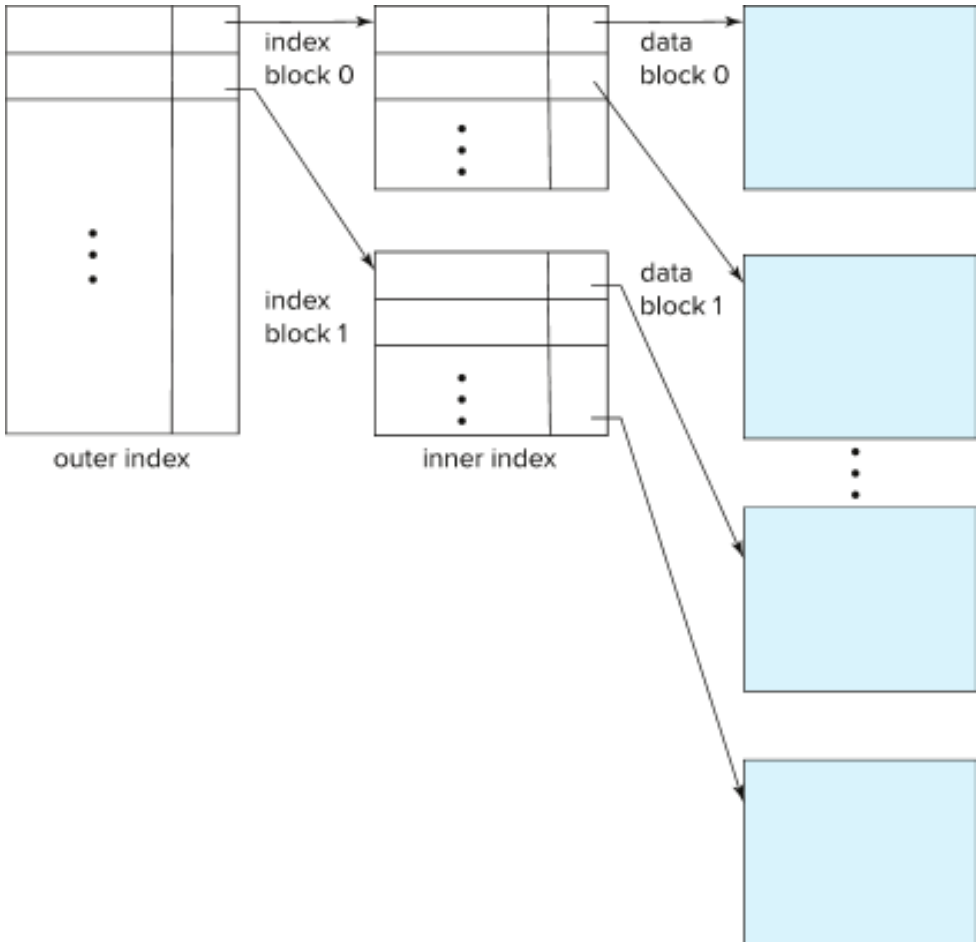


- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense
  - Why?

Multilevel Index

- If index does not fit in memory, access becomes expensive
  - Usually databases try to keep the index in memory if they can!
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.
- File systems often use a multilevel index (e.g., nested directories)

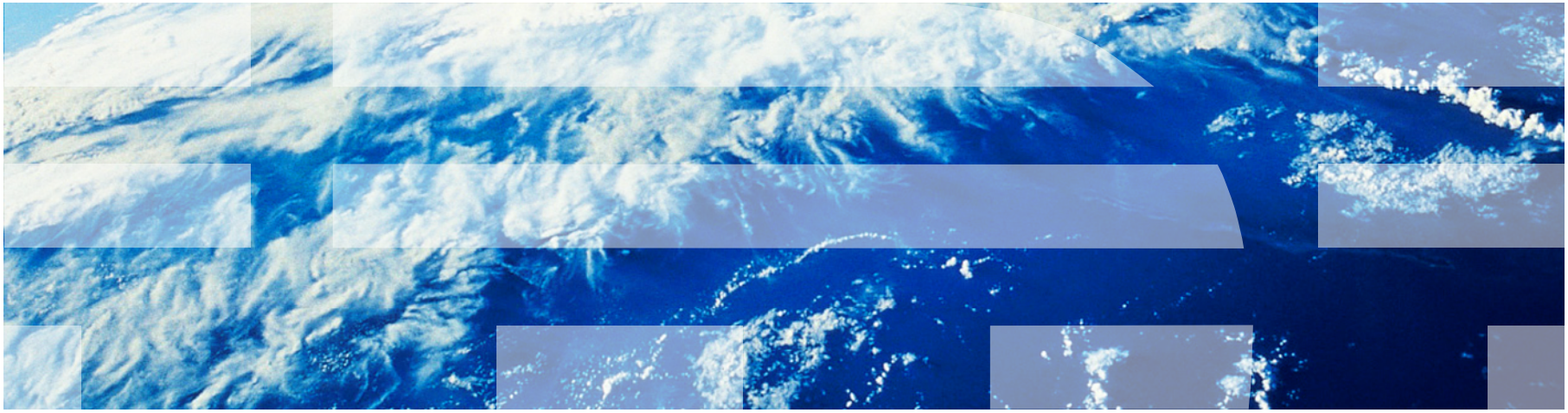
Multilevel Index (continued)



Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

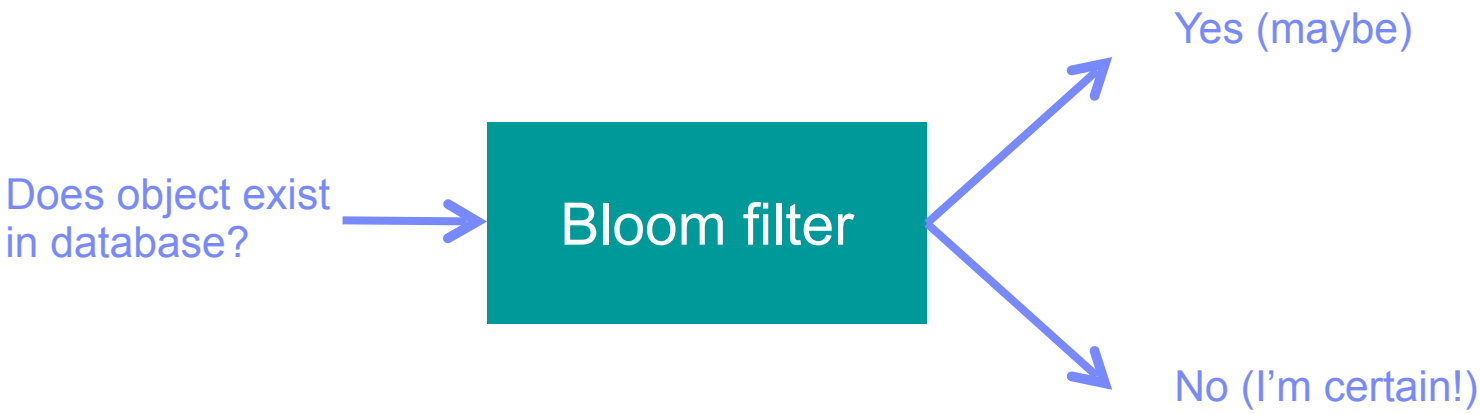
Bloom Filter



Motivation: Index doesn't fit in memory

- Example of a typical database:
  - 64GB memory
  - 2TB flash disk
  - Average key+value (e.g., entry) size: 50B → database can store up to 40 billion keys!
- A pointer needs to locate a key among 40 billion keys
  - Minimum pointer size: 36 bits ~ 4.5 bytes (let's round it to 5 bytes)
  - Index size = number of keys * byte rounded pointer size ~ 200GB
  - Does not fit in memory
- Therefore, we must use a multi-level index
  - Outer index in memory
  - Inner index on disk
- Minimum time reading a single object: 1 memory access + 2 flash accesses ~ 200us
- **What if object doesn't exist in the database?**
  - Still 200us!
- Can we do better?

Bloom filters [Bloom 1970]:
Approximate way to determine if object exists



Bloom filter parameters

- An array of m bits (can only be ‘0’ or ‘1’)
  - Array initialized to 0
- k independent **hash functions** $h_1, h_2, \ldots h_k$ that return a number between $1, \ldots m$

What is a hash function?

- What is a hash function?
  - $h(x) = y$, where x is an input (e.g., a key) and y is a uniformly random number
  - In our case, y is a random number between $1, .. \ m$

- If $h(x_1) = h(x_2)$, there is some probability that $x_1 = x_2$
- If $h(x_1) \ != h(x_2)$, we know **for sure** that $x_1 \ != x_2$

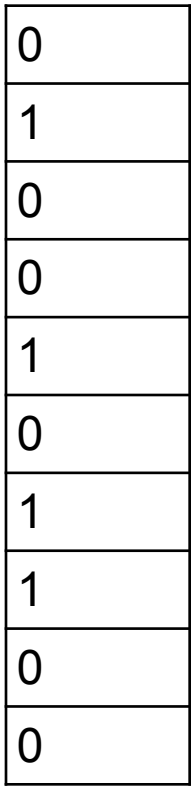
Algorithms: check membership and add membership

- To check if x is a member of the bloom filter, check whether $h_1(x), \ldots h_k(x)$ are all set to 1
  - If not, x is definitely not a member
  - If yes, x might be a member
    - There can be false positives!

- To add x, set the positions of $h_1(x), \ldots h_k(x)$ all to 1
  - Some positions might already have been set as 1

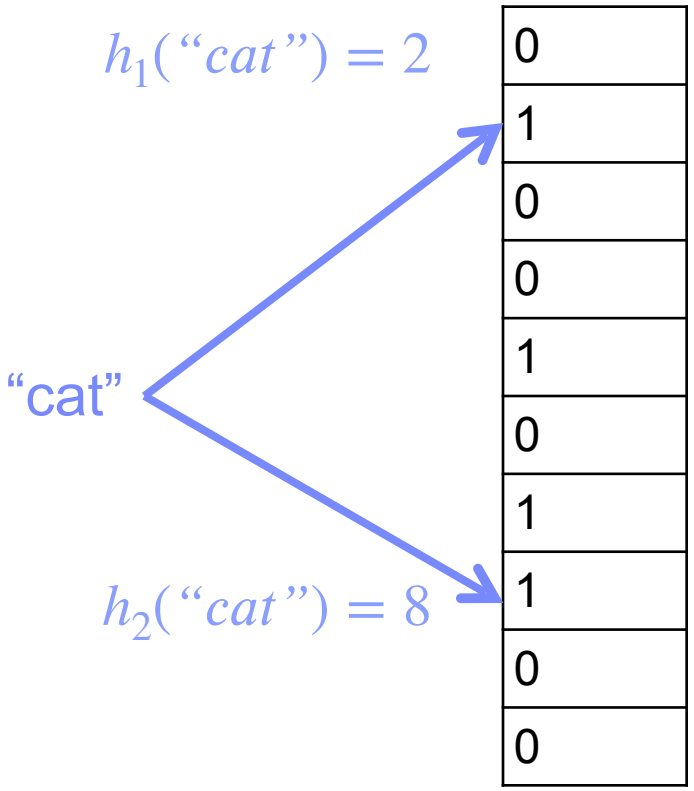
How does a bloom filter work? Example

m = 10
k = 2



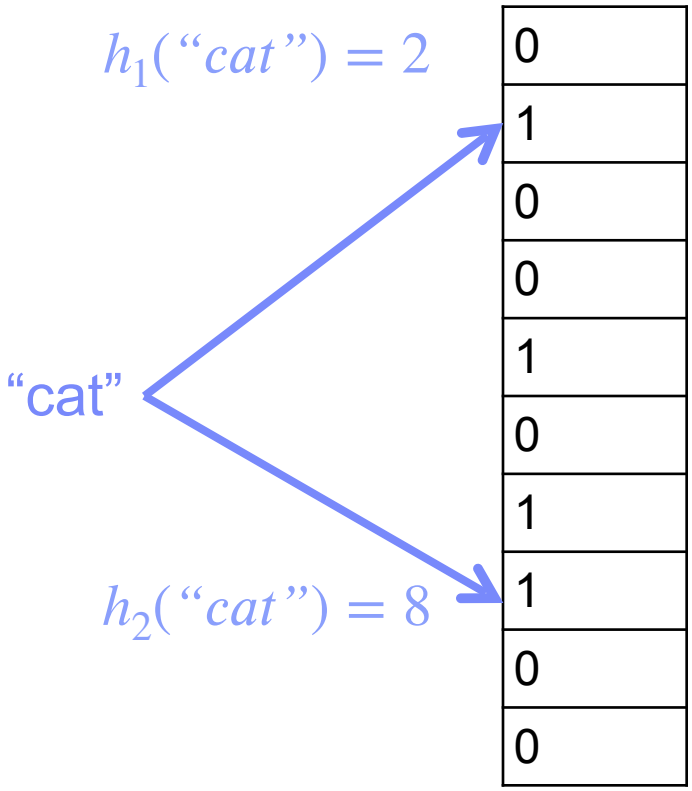
Is cat in DB?

m = 10
k = 2



Is cat in DB? Maybe!

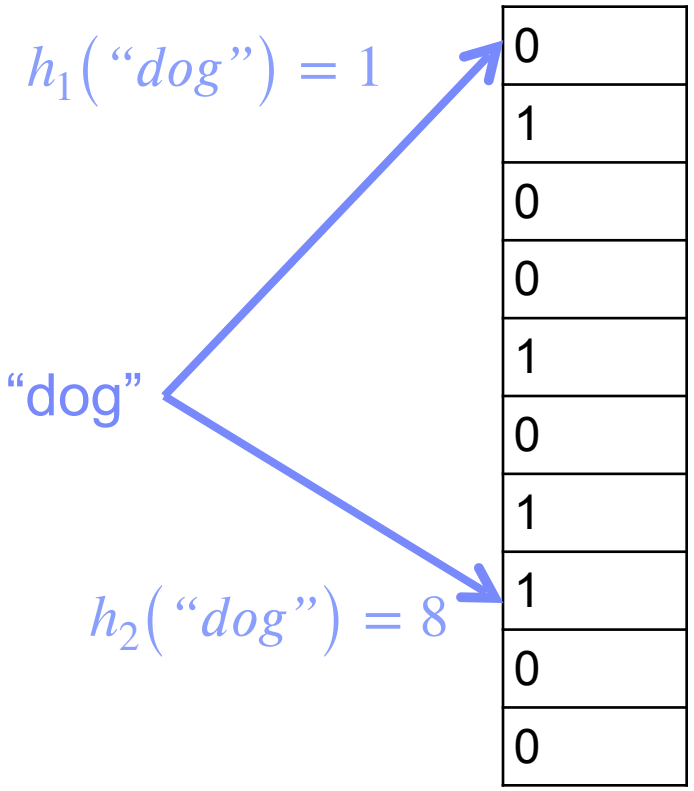
m = 10
k = 2



Cat might exist in DB!

Is dog in DB?

m = 10
k = 2

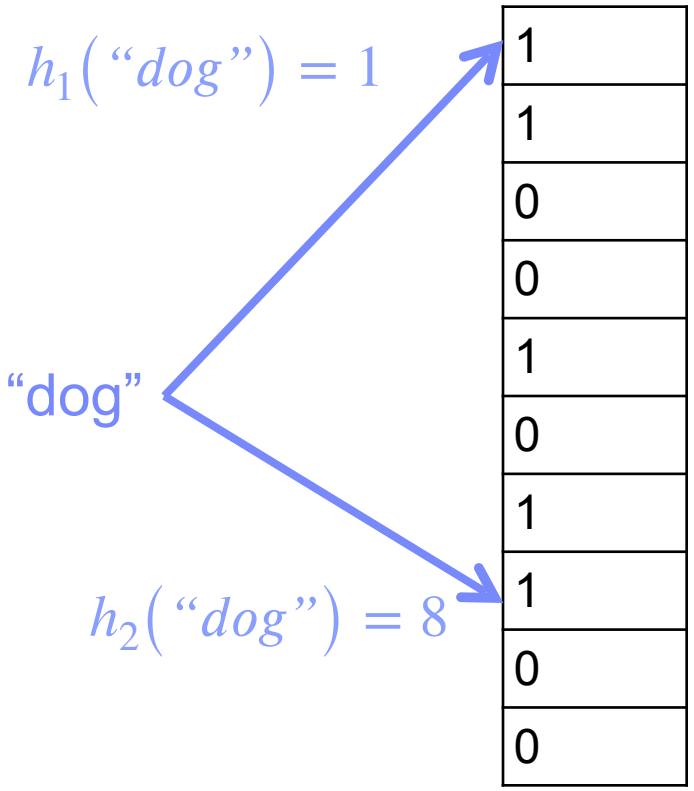


Dog definitely does not exist in DB

→ We don’t need to read from disk

Add dog to DB

m = 10
k = 2

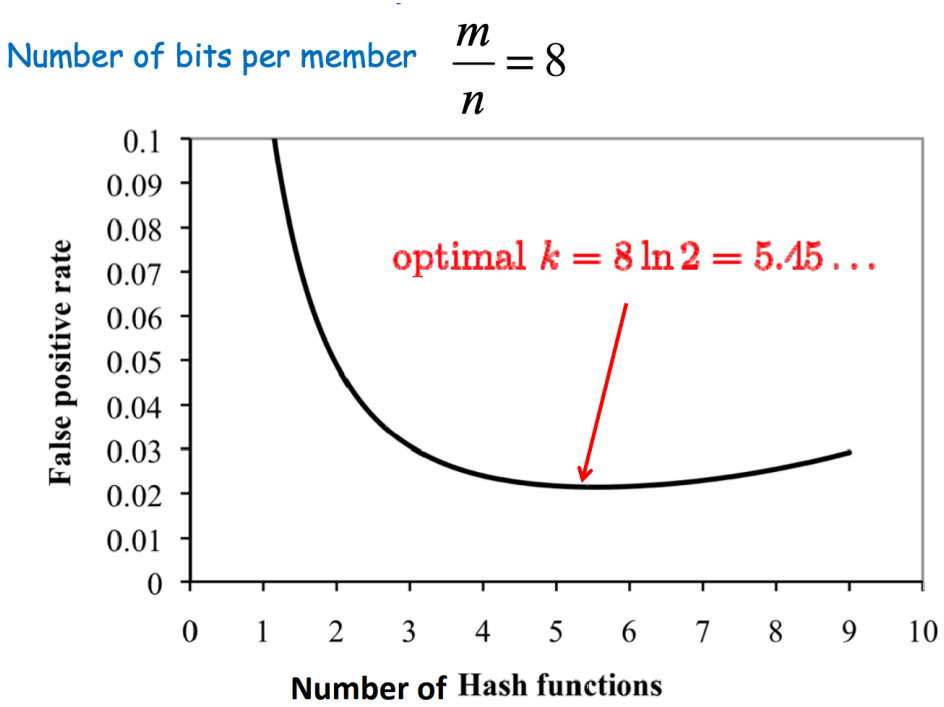


Goal: low false positives

- Define n as the total number of unique objects that might ever be inserted into the database
  - For example, for a bank database that uses account ID as keys, this is the total number of accounts the bank will ever have
- Let's assume $kn < m$

False positive probability (credit: Simon S. Lam)

- The size of k is a trade-off:
  - A higher k increase the number of hash functions that might map to 0
  - But also ”depletes” the available 0 slots in the bloom filter
- Optimal k: $k = \frac{m}{n} ln2$



Bloom filter calculator (https://hur.st/bloomfilter/)

☐ Bloom Filter Calculator ☐

Bloom filters are space-efficient probablistic data structures used to test whether an element is a member of a set.

They're surprisingly simple: take an array of **m** bits, and for up to **n** different elements, either test or set **k** bits using positions chosen using hash functions. If all bits are set, the element *probably* already exists, with a false positive rate of **p**; if any of the bits are not set, the element *certainly* does not exist.

Bloom filters find a wide range of uses, including tracking which articles you've read, speeding up Bitcoin clients, detecting malicious web sites, and improving the performance of caches.

This page will help you choose an optimal size for your filter, or explore how the different parameters interact.

---

n

Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)

4000

p

Probability of false positives, fraction between 0 and 1 or a number indicating 1-in-p

1.0E-7

m

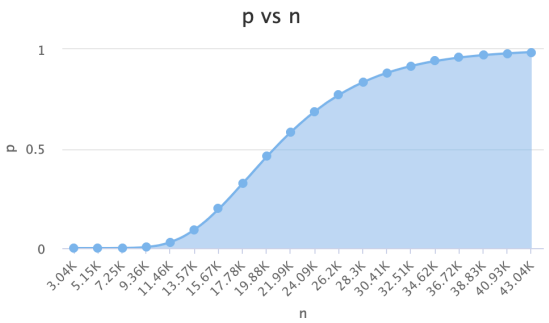
Nu**m**ber of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)

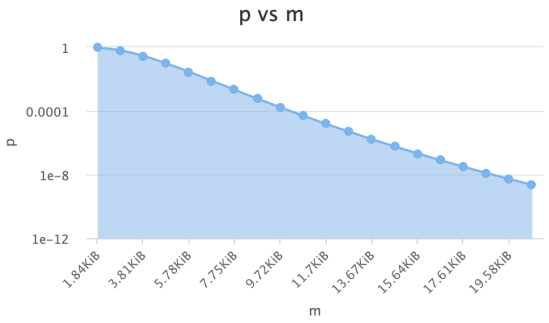
k

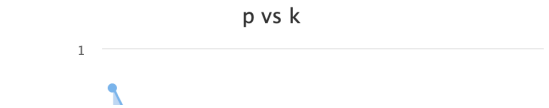
Number of hash fun**c**tions

Submit

n = 4,000
p = 0.0000001 (1 in 9,994,297)
m = 134,191 (16.38KiB)
k = 23





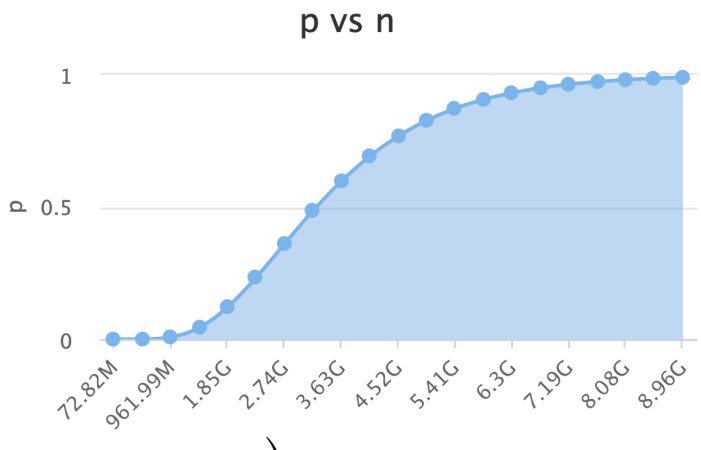


Trade off memory space vs. speed

- Larger bloom filter:
  - Reduces false positives → fewer reads to disk → lower latency, higher throughput
- But…
  - Takes up more space in memory → less space for caching database index entries in memory → higher chance of going to disk → higher latency, lower throughput

Example of space vs. speed trade off

- Example: database has 2GB of memory for caching and 100GB of flash
  - Needs to support 1 billion entries (each ~100B)
- Flash access is 100us, memory access is 100ns
- Which bloom filter parameters would you choose?
  1. 600MB bloom filter:
     - Bloom filter false positive rate of 9%
     - 1.4GB of DRAM left → index cache hit rate of 90%
  2. 1.1GB bloom filter:
     - Bloom filter false positive rate of 1%
     - 0.9GB of DRAM left → index cache hit rate of 70%



- 60% of requests return object does not exist:
- $$\textbf{avg latency} = \mathbb{P}(exists) \bullet \left(\mathbb{P}(cached) \bullet 100ns + \mathbb{P}(not\_cached) \bullet 100\mu s\right) +$$

$$\mathbb{P}(doesn't\_exist) \bullet \left(\mathbb{P}(true\ negative) \bullet 100ns + \mathbb{P}(false\ positive) \bullet 100\mu s\right)$$

- Scenario 1
  - 0.4 * (0.9 * 100ns + 0.1 * 100us) + 0.6 * (0.91 * 100ns + 0.09 * 100us) = **9.49us**
- Scenario 2:
  - **12.64us**

Other issues with bloom filters

- Can get “depleted” over time
- Need to estimate number of unique entries in advance
- Do not support deletes!
  - Why?
- Improvements: counting bloom filters, cuckoo filters, learned bloom filters, elastic bloom filters… This is a hot research area!

Algorithmic Nuggets in Content Delivery

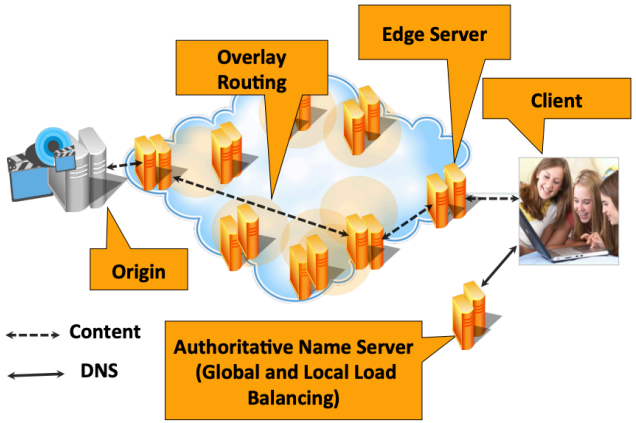
Bruce M. Maggs
Duke and Akamai
bmm@cs.duke.edu

Ramesh K. Sitaraman
UMass, Amherst and Akamai
ramesh@cs.umass.edu

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

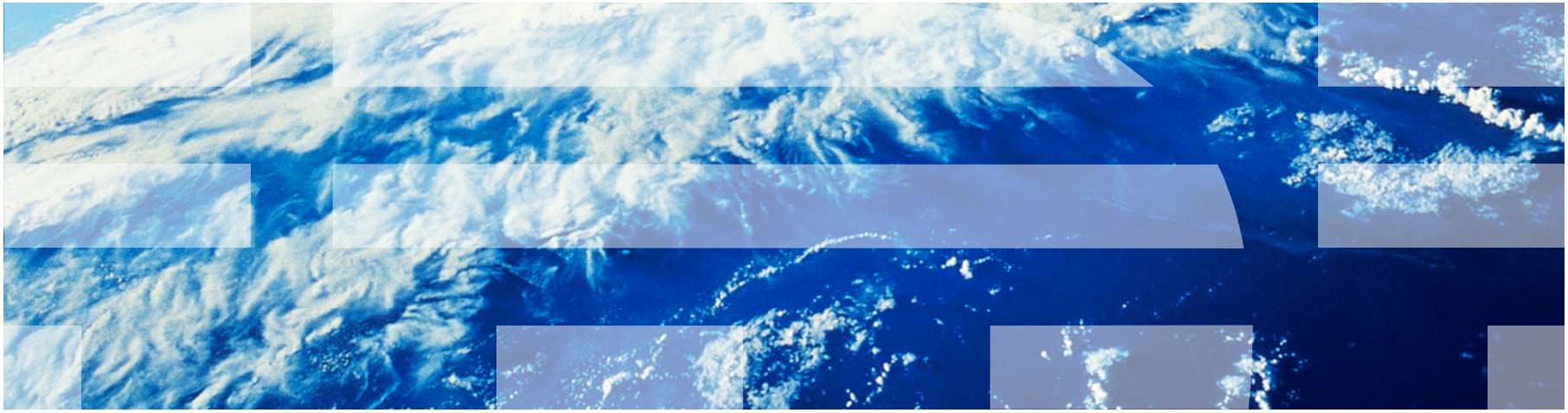
ABSTRACT

This paper "peeks under the covers" at the subsystems that provide the basic functionality of a leading content delivery network. Based on our experiences in building one of the largest distributed systems in the world, we illustrate how sophisticated algorithmic research has been adapted to balance the load between and within server clusters, manage the caches on servers, select paths through an overlay routing network, and elect leaders in various contexts. In each instance, we first explain the theory underlying the algorithms, then introduce practical considerations not captured by the theoretical models, and finally describe what is implemented in practice. Through these examples, we highlight the role of algorithmic research in the design of complex networked systems. The paper also illustrates the close synergy that exists between research and industry where research ideas cross over into products and product requirements drive future research.



https://people.cs.umass.edu/~ramesh/Site/PUBLICATIONS_files/CCRpaper_1.pdf

Caching



Motivation

- Many systems use caches for reducing work/transport duplication
  - Databases cache frequently accessed keys in memory
  - Computers use memory as a cache
  - Content delivery networks are a cache
  - The file system has a cache, called a page cache
  - …

- Big data systems even use dedicated cache servers (key-value caches)
  - All data is stored in memory (does not use disk)
  - Before going to database, check key-value cache
  - Examples:
    - Memcached
    - Redis
    - Amazon Elasticache

Important questions for caching

- Assignment: where do you put the data?
  - In memory? How? Optimize read speed
- Consistency: what happens when data is updated/deleted?
  - Passive—cache entry expires
  - Active—cache entry is invalidated
- Eviction: who do you kick out?
  - Cache space is expensive—otherwise, we would put everything there
  - What do you do when you run out of space?

Standard Eviction Policy Model

- Cache has fixed size (C)
  - Each item has the same size (1)
  - Each item has a unique key
- Cache is full
- Requests come in
  - If the request is for an item that exists in the cache, it is read
  - If the request is for an item that does not exist in the cache, it is fetched (from a second storage system, e.g., a database), and another item is evicted to make room for it
- Eviction policy decides which object to evict when a new item is inserted

Metric for success

- Overall goal might be total latency or throughput (usually latency)
  - E.g., what's the performance of the total system given a certain eviction policy
- Typically average latency is equivalent to cache hit rate
  - Over time window, percentage of read requests whose objects existed in the cache

- Example (includes the network latency):
  - Average latency to flash database: 5 millisecond
  - Average latency to in-memory key-value cache: 100 microseconds

  - Average latency (in microseconds) when cache hit rate is 98%:
    - 0.98 * 100 + 0.02 * 5000 = 198 microseconds
  - Average latency (in microseconds) when cache hit rate is 99%:
    - 0.99 * 100 + 0.01 * 5000 = 149 microseconds

  - 1% cache hit rate increase improves latency by 25%!

Common eviction policies

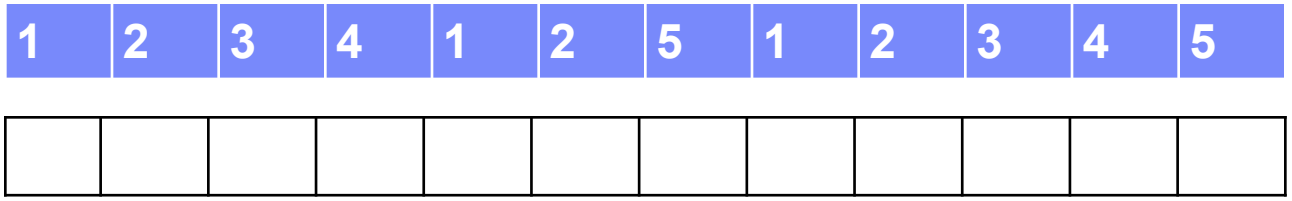
- FIFO (first in first out)
- LRU (least recently used)
- CLOCK (approximates LRU)
- LFU (least frequently used)
- OPT (Belady's algorithm: knows the future)

FIFO (First In First Out)

- Evict the item that was **last inserted** into the cache
- Idea: the item that was last inserted is the oldest, and hence the least relevant
- Advantages:
  - Simple and easy to implement
  - Can be implemented as a list, where new item are inserted to the tail of the list, and items are evicted from the head
- Why might it work?
  - Sometimes the item that was brought in the cache last is old and is not read anymore
- Why might it not work?
  - Because maybe an old item is actually a popular item that is accessed frequently
- In general, usually suffers from a low hit rate

FIFO example

- List of cache accesses (by key):

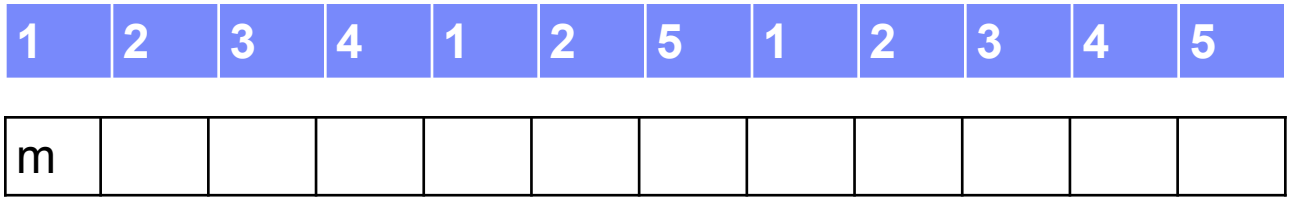


- Cache (size 4):

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):



- Cache (size 4):

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m										

- Cache (size 4):

		1	2

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

- Cache (size 4):

1	2	3	4

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

- Cache (size 4):

1	2	3	4

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

1 evicted

- Cache (size 4):

2	3	4	5

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	m				

2 evicted

- Cache (size 4):

3	4	5	1

FIFO example

- List of cache accesses (by key):

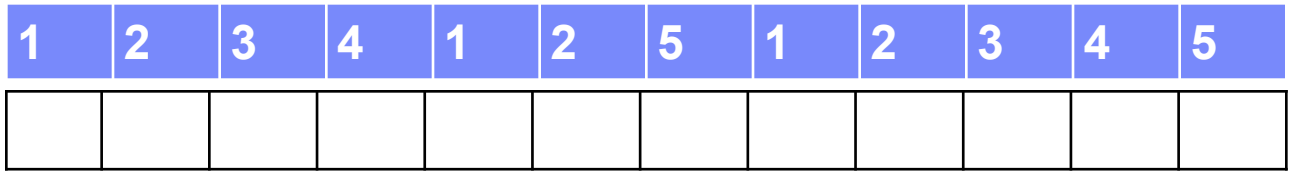
1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	m	m	m	m	m

Total # misses: 10

LRU (Least Recently Used)

- Evict the object that was **last accessed** from the cache
- Idea: recently accessed item likely to be accessed again in the future
- How is LRU different than FIFO?
- Advantages:
  - The default eviction policy, works reasonably well for many workloads
- Disadvantages:
  - Sounds simple, but can be expensive to implement
  - Need to track the last access time of each item, and find item with the oldest access item

LRU example



LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

1	2	3	4

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

2	3	4	1

Need to rearrange cache order even if item is hit

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

3	4	1	2

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

3 is evicted

4	1	2	5

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h				

4	2	5	1

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h			

4	5	1	2

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m		

4 is evicted

5	1	2	3

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	m	

5 is evicted

1	2	3	4

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	m	m

1 is evicted

2	3	4	5

Total: 8 misses
Better than FIFO, but not optimal…

LRU Approximations

- LRU is expensive to fully implement
  - Need to update a list of items at every read
  - Maintaining cache ranking of all items can be expensive
- In practice, many caches use approximations of LRU
- Sampled LRU
  - Instead of maintaining full ranking of all items (in a list), every time a new item is accessed, simply update its access time
  - At eviction time, sample N items, and pick the one with the oldest access time
  - When N is large enough (e.g., 50/100 or more) this approximates LRU very well
- CLOCK
  - Maintain 1 bit per item in the cache (can be 0 or 1)
  - When an item is accessed set bit to 1
  - Occasionally use a clock “hand” that decrements bits to 0 one by one
  - At eviction time, evict next item that has a bit 0
  - This is the algorithm originally used to cache memory pages in the operating system

LFU (Least Frequently Used)

- Evict the object that was **accessed the fewest total time** from the cache
  - Usually # of times over a period time (i.e., frequency)
- Idea: items that were accessed many times likely to be accessed again in the future
- How is LFU different than LRU?
- Advantages:
  - Works well for workloads that have a relatively static number of popular items
- Disadvantages:
  - Does not work well if new items become popular
  - Can be difficult to implement: need to track item frequency for all keys, find the one with the lowest frequency
    - To fully implement, also need to track frequencies of evicted items

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

- # accesses:

1	2	3	4
1	1	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

- # accesses:

1	2	3	4
2	1	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

- # accesses:

1	2	3	4
2	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

Evict 3/4 (randomly chose 3)

1	2	4	5

- # accesses:

2	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h				

- # accesses:

1	2	4	5
3	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h			

- # accesses:

1	2	4	5
3	3	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m		

Evict 4/5 (randomly chose 5)

- # accesses:

1	2	3	4
3	3	2	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	h	

- # accesses:

1	2	3	4
3	3	2	2

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	h	m

Evict 3/4 (randomly chose 4)

- # accesses:

1	2	3	5
3	3	2	2

Total misses: 7

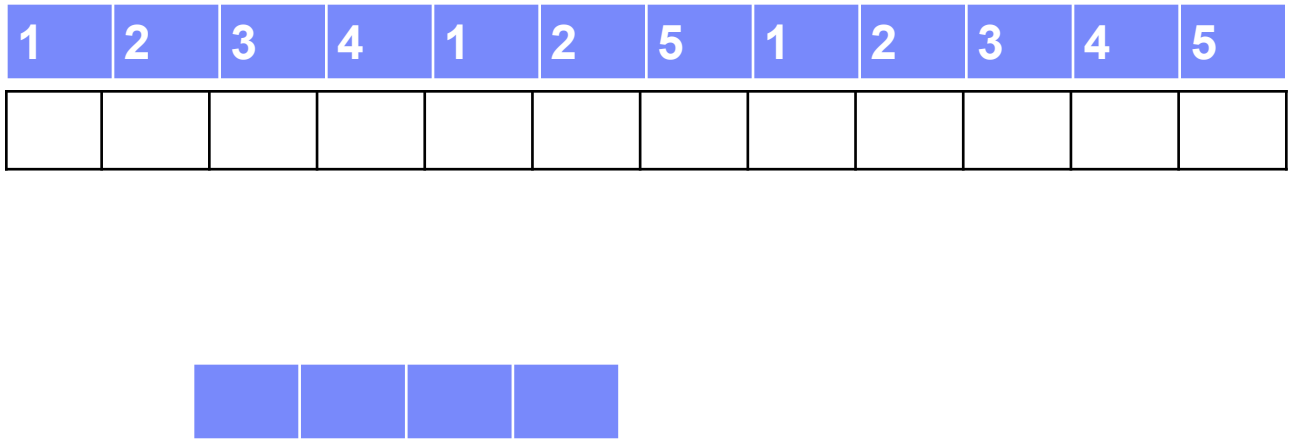
LRU/LFU combinations

- LRU prioritizes recent accesses, while LFU prioritizes frequently accessed items
- Recency vs. frequency are both important features, each works better for different workloads
- Many hybrid frequency and recency algorithms
- For example, Segmented LRU
  - The first time an object is inserted into the cache, insert it into the middle of the cache (not the head)
  - Afterwards, at every access insert it to the top of the cache
  - Mostly behaves like LRU, but penalizes objects that might be accessed recently but just once

OPT/Optimal (Belady's Algorithm)

- Each one of the algorithms we surveyed earlier can perform better (or worse) on certain workloads
- The Optimal algorithm (invented by Belady) is the optimal eviction algorithm
  - Assumes it knows the future
  - Evicts page that won't be accessed for the longest time in the future
- This is not an algorithm that can be used in practice (since it needs to know the future!)
  - But can be used to evaluate how far off LRU/LFU/etc. are from the optimal for a given workload
  - Some recent work uses OPT to train machine learning eviction policies ("Learning Relaxed Belady", Song et al.)

OPT example



OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

4	3	2	1

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

4	3	1	2

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

4	3	2	1

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

Evict 4 (furthest away in the future)

5	3	2	1

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h		

3	2	1	5

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h	m	

4	2	1	5

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h	m	h

4	2	1	5

Total misses: 6 (this is the optimal!)

Eviction policies in the real world

- All of the eviction policies we surveyed are used by real systems
  - Except for OPT!
- LRU and its approximations are the most common
- Complications:
  - We assumed all items are the same size
  - Things get more complicated when items can have different sizes
    - E.g., Bigger items might cause the eviction of many small items

Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li,
Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung,
Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: `Memcached` is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages `memcached` as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a billion users around the world.

1 Introduction

Popular and engaging social networking sites present significant infrastructure challenges. Hundreds of millions of people use these networks every day and impose computational, network, and I/O demands that traditional web architectures struggle to satisfy. A social network's infrastructure needs to (1) allow near real-time communication, (2) aggregate content on-the-fly from multiple sources, (3) be able to access and update very popular shared content, and (4) scale to process millions of user requests per second.

We describe how we improved the open source ver-

however, web pages routinely fetch thousands of key-value pairs from `memcached` servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities require more effort to achieve than others. For example, maintaining data consistency can be easier at small scales if replication is minimal compared to larger ones where replication is often necessary. Additionally, the importance of finding an optimal communication schedule increases as the number of servers increase and networking becomes the bottleneck.

This paper includes four main contributions: (1) We describe the evolution of Facebook's `memcached`-based architecture. (2) We identify enhancements to `memcached` that improve performance and increase memory efficiency. (3) We highlight mechanisms that improve our ability to operate our system at scale. (4) We characterize the production workloads imposed on our system.

https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf

TCP/IP
5-Layer Model

How Data Actually Travels Across the Internet

1 Physical | 2 Data Link | 3 Network | 4 Transport | 5 Application

Graduate Course in Computer Networking

The 5-Layer Stack

5 — Application

HTTP · HTTPS · SMTP · FTP · SSH · DNS

4 — Transport

TCP · UDP · QUIC · Port numbers

3 — Network

IPv4 · IPv6 · BGP · Routers

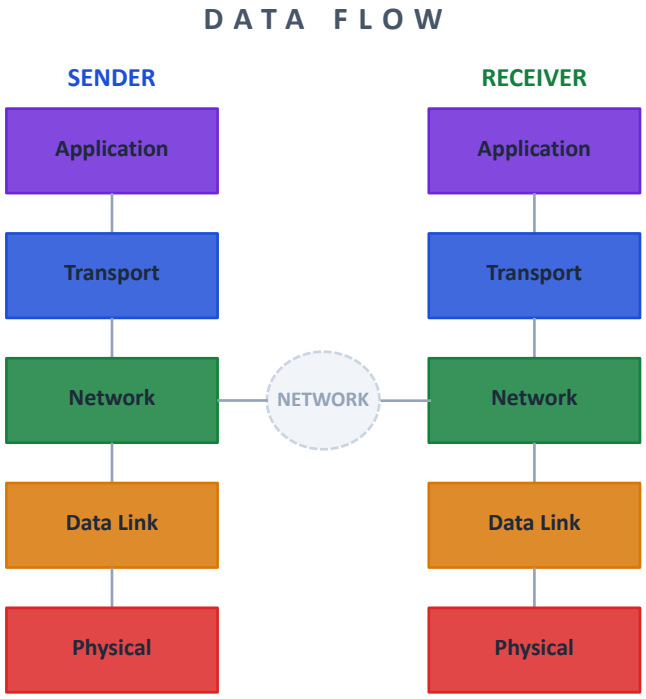
2 — Data Link

Ethernet frames · MAC · ARP · DHCP

1 — Physical

Fiber · Copper · Coax · WiFi · 5G · Satellite · Microwave

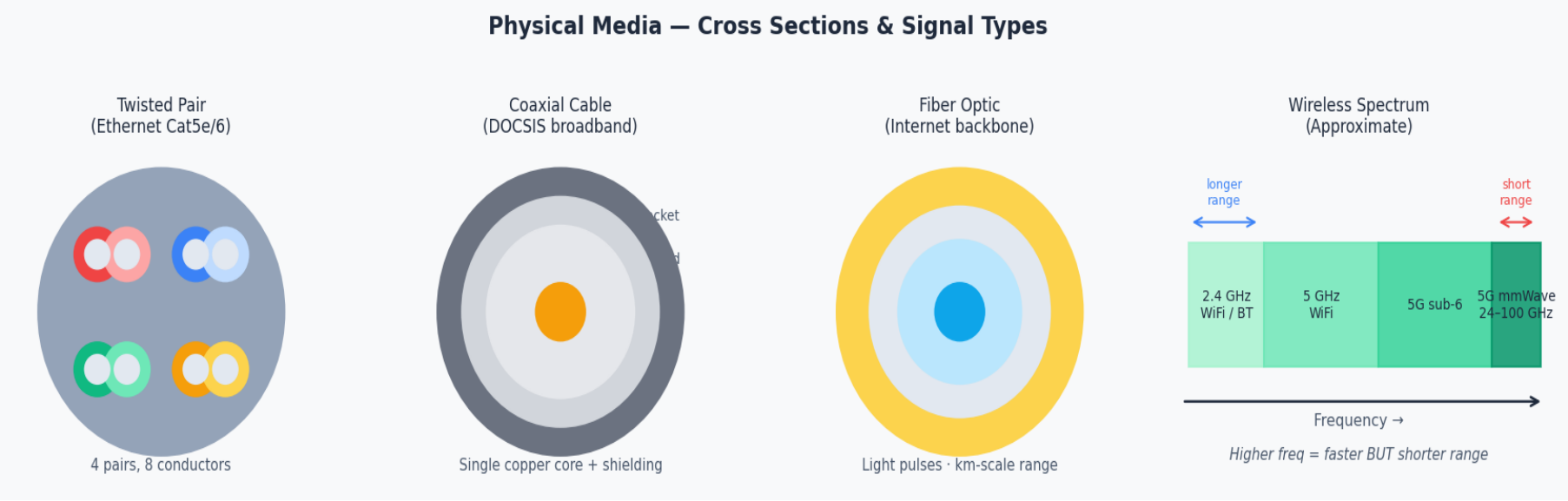
← Encapsulation (sending) · Decapsulation (receiving) →



Layer 1: Physical

L1 Physical

Converts bits to physical signals. The medium determines speed, range, and latency.



Ethernet spans L1 (signaling) + L2 (framing) — the same cable serves both layers

Microwave travels faster than fiber (air vs. glass) — HFT firms use this for trading

GEO satellite: ~600ms latency (physics). Starlink LEO: 20–40ms — viable for apps

KEY CONCEPT Layer 1 is protocol-agnostic — it just moves signals. A broken cable or weak signal kills everything above it regardless of how good your software is.

Layer 2: Data Link

L2 Data Link

Organizes bits into frames · handles local network delivery · MAC addresses identify hardware.

ETHERNET FRAME STRUCTURE

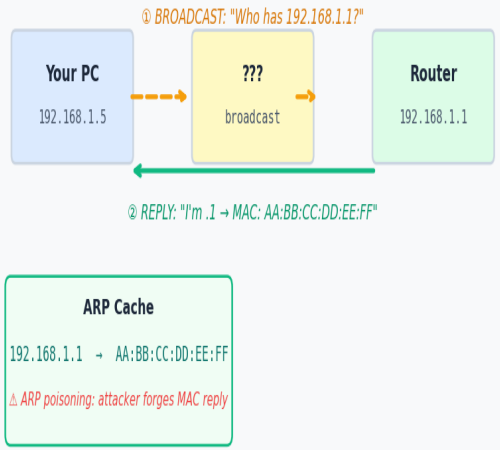
Preamble	Dst MAC	Src MAC	Type	Payload	FCS
8B	6B	6B	2B	46-1500B	4B

MAC Address

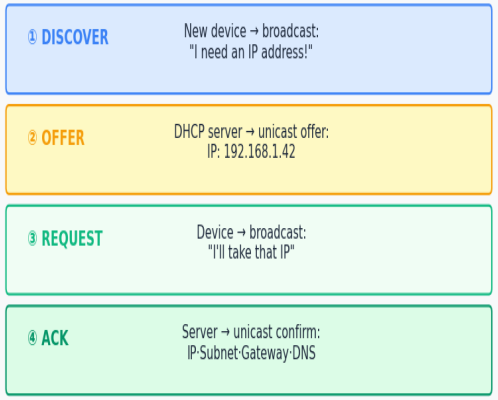
3C:22:FB:4A:9E:01

48-bit hardware identifier · burned in at manufacture
Local only — invisible beyond the router
Switches use MAC tables; routers do not

ARP – Address Resolution Protocol



DHCP – Dynamic Host Configuration



KEY CONCEPT MAC addresses solve local delivery. ARP bridges L2 (MAC) and L3 (IP). DHCP bootstraps new devices automatically. Both are critical infrastructure — and both have well-known attacks.

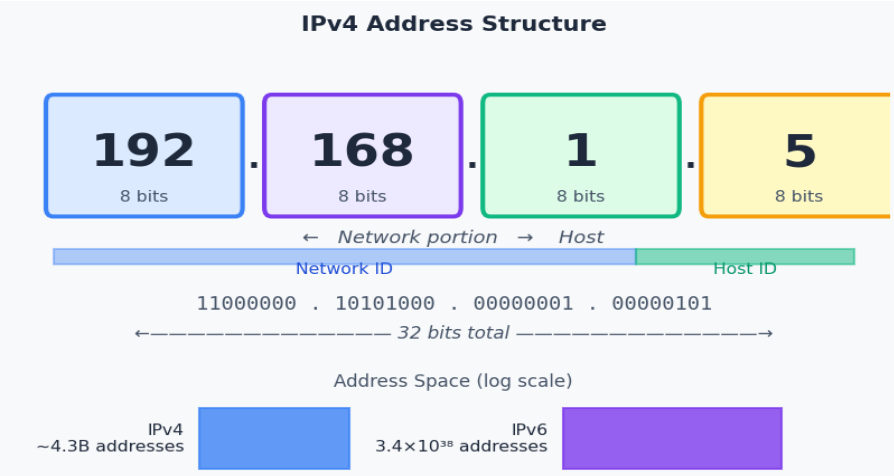
Layer 3: Network

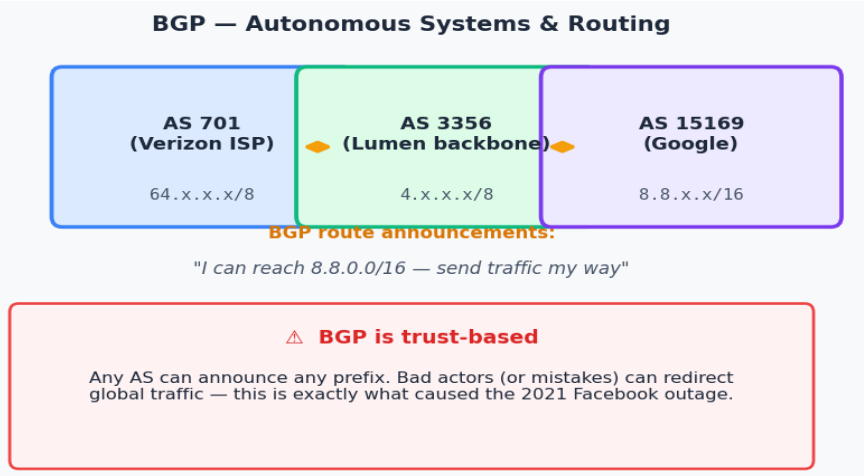
L3 Network

Routes packets across multiple networks using IP addresses. Makes the global internet possible.

IPv4 → 32-bit · ~4.3B addresses · Exhausted (NAT extends it)
IPv6 → 128-bit · 340 undecillion · Eliminates NAT · Dominant on mobile

```
Run  traceroute google.com  in a terminal to see every router hop in real time.
```





KEY CONCEPT IP addresses are logical and hierarchical — they encode location. No router has a complete map: each makes a local best-hop decision. BGP's trust model is its greatest vulnerability.

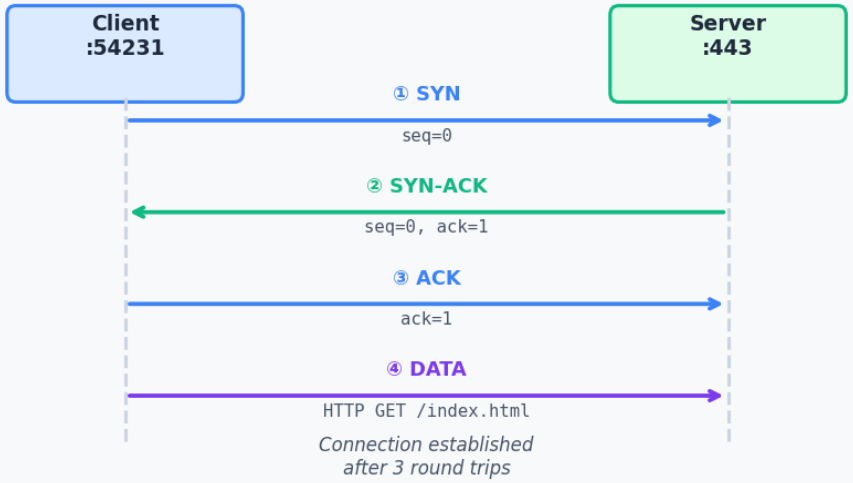
Layer 4: Transport

L4 Transport

End-to-end delivery between applications. Ports multiplex streams. TCP vs UDP is the central reliability/speed tradeoff.

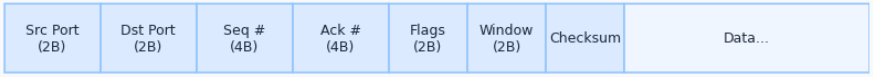
QUIC (RFC 9000 / HTTP/3) Runs over UDP but reimplements reliability. Integrates TLS 1.3 (0-RTT vs TCP+TLS's 3+ round trips). No head-of-line blocking. Survives WiFi→5G handoff without dropping connections. Used by Google, YouTube, Meta for a major fraction of traffic.

TCP — 3-Way Handshake



TCP vs UDP — Header Comparison

TCP Header (20-60 bytes)



UDP Header (8 bytes only!)

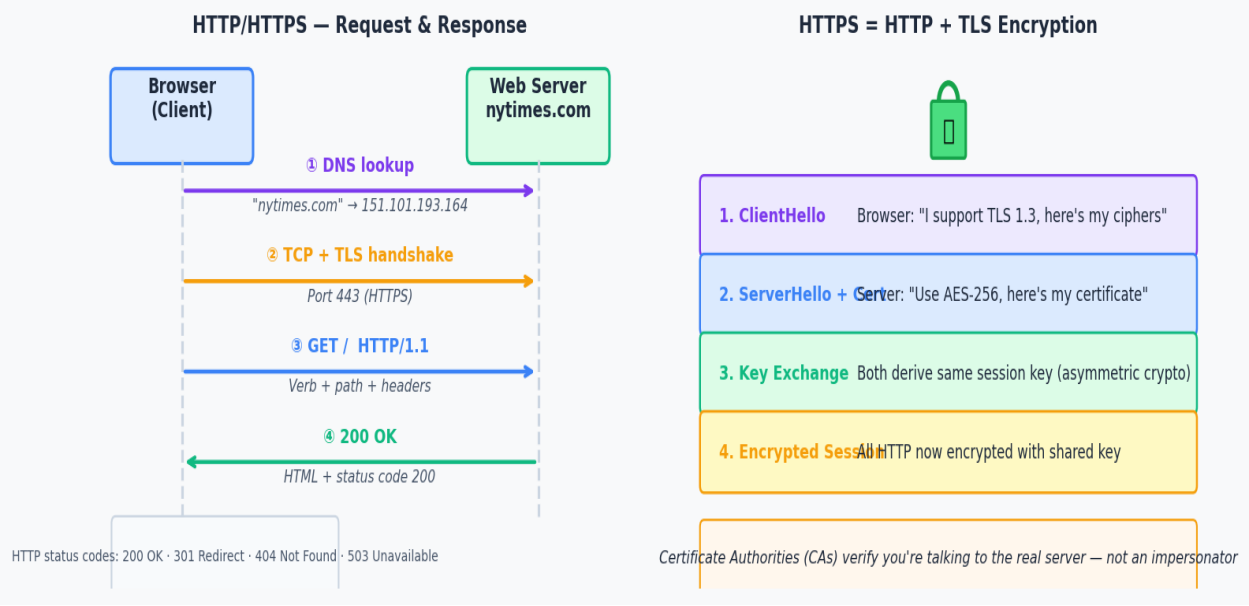
Feature	TCP	UDP
Connection	Required (handshake)	None
Reliability	Guaranteed	Best-effort
Ordering	Preserved	Not guaranteed
Speed	Slower	Faster
Use case	HTTP,SSH,SMTP,FTP	DNS,Zoom,games

KEY CONCEPT TCP trades latency for reliability. UDP trades reliability for speed. QUIC is the modern attempt to get TCP-level guarantees with UDP-level latency.

Layer 5: Application

L5 Application

Where developers work. Protocols define the language two applications speak over a network.



SMTP — Port 25 / 587

Mail transfer between servers. Plaintext by default (1980s!). STARTTLS adds encryption. Weak auth = endemic spam.

SSH — Port 22

Encrypted remote terminal. Replaced Telnet. Key-based auth. SSH tunnels forward arbitrary TCP over encrypted channel.

FTP/SFTP — Port 21 / 22

FTP (1971): plaintext incl. credentials. SFTP (over SSH): encrypted. FTP survives in legacy industrial systems.

DNS — Port 53 UDP

Hostname → IP. Hierarchical: root→TLD→authoritative. TTL caching. DoH/DoT encrypt queries. Critical infrastructure.

KEY CONCEPT Application protocols define message format, verbs, and session rules. Many were designed in the 1980s without security assumptions — which is why plaintext protocols and BGP hijacking remain live threats.

Summary: The 5-Layer Model

Every network problem has a layer. Identifying it is half the diagnosis.

5	**Application** *Software communicates*	`HTTP · HTTPS · SMTP · FTP · SSH · DNS · DoH`
4	**Transport** *App-to-app, reliability/speed*	`TCP · UDP · QUIC · Port numbers`
3	**Network** *Global routing, IP addressing*	`IPv4 · IPv6 · BGP · Routers`
2	**Data Link** *Local delivery, MAC addressing*	`Ethernet · 802.11 MAC · ARP · DHCP · Switches`
1	**Physical** *Bits → signals*	`Fiber · Copper · Coax · WiFi · 5G · Microwave · Satellite`

Mnemonic (bottom → top): **P**eople **D**on't **N**eed **T**o **A**sk (Physical → Data Link → Network → Transport → Application)

KEY CONCEPT The TCP/IP model is a mental framework for diagnosing failures: is this a physical issue, a routing issue, or an application protocol issue? Layer-based thinking is how professionals isolate problems fast.

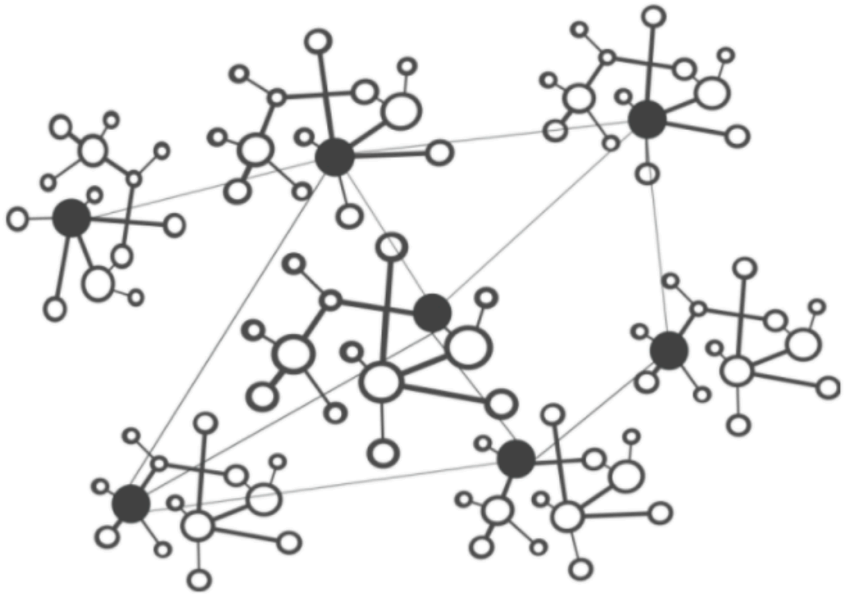
Understanding how Facebook disappeared from the Internet

2021-10-04

 Celso Martinho  Tom Strickx

5 min read

This post is also available in 简体中文, Français, Deutsch, Italiano, 日本語, 한국어, Português, Español, Русский and 繁體中文.



https://blog.cloudflare.com/october-2021-facebook-outage/

The Internet - A Network of Networks

"*Facebook can't be down, can it?*", we thought, for a second.