The Book System Deisgn and Architecture Edition 2 is now work in progress at https://github.com/puncsky/system-design-and-architecture.

https://tianpan.co/

I am organizing contents of this docs into https://puncsky.com/hacking-the-software-engineer-interview/?isAdmin=true
Latest update: May 13, 2019

Fork me at https://github.com/puncsky/system-design-and-architecture
Follow the author at https://github.com/puncsky

Telegram Group for Discussion: t.me/system_design_and_archiecture

Tags to Goto: [TODO], [不重要], [Lab]

Experience Deep Dive

[Technical Communication]

Intended Audience: Moderate experience or less, or anyone who was not in a leadership or design position (either formal or informal) in their previous position

Question Description: **Describe one of your previous projects that was particularly interesting or memorable to you.** Followup questions:

- What about it made it interesting?
- What was the most challenging part of the project, and how did you address those challenges?
- What did you learn from the project, and what do you wish you had known before you started?
- What other designs/implementation methods did you consider? Why did you choose the one that you did? If you were to do the same project over again, what would you do differently?

Interviewer tips: Since the goal here is to assess the technical communication skill and interest level of someone who has not necessarily ever been in a role that they could conduct a crash course in, you should be prepared to keep asking them questions (either for more details, or about other aspects of the project). If they are a recent grad and did a thesis, that's often a good choice to talk about. While this question is in many ways similar to the Resume Questions question from phone screen one, this question is intended to be approximately four times as long, and should get into proportionally more detail about what it is that they have done. As such, the scoring criteria are similar, but should be evaluated with both higher expectations and more data.

Scoring:

A great candidate will
- Be able to talk for the full time about the project, with interaction from the interviewer being conversational rather than directing
- Be knowledgeable about the project as a whole, rather than only their area of focus, and be able to articulate the intent and design of the project
- Be passionate about whatever the project was, and able to describe the elements of the project that inspired that passion clearly
- Be able to clearly explain what alternatives were considered, and why they chose the implementation strategy that they did.

Have reflected on and learned from their experiences

A good candidate will
- May have some trouble talking for the full time, but will be able to with some help and questions from the interviewer
- May lack some knowledge about the larger scope of the project, but still have strong knowledge of their particular area and pieces that directly interacted with them
- May seem passionate, but be unable to clearly explain what inspired that passion
- May be able to discuss alternatives to what they did, but not have considered them in depth
- Have reflected on and learned from their experiences

A bad candidate will
- Have difficulty talking for the full time. The interviewer may feel as if they are interrogating rather than conversing with the candidate
- May lack detailed knowledge of the project, even within the area that they were working. They may not understand why their piece was designed the way it was, or may not understand how it interacted with other systems
- Does not seem very interested in the project - remember that you are asking them about the most interesting project that they have done, they should be very interested in whatever it was
- May not be familiar with potential alternatives to their implementation method
- Does not seem to have learned from or reflected on their experiences with the project. A key sign of this is that the answer to 'what did you learn' and 'what would you do differently' are short and/or nearly identical.

A terrible candidate will
- Be unable to talk for the full time, even with significant interviewer assistance.
- Not seem to understand anything about the project beyond their particular implementation details
- Not display interest in the project

	- Did not consider or think to research alternative implementation methods - Be unable to answer questions like 'what would you do differently if you were to redo the project?' and similar.

Introduction to Architecture

What is architecture?

Architecture is the shape of the software system. Thinking it as a big picture of physical buildings.

- paradigms are bricks.
- design principles are rooms.
- components are buildings.

Together they serve a specific purpose like a hospital is for curing patents and a school is for educating students.

Why do we need architecture?

Behavior vs. Structure

Every software system provides two different values to the stakeholders: behavior and structure. Software developers are responsible for ensuring that both those values remain high.

Software architects are, by virtue of their job description, more focused on the structure of the system than on its features and functions.

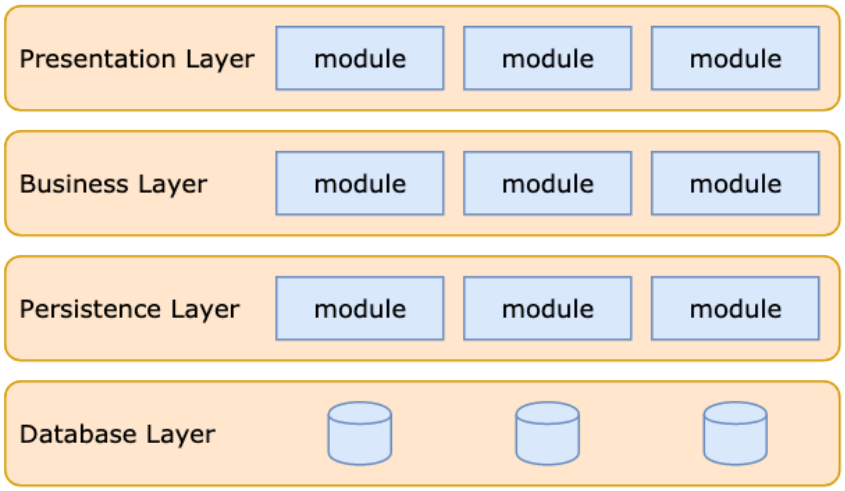
Ultimate Goal - saving human resources costs per feature

Architecture serves the full lifecycle of the software system to make it easy to understand, develop, test, deploy, and operate. The goal is to minimize the human resources costs per business use-case.

The O'Reilly book Software Architecture Patterns by Mark Richards is a simple but effective introduction to these five fundamental architectures.

1. Layered Architecture

The layered architecture is the most common in adoption, well-known among developers, and hence the de facto standard for applications. If you do not know what architecture to use, use it.



Examples

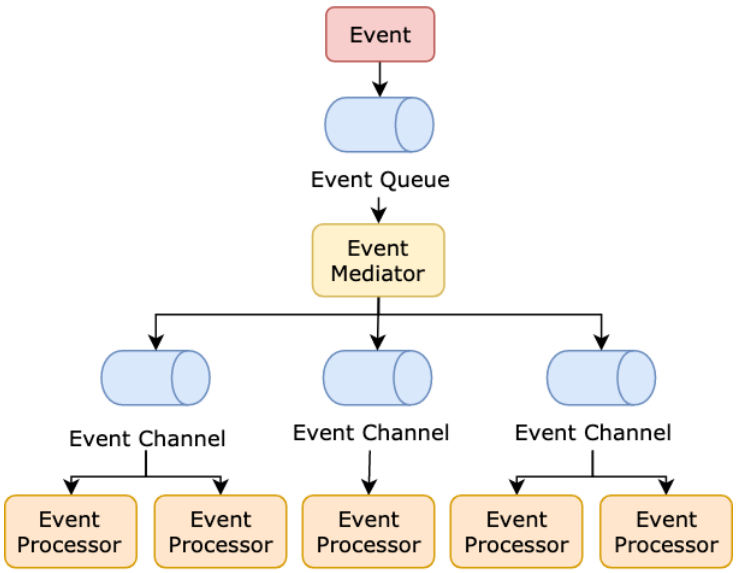
- TCP / IP Model: Application layer > transport layer > internet layer > network access layer
- Facebook TAO: web layer > cache layer (follower + leader) > database layer

Pros and Cons

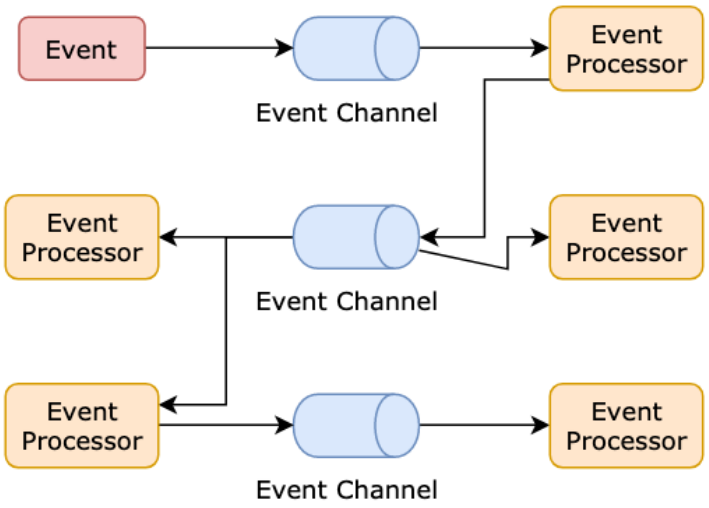
- Pros
  - ease of use
  - separation of responsibility
  - testability
- Cons
  - monolithic
    - hard to adjust, extend or update. You have to make changes to all the layers.

2. Event-Driven Architecture

A state change will emit an event to the system. All the components communicate with each other through events.



A simple project can combine the mediator, event queue, and channel. Then we get a simplified architecture:



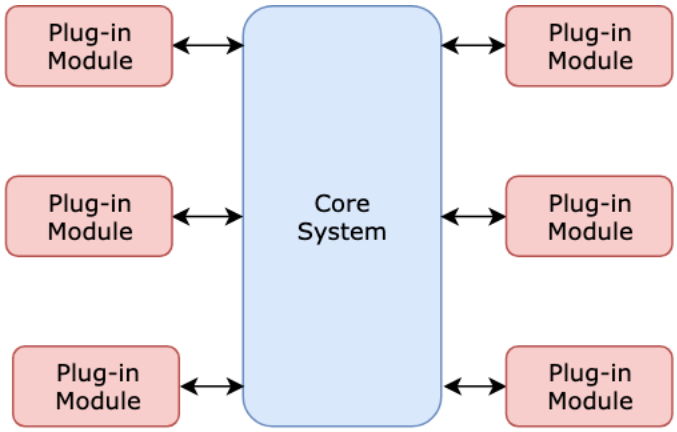
Examples

- QT: Signals and Slots
- Payment Infrastructure: Bank gateways usually have very high latencies, so they adopt async technologies in their architecture design.

3. Micro-kernel Architecture (aka Plug-in Architecture)

The software's responsibilities are divided into one "core" and multiple "plugins". The core contains the

bare minimum functionality. Plugins are independent of each other and implement shared interfaces to achieve different goals.

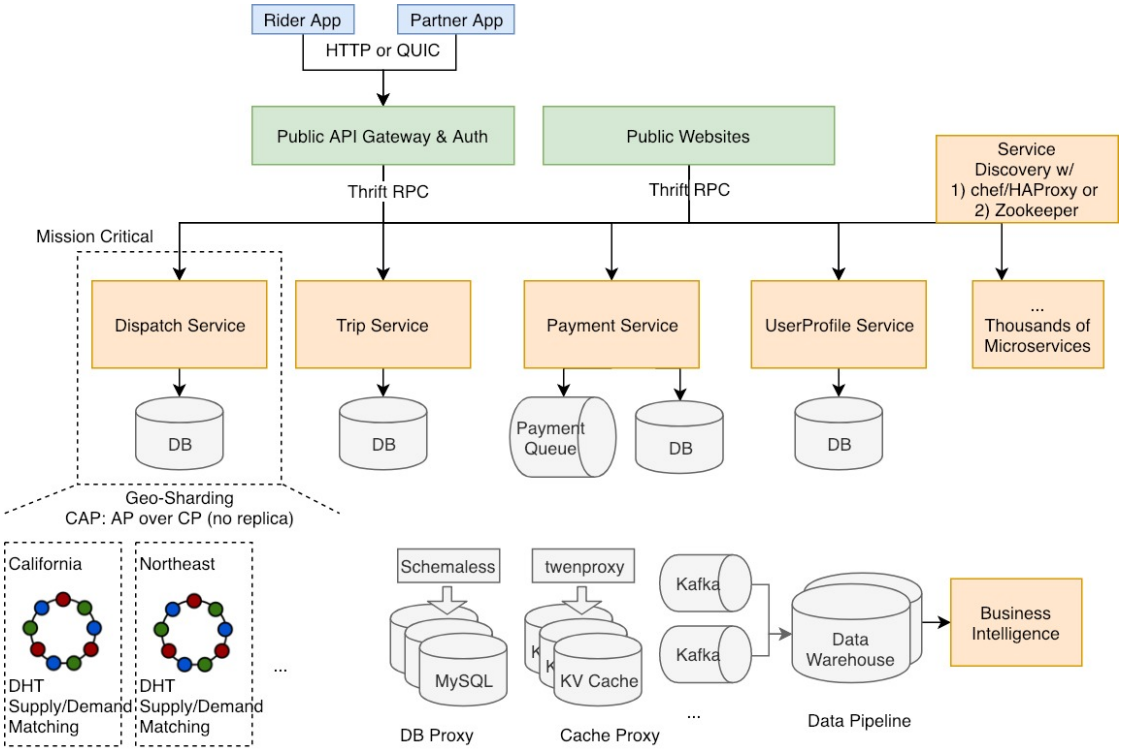


Examples

- Visual Studio Code, Eclipse
- MINIX operating system

4. Microservices Architecture

A massive system is decoupled to multiple micro-services, each of which is a separately deployed unit, and they communicate with each other via RPCs.

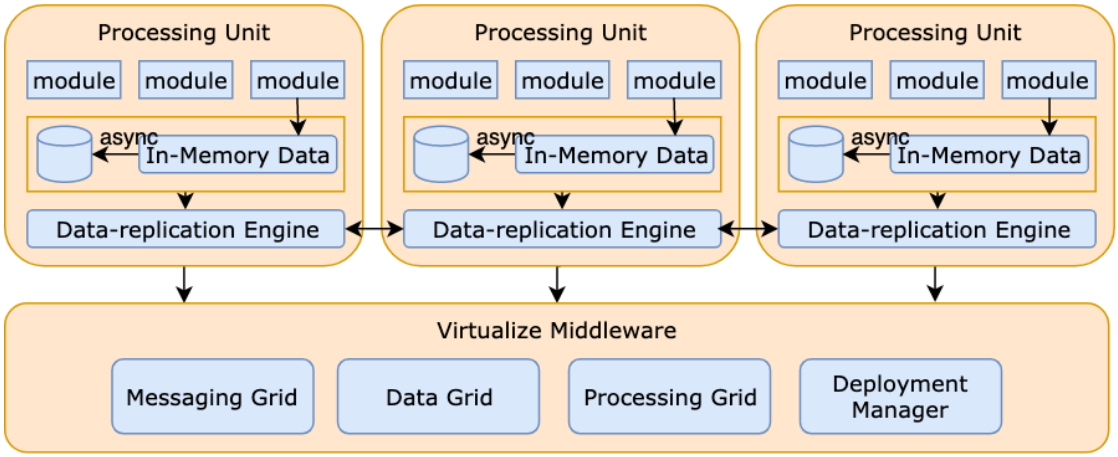


Examples

- Uber: See designing Uber
- Smartly

5. Space-based Architecture

This pattern gets its name from “tuple space”, which means “distributed shared memory". There is no database or synchronous database access, and thus no database bottleneck. All the processing units share the replicated application data in memory. These processing units can be started up and shut

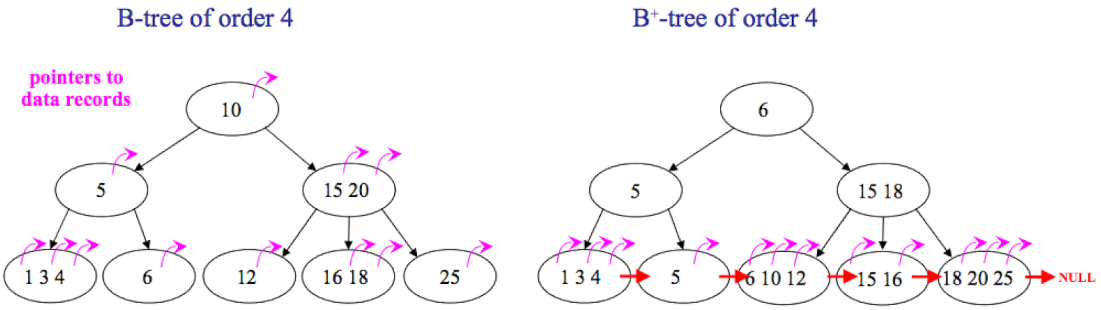
	down elastically.  Examples: See Wikipedia • Mostly adopted among Java users: e.g., JavaSpaces

Data Partition and Routing

scale out 产生 data shard / partition, 需要 routing, partition 需要 replica 保证 availability

(+) availability
(+) read(parallelization, single read efficiency)
(-) consistency

Routing abstract model	two maps: key-partition map, partition-machine map
Hash partition	Round robin, virtual buckets, consistent hashing O(1) ~ O(log n) **hash and mod** (+) simple (-) flexibility (tight coupling two maps: adding and removing nodes (partition-machine map) disrupt existing key-partition map) **Virtual buckets: key--(hash)-->vBucket, vBucket--(table lookup)-->servers** Usercase: Membase a.k.a Couchbase, Riak (+) flexibility, decoupling two maps (-) [TODO] ? centralized in-memory lookup table **Consistent hashing and DHT** [Chord] implementation virtual nodes: for load balance in heterogeneous data center Usercase: Dynamo, Cassandra (+) flexibility, hashing space decouples two maps. two maps use the same hash, but adding and removing nodes only impact succeeding nodes. (-) network complexity, hard to maintain
Range partition	sort by primary key, shard by range of primary key range-server lookup table (e.g. HBase .META. table) + local tree-based index (e.g. LSM, B+) (+) search for a range (-) log(n) Usercase: Yahoo PNUTS, Azure, Bigtable

[Chord]	[TODO] O(log n), finger table
B tree vs. B+ tree [不重要]	http://stackoverflow.com/questions/870218/b-trees-b-trees-difference Advantages of B+ trees: • Because B+ trees don't have data associated with interior nodes, more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node. • The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree. This full-tree traversal will likely involve more cache misses than the linear traversal of B+ leaves. Advantage of B trees: • Because B trees contain data with each key, frequently accessed nodes can lie closer to the root, and therefore can be accessed more quickly. • A B⁺-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves 

Load Balance	https://puncsky.com/notes/2018-07-23-load-balancer-types
	
	Generally speaking, load balancers fall into three categories:
	
	• DNS Round Robin (rarely used): clients get a randomly-ordered list of IP addresses.
	  ○ pros: easy to implement and free
	  ○ cons: hard to control and not responsive, since DNS cache needs time to expire
	• Network (L3/L4) Load Balancer: traffic is routed by IP address and ports.L3 is network layer (IP). L4 is session layer (TCP).
	  ○ pros: better granularity, simple, responsive
	• Application (L7) Load Balancer: traffic is routed by what is inside the HTTP protocol. L7 is application layer (HTTP).

Replica and Consistency

Common practice 是 3 replicas
(+) availability
(+) read parallelization
(-) consistency

Principles

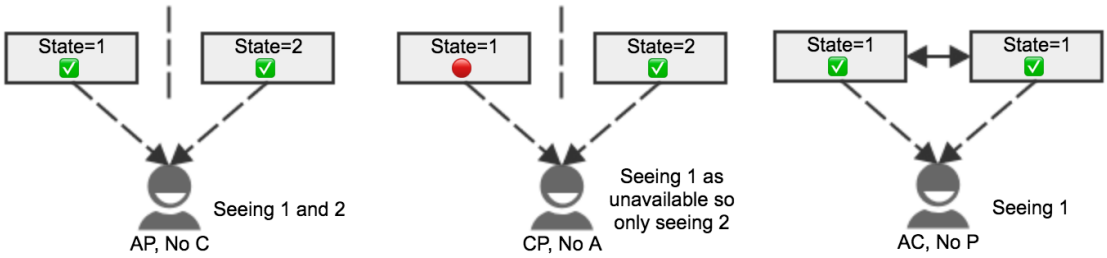
CAP

Why replica and consistency?

large dataset → scale out → data shard / partition → 1) routing for data access

2) replica for availability → consistency challenge

Consistency trade-offs for CAP theorem



- Consistency: all nodes see the same data at the same time
- Availability: a guarantee that every request receives a response about whether it succeeded or failed
- Partition tolerance: system continues to operate despite arbitrary message loss or failure of part of the system

Any networked shared-data system can have only two of three desirable properties.

- rDBMS prefer CP → ACID
- NoSQL prefer AP → BASE

"2 of 3" is mis-leading

12 years later, the author Eric Brewer said "2 of 3" is mis-leading, because

1. partitions are rare, there is little reason to forfeit C or A when the system is not partitioned.
2. choices can be actually applied to multiple places within the same system at very fine granularity.
3. choices are not binary but with certain degrees.

Consequently, when there is no partition (nodes are connected correctly), which is often the case, we should have both AC. When there are partitions, deal them with 3 steps:

1. detect the start of a partition,
2. enter an explicit partition mode that may limit some operations, and
3. initiate partition recovery (compensate for mistakes) when communication is restored.

CAP reloaded

Eric Brewer: CAP with finer granularity，在没有P的时候保证AC，在P发生的时候，意识到P的发生，P结束后要 resolve 产生分歧的状态。

ACID

Atomicity
Consistency: from one valid state to another. (diff from C in CAP)
Isolation: concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
Durability

BASE

Basic Availability:
Soft State: between state and stateless
Eventual Consistency:

NoSQL 采纳 BASE，但是正在向局部 ACID 发展，比如 Google MegaStore

CAP vs. ACID vs. BASE	
Idempotent	比如在杀Hadoop job的时候，有一个重要的线上job还在跑，被我杀掉了，当时吓了一跳，赶紧找熟悉这些job的oncall来看，还好这个job是幂等的，重跑就行了。如果运气不好，干掉一些特殊的job，结果或许就糟糕了。前年年底就有一个组内job的问题因为维护的工程师的操作触发了sev 1（最高级别）的线上问题。

Consistency Models

http://en.wikipedia.org/wiki/Consistency_model

Strong consistency	Instructions will be executed on processor in the order as they were issued
Eventual consistency	After a write, read will eventually see at(may be in couple of miliseconds).Data is replicated asynchronously.
Causal consistency	
Read-your-writes Consistency	Read-after-write consistency tightens things up a bit, guaranteeing immediate visibility of *new*data to all clients. With read-after-write consistency, a newly created object or file or table row will immediately be visible, without any delays
Session consistency	
Monotonic read consistency	
Monotonic write consistency	

Replication Strategies (How to update?)

All at once	
Master-slave / leader-follower	
Any node first	

Consistency Protocols

2PC	
Vector clock + RWN = eventual consistency	Pigeonhole principle Usercase: Dynamo
Paxos	2f + 1 个节点有 f + 1 个节点同意了，结果就决定了
ISR	In-sync replica Usercase: Kafka
2PC vs. Paxos	2PC blocks if the transaction manager fails, requiring human intervention to restart. 3PC algorithms (there are several such algorithms) try to fix 2PC by electing a new transaction manager when the original manager fails. Paxos does not block as long as a majority of processes (managers) are correct. Paxos actually solves the more general problem of consensus, hence, it can be used to implement transaction commit as well. In comparison to 2PC it requires more messages, but it is resilient to manager failures. In comparison to most 3PC algorithms, Paxos renders a simpler, more efficient algorithm (minimal message delay), and has been proved to be correct.

Big Data Algorithms and Data Structures

Bloom filter	
Skiplist	A skip-list is essentially a linked list that allows you to binary search on it. The way it accomplishes this is by adding extra nodes that will enable you to 'skip' sections of the linked-list. Given a random coin toss to create the extra nodes, the skip list should have O(logn) searches, inserts and deletes. Usecases ● LevelDB MemTable ● Redis SortedSet ● Lucene inverted index
LSM tree	high volume insertion logn
Merkle Hash Tree	
Snappy and LZSS	compression Snappy Usercase: Google LevelDB
Cuckoo Hashing	Motivation: 解决 collision 问题, with worst-case O(1) lookup time x => bucket1 = hash1(x), bucket2 = hash2(x), if either of them is empty, then use it; else push one of them to a different location in the table. Using just three hash functions increases the load to 91%. Using just 2 keys per bucket permits a load factor above 80%. Usercase: CMU SILT (Small Index Large Table) a memory-efficient, high-performance key-value store system based on flash storage. write-optimized and append-only, and indexed by a hash table that uses partial-key cuckoo hashing., a = hash1(key), b = hash2(key) 如果某个位置要被替换, 可以直接从位置存储的内容获得下一个应该存储的位置(a 处存 b, b 处存 a, in-memory index 存 location tag 不存 key), 省去了读 flash 的过程。

Resource Management and Scheduler

Abstraction Models

Basic Design Problems

Paradigms

Strategies

Examples

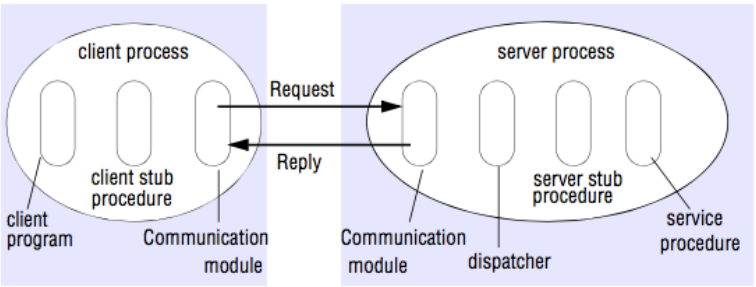
Distributed Coordination System

Chubby	

ZooKeeper	centralized

Distributed Communication

Serialization and RPC Frameworks

HTTP (RESTful api), RPC, RMI, UDP, TCP ? Tradeoffs	- UDP and TCP are both transport layer. TCP is reliable and connection-based. UDP is connectionless and unreliable. - HTTP is application layer. It is normally TCP based since HTTP assumes a reliable transport. - RPC, application layer protocol. [What is RPC? How to implement it?] - [RESTful]
What is RPC? How to implement it?	a remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation (RMI).  Stub procedure: a local procedure to marshal the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results.
Protobuf vs Thrift vs Avro	Google Protobuf, open source version does not have RPC implementations but only API. Smaller serialized data and slightly faster. Better documentations and cleaner APIs. Facebook Thrift, support more languages, richer data structures(support richer data structures: List/Set/Map, etc, Protobuf不支持。Usercase: Hbase/Cassandra/Hypertable/Scrib/.. Incomplete documentation and hard to find good examples. Apache Avro: Avro is heavily used in the hadoop ecosystem. It is based on dynamic schemas in Json. . Avro differs from these systems in the following fundamental aspects. • *Dynamic typing*: Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation, static datatypes, etc. This facilitates construction of generic data-processing systems and languages. • *Untagged data*: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size. • *No manually-assigned field IDs*: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names.
Distributed Objects	refers to software modules that are designed to work together, but reside either in multiple computers connected via a network or in different processes inside the same computer. One object sends a message to another object in a remote machine or process to perform some task. The results are sent back to the calling object.

	Distributed objects are used in Java RMI. CORBA lets one build distributed mixed object systems. DCOM is a framework for distributed objects on the Microsoft platform. JavaSpaces is a Sun specification for a distributed, shared memory (spaces based)

HTTP and RESTful API

RESTful	REST(Representational state transfer of resources) - Best practice of HTTP API to interact with resources. - URL only decides the location. Headers (Accept and Content-Type, etc.) decide the representation. HTTP methods(GET/POST/PUT/DELETE) decide the state transfer. - minimize the coupling between client and server (a huge number of HTTP infras on various clients, data-marshalling). - Stateless and scaling out. - service partitioning feasible. - Public API.
OAuth2	

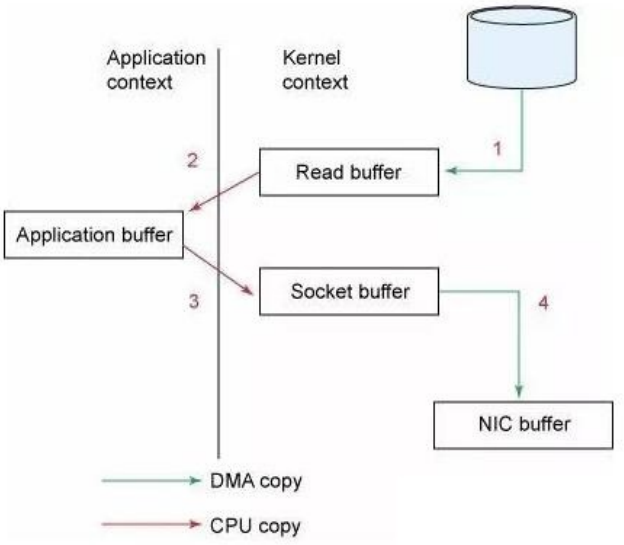
Ratelimiting	Token Bucket and leaky bucket
	
	http://zhengyun-ustc.iteye.com/blog/1895814
	
	public class RateLimiter {
	    private long _lastTime = System.currentTimeMillis();
	    private double _permitsPerSecond;
	    private double _allowance = _permitsPerSecond;
	
	    public RateLimiter(double permitsPerSecond) {
	        _permitsPerSecond = permitsPerSecond;
	    }
	
	    public boolean acquire() {
	        long curTime = System.currentTimeMillis();
	        double timeEplasedInSec = (curTime - _lastTime) / 1000.0;
	        _lastTime = curTime;
	        _allowance += timeEplasedInSec * _permitsPerSecond;
	
	        if (_allowance > _permitsPerSecond) {
	            _allowance = _permitsPerSecond;
	        }
	
	        if (_allowance < 1) {
	            return false;
	        } else {
	            _allowance -= 1;
	            return true;
	        }
	    }
	
	    public static void main(String[] args) {
	        RateLimiter limiter = new RateLimiter(2);
	        Scanner keyboard = new Scanner(System.in);
	        while (keyboard.nextLine() != null) {
	            System.out.println(limiter.acquire() + new Timestamp(System.currentTimeMillis()).toString());
	        }
	    }
	}

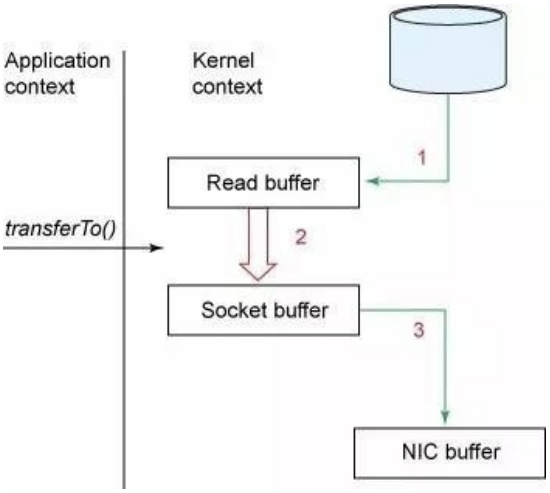
SSL	

Messaging Queue

Application-Level Multi-Broadcast

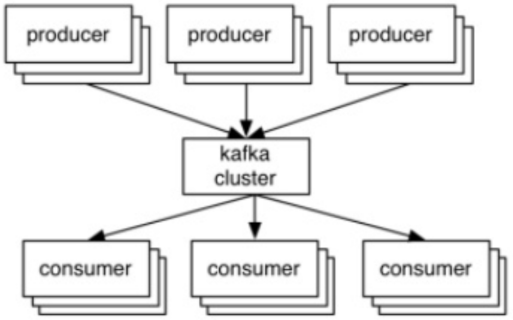
Data Pipeline

Kafka	https://puncsky.com/notes/61-what-is-apache-kafka **# What is Apache Kafka?** Apache Kafka is a distributed streaming platform. **# Why use Apache Kafka?** Its abstraction is a queue and it features ● a distributed pub-sub messaging system that resolves N^2 relationships to N. Publishers and subscribers can operate at their own rates. ● super fast with zero-copy technology ● support fault-tolerant data persistence It can be applied to ● logging by topics ● messaging system ● geo-replication ● stream processing **# Why is Kafka so fast?** Kafka is using zero copy in which that CPU does not perform the task of copying data from one memory area to another. Without zero copy:  With zero copy:

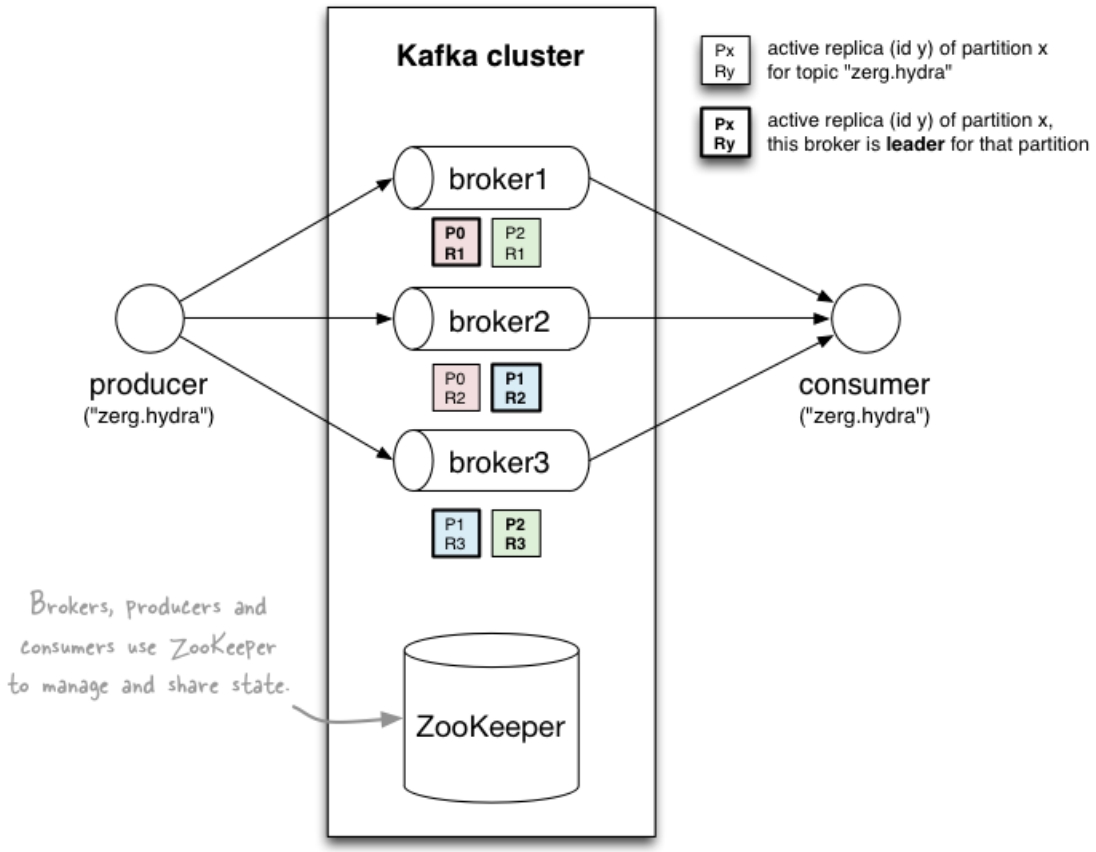


Architecture

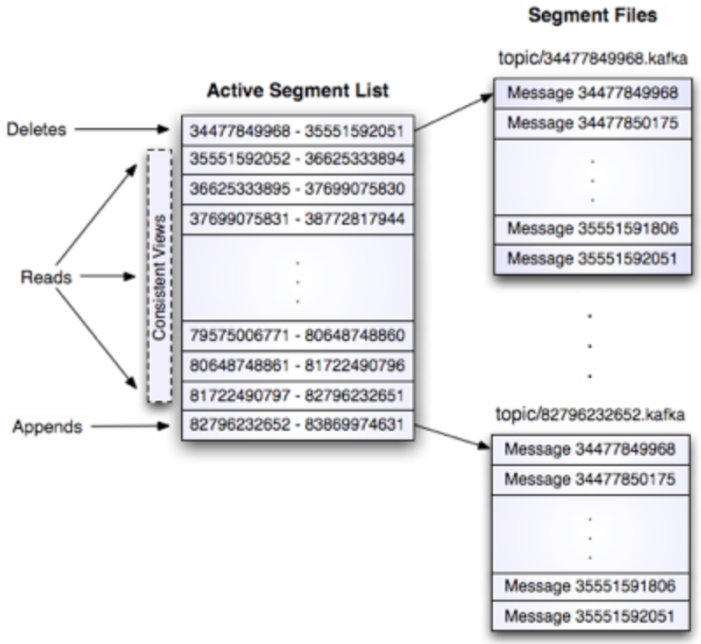
Looking from outside, producers write to brokers, and consumers read from brokers.



Data is stored in topics and split into partitions which are replicated.



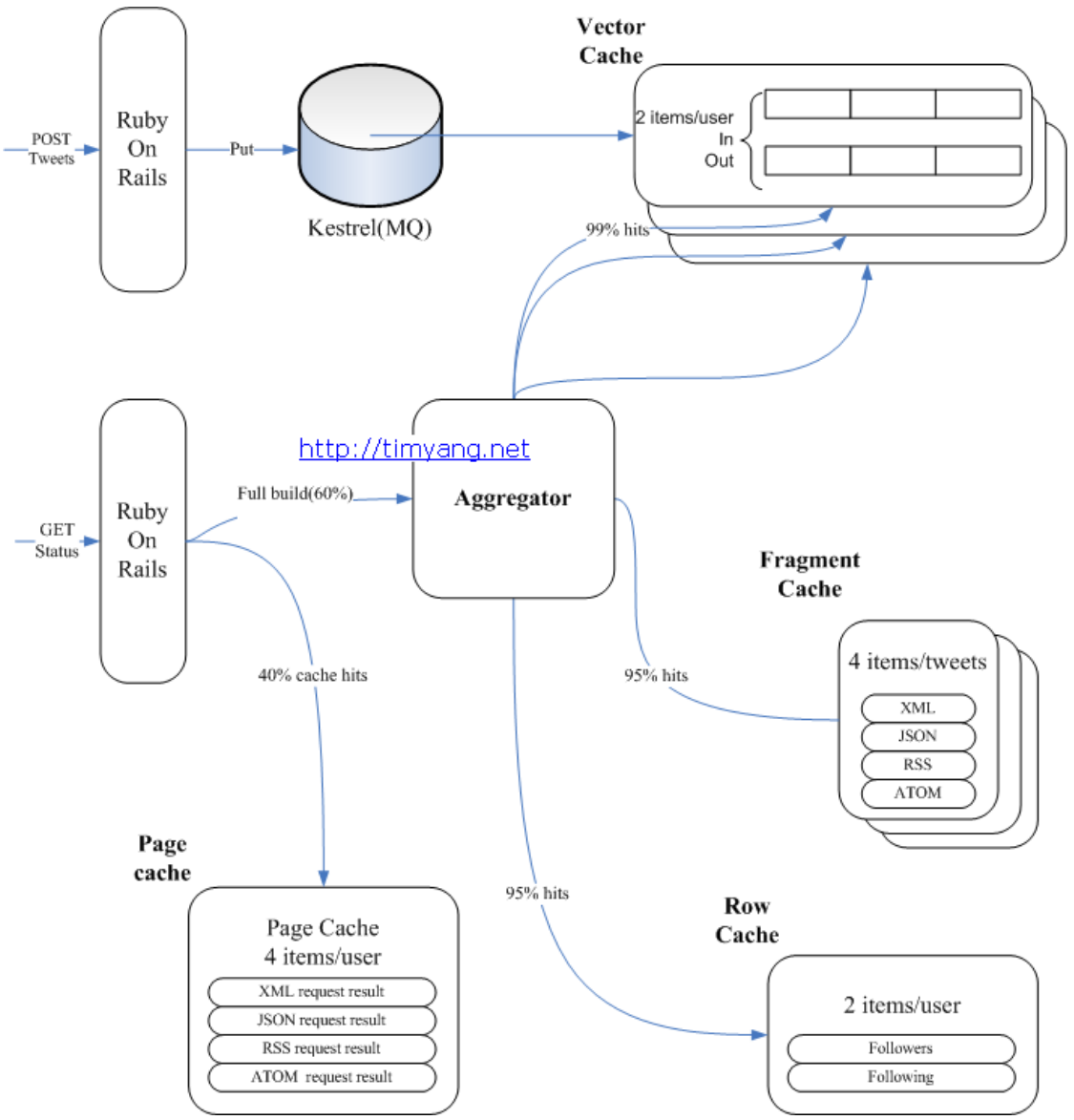
1. Producer publishes messages to a specific topic.
    - Write to in-memory buffer first and flush to disk.
    - append-only sequence write for fast write.
    - Available to read after write to disks.
2. Consumer pulls messages from a specific topic.

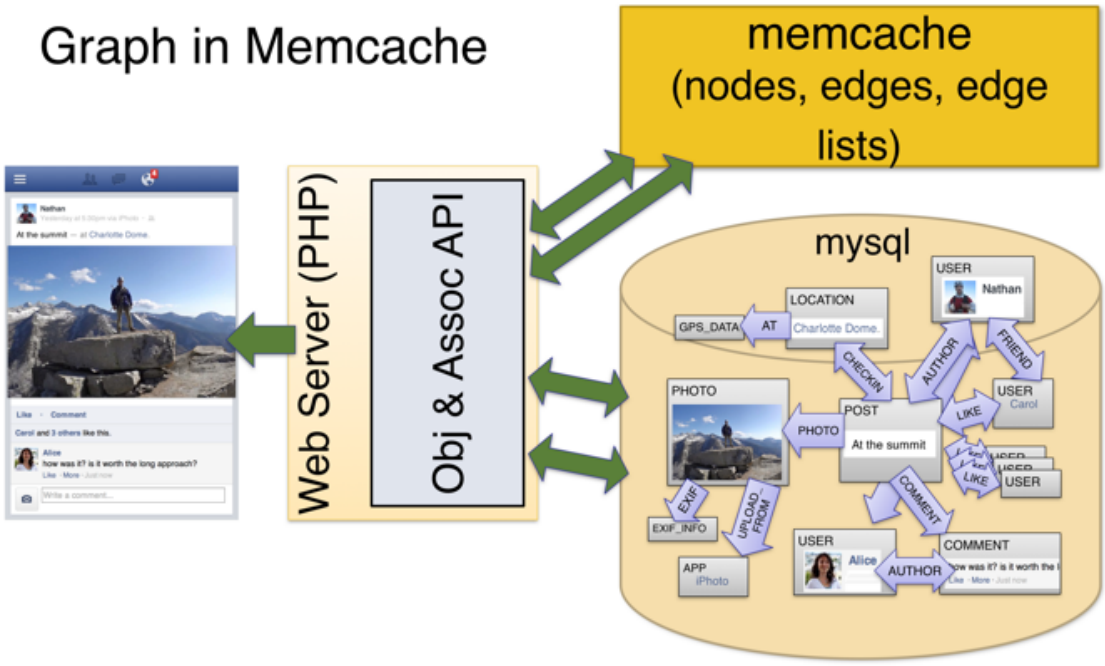
	○ use an “offset pointer” (offset as seqId) to track/control its only read progress. 3. A topic consists of partitions, load balance, partition (= ordered + immutable seq of msg that is continually appended to) ○ Partitions determine max consumer (group) parallelism. Only once consumer can read from one partition at the same time. How to serialize data? Avro What is its network protocol? TCP What is a partition’s storage layout? O(1) disk read  **# How to tolerate fault?** In-sync replica (ISR) protocol. It tolerates (numReplicas - 1) dead brokers. Total Replicas = ISRs + out-of-sync replicas 1. ISR is the set of replicas that are alive and have fully caught up with the leader (note that the leader is always in ISR). 2. When a new message is published, the leader waits until it reaches all replicas in the ISR before committing the message. 3. If a follower replica fails, it will be dropped out of the ISR and the leader then continues to commit new messages with fewer replicas in the ISR. Notice that now, the system is running in an under replicated mode. If a leader fails, an ISR is picked to be a new leader. 4. Out-of-sync replica keeps pulling message from the leader. Once catches up with the leader, it will be added back to the ISR. **# Is Kafka an AP or CP system in CAP theorem?** Jun Rao says it is CA, because “Our goal was to support replication in a Kafka cluster within a single datacenter, where network partitioning is rare, so our design focuses on maintaining highly available and strongly consistent replicas.” However, it actually depends on the configuration. 1. Out of the box with default config (min.insync.replicas=1, default.replication.factor=1) you are getting AP system (at-most-once). 2. If you want to achieve CP, you may set min.insync.replicas=2 and topic replication factor of 3 - then producing a message with acks=all will guarantee CP setup (at-least-once), but (as expected) will block in cases when not enough replicas (<2) are available for particular topic/partition pair. (see design_ha, producer config docs)
Facebook Scribe	
Facebook Wormhole	https://www.facebook.com/notes/facebook-engineering/wormhole-pubsub-system-moving-data-through-space-and-time/10151504075843920

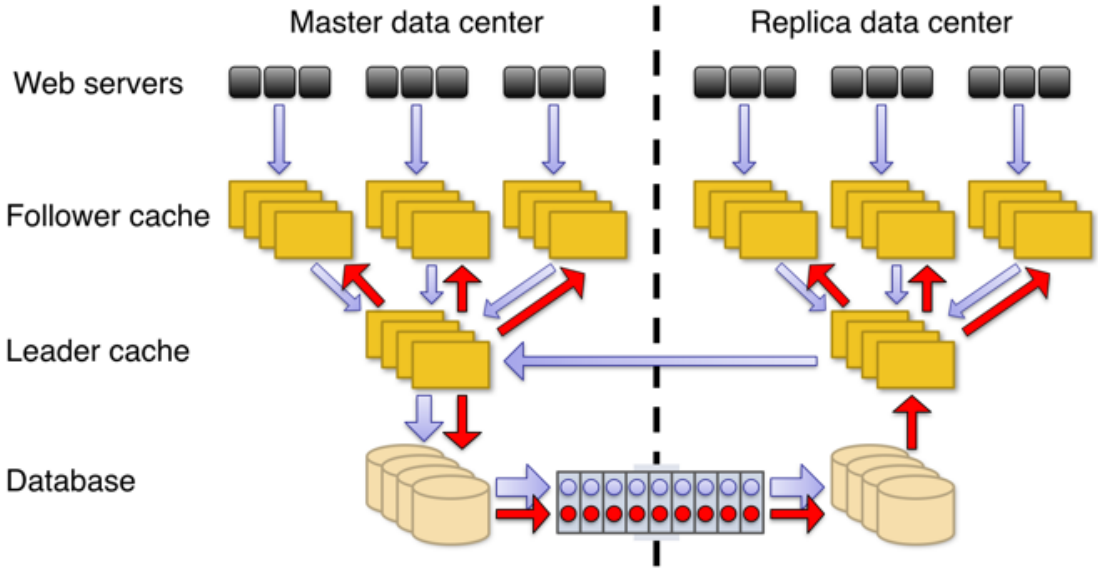
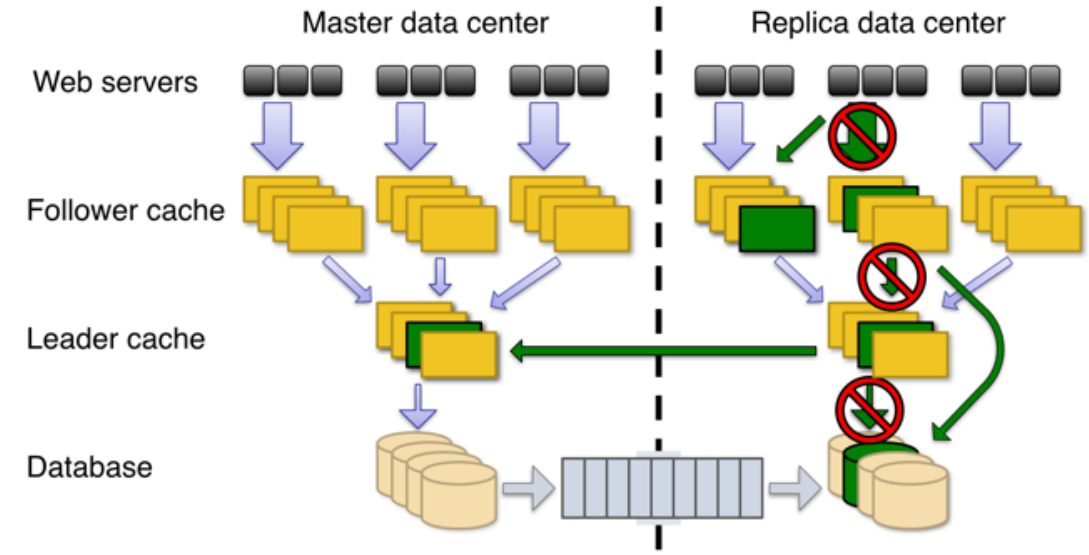
Distributed File System

In-memory KV store

`Out-of-box choices`	`Redis, Memcache, Riak, Berkeley DB, HamsterDB, Amazon Dynamo, Project Voldemort, etc.`
`[Cache]`	KV cache is like a giant hash map and used to reduce the latency of data access, typically by 1. Putting data from slow and cheap media to fast and expensive ones. 2. Indexing from tree-based data structures of `O(log n)` to hash-based ones of`O(1)` to read and write There are various cache policies like read-through/write-through(or write-back), and cache-aside. By and large, Internet services have a read to write ratio of 100:1 to 1000:1, so we usually optimize for read. In distributed systems, we choose those policies according to the business requirements and contexts, under the guidance of CAP theorem. **Regular Patterns** ● Read ○ Read-through: the clients read data from the database via the cache layer. The cache returns when the read hits the cache; otherwise, it fetches data from the database, caches it, and then return the vale. ● Write ○ Write-through: clients write to the cache and the cache updates the database. The cache returns when it finishes the database write. ○ Write-behind / write-back: clients write to the cache, and the cache returns immediately. Behind the cache write, the cache asynchronously writes to the database. ○ Write-around: clients write to the database directly, around the cache. **Cache-aside pattern** When a cache does not support native read-through and write-through operations, and the resource demand is unpredictable, we use this cache-aside pattern. ● Read: try to hit the cache. If not hit, read from the database and then update the cache. ● Write: write to the database first and then delete the cache entry. A common pitfall here is that people mistakenly update the cache with the value, and double writes in a high concurrency environment will make the cache dirty. There are still chances for dirty cache in this pattern. It happens when these two cases are met in a racing condition: 1. read database and update cache 2. update database and delete cache **Where to put the cache?** ● client-side ● distinct layer ● server-side **What if data volume reaches the cache capacity? Use cache replacement policies** ● LRU(Least Recently Used): evict the most recently used entries and keep the most recently used ones. ● LFU(Least Frequently Used): evict the most frequently used entries and keep the most frequently used ones.

	• ARC(Adaptive replacement cache): it has a better performance than LRU. It is achieved by keeping both the most frequently and frequently used entries, as well as a history for eviction. (Keeping MRU+MFU+eviction history.) **Who are the King of the cache usage?** Facebook TAO
Redis (though supports persistence)	Redis is single threaded. Supports persistence (snapshot and may lose data from failover). Redis Cluster is a distributed implementation and supports 1000 nodes. INCR and SET are atomic
Memcached	
Usercase: Twitter cache	读写分离 + 为不同层级的数据，不同格式的数据做 cache  - Cache分Page cache, fragment cache, row cache, vector Cache, cache命中率见图。 - Fragment cache存放了API各种请求格式的数据，包括XML, JSON, RSS, ATOM。 - 发表Tweets是先放入Kestrel，再异步处理，Kestrel用的也是memcached协议。 - API requests: 550 r/s。 - POST tweets: 峰值：平时 80tweets/s，奥巴马就任时达到 350tweets/s。 - Aggregator模块需要访问memcached multi get 数百个/s。 - Ruby on Rails前面还用了Varnish作前端反向代理。

	http://timyang.net/architecture/twitter-cache-architecture/
Facebook [TAO]	https://puncsky.com/notes/49-facebook-tao **# What are the challenges?** Before TAO, use cache-aside pattern  Social graph data is stored in MySQL and cached in Memcached 3 problems: 1. list update operation in Memcached is inefficient. cannot append but update the whole list. 2. clients have to manage cache 3. Hard to offer read-after-write consistency To solve those problems, we have 3 goals: ● online data graph service that is efficiency at scale ● optimize for read (its read-to-write ratio is 500:1) ○ low read latency ○ high read availability (eventual consistency) ● timeliness of writes (read-after-write) **# Data Model** ● Objects (e.g. user, location, comment) with unique IDs ● Associations (e.g. tagged, like, author) between two IDs ● Both have key-value data as well as a time field **# Solutions: TAO** 1. Efficiency at scale and reduce read latency ○ graph-specific caching ○ a standalone cache layer between the stateless service layer and the DB layer (aka Functional Decomposition) ○ subdivide data centers (aka Horizontal Data Partitioning) 2. Write timeliness ○ write-through cache ○ follower/leader cache to solve thundering herd problem ○ async replication 3. Read availability ○ Read Failover to alternate data sources **# TAO's Architecture**

	• MySQL databases → durability • Leader cache → coordinates writes to each object • Follower caches → serve reads but not writes. forward all writes to leader.  Read failover  Related: Twitter FlockDB

Document Store

Out-of-box choices	MongoDB, CouchDB, Terrastore, OrientDB, RavenDB, etc

SSD or Rotating Disk KV Store

LevelDB	C++ 代码不到 1000 行，值得上手学习 Sorted by key Operations: Put, Get, Delete 单进程

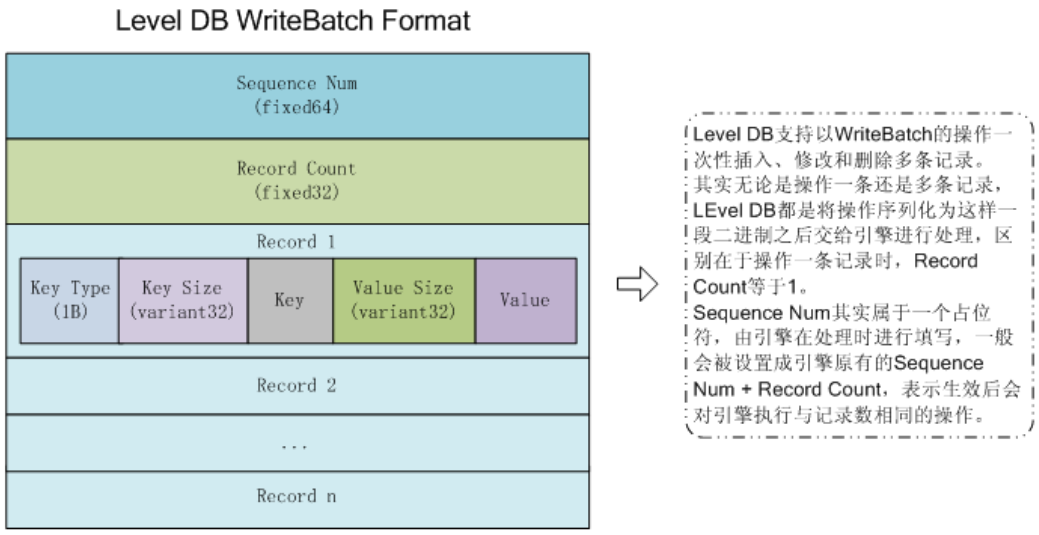
Motivation

Optimize for high volumn write (O(log n) 但是是 batch write, 不是 random write)
可遍历

Key Ideas

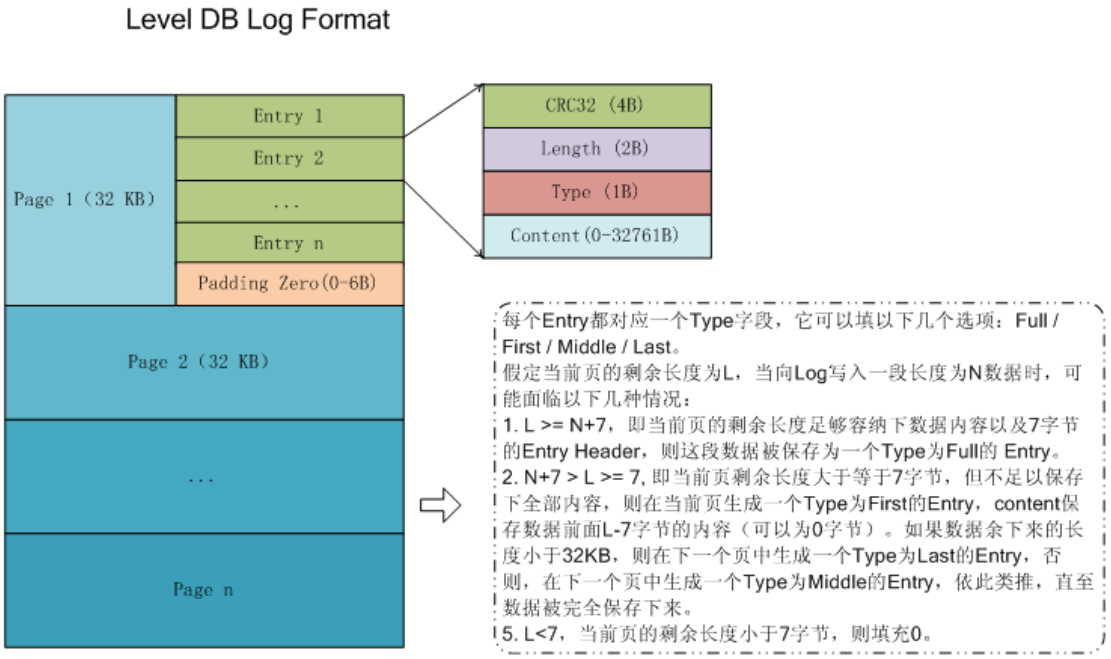
- [LSM tree](Log structured merge tree): LSM trees maintain data in two or more separate structures, each of which is optimized for its respective underlying storage medium; data is synchronized between the two structures efficiently, in batches.
    - 此处 LevelDB 的做法是 Write to the in-memory skiplist-based memtable first, and append-only to the disk. Compact if necessary. 最后形成 memtable 和 SSTables in levels 的混合形态。
- Failure caused by missing memtable? [WAL](write ahead log): all modifications are written to a log before they are applied.

Write Batch

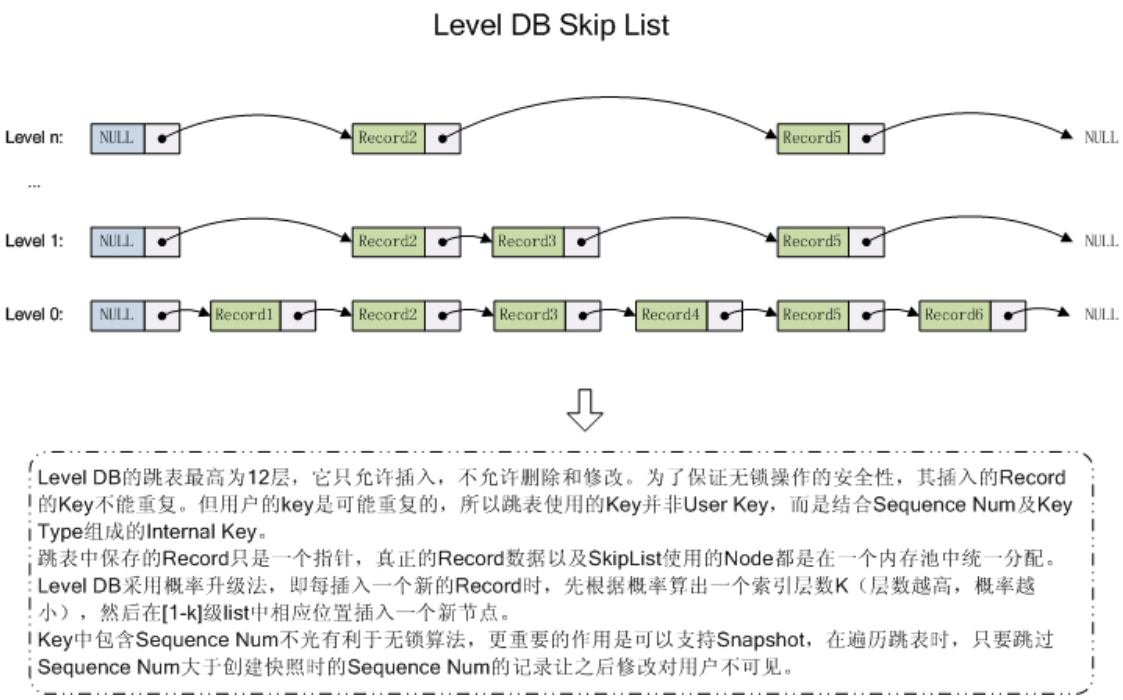


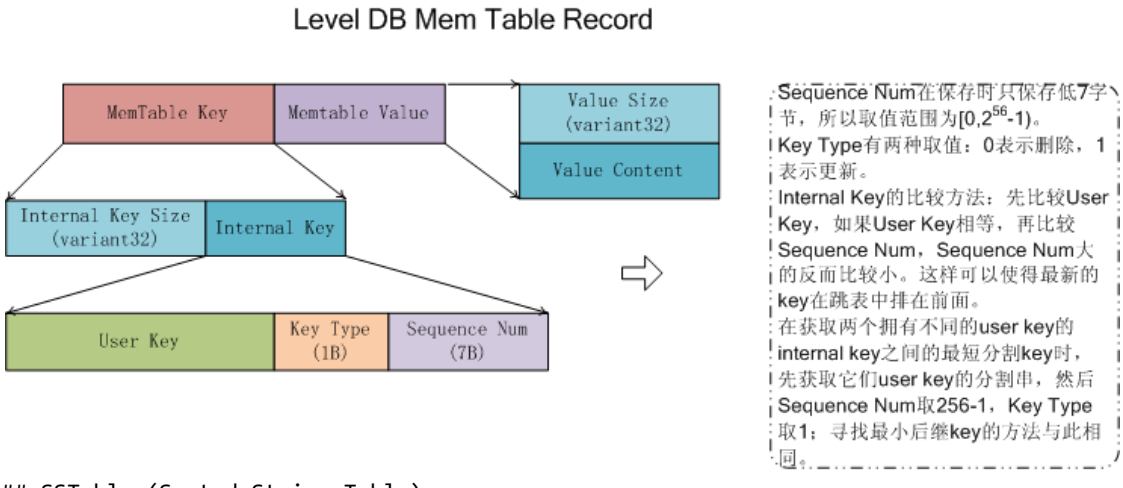
Log

LevelDB使用Memory Mapping的方式对log数据进行访问：如果前一次映射的空间已写满，则先将文件扩展一定的长度（每次扩展的长度按64KB,128KB,...的顺序逐次翻倍，最大到1MB），然后映射到内存，对映射的内存再以32KB的Page进行切分，每次写入的日志填充到Page中，攒积一定量后Sync到磁盘上（也可以设置WriteOptions，每写一条日志就Sync一次，但是这样效率很低）

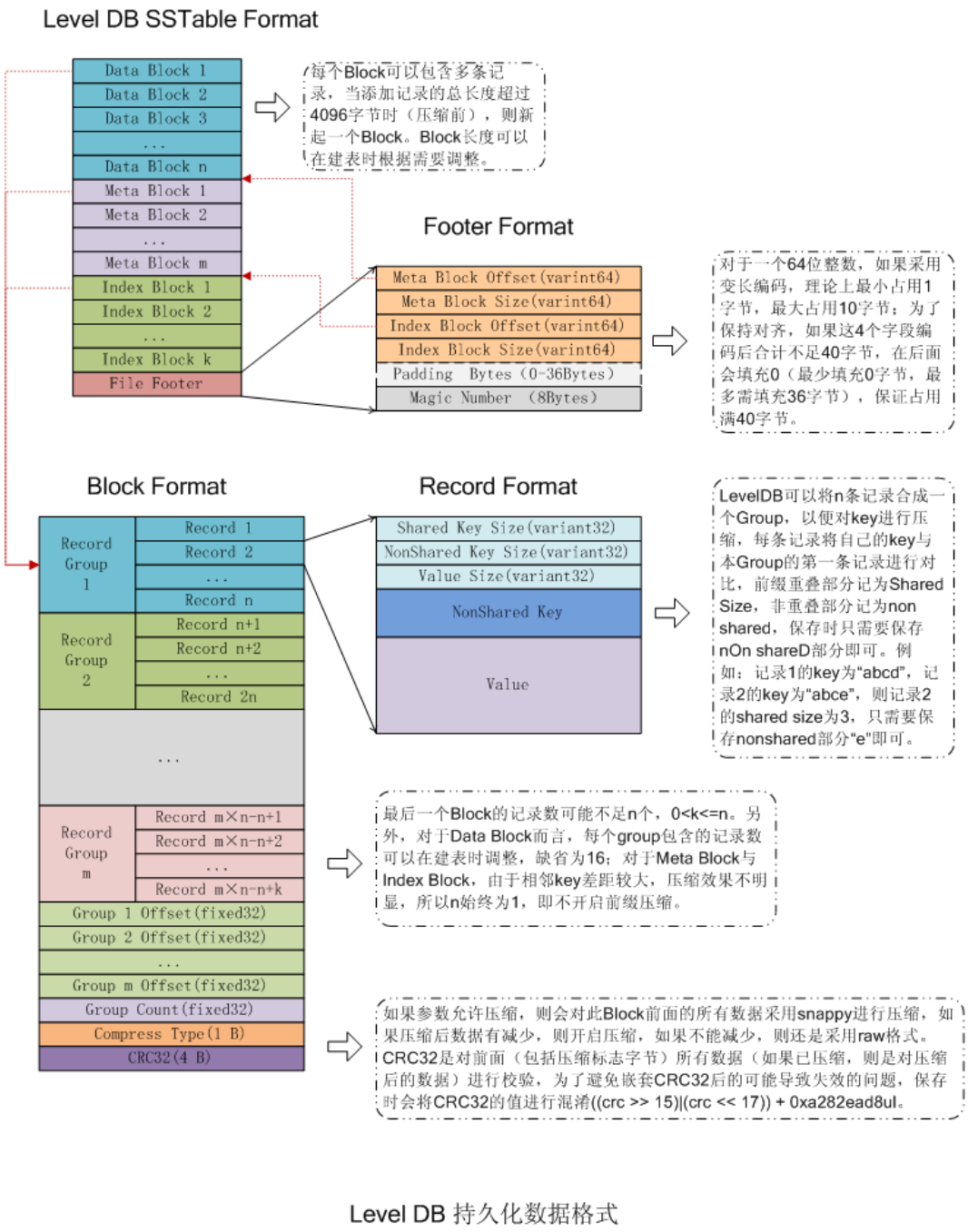


memtable and skip list





SSTable (Sorted String Table)

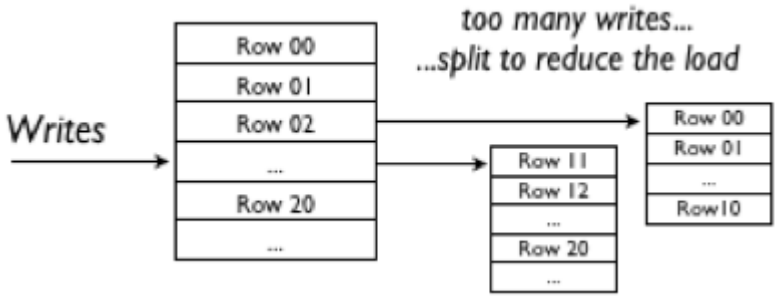
	Level DB间歇性地将内存中的SkipList对应的数据集合Dump到磁盘上，生成一个sst的文件  Level DB 持久化数据格式 ## SSTables in Levels Level DB可能dump多个sst文件，这些文件的key范围可能重叠。按照Level DB的设计，其会将sst分为7个等级，可以视为代龄，其中，只有Level 0中的sst可能存在key的区间重叠的情况，而level1 - level6中，同一level中的sst可以保证不重叠，但不同level之间的sst依然可能key重叠。因此，如果查询一个key，其最多可能在6+n个sst中同时存在，n为level0中sst的个数；同时，由于这些文件的生成有先后关系，查询时还需要注意顺序 [TODO] 补图 Usercase: Chrome ## Ref http://blog.csdn.net/icefireelf/article/details/7515816
Bitcask	Append only Only one Active data file (for write), older data file(s) (for read only) ## Components:

	1. In-memory HashMap<Key, <fildId, value_offset, value_size, timestamp>> 2. Data file, layout: |crc|timestamp|key_size|value_size|key|value| 3. (index) hint file ## Operations Delete: set the value to a magic number Put: append to the active data file and update in-memory hash map Periorical compaction: Copy latest entries: In-memory hashmap is always up-to-date. Stop and copy into new files. O(n) n is the number of valid entries. + Efficient for lots of entries out-dated or deleted. - Consume storage if little entries out-dated. May double the space. (can be resolved by having a secondary node do the compression work with GET/POST periordically. E.g. Hadoop secondary namenode). Scan and move: foreach entry, if it is up-to-date, move to the tail of the validated section. O(n) n is the number of all the entries. + shrink the size + no extra storage space needed - Complex and need to sync hashmap and storage with transactions. May hurt performance. How to detect records that can be compacted? Timestamp. How to compact multiple files together (if sorted)? [TODO] check bitcask paper compaction 怎么做 ## What if hash map cannot fit into a single machine's memory? Consistent hashing, chord DHT, O(logn)

Dynamo, distributed KV store: DHT, virtual nodes, N(by default 3) replicas, R+W>N, hash tree to sync, gossip protocol to maintain membership, vector clock

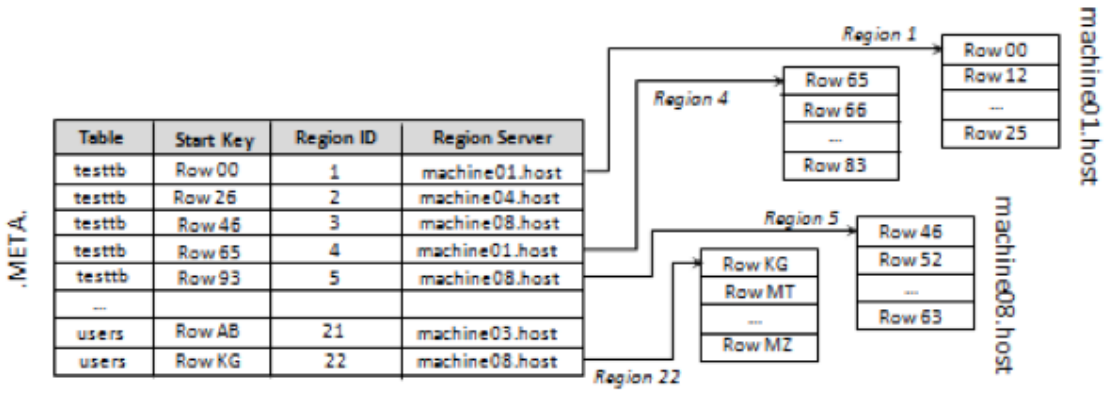
Column-oriented DBMS

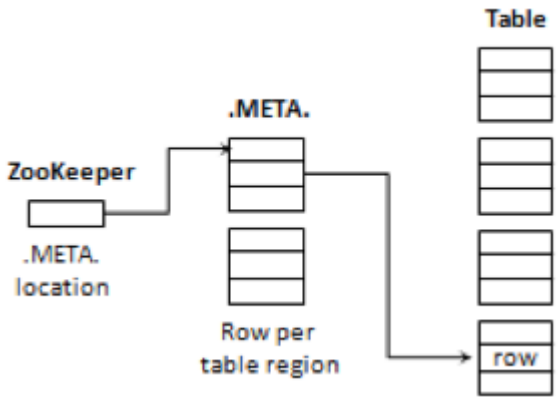
Out-of-box choices	Cassandra, HBase, Hypertable, Amazon SimpleDB, etc.
HBase	Motivation? petabytes structured/semi-structured data storage manager for low-latency random reads/writes on HDFS Time complexity? LSM tree indexing O(log n) Abstraction? big table, column-based, LSM tree indexing by (RowKey, ColumnKey, Timestamp/Version) Scale? horizontal scalability with Regions (a contiguous, sorted range of rows that are stored together.)



Components?

HMaster(META system table, region assignment, and balancing) Which node as master?
ZooKeeper





HRegionServer. Slaves are called region servers. One region per region server.

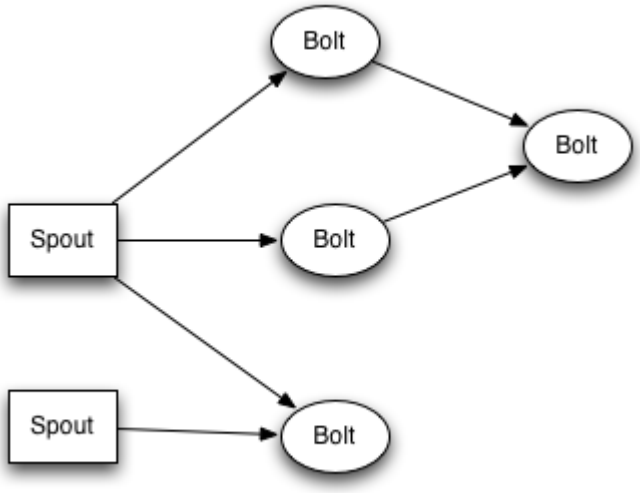
Client, can go to region servers DIRECTLY (no dependency on master). Query META table on master and identify the region (cache locations for future, get exception if miss), and clients go to region servers for read and write (HTable). The Master is used by the client only for table creation, modification, and deletion operations (HBaseAdmin).

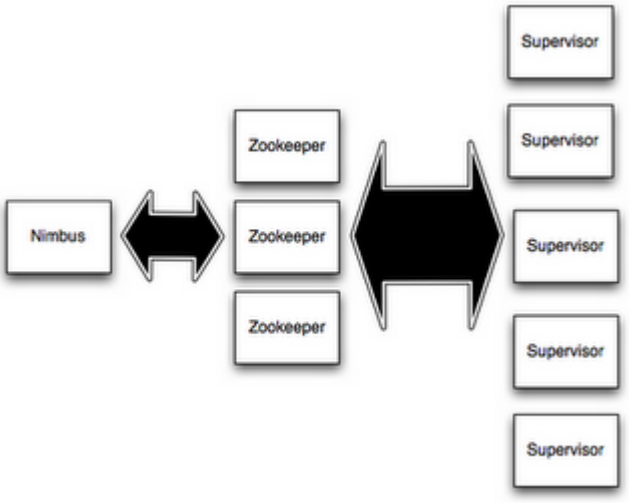
Usercase: Facebook messaging system

Ref
http://blog.cloudera.com/blog/2013/04/how-scaling-really-works-in-apache-hbase/

Large-scale Batch Processing

Stream Processing

Motivation?	Real-time data analysis (financial data, twitter feeds, news data ...) High-frequency trading Complex Event Processing Real-time Search Social Networks Incorporating Web application user feedback in real time Search advertising personalization Low-latency data processing pipelines Online algorithm development Requirements: low latency, robust and fault-tolerant, scaling out, flexibility
Examples?	Apache S4 (Yahoo), Apache Storm (Twitter), Google MillWheel, Apache Samza (Linkedin), Spark DStream, hop - Hadoop Online Prototype
General Architecture?	Master-slave 简单: Storm, MillWheel, Samza - 比较特别的情况: Samza on YARN and Kafka P2P 复杂: S4
Storm	## Motivation run “topologies” (A topology is a graph of computation) that never end for input stream (unbounded sequence of tuples) ## Abstraction spout: source of streams (input: tuples) bolt: consumes any number of input streams, does some processing, and possibly emits new streams. A topology is a graph of stream transformations where each node is a spout or bolt. ## Architecture (Master-slave) 



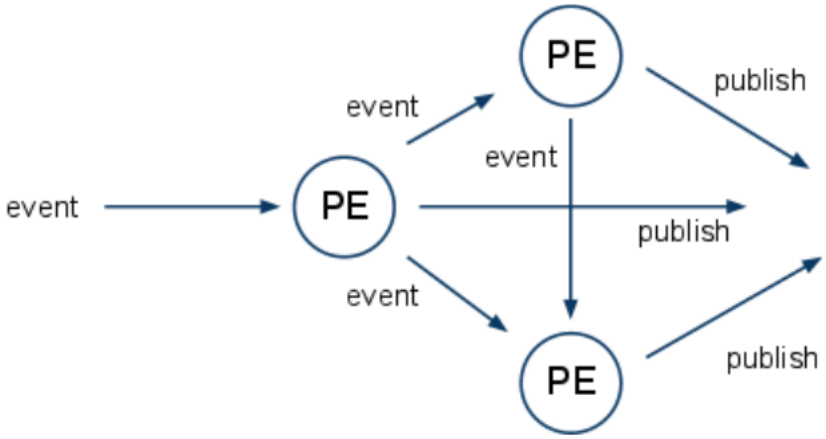
Master: a nimbus daemon (job tracker). 分发计算代码，分配计算任务，故障检测... (Flying numbus 筋斗云)

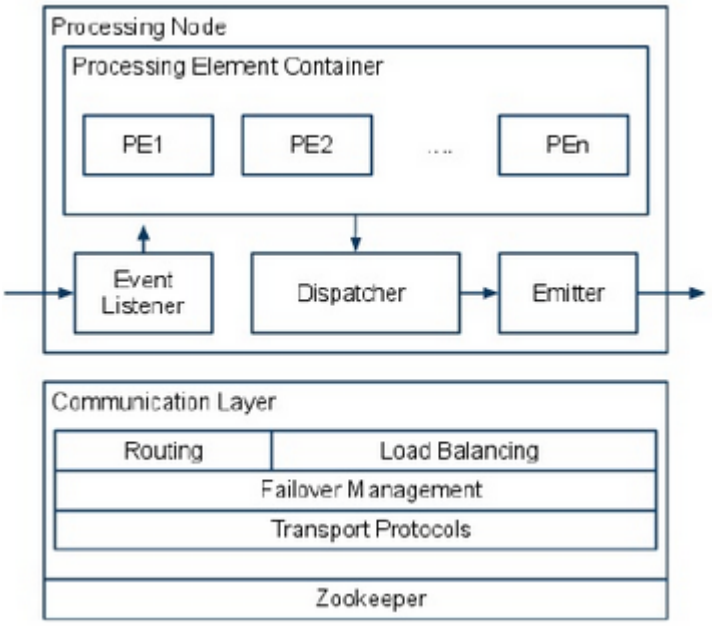
Worker: a supervisor daemon(listen on tasks)

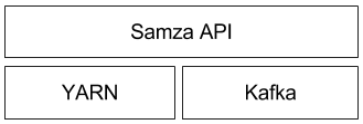
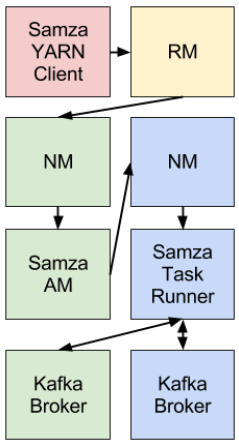
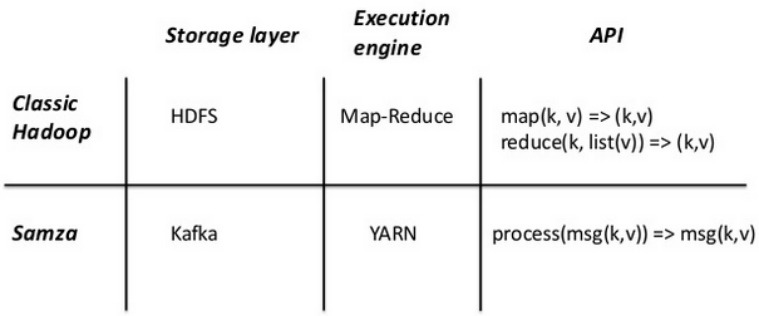
Interaction: through zookeeper cluster. daemons are fail-fast and stateless. all state is kept in Zookeeper or on local disk.

S4 Architecture

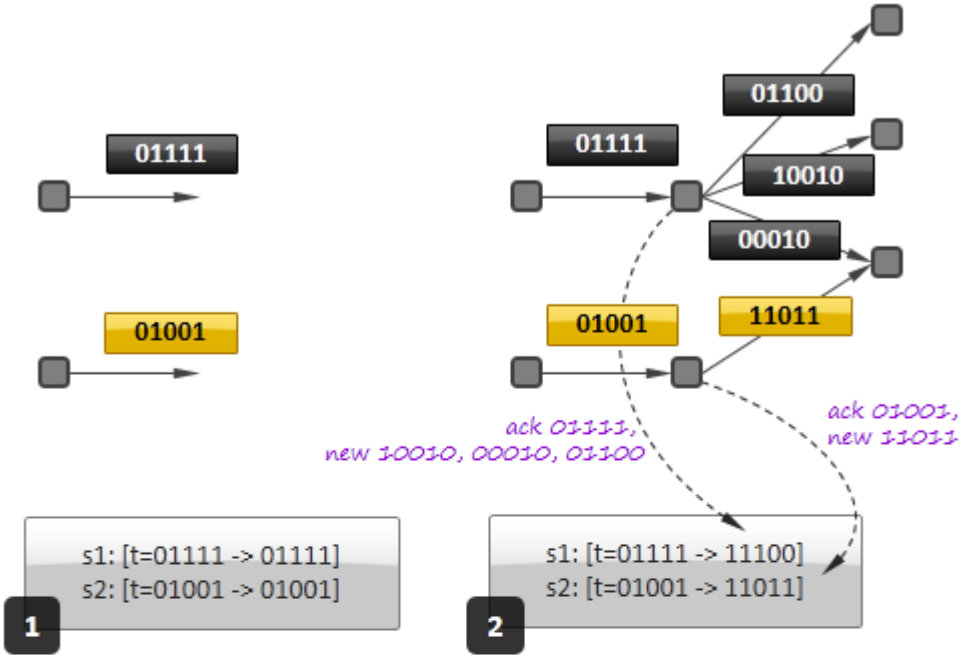
(+) scaling out and fault-tolerant. no single point of failure
(-) complex to implement

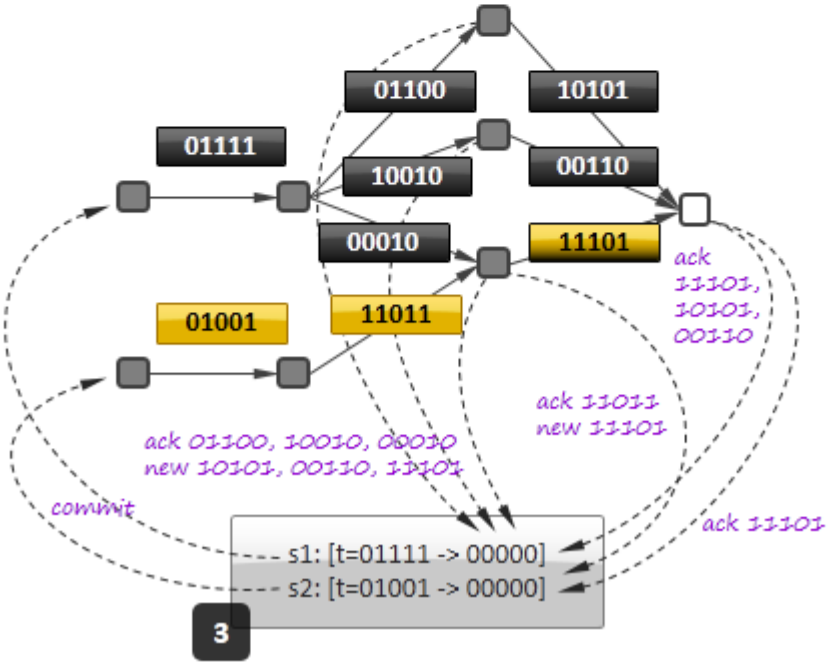




	严重的问题是 没有合理的 persistence strategy，机器出故障会丢数据
Samza Architecture	 Samza uses YARN and Kafka to provide a framework for stage-wise stream processing and partitioning. Everything, put together, looks like this (different colors indicate different host machines):   
DAG topology	## Nodes Source Nodes: Storm spout, MillWheel Injector, S4 Keyless Event Computing Nodes: Storm bolt, ## Data Flow MillWheel: (K, V, Timestamp) Storm: Tuple S4: [K, Attribute] ## Topology 常见的 topology: pipeline, shuffle grouping, field grouping, broadcasting Storm 的 topology 定义最灵活，Samza 受限 kafka，YARN 缺乏灵活性，只能表达一些简单的任务

	TopologyBuilder builder = new TopologyBuilder(); builder.setSpout("spout", new KestrelSpout("kestrel.everything.me", 22133, "my_queue"), 5); builder.setBolt("split", new SplitSentence(), 8) .shuffleGrouping("spout"); builder.setBolt("count", new WordCount(), 12) .fieldsGrouping("split", new Fields("word"));
Delivery Guarantees	At-Least Once Delivery, At-Most Once Delivery, Exactly-Once Delivery (最重要) ## Storm Exactly-Once Delivery https://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/ Key idea is to utilize the fact that the result of XOR of two exact same signatures is 0. framework maintains a set of pairs [event ID -> random signature] for each initial event Downstream nodes 对生成的 events 生成新的 random signature, 每个 node 对所有 edges 的所有 event signatures 做 XOR, 做完后更新 [event ID -> result], 所有的计算结束后, 要保证 [event ID -> 0]





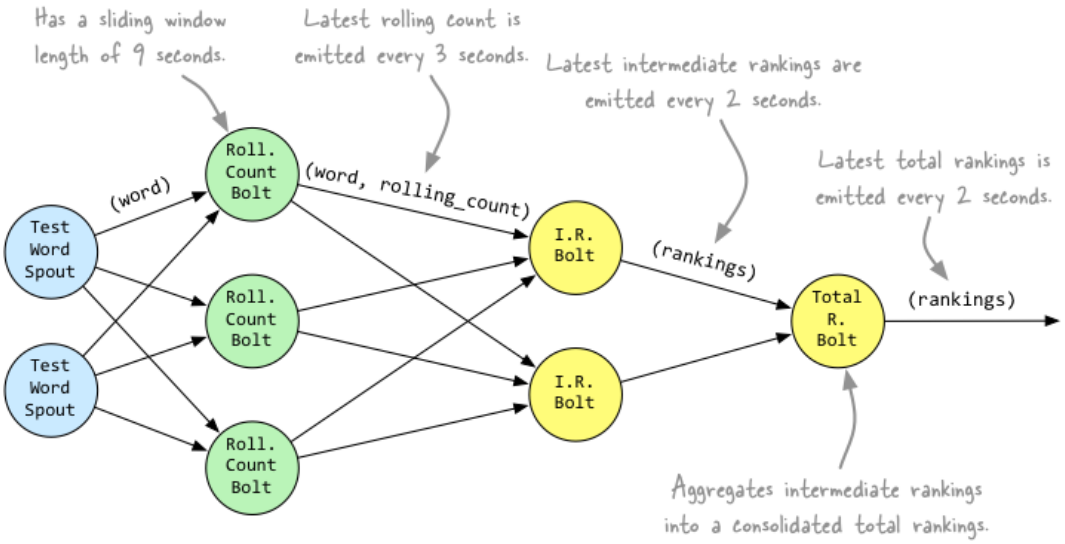
MillWheel Exactly-Once Delivery

ACK + 检查重复 (bloom filter, external database: msgId, states, ...)

[TODO]

Persistence /Fault-tolerance/ failover / availability	Cold Standby: Use heartbeat or ZooKeeper to track failure. provision new standby nodes when failure occurs. Only suitable for stateless services. Hot Standby: Keep two active nodes undertaking the same role. Downstream needs a stream selector to resolve duplications. Warm Standby: Keep two active nodes but the secondary node does not take traffic unless failure occurs. Checkpointing (or like Redis snapshot): write ahead log , standby node recovers from the log (-) time-consuming for large logs (-) lose data Usercase: Storm, WhillWheel, Samza

Storm State Persistence	[TODO]
MillWheel and Samza State Persistence	[TODO]

Trending Topics
 	
top 10 topics in the last week
 	
[Lab]
	## Motivation
	
	a topic is considered trending when it has been among the top N topics in a given window of time.
	
	## Key Ideas
	
	(rolling counts (a.k.a. sliding windows) + ranking) in distributed stream processing
	
	## Topology
	
	 
	
	class RollingTopWords with main()
	// 一种 spout，三种 bolt
	- builder.setSpout(spoutId, new TestWordSpout(), 2);
	- builder.setBolt(counterId, new RollingCountBolt(9, 3), 3)
	.fieldsGrouping(spoutId, new Fields("word"));
	- builder.setBolt(intermediateRankerId, new IntermediateRankingsBolt(TOP_N), 2)
	.fieldsGrouping(counterId, new Fields("obj"));
	- builder.setBolt(totalRankerId, new TotalRankingsBolt(TOP_N))
	.globalGrouping(intermediateRankerId);
	
	class TestWordSpout emits tuples like (“nathan”, “mike”, “jackson”, “golda”, “bertels”) every 100 ms.
	
	class RollingCountBolt
	
	@Override
	public void execute(Tuple tuple) {
	// tick tuple is received at periodic intervals, set in conf
	if (TupleHelpers.isTickTuple(tuple)) {
	LOG.info("Received tick tuple, triggering emit of current window counts");
	emitCurrentWindowCounts();
	}
	else {
	countObjAndAck(tuple);
	}
	}

	private void emitCurrentWindowCounts() {
	  Map<Object, Long> counts = counter.getCountsThenAdvanceWindow();
	  ...
	  emit(counts, actualWindowLengthInSeconds);
	}
	
	private void emit(Map<Object, Long> counts) {
	  for (Entry<Object, Long> entry : counts.entrySet()) {
	  Object obj = entry.getKey();
	  Long count = entry.getValue();
	  collector.emit(new Values(obj, count));
	  }
	}
	
	private void countObjAndAck(Tuple tuple) {
	  Object obj = tuple.getValue(0);
	  counter.incrementCount(obj);
	  collector.ack(tuple);
	}
	
	class IntermediateRankingsBolt
	@Override
	void updateRankingsWithTuple(Tuple tuple) {
	  Rankable rankable = RankableObjectWithFields.from(tuple);
	  super.getRankings().updateWith(rankable);
	}
	
	class TotalRankingsBolt (Single instance)
	@Override
	void updateRankingsWithTuple(Tuple tuple) {
	  Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
	  super.getRankings().updateWith(rankingsToBeMerged);
	}
	
	最后的 rankings 似乎就只是个 ArrayList of <K, V>，合并后插入更新
	
	## Ref
	
	http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/

Interactive Data Analysis

Graph Database: Architecture and Algorithm

Out-of-box choices	Neo4J, Infinitegraph, OrientDB, FlockDB, etc.

Machine Learning: Paradigm and Architecture

Machine Learning: Distributed Algorithm

Incremental Processing

rDBMS

[Denormalization]	Denormalization is the process of attempting to **optimize the read performance** of a database by adding redundant data or by grouping data. 一般的设计中，我们常遵循 3NF ： all the attributes in a table are determined only by the candidate keys of that table and not by any non-prime attributes. 非主键字段间不存在依赖关系，减少 redundancy，加强 consistency 然而，disk seek 10 millisecond，local join spends 99% time on disk seek. It is worse for distributed join.

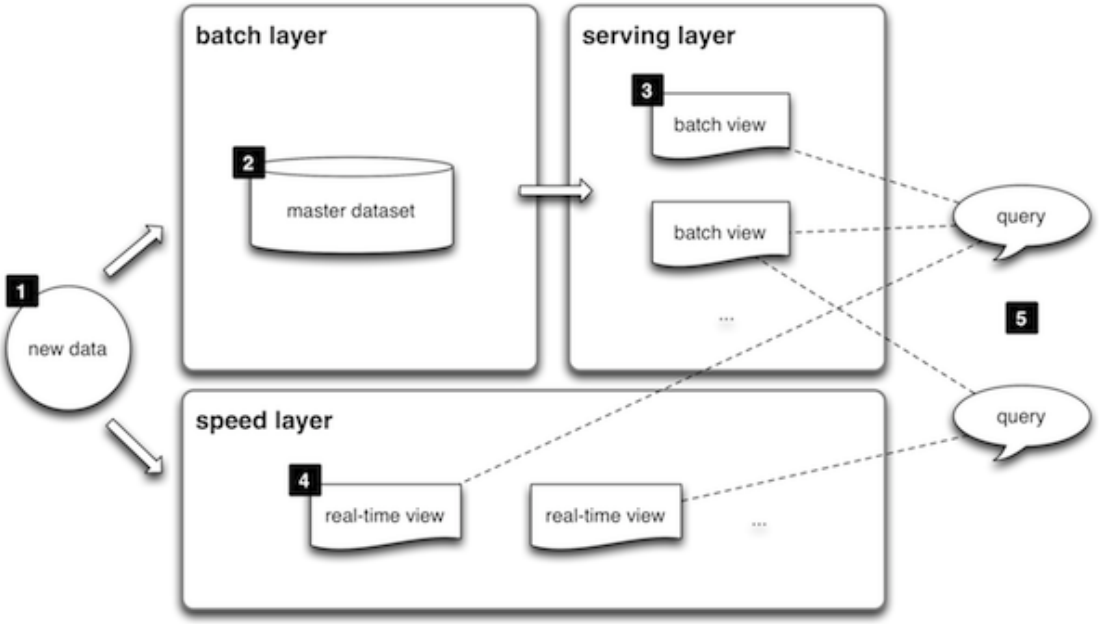
How to shard rDBMS?	58同城mysql实战（纯干货）
	http://chuansong.me/n/1283941
	
	1）用户库如何拆分
	用户库，10亿数据量
	user(uid, uname, passwd, age, sex, create_time);
	业务需求如下
	a）1%登录请求 => where uname=XXX and passwd=XXX
	b）99%查询请求 => where uid=XXX
	结论：“单key”场景使用“单key”拆库
	
	2）帖子库如何拆分 1 to many
	帖子库，15亿数据量
	tiezi(tid, uid, title, content, time);
	业务需求如下
	a）查询帖子详情（90%请求）
	SELECT * FROM tiezi WHERE tid=$tid
	b）查询用户所有发帖（10%请求）
	SELECT * FROM tiezi WHERE uid=$uid
	结论：“1对多”场景使用“1”分库，例如帖子库1个uid对应多个tid，则使用uid分库，tid生成时加入分库标记
	
	3）好友库如何拆分 many to many
	好友库，1亿数据量
	friend(uid, friend_uid, nick, memo, XXOO);
	业务需求如下
	a）查询我的好友（50%请求） => 用于界面展示
	SELECT friend_uid FROM friend WHERE uid=$my_uid
	b）查询加我为好友的用户（50%请求） => 用户反向通知
	SELECT uid FROM friend WHERE friend_uid=$my_uid
	结论：“多对多”场景，使用数据冗余方案，多份数据使用多种分库手段
	
	4）订单库如何拆分
	订单库，10亿数据量
	order(oid, buyer_id, seller_id, order_info, XXOO);
	业务需求如下
	a）查询订单信息（80%请求）
	SELECT * FROM order WHERE oid=$oid
	b）查询我买的东东（19%请求）
	SELECT * FROM order WHERE buyer_id=$my_uid
	c）查询我卖出的东东（1%请求）
	SELECT * FROM order WHERE seller_id=$my_uid
	结论：“多key”场景一般有两种方案
	a）方案一，使用2和3综合的方案
	b）方案二，1%的请求采用多库查询

Db proxy, MySQL Clustering vs Sharding	http://dba.stackexchange.com/questions/8889/mysql-sharding-vs-mysql-cluster http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.htmlClustering%20-%20Everything%20Is%20Automatic Clustering: everything is automatic Decentralized: Data are distributed, moved, rebalanced automatically. Nodes gossip with each other. (Gossiping may cause group isolation). Pros

	- automatic scale - easy to setup - can be spatially distributed to different regions and db takes care of it - high availability - load balancing - (claim to be) no single point of failure Cons - still fairly young - fundamentally complicated - less community support - fewer engineers with working knowledge - Difficult and scary upgrade mechanisms - Cluster management algo DO have a single point of failure, which impacts every node. (It takes pinterest down 4 times.) - data corruption unnoticed Sharding: everything is manual Centralized: Get rid of properties of clustering that you don't like and you get sharding. Data are distributed manually and will not move. Nodes are not aware of each other. Pros: - can split db and add more capacity - spatially distributed - high availability - load balancing - simple code for splitting data - id generation is simple Cons: - cannot perform most joins. - lost transaction capacities. - many constraints and operations (like join) are moved to the application level - schema changes require more planning - reports require running queries on all shards and then perform all the aggregation yourself When to shard? - If your project will have a few TBs of data then you should shard as soon as possible. - When the Pin table went to a billion rows the indexes ran out of memory and they were swapping to disk. - They picked the largest table and put it in its own database. - Then they ran out of space on the single database.

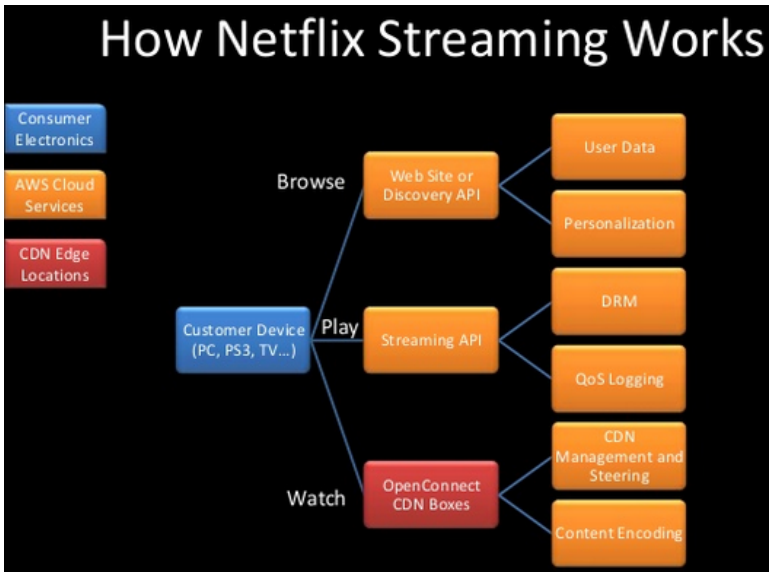
Misc

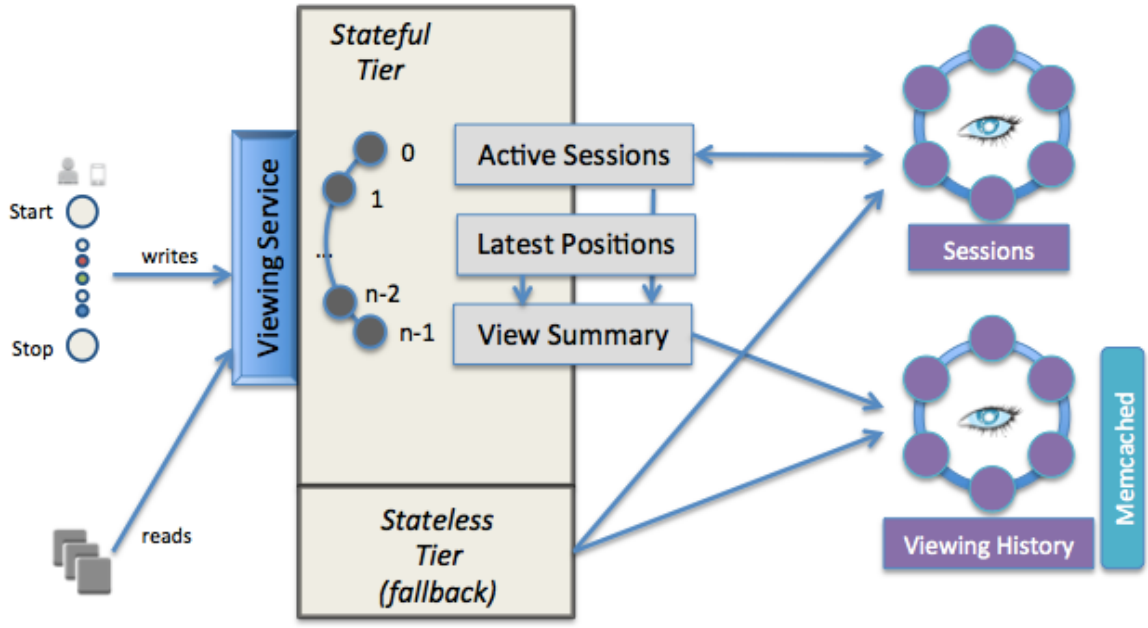
Lambda Architecture	Nathan Marz: generic, scalable, fault-tolerant data processing architecture

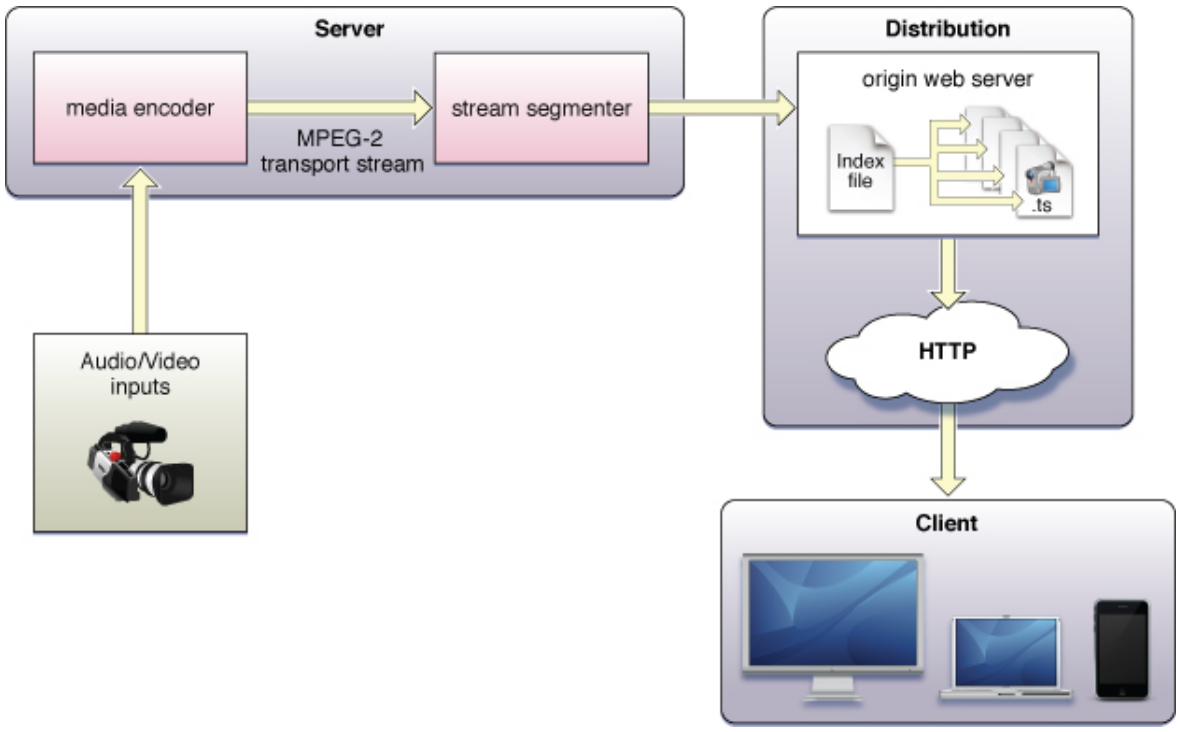
	 1. All **data** entering the system is dispatched to both the batch layer and the speed layer for processing. 2. The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views. 3. The **serving layer** indexes the batch views so that they can be queried in low-latency, ad-hoc way. 4. The **speed layer** compensates for the high latency of updates to the serving layer and deals with recent data only. 5. Any incoming **query** can be answered by merging results from batch views and real-time views
[Reverse Proxy]	Server side proxy having any number of features: load balancing, caching, security, SSL acceleration. 常见的有 Varnish, Squid varnish测试，用ab -c 200，内网，没keep-alive，文档长度20(不含http头)，Requests per second: squid 6k，nginx 10k，varnish 23k
AKF Scaling Cube	x: Horizontal Duplication y: Functional or Service Splits z: Look-up Oriented Split
POI 任给一个手机的位置信号(经纬度)，需要返回附近5mile 的POI (point of interest)	两种做法 KD Tree, Geohash ## Geohash HBase in Action, Chapter 8 http://architects.dzone.com/articles/designing-spacial-index Goal 1: 地理上相近的点存在disk上相近的地方 Goal 2: 每次query的时候返回尽可能少的点 Geohash encodes longitude/latitude by subdividing space into buckets of grid shape. Longtitude: [-180, 180], latitude: [-90, 90]. precision. If the point is greater than or equal to the midpoint, it's a 1-bit. Otherwise, it's a 0-bit. This process is repeated, again cutting the range in half and selecting a 1 or 0 based on where the target point lies. Then we can find a potential list of POI by a calculated prefix.

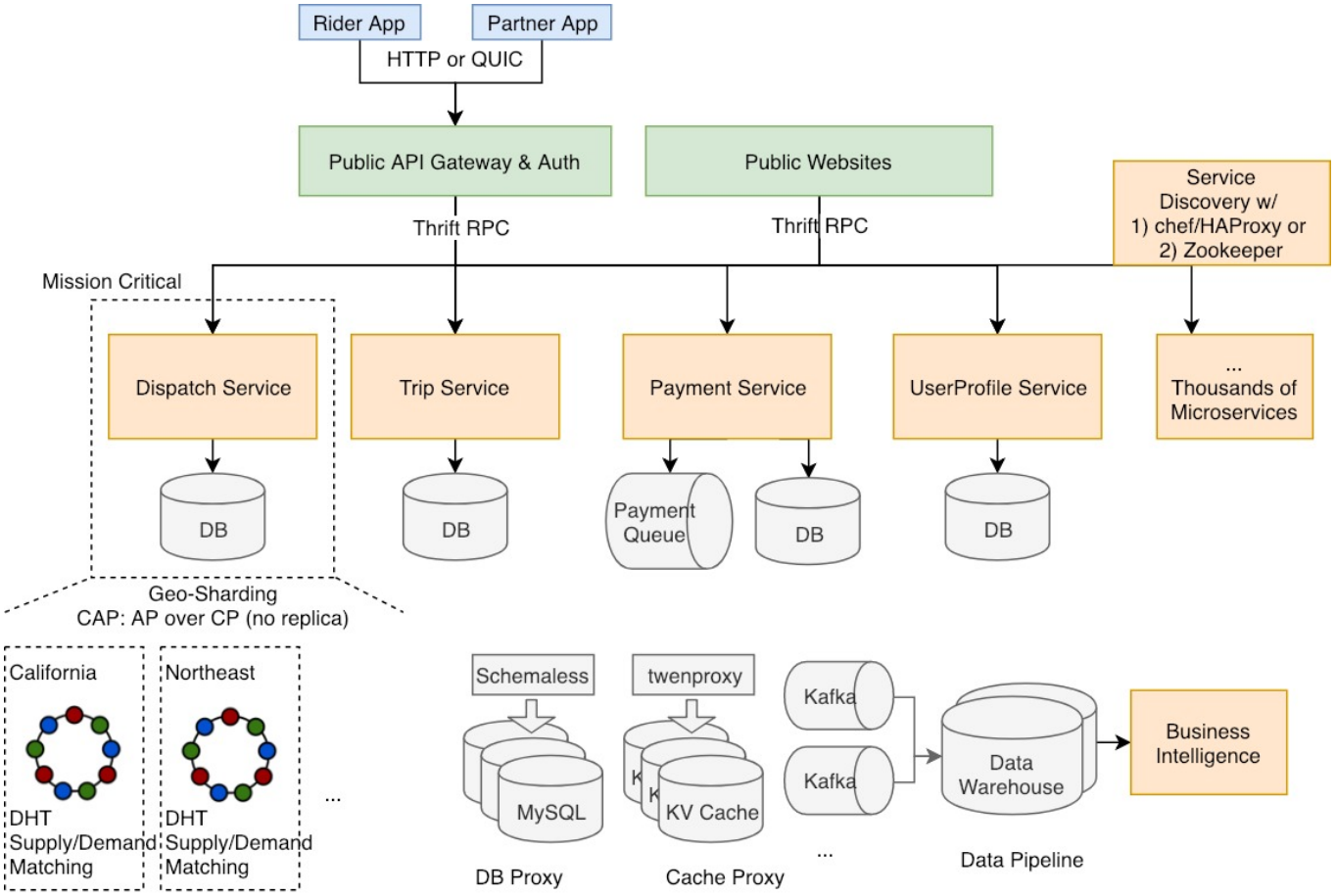
	## HBase (GEOHASH(row key), X, Y, ID, NAME) Related: find KNN (K nearest neighbors) quadtree

System Design and Architecture

Twitter	http://timyang.net/architecture/notes-timelines-twitter/ IO: 3k qps, [TODO]
Wechat	[TODO]
Netflix Architecture	AWS为例：数据存储使用s3，配合Relational db / Non sql database；然后是Service layer，功能包括：User authentication, session management, data streaming and other business logic；前台则主要是用户界面。优化包括：如何Cache，如何利用CDN network replicate data close to the users. 因为Netflix的数据大部分是静态数据，很少更新，电影电视剧的内容完全可以Replicate很多份放到Internet的Edge server上。 http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=202742077&idx=1&sn=77980c8a5d741399cd4a6efbd5fe2fa9#rd ## Architecture Use AWS stack: • AWS – Amazon Web Services (common name for Amazon cloud) • AMI – Amazon Machine Image (archived boot disk, Linux, Windows etc. plus application code) • EC2 – Elastic Compute Cloud – Range of virtual machine types m1, m2, c1, cc, cg. Varying memory, CPU and disk configurations. – Instance – a running computer system. Ephemeral, when it is de-allocated nothing is kept. – Reserved Instances – pre-paid to reduce cost for long term usage – Availability Zone – datacenter with own power and cooling hosting cloud instances – Region – group of Availability Zones – US-East, US-West, EU-Eire, Asia-Singapore, Asia-Japan • ASG – Auto Scaling Group (instances booting from the same AMI) • S3 – Simple Storage Service (http access) • EBS – Elastic Block Storage (network disk filesystem can be mounted on an instance) • RDB – Relational Data Base (managed MySQL master and slaves) • SDB – Simple Data Base (hosted http based NoSQL data store) • SQS – Simple Queue Service (http based message queue) • SNS – Simple Notification Service (http and email based topics and messages) • EMR – Elastic Map Reduce (automatically managed Hadoop cluster) • ELB – Elastic Load Balancer • EIP – Elastic IP (stable IP address mapping assigned to instance or ELB) • VPC – Virtual Private Cloud (extension of enterprise datacenter network into cloud) • IAM – Identity and Access Management (fine grain role based security keys)  Chaos Monkey

Netflix View State Service	## View Service http://techblog.netflix.com/2015/01/netflixs-viewing-data-how-we-know-where.html  The stateful tier(for memory speed) has the latest data for all active views stored in memory. Data is partitioned into N stateful nodes by a simple mod N of the member's account id. When stateful nodes come online they go through a slot selection process to determine which data partition will belong to them. Cassandra is the primary data store for all persistent data. Memcached is layered on top of Cassandra as a guaranteed low latency read path for materialized, but possibly stale, views of the data Consistency > Availability, 所以 stateful, 缺点是failure 导致 1/n 用户受到影响, 为了补充 Availability, 增加 stateless tier as standby service, 缺点是 stale data We use an eventually consistent approach to handling multiple writers, accepting that an inconsistent write may happen but will get corrected soon after due to a short cache entry TTL and a periodic cache refresh. Use Cassandra for very high volume, low latency writes. Use Redis for very high volume, low latency reads
HLS Http Live Streaming	## HTTP Live Streaming Apple: HTTP Live Streaming architecture: https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/HTTPStreamingArchitecture/HTTPStreamingArchitecture.html#//apple_ref/doc/uid/TP40008332-CH101-SW2 ## Motivation Video service over Http Live Streaming for mobile devices, which have limited memory/storage and suffer from unstable network connection and variable bandwith. Need midstream quality adjustments ## Solution / Abstraction In a typical configuration, a hardware encoder takes audio-video input, encodes it as H.264 video and AAC audio, and outputs it in an MPEG-2 Transport Stream, which is then broken into a series of short media files (.ts 可能10s) by a software stream segmenter. These files are placed on a web server. The segmenter also creates and maintains an index(.m3u8) file containing a list of the media files. The URL of the index file is published on the web server. Client software reads the index, then requests the listed media files in order and displays them without any pauses or gaps between segments. ## Architecture

	
滴滴打车	https://www.evernote.com/shard/s75/sh/40ba3b04-e8d1-4508-8aea-3a1e2000ba8c/91e62d61f4fda6f056ea1900bdcee9de
Designing Uber	Disclaimer: All things below are collected from public sources or purely original. No Uber-confidential stuff here. ## Requirements - ride hailing service targeting the transportation markets around the world - realtime dispatch in massive scale - backend design ## Architecture



Why micro services?

Conway's law says structures of software systems are copies of the organization structures.

	Monolithic Service	**Micro Services**
Productivity, when teams and codebases are small	✅ High	❌ Low
Productivity, when teams and codebases are large	❌ Low	✅ High (Conway's law)
Requirements on Engineering Quality	❌ High (under-qualified devs break down the system easily)	✅ Low (runtimes are segregated)
Dependency Bump	✅ Fast (centrally managed)	❌ Slow

Multi-tenancy support / Production-staging Segregation	✅ Easy	❌ Hard (each individual service has to either 1) build staging env connected to others in staging 2) Multi-tenancy support across the request contexts and data storage)
Debuggability, assuming same modules, metrics, logs	❌ Low	✅ High (w/ distributed tracing)
Latency	✅ Low (local)	❌ High (remote)
DevOps Costs	✅ Low (High on building tools)	❌ High (capacity planning is hard)

Combining monolithic codebase and micro services can bring benefits from both sides.

Dispatch Service

- consistent hashing sharded by geohash
- data is transient, in memory, and thus there is no need to replicate. (CAP: AP over CP)
- single-threaded or locked matching in a shard to prevent double dispatching

Payment Service

The key is to have an async design, because payment systems usually have a very long latency for ACID transactions across multiple systems.

- leverage event queues
- payment gateway w/ Braintree, PayPal, Card.io, Alipay, etc.
- logging intensively to track everything
- APIs with idempotency, exponential backoff, and random jitter

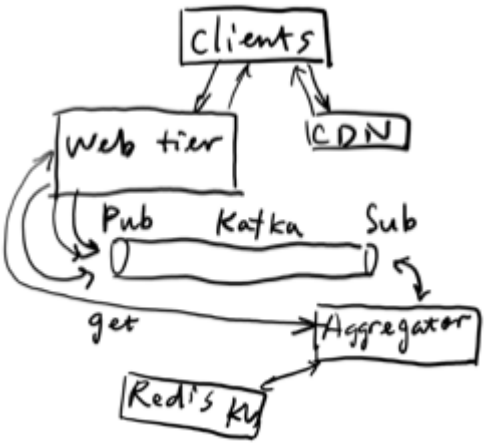
UserProfile Service and Trip Service

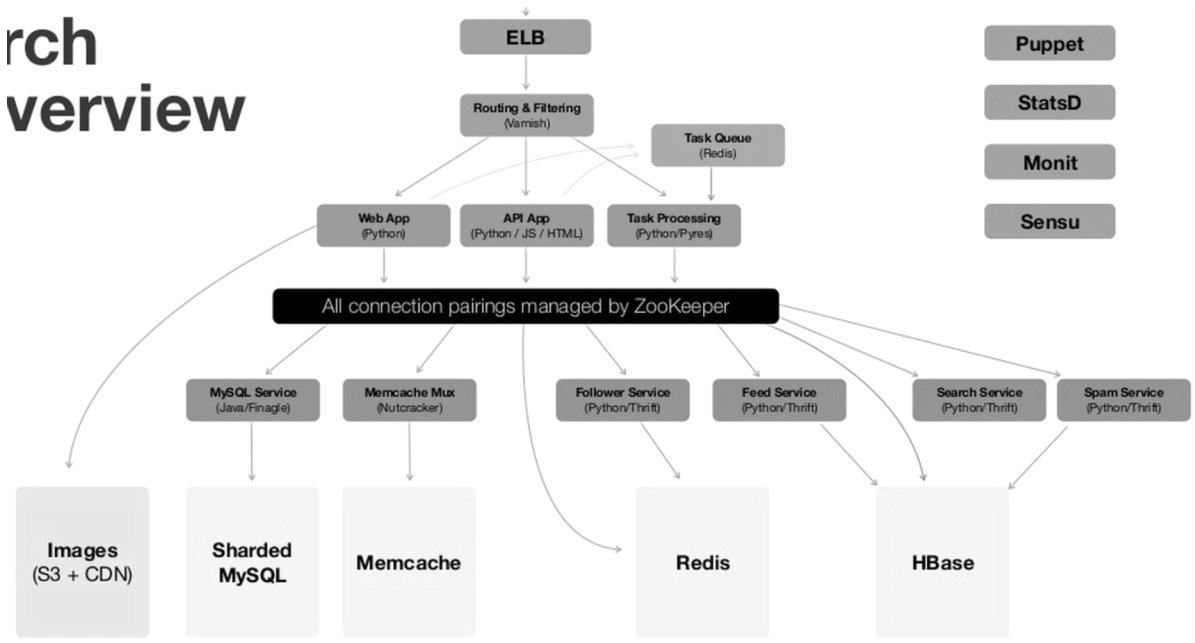
- low latency with caching
- UserProfile Service has the challenge to serve users in increasing types (driver, rider, restaurant owner, eater, etc) and user schemas in different locations

Push Notification Service

- Apple Push Notifications Service (not quite reliable)
- Google Cloud Messaging Service GCM （it can detect the deliverability) or
- SMS service is usually more reliable

```
Question: Why hybrid of MySQL, Oracle, and PostgresSQL? Why Vertica? How to set geo targeting price?
```

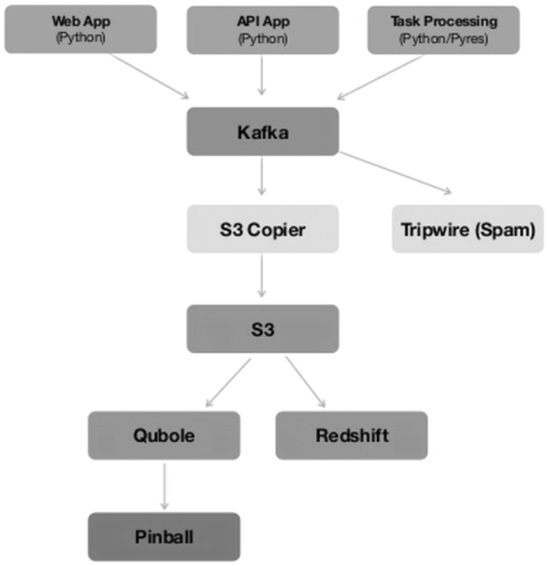
Design Spotify top 10 playing music 区别于 trending shares 某一时间段最多的，这里的问题是某一刻正在播放的最多的	 - minHeap size of 10, Hashtable -> KV storage - 用户点击播放或者暂停或者下一首后，这个更新POST到服务器之后服务器应该做什么，这里可以用到消息队列，把更新TOP10和数据库的功能交给worker去做，服务器只负责传递信息。 - worker处理完信息发现需要更新TOP10之后，如何通知服务器并且在客户端那边更新。这里就是一个pub / sub的概念，同时需要有socket池来通知客户端去更新（Python的话就Socket.IO咯）。 - 如何减少操作量，比如把一些server的操作移到worker，增加一些条件检查不用每次把整个流程跑一遍（可以提前结束或者跳过某些部分）。 - Redis is single threaded.线程安全和原子操作性（Redis INCR and SET are atomic）。如果不是redis，如何保证多个worker接到同一个歌的update能正确操作。（ReadWriteLock, RCU锁之类的）。 - worker挂了怎么办，server挂了怎么办，存heap和table的服务器挂了怎么办，整体的瓶颈可能是哪里（大量人听同一首歌，会导致一个歌不停被update + 1，这个其实可以aggregate起来），如何优化（就增加一个aggregator），这个优化的服务器挂了怎么办。大致来说都是用备份服务器，然后备份里放一段时间内的起始备份和一个追加操作的log，类似GFS那样。 - 假如我们不追求performance，只追求用户体验，上面提出来的方案应该如何优化。这里就是让备份服务器牺牲空间和效率来定期完整备份主服务器，主服务器挂了直接换备份的上。所以要增加一台Master服务器来管理这些，然后Master服务器本身又要备份。。。就看你愿意说多细了 - 有哪些现存技术可以直接实现这些东西，改进的可能性，这里就是show你的buzz word储量了。
Estimate number of machines	Estimate hardware cost of the server hardware needed to build a photo sharing app used by every person on the earth. The cost is a function of CPU, RAM, storage, bandwidth, number and size of the images uploaded each day. N users 10^10 i images / (user * day) 10 s size in bytes / image 10^6 viewed v times / image 100 d days h requests / sec 10^4 b bandwidth Server cost: $1000 / server Storage cost: $0.1 / GB Storage = Nisd Number of required servers = max(Niv/h, Nivs/b) This estimation leaves out many costs, such as electricity, cooling, network. It also neglects computations such as computing trending data and spam analysis. There is no measure of the cost of redundancy, such as replicated storage, or the ability to handle nonuniform loads. Nevertheless, it is a decent starting point.
Design Pinterest/Instagram	## Architecture



亮点有两处：一是 web tier 直接联 Images (S3 + CDN)，二是 ZooKeeper 管理 Services zookeeper watcher

Web app = NginX + Python, JS, HTML and API

Data Pipeline

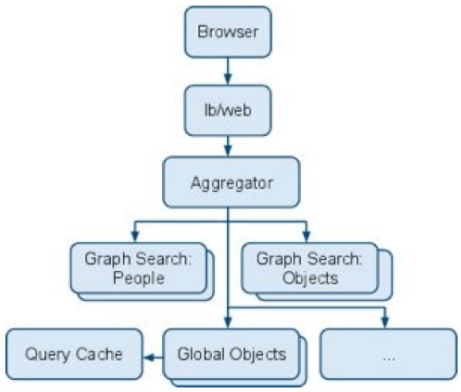


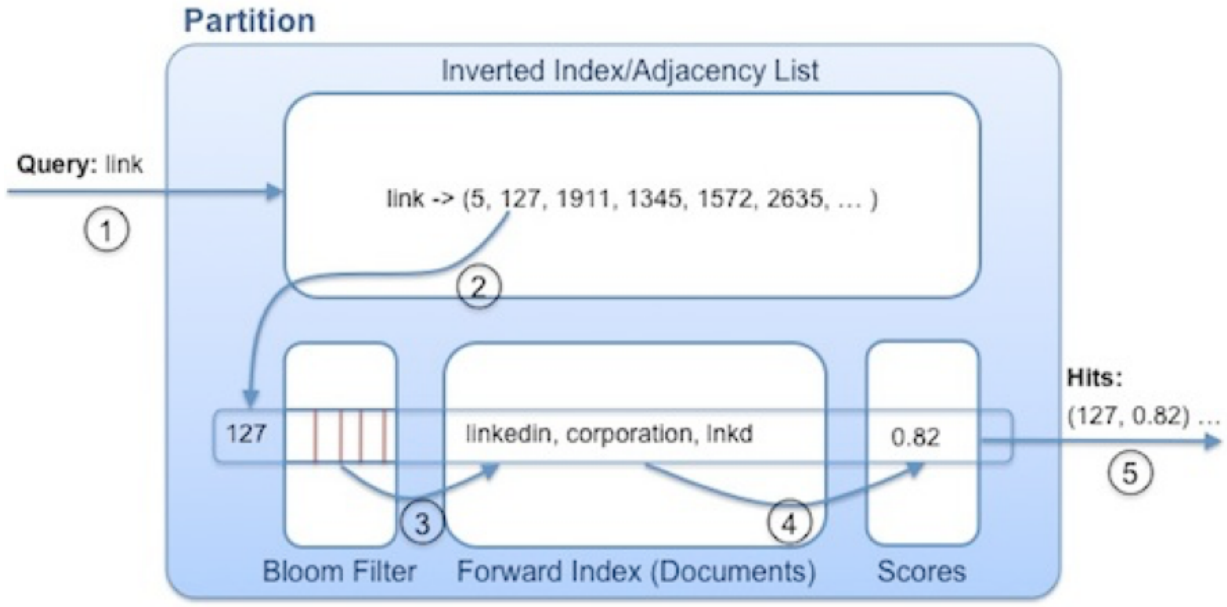
Tradeoffs

Why Amazon EC2/S3?
· Very good reliability, reporting, and support
· Very good peripherals, such as managed cache,
DB, load balancing, DNS, map reduce, and more...
· New instances ready in seconds
Scaling Pinterest
· Con: Limited choice
· Pro: Limited choice

Why Memcache?
· Extremely mature
· Very good performance
· Well known and well liked
· Never crashes, and few failure modes
· Free

Why Redis?

	· Variety of convenient data structures · Has persistence and replication · Well known and well liked · Consistently good performance · Few failure modes · Free [MySQL Clustering and Sharding]
Yelp	似乎没有现成资料，我猜 ELB --> Routing & Filtering (Varnish) --> (WebApp, WebApi, TaskProcessing) All connection parings managed by ZooKeeper Services: Follower(Redis), Feed(HBase), Search, Spam(HBase) Storage: Blob存S3，内容多有CDN
Typeahead Search aka autocomplete	1. As soon as you focus the search box, before you've even typed anything, we query to retrieve your *first-degree* connections: those people and entities (pages, apps, events, places, etc.) that you have an explicit, structured relationship with on the site. By the time you type the first character of your search, your web browser will have a few decent guesses client-side. 2. As you start typing, we send your partial query to a complex back-end set of services that find more distant objects in the social graph that match your query. If you're curious about how this back-end works, I did a tech talk on it back after we launched[2]. [1] https://www.facebook.com/notes/f... [2] https://www.facebook.com/video/v... Architecture 是典型的 multi tier 系统  Architecture of Typeahead Search 对单个service： 原始的搜索数据本身是存成 <term, ids> inverted index 的。以搜索 Companies service 为例子。 对每个 document 预处理生成 8 byte bloom filter，hash 的 key是 document 中的每个 term；类似的生成 forward index <id, terms>。 对每个可能的 prefix 生成 <prefix, ids> 首先根据 prefix 能拿到一堆结果，aggregator 根据情况排序返回部分结果。 随着prefix 增多，变成完整的 term，这时候 bloom filter 和 forward index 就能够起 filter 没有的 term 的 document

<table>
<tr><td></td><td> 
</td></tr>
<tr><td>chat application</td><td>http://stackoverflow.com/questions/10028770/html5-websocket-vs-long-polling-vs-ajax-vs-webrtc-vs-server-sent-events</td></tr>
<tr><td>Shorten URL

Design a system to take user-provided URLs and transform them to a shortened URLs that redirect back to original. Describe how the system works. How would you allocate the shorthand URLs? How would you store the shorthand to original URL mapping? How would you implement the redirect servers? Keep click stats. How would you store the stats?

Assumptions: I generally don't include these assumptions in the initial problem presentation. Good</td><td>## Assumptions

1B new urls per day, 100B entries in total
the shorter, the better
show statics (real-time and daily/monthly/yearly)

Encode Url:
http://blog.codinghorror.com/url-shortening-hashes-in-practice/

Choice 1. md5(128 bit, 16 hex numbers, collision, birthday paradox, 2^(n/2) = 2^64) truncate? (64bit, 8 hex number, collision 2^32), Base64.

+: hash is simple and horizontally scalable.
-: too long, how to purify expired urls?

Choice 2. Distributed Seq Id Generator.(Base62: a~z, A~Z, 0~9, 62 chars, 62^7), sharding: each node maintains a section of ids.

+: easy to outdate expired entries, shorter
-: coordination (zookeeper)

KV store

MySQL(10k qps, slow, no relation), KV (100k qps, Redis, Memchached)

A great candidate will ask about the lifespan of the aliases and design a system that purges aliases past their expiration

Redirect Service

Apache, Nginx, Jetty etc. 302 Found (301 Moved Permanently 服务器压力会小点，但是没法统计点击次数)

Statics

Hit counter, ??? messaging system to buffer click data, aggregation layer

Simple Single Node Implementation

var base = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
var baseSize = base.length;
var digitNum = 3;
var idSize = Math.pow(baseSize, digitNum);
console.log(idSize);</td></tr>
</table>

candidates will ask about scale when coming up with a design.
- Total number of unique domains registering redirect URLs is on the order of 10s of thousands
- New URL registrations are on the order of 10,000,000/day (100/sec)
- Redirect requests are on the order of 10B/day (100,000/sec)
- Remind candidates that those are average numbers - during peak traffic (either driven by time, such as 'as people come home from work' or by outside events, such as 'during the superbowl') they may be much higher.
- Recent stats (within the current day) should be aggregated and available with a 5 minute lag time
- Long look-back stats can be computed daily

```
var idCur = 0;
var urlToId = {};
var idToUrl = {};
var urlBase = "http://bit.ly/";

var getShortenUrl = function(url) {
  return urlBase + getId(url);
};

var getId = function(url) {
  if (urlToId[url] != undefined) {
    return urlToId[url];
  }

  var encodedId = baseEncode(idCur);
  urlToId[url] = encodedId;
  idToUrl[encodedId] = url;

  idCur++;
  if (idCur >= idSize) idCur = 0;

  var encodedCurId = baseEncode(idCur);
  if (idToUrl[encodedCurId] != undefined) {
    var curUrl = idToUrl[encodedCurId];
    delete idToUrl[encodedCurId];
    delete urlToId[curUrl];
  }

  return encodedId;
};

var getOriginUrl = function(shortenUrl) {
  var id = shortenUrl.substring(urlBase.length);
  return idToUrl[id];
};

var baseEncode = function(id) {
  if (id === 0) return "a";
  var rst = "";
  while (id > 0) {
    rst = base[id % baseSize] + rst;
    id = Math.floor(id / baseSize);
  }
  return rst;
};


var shortens = [];
for (var i = 0; i < 238328; i++) {
  shortens[i] = getShortenUrl("https://bitly.com/shorten/" + i);
//  console.log(shortens[i]);
}


var urls = [];
for (var i = 0; i < 238328; i++) {
  urls[i] = getOriginUrl(shortens[i]);
//  console.log(urls[i]);
}

console.log(getOriginUrl(shortens[238327]));

var expectedUrl =
"https://www.evernote.com/shard/s75/sh/40ba3b04-e8d1-4508-8aea-3a1e2000ba8c/91e62d61f4fda6f056ea1900bd
cee9de/res/3915a501-e697-430b-877a-b5a06a167abc/%E6%BB%B4%E6%BB%B4%E6%89%93%E8%BD%A6%E6%9E%B6%E6%9E%84
%E6%BC%94%E5%8F%98%E5%8F%8A%E5%BA%94%E7%94%A8%E5%AE%9E%E8%B7%B5.pdf";
var bitly = getShortenUrl(expectedUrl);

## Followup TODO

怎 么盈利，怎么避免为黄色网址服务，怎么得到实时logging metrics，etc。
```

	Q: How will shortened URLs be generated? A poor candidate will propose a solution that uses a single id generator (single point of failure) or a solution that requires coordination among id generator servers on every request. For example a single database server using an auto-increment primary key. An acceptable candidate will propose a solution using an md5 of the URL, or some form of UUID generator that can be done independently on any node. While this allows distributed generation of non-colliding IDs, it yields large "shortened" URLs A good candidate will design a solution that utilizes a cluster of id generators that reserve chunks of the id space from a central coordinator (e.g. ZooKeeper) and independently allocate IDs from their chunk, refreshing as necessary. Q: How to store the mappings? A poor candidate will suggest monolithic database. There are no relational aspects to this store. It is a pure key-value store. A good candidate will propose using any light-weight, dis e store. MongoDB/hbase/Voldemort/etc. A great candidate will ask about the lifespan of the aliases and design a system that purges aliases past their expiration Q: How to implement the redirect servers? A poor candidate will start designing something from scratch to solve an already solved problem A good candidate will propose using an off-the-shelf HTTP server with a plug-in that parses the shortened URL key, looks the alias up in the DB, updates click stats and returns a 303 back to the original URL. Apache/Jetty/Netty/tomcat/etc. are all fine. Q: How are click stats stored? A poor candidate will suggest write-back to a data store on every click A good candidate will suggest some form of aggregation tier that accepts clickstream data, aggregates it, and writes back a persistent data store periodically Q: How will the aggregation tier be partitioned? A great candidate will suggest a low-latency messaging system to buffer the click data and transfer it to the aggregation tier. A candidate may ask how often the stats need to be updated. If daily, storing in hdfs and running map/reduce jobs to compute stats is a reasonable approach If near real-time, the aggregation logic should compute stats
Newsfeed news feed 整个流程（twitter，facebook类似但不一样，不一样在哪里），是用pull还是push，是为每个用户保存一个queue吗？new year时high traffic会出现哪些 特殊情况，怎么解决？	TODO

Facebook

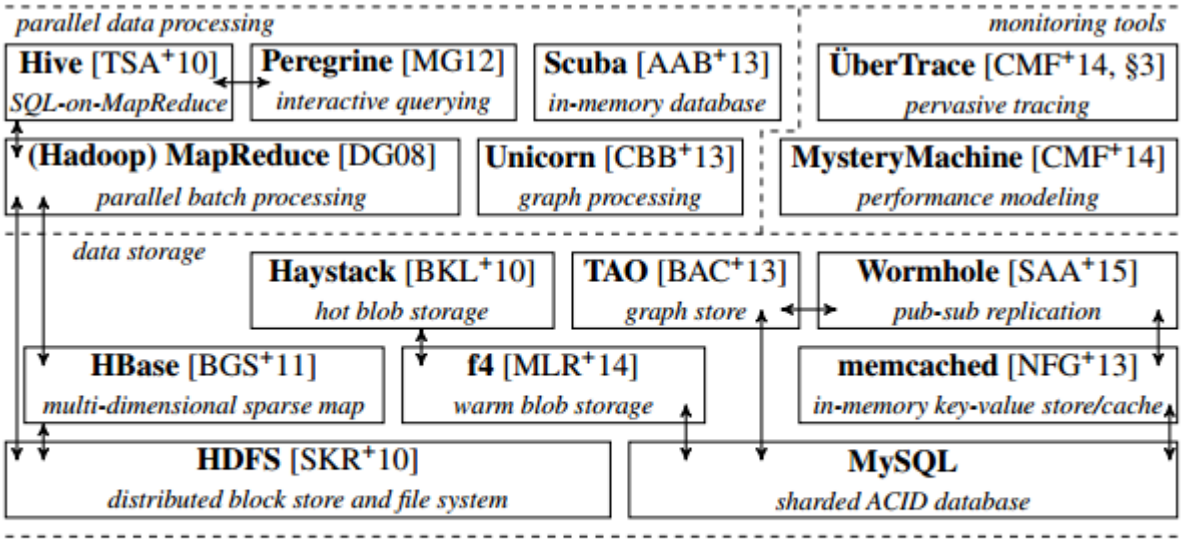


Figure 1: The Facebook infrastructure stack. I omit front-end serving systems about which details are unknown. Arrows indicate data exchange and dependencies between systems; simple layering does *not* imply a dependency or relation.

https://www.quora.com/What-is-Facebooks-architecture

Amazon Shopping Cart

facebook photo storage

Motivation & Assumptions

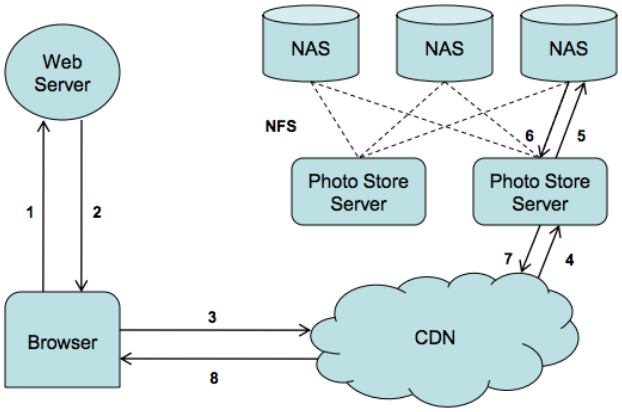
- PB-level Blob storage
- Traditional NFS based desgin (Each image stored as a file) has metadata bottleneck: large metadata size severely limits the metadata hit ratio.
    - Explain more about the metadata overhead

For the Photos application most of this metadata, such as permissions, is unused and thereby wastes storage capacity. Yet the more significant cost is that the file's metadata must be read from disk into memory in order to find the file itself. While insignificant on a small scale, multiplied over billions of photos and petabytes of data, accessing metadata is the throughput bottleneck.

Solution

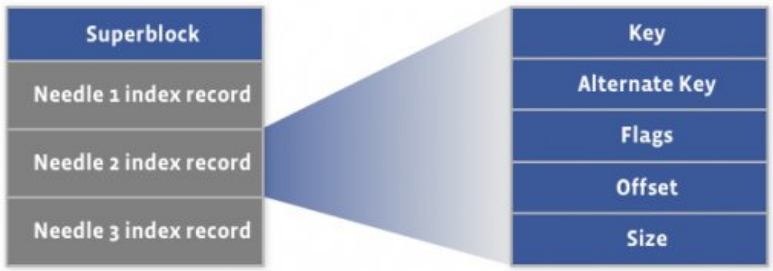
Eliminates the metadata overhead by aggregating hundreds of thousands of images in a single haystack store file.

Architecture

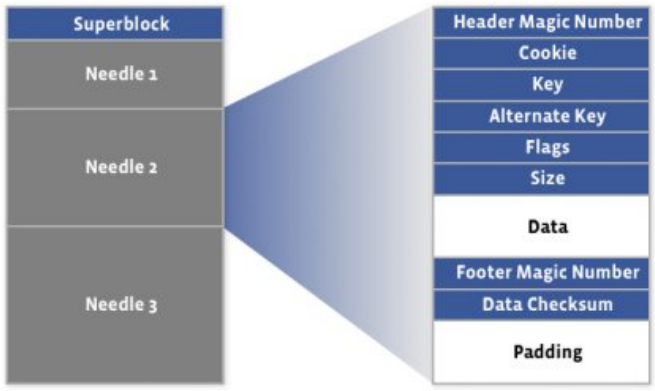


Data Layout

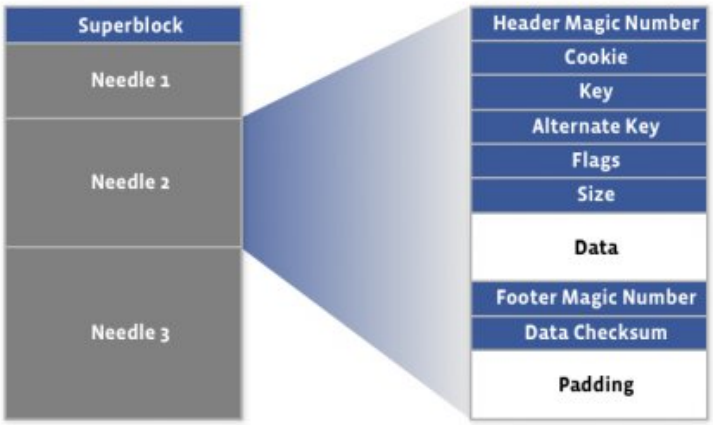
index file (for quick memory load) + haystack store file containing needles.



index file layout



haystack store file



CRUD Operations

C: write to store file and then async write index file, because index is not critical

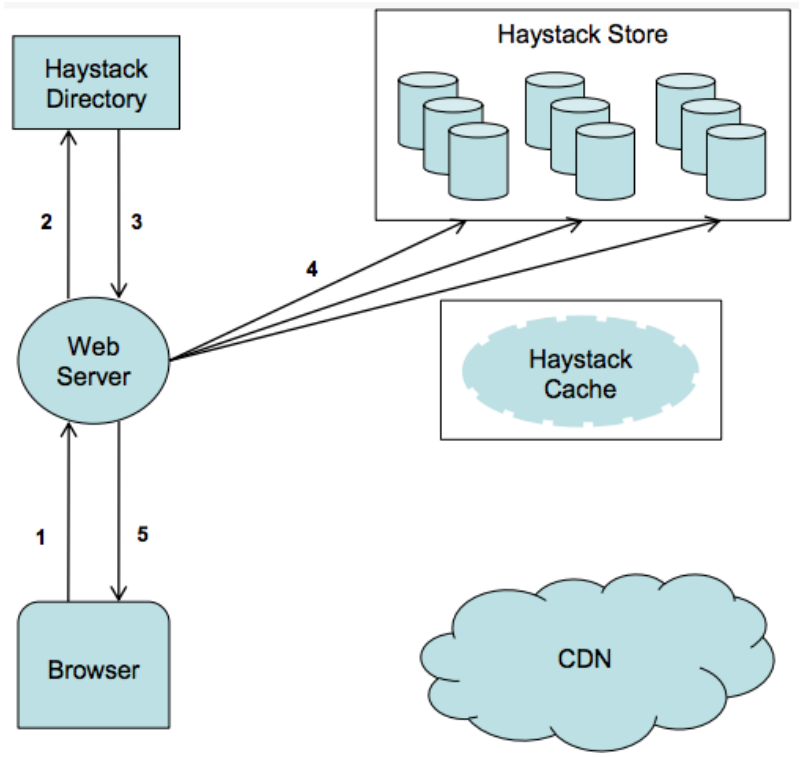
R: read(offset, key, alternate_key, cookie, data_size)

U: Append only, duplicate keys and app can choose one with largest offset.

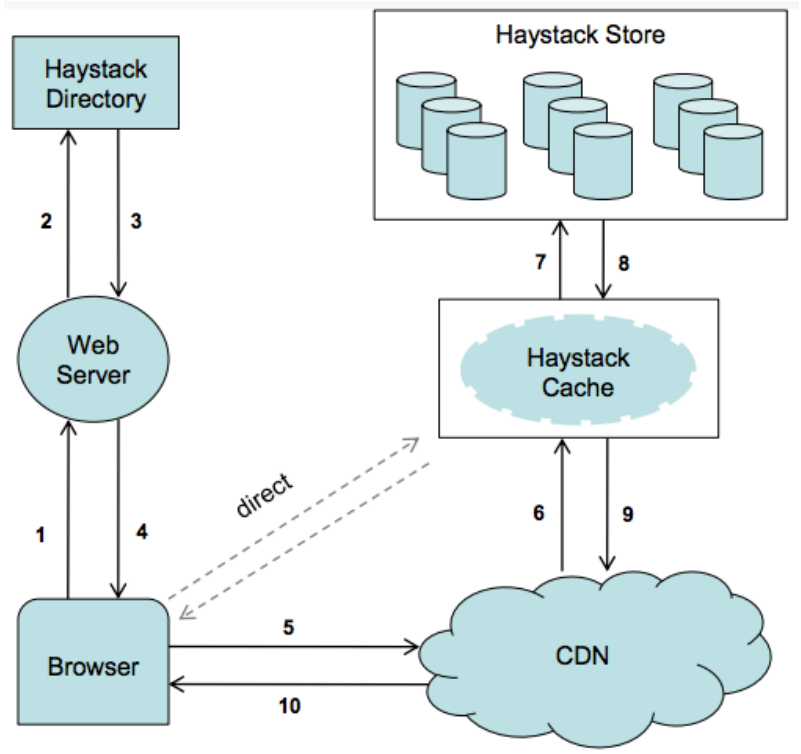
D: soft delete by marking the deleted bit in the flag field. Hard delete is executed by the compact operation.

Usecases

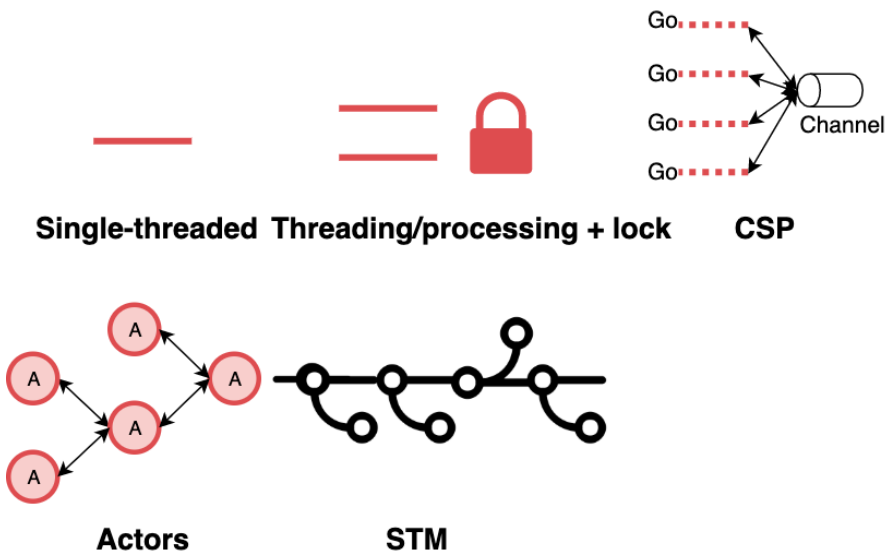
Upload



Download

Concurrency

RCU	read-copy update sometimes be used as an alternative to a readers-writer lock
Concurrency Models	 * Single-threaded - Callbacks, Promises, Observables and async/await: vanilla JS * threading/multiprocessing, lock-based concurrency * protecting critical section vs. performance * Communicating Sequential Processes (CSP) * Golang or Clojure's `core.async`. * process/thread passes data through channels. * Actor Model (AM): Elixir, Erlang, Scala * asynchronous by nature, and have location transparency that spans runtimes and machines - if you have a reference (Akka) or PID (Erlang) of an actor, you can message it via mailboxes. * powerful fault tolerance by organizing actors into a supervision hierarchy, and you can handle failures at its exact level of hierarchy.

	* Software Transactional Memory (STM): Clojure, Haskell * like MVCC or pure functions: commit / abort / retry

Tradeoffs

Read vs. Write	“写”比“读”慢40倍左右，而“读”在一般的互联网产品中，比如twitter，读写比是100:1到1000:1

100 open source Big Data architecture papers for data professionals.

Jun 18, 2015

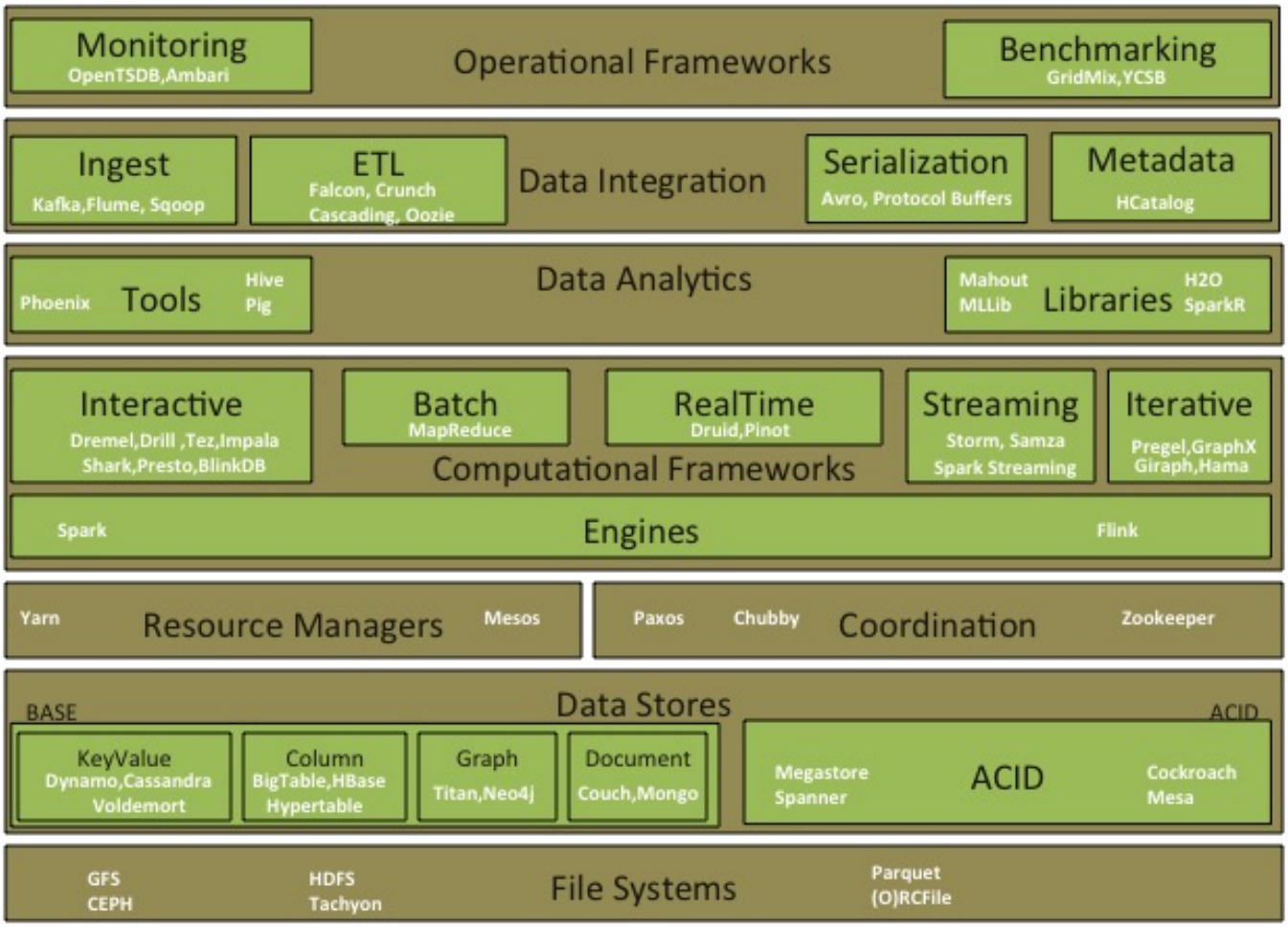
- 17,232views
  - 555Likes
  - 57Comments
- Share on LinkedIn
  - Share on Facebook
  - Share on Google Plus
  - Share on Twitter

Big Data technology has been extremely disruptive with open source playing a dominant role in shaping its evolution. While on one hand it has been disruptive, on the other it has led to a complex ecosystem where new frameworks, libraries and tools are being released pretty much every day, creating confusion as technologists struggle and grapple with the deluge.

If you are a Big Data enthusiast or a technologist ramping up (or scratching your head), it is important to spend some serious time deeply understanding the architecture of key systems to appreciate its evolution. Understanding the architectural components and subtleties would also help you choose and apply the appropriate technology for your use case. In my journey over the last few years, some literature has helped me become a better educated data professional. My goal here is to not only share the literature but consequently also use the opportunity to put some sanity into the labyrinth of open source systems.

One caution, most of the reference literature included is hugely skewed towards deep architecture overview (in most cases original research papers) than simply provide you with basic overview. I firmly believe that deep dive will fundamentally help you understand the nuances, though would not provide you with any shortcuts, if you want to get a quick basic overview.

Jumping right in…



Key architecture layers

- **File Systems**- Distributed file systems which provide storage, fault tolerance, scalability, reliability, and availability.
- **Data Stores**– Evolution of application databases into Polyglot storage with application specific databases instead of one size fits all. Common ones are Key-Value, Document, Column and Graph.
- **Resource Managers**– provide resource management capabilities and support schedulers for high utilization and throughput.
- **Coordination**– systems that manage state, distributed coordination, consensus and lock management.
- **Computational Frameworks**– a lot of work is happening at this layer with highly specialized compute frameworks for Streaming, Interactive, Real Time, Batch and Iterative Graph (BSP) processing. Powering these are complete computation runtimes like BDAS (Spark) & Flink.
- **DataAnalytics** –Analytical (consumption) tools and libraries, which support exploratory, descriptive, predictive, statistical analysis and machine learning.
- **Data Integration**– these include not only the orchestration tools for managing pipelines but also metadata management.
- **Operational Frameworks** – these provide scalable frameworks for monitoring & benchmarking.

Architecture Evolution

The modern data architecture is evolving with a goal of reduced latency between data producers and consumers. This consequently is leading to real time and low latency processing, bridging the traditional batch and interactive layers into hybrid architectures like Lambda and Kappa.

- Lambda - Established architecture for a typical data pipeline. More details.
- Kappa– An alternative architecture which moves the processing upstream to the Stream layer.
- SummingBird– a reference model on bridging the online and traditional processing models.

Before you deep dive into the actual layers, here are some general documents which can provide you a great background on NoSQL, Data Warehouse Scale Computing and Distributed Systems.

- Data center as a computer– provides a great background on warehouse scale computing.
- NOSQL Data Stores– background on a diverse set of key-value, document and column oriented stores.
- NoSQL Thesis– great background on distributed systems, first generation NoSQL systems.
- Large Scale Data Management- covers the data model, the system architecture and the consistency model, ranging from traditional database vendors to new emerging internet-based enterprises.
- Eventual Consistency– background on the different consistency models for distributed systems.
- CAP Theorem– a nice background on CAP and its evolution.

There also has been in the past a fierce debate between traditional Parallel DBMS with Map Reduce paradigm of processing. Pro parallel DBMS (another) paper(s) was rebutted by the pro MapReduce one. Ironically the Hadoop community from then has come full circle with the introduction of MPI style shared nothing based processing on Hadoop - SQL on Hadoop.

File Systems

As the focus shifts to low latency processing, there is a shift from traditional disk based storage file systems to an emergence of in memory file systems - which drastically reduces the I/O & disk serialization cost. Tachyon and Spark RDD are examples of that evolution.

- Google File System- The seminal work on Distributed File Systems which shaped the Hadoop File System.
- Hadoop File System– Historical context/architecture on evolution of HDFS.
- Ceph File System– An alternative to HDFS.
- Tachyon– An in memory storage system to handle the modern day low latency data processing.

File Systems have also seen an evolution on the file formats and compression techniques. The following references gives you a great background on the merits of row and column formats and the shift towards newer nested column oriented formats which are highly efficient for Big Data processing. Erasure codes are using some innovative techniques to reduce the triplication (3 replicas) schemes without compromising data recoverability and availability.

- Column Oriented vs Row-Stores– good overview of data layout, compression and materialization.
- RCFile– Hybrid PAX structure which takes the best of both the column and row oriented stores.
- Parquet– column oriented format first covered in Google's Dremel's paper.
- ORCFile– an improved column oriented format used by Hive.
- Compression– compression techniques and their comparison on the Hadoop ecosystem.
- Erasure Codes– background on erasure codes and techniques; improvement on the default triplication on Hadoop to reduce storage cost.

Data Stores

Broadly, the distributed data stores are classified on ACID & BASE stores depending on the continuum of strong to weak consistency respectively. BASE further is classified into KeyValue, Document, Column and Graph - depending on the underlying schema & supported data structure. While there are multitude of systems and offerings in this space, I have covered few of the more prominent ones. I apologize if I have missed a significant one...

BASE

Key Value Stores

Dynamo – key-value distributed storage system

Cassandra – Inspired by Dynamo; a multi-dimensional key-value/column oriented data store.

Voldemort – another one inspired by Dynamo, developed at LinkedIn.

Column Oriented Stores

BigTable – seminal paper from Google on distributed column oriented data stores.

HBase – while there is no definitive paper , this provides a good overview of the technology.

Hypertable – provides a good overview of the architecture.

Document Oriented Stores

CouchDB – a popular document oriented data store.

MongoDB – a good introduction to MongoDB architecture.

Graph

Neo4j – most popular Graph database.

Titan – open source Graph database under the Apache license.

ACID

I see a lot of evolution happening in the open source community which will try and catch up with what Google has done – 3 out of the prominent papers below are from Google , they have solved the globally distributed consistent data store problem.

Megastore – a highly available distributed consistent database. Uses Bigtable as its storage subsystem.

Spanner – Globally distributed synchronously replicated linearizable database which supports SQL access.

MESA – provides consistency, high availability, reliability, fault tolerance and scalability for large data and query volumes.

CockroachDB – An open source version of Spanner (led by former engineers) in active development.

Resource Managers

While the first generation of Hadoop ecosystem started with monolithic schedulers like YARN, the evolution now is towards hierarchical schedulers (Mesos), that can manage distinct workloads, across different kind of compute workloads, to achieve higher utilization and efficiency.

YARN – The next generation Hadoop compute framework.

Mesos – scheduling between multiple diverse cluster computing frameworks.

These are loosely coupled with schedulers whose primary function is schedule jobs based on scheduling policies/configuration.

Schedulers

Capacity Scheduler - introduction to different features of capacity scheduler.

FairShare Scheduler - introduction to different features of fair scheduler.

Delayed Scheduling - introduction to Delayed Scheduling for FairShare scheduler.

Fair & Capacity schedulers – a survey of Hadoop schedulers.

Coordination

These are systems that are used for coordination and state management across distributed data systems.

Paxos – a simple version of the classical paper; used for distributed systems consensus and coordination.

Chubby – Google's distributed locking service that implements Paxos.

Zookeeper – open source version inspired from Chubby though is general coordination service than simply a locking service

Computational Frameworks

The execution runtimes provide an environment for running distinct kinds of compute. The most common runtimes are

Spark – its popularity and adoption is challenging the traditional Hadoop ecosystem.

Flink – very similar to Spark ecosystem; strength over Spark is in iterative processing.

The frameworks broadly can be classified based on the model and latency of processing

Batch

MapReduce – The seminal paper from Google on MapReduce.

MapReduce Survey – A dated, yet a good paper; survey of Map Reduce frameworks.

Iterative (BSP)

Pregel – Google's paper on large scale graph processing

Giraph - large-scale distributed Graph processing system modelled around Pregel

GraphX - graph computation framework that unifies graph-parallel and data parallel computation.

Hama - general BSP computing engine on top of Hadoop

Open source graph processing survey of open source systems modelled around Pregel BSP.

Streaming

Stream Processing – A great overview of the distinct real time processing systems

Storm – Real time big data processing system

Samza - stream processing framework from LinkedIn

Spark Streaming – introduced the micro batch architecture bridging the traditional batch and interactive processing.

Interactive

Dremel – Google's paper on how it processes interactive big data workloads, which laid the groundwork for multiple open source SQL systems on Hadoop.

Impala – MPI style processing on make Hadoop performant for interactive workloads.

Drill – A open source implementation of Dremel.

Shark – provides a good introduction to the data analysis capabilities on the Spark ecosystem.

Shark – another great paper which goes deeper into SQL access.

Dryad – Configuring & executing parallel data pipelines using DAG.

Tez – open source implementation of Dryad using YARN.

BlinkDB - enabling interactive queries over data samples and presenting results annotated with meaningful error bars

RealTime

Druid – a real time OLAP data store. Operationalized time series analytics databases

Pinot – LinkedIn OLAP data store very similar to Druid.

Data Analysis

The analysis tools range from declarative languages like SQL to procedural languages like Pig. Libraries on the other hand are supporting out of the box implementations of the most common data mining and machine learning libraries.

Tools

Pig – Provides a good overview of Pig Latin.

Pig – provide an introduction of how to build data pipelines using Pig.

Hive – provides an introduction of Hive.

Hive – another good paper to understand the motivations behind Hive at Facebook.

Phoenix – SQL on Hbase.

Join Algorithms for Map Reduce – provides a great introduction to different join algorithms on Hadoop.

Join Algorithms for Map Reduce – another great paper on the different join techniques.

Libraires

MLlib – Machine language framework on Spark.

SparkR – Distributed R on Spark framework.

Mahout – Machine learning framework on traditional Map Reduce.

Data Integration

Data integration frameworks provide good mechanisms to ingest and outgest data between Big Data systems. It ranges from orchestration pipelines to metadata framework with support for lifecycle management and governance.

Ingest/Messaging

Flume – a framework for collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store.

Sqoop– a tool to move data between Hadoop and Relational data stores.

Kafka – distributed messaging system for data processing

ETL/Workflow

Crunch – library for writing, testing, and running MapReduce pipelines.

Falcon – data management framework that helps automate movement and processing of Big Data.

Cascading – data manipulation through scripting.

Oozie – a workflow scheduler system to manage Hadoop jobs.

Metadata

HCatalog - a table and storage management layer for Hadoop.

Serialization

ProtocolBuffers – language neutral serialization format popularized by Google. Avro – modeled around Protocol Buffers for the Hadoop ecosystem.

Operational Frameworks

Finally the operational frameworks provide capabilities for metrics, benchmarking and performance optimization to manage workloads.

Monitoring Frameworks

OpenTSDB – a time series metrics systems built on top of HBase.

Ambari - system for collecting, aggregating and serving Hadoop and system metrics

Benchmarking

YCSB – performance evaluation of NoSQL systems.

GridMix – provides benchmark for Hadoop workloads by running a mix of synthetic jobs

Background on big data benchmarking with the key challenges associated.

Summary

I hope that the papers are useful as you embark or strengthen your journey. I am sure there are few hundred more papers that I might have inadvertently missed and a whole bunch of systems that I might be unfamiliar with - apologies in advance as don't mean to offend anyone though happy to be educated....

1. 如何设计数据库系统，具体地，可以是设计 taobao/Facebook 或是任何公司的员工数据库等；
2. 如何设计用户系统，具体地，可以是 Netflix/Youtube 的用户系统等；

3. 如何设计支付系统，具体地，可以是 alipay 等；
4. 如何设计爬虫系统，具体地，可以是 baidu 的搜索引擎等；
5. 如何设计短网址系统，具体地，可以是新浪的短网址等；
6. 如何设计“秒杀”系统，具体地，可以是淘宝双十一系统等；
7. 如何设计 message 和 news feed 系统，具体地，可以是 facebook/人人 /微信朋友圈 /whatsapp/snapchat 等；
8. OOD 面向对象系统设计，具体地，可以是电梯问题、停车问题等；
9. 如何设计分布式文件系统，具体地，可以是 google 的文件分布系统 BFS 等；
10.