github.com

system-design-primer/README.md at master · donnemartin/system-design-primer

9-12 minutes

Design a key-value cache to save the results of the most recent web server queries

Note: This document links directly to relevant areas found in the system design topics to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

> Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** sends a search request resulting in a cache hit

- **User** sends a search request resulting in a cache miss
- **Service** has high availability

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Popular queries should almost always be in the cache
- Need to determine how to expire/refresh
- Serving from cache requires fast lookups
- Low latency between machines
- Limited memory in cache
- Need to determine what to keep/remove
- Need to cache millions of queries
- 10 million users
- 10 billion queries per month

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

- Cache stores ordered list of key: query, value: results
- `query` - 50 bytes
- `title` - 20 bytes
- `snippet` - 200 bytes

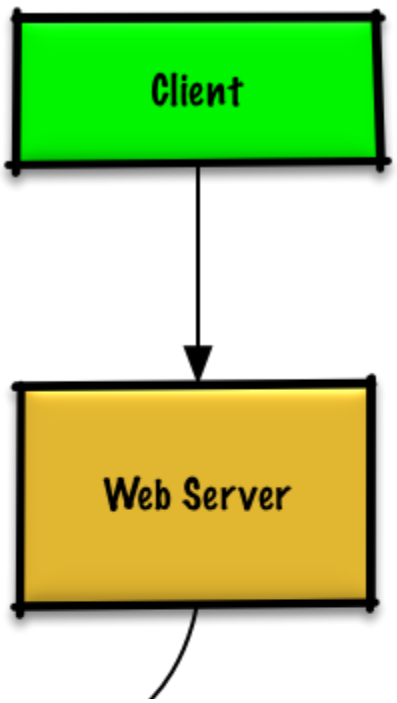
- Total: 270 bytes
- 2.7 TB of cache data per month if all 10 billion queries are unique and all are stored
- 270 bytes per search * 10 billion searches per month
- Assumptions state limited memory, need to determine how to expire contents
- 4,000 requests per second

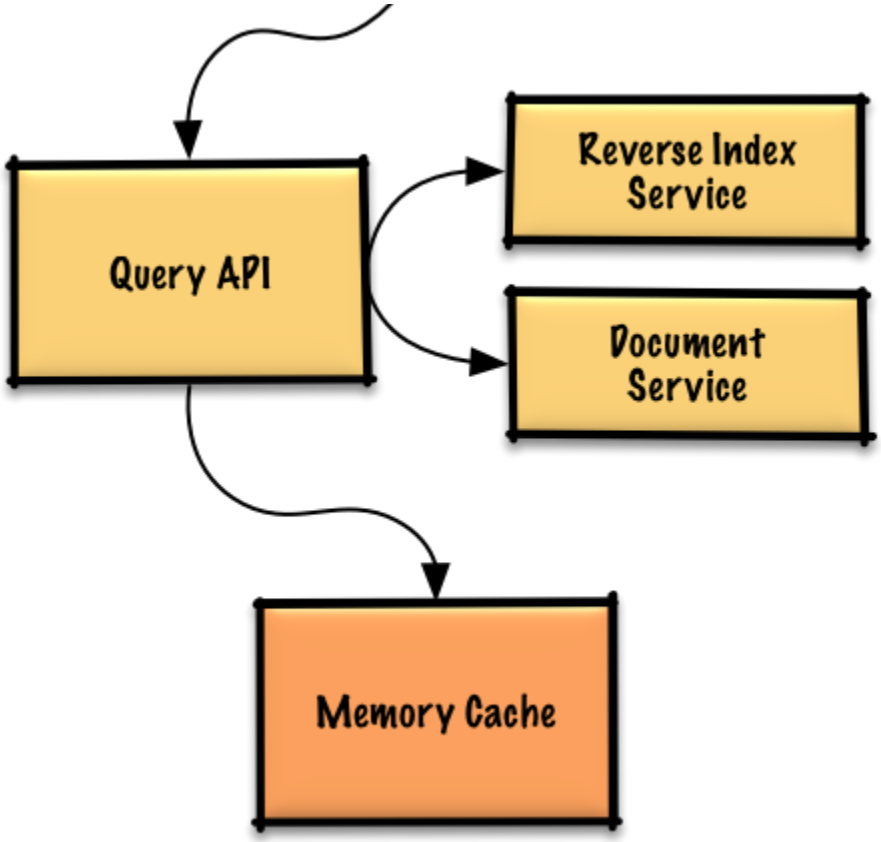
Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.





Step 3: Design core components

Dive into details for each core component.

Use case: User sends a request resulting in a cache hit

Popular queries can be served from a **Memory Cache** such as Redis or Memcached to reduce read latency and to avoid overloading the **Reverse Index Service** and **Document Service**. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.[1]

Since the cache has limited capacity, we'll use a least recently used (LRU) approach to expire older entries.

- The **Client** sends a request to the **Web Server**, running as a

reverse proxy

- The **Web Server** forwards the request to the **Query API** server
- The **Query API** server does the following:
- Parses the query
- Removes markup
- Breaks up the text into terms
- Fixes typos
- Normalizes capitalization
- Converts the query to use boolean operations
- Checks the **Memory Cache** for the content matching the query
- If there's a hit in the **Memory Cache**, the **Memory Cache** does the following:
- Updates the cached entry's position to the front of the LRU list
- Returns the cached contents
- Else, the **Query API** does the following:
- Uses the **Reverse Index Service** to find documents matching the query
- The **Reverse Index Service** ranks the matching results and returns the top ones
- Uses the **Document Service** to return titles and snippets
- Updates the **Memory Cache** with the contents, placing the entry at the front of the LRU list

Cache implementation

The cache can use a doubly-linked list: new items will be added to

the head while items to expire will be removed from the tail. We'll use a hash table for fast lookups to each linked list node.

Clarify with your interviewer how much code you are expected to write.

Query API Server implementation:

```
class QueryApi(object):

    def __init__(self, memory_cache, reverse_index_service):
        self.memory_cache = memory_cache
        self.reverse_index_service = reverse_index_service

    def parse_query(self, query):
        """Remove markup, break text into terms, deal with typos,
        normalize capitalization, convert to use boolean operations.
        """
        ...

    def process_query(self, query):
        query = self.parse_query(query)
        results = self.memory_cache.get(query)
        if results is None:
            results =
self.reverse_index_service.process_search(query)
            self.memory_cache.set(query, results)
        return results
```

Node implementation:

```
class Node(object):
```

```
    def __init__(self, query, results):
        self.query = query
        self.results = results
```

LinkedList implementation:

```
class LinkedList(object):

    def __init__(self):
        self.head = None
        self.tail = None

    def move_to_front(self, node):
        ...

    def append_to_front(self, node):
        ...

    def remove_from_tail(self):
        ...
```

Cache implementation:

```
class Cache(object):

    def __init__(self, MAX_SIZE):
        self.MAX_SIZE = MAX_SIZE
        self.size = 0
        self.lookup = {}  # key: query, value: node
        self.linked_list = LinkedList()

    def get(self, query)
```

```
        """Get the stored query result from the cache.

        Accessing a node updates its position to the front of the LRU
list.
        """
        node = self.lookup[query]
        if node is None:
            return None
        self.linked_list.move_to_front(node)
        return node.results

    def set(self, results, query):
        """Set the result for the given query key in the cache.

        When updating an entry, updates its position to the front of
the LRU list.
        If the entry is new and the cache is at capacity, removes the
oldest entry
        before the new entry is added.
        """
        node = self.lookup[query]
        if node is not None:
            # Key exists in cache, update the value
            node.results = results
            self.linked_list.move_to_front(node)
        else:
            # Key does not exist in cache
            if self.size == self.MAX_SIZE:
                # Remove the oldest entry from the linked list and
lookup
```

```
            self.lookup.pop(self.linked_list.tail.query, None)
            self.linked_list.remove_from_tail()
        else:
            self.size += 1
        # Add the new key and value
        new_node = Node(query, results)
        self.linked_list.append_to_front(new_node)
        self.lookup[query] = new_node
```

When to update the cache

The cache should be updated when:

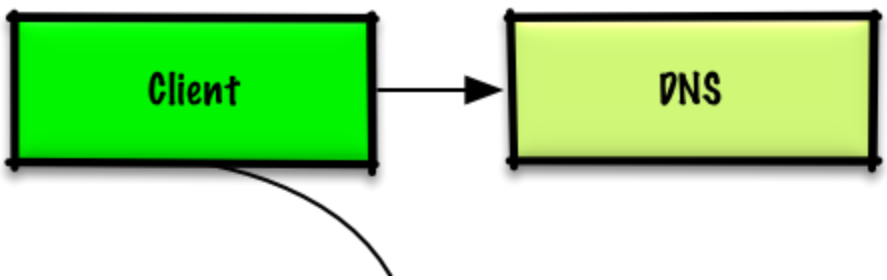
- The page contents change
- The page is removed or a new page is added
- The page rank changes

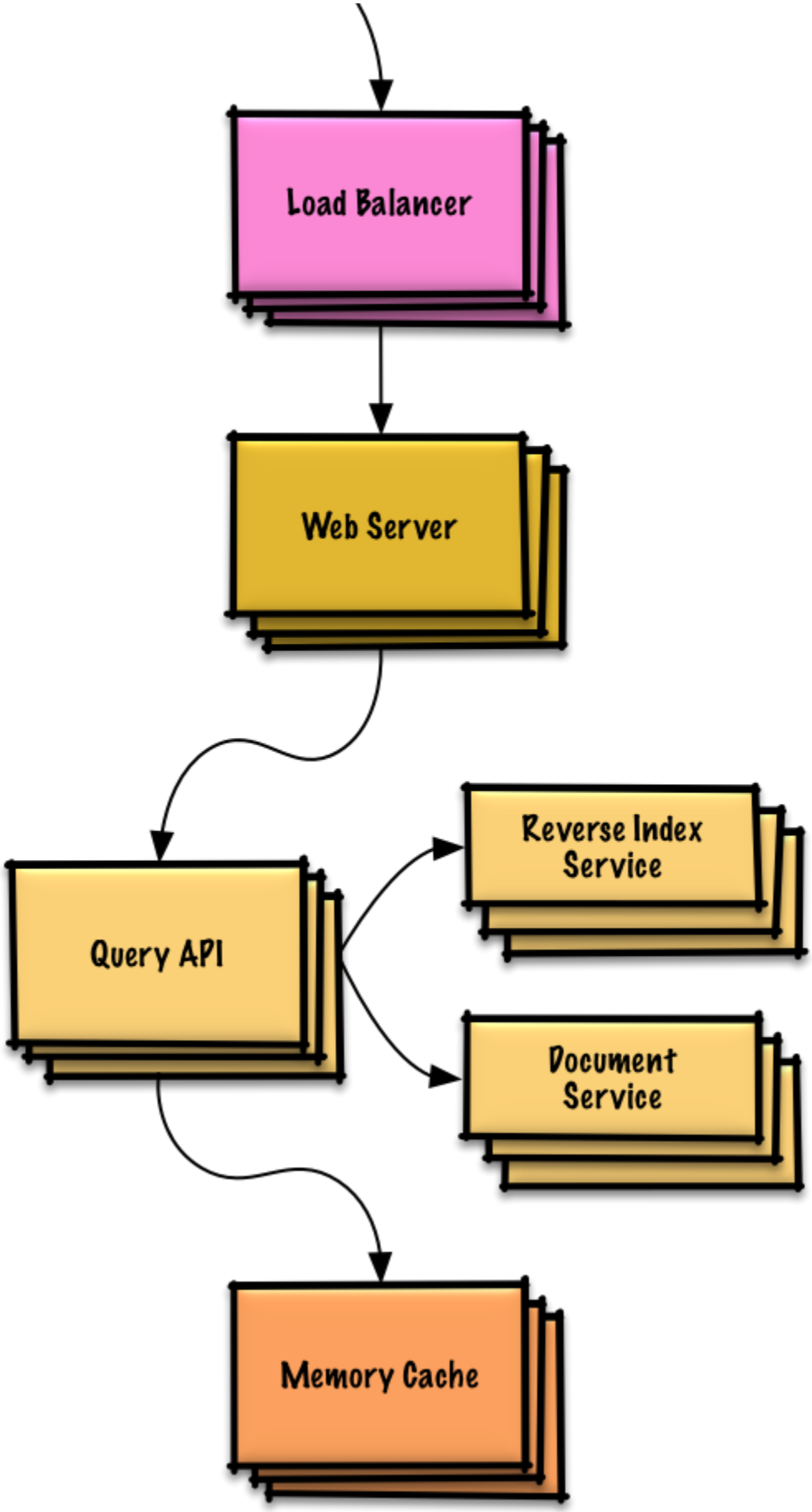
The most straightforward way to handle these cases is to simply set a max time that a cached entry can stay in the cache before it is updated, usually referred to as time to live (TTL).

Refer to When to update the cache for tradeoffs and alternatives. The approach above describes cache-aside.

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.




Load Balancer
Web Server
Query API
Reverse Index Service
Document Service
Memory Cache

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See Design a system that scales to millions of users on AWS as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers**? **CDN**? **Master-Slave Replicas**? What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following system design topics for main talking points, tradeoffs, and alternatives:

- DNS
- Load balancer
- Horizontal scaling
- Web server (reverse proxy)
- API server (application layer)
- Cache
- Consistency patterns
- Availability patterns

Expanding the Memory Cache to many machines

To handle the heavy request load and the large amount of memory needed, we'll scale horizontally. We have three main options on how to store the data on our **Memory Cache** cluster:

- **Each machine in the cache cluster has its own cache** - Simple, although it will likely result in a low cache hit rate.
- **Each machine in the cache cluster has a copy of the cache** - Simple, although it is an inefficient use of memory.
- **The cache is** sharded **across all machines in the cache cluster** - More complex, although it is likely the best option. We could use hashing to determine which machine could have the cached results of a query using `machine = hash(query)`. We'll likely want to use consistent hashing.

Additional talking points

> Additional topics to dive into, depending on the problem scope and time remaining.

SQL scaling patterns

- Read replicas
- Federation
- Sharding
- Denormalization
- SQL Tuning

NoSQL

- Key-value store
- Document store
- Wide column store
- Graph database
- SQL vs NoSQL

Caching

- Where to cache
- Client caching
- CDN caching
- Web server caching
- Database caching
- Application caching
- What to cache
- Caching at the database query level
- Caching at the object level
- When to update the cache
- Cache-aside
- Write-through
- Write-behind (write-back)
- Refresh ahead

Asynchronism and microservices

- Message queues

- Task queues
- Back pressure
- Microservices

Communications

- Discuss tradeoffs:
- External communication with clients - HTTP APIs following REST
- Internal communications - RPC
- Service discovery

Security

Refer to the security section.

Latency numbers

See Latency numbers every programmer should know.

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process