APM

INTERVIEW KICKSTART

Stream Processing and Analytics

{ik} INTERVIEW KICKSTART

Precursor

Why Scalable Systems Design

- Looking back around 2014-2015
  - Systems Design Interviews used to be Niche
  - More Focus on Coding, Data Structures and Algorithms

- Pros and Cons
  - Pro: A common platform to evaluate any software engineering candidate
  - Con: Somewhat unaligned for experienced candidates
    - I have never had to deal with combinatorial recursions in my career

Why Scalable Systems Design

- Systems Design gives a fair play platform for experienced candidates
  - Assumption is that experienced candidates have been involved in designing and productizing systems in their career
- Scalable Systems Design
  - Last decade has been all about scale
  - Data, application, and traffic deluge
  - Silicon Valley is all about scale and expansion
- Fast forward to 2020
  - Scalable Systems Design is ***not Niche any more***

Differences

DS and Algo

- Deterministic
- More One-way communication
- Less Discussion Involved
- Depth oriented
- Evaluates a software engineer persona

Systems Design

- Fuzzy (no solution is 100% perfect)
- More Two-way communication
- Purely discussion driven
- Both breadth and depth oriented
- Evaluates all personas
    - Product managers
    - Engineering leaders
    - Software engineers
    - Performance engineers

How to Approach Mentally

- Personas
  - Interviewer is not the teacher but the end-user/customer/peer
  - Candidate is not a student but a peer
- Interviewing style
  - Not about answering a question
  - More about engaging in a fruitful discussion
  - Interviewer should feel confident that both can work together with full alignment
- **Demonstrate thought leadership**
  - ***It is more about how you would build the system in real world***

High Level Thought Process

Step 1: Wear the Product Management Hat

- **Treat interviewer as an end-user/customer**
- **Collect functional requirements**
  - This is the detailed problem statement
  - High level
  - Spend a few minutes to show that you can communicate given an unknown problem
  - Ask questions to clarify all doubts as much as possible.
  - The objective would be to be able to visualize APIs from the requirements
- **Collect design constraints (NFR non functional requirements)**
  - Numbers, how many, how much
  - Required for answering scalability
  - Often interviewers throw these back to the candidate, so it is beneficial to research on them
  - Can be collected at a later step

Problem statement

Imagine a data center having hundreds of servers (1000) each emitting thousands of metrics per second (2000) (such as CPU utilization, memory utilization,....,...)

We need to build a central brain that ingests and serves a dashboard

1. Given a server id, return min, max, avg of all metrics within a time window of 2 days (at a granularity of 1 min)
2. Given a metric id, return min, max, avg of all servers within a time window of 2 days (at a granularity of 1 min)
3. Given a server id and a time range, return min, max, avg of all metrics
4. Given a metric id and a time range, return min, max, avg of all servers

Data will live for an year

Step 2: Identify Building Blocks

- **From this step, start treating the interviewer as a peer**
- **Bucketize functional requirements into Microservices**
  - Simple high level clustering of requirements
  - Imagine if all requirements can be handled by the same team or not
  - One approach can be: if data models and APIs to address two requirements do not look same, then put them in different buckets
  - Not deterministic, depends on every individual
  - https://microservices.io/
- A microservice corresponds to a ***building block***
- From this, it gets clear whether problem is breadth-oriented or depth-oriented

Services

1. Data collection - pub/sub
2. Data aggregation and API service (central brain) (stream processor)
    a. App server tier - consumption
    b. In-memory tier - has the same logic and data model, but organization and amount of data differs (here 2 days)
    c. Storage tier (1 year)

Step 3: Propose Umbrella Architecture

- **Propose logical architecture**
  - Block diagram of each Microservice
  - API gateway to interface between users and system (optional)

- **Propose and explain data/logic flow between them**
  - Rules of thumb:
    - If high volume of data needs to be pushed in near real time between two microservices, use publisher subscriber
    - Pub-sub is a microservice of its own
    - If data needs to be pulled from server to client, use REST APIs
    - If data transfer is offline, you may use batched ETL (extract transform load) jobs

Step 4: Delegate Design of Building Blocks

- Treat this step as if you are delegating each building block to a development team

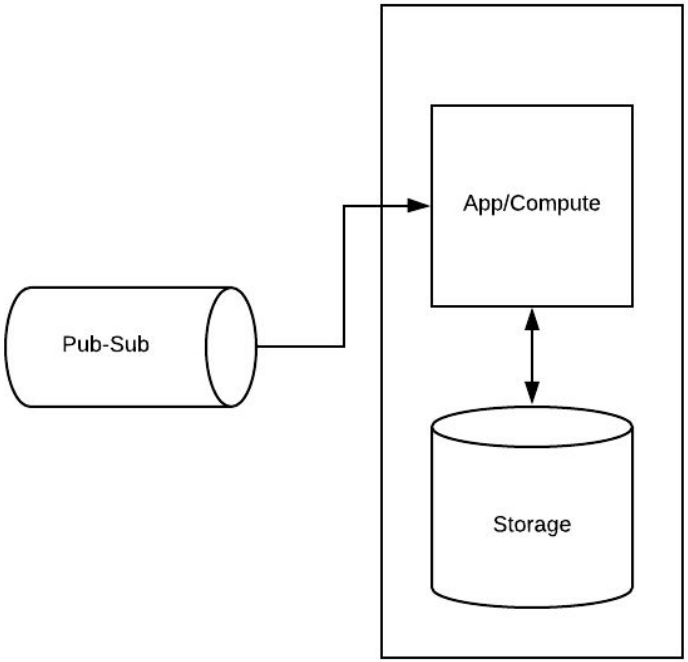
- Deep dive on each building block at a time
- **Try to map each building block as one of the technology types**
- Iterate over and discuss each
- Rinse and repeat

- If too many microservices or building blocks,
  - Negotiate with the interviewer on which ones to focus on
  - You only have constant time
  - Depending on the number of microservices to focus on
    - Budget your time per microservice

Designing Stream Processing and Analytics building block

Step 1:
Zoom in the Building Block (Generic)

Zoom in the Building Block



- App tier
  - Handles consumption of events from pub-sub or message queues
- In-memory tier
  - Stores a window of most recent time series events
  - Performs lightweight workloads on recent events
- Storage tier
  - Stores for a larger time window duration
  - Performs more heavyweight analytics on historical data
- Tier handling
  - App tier handled by application teams
  - Compute tier logic handled by application teams, infra handled by infrastructure teams
  - Storage tier handled by infrastructure teams

Building Block Design

Step 2:
Solve Dataplane (Data Model and API Logic)
Data Model and Standard APIs: Category Specific
Business Logic: Problem Specific

Step 2 (Dataplane).

- Solve each tier logically (as if solving for a single server)
  - **Scale is not in the picture yet**
  - Propose Data Model (what data needs to be stored) to match the functional requirements
  - Propose APIs to match the functional requirements
  - Discuss how data will be stored in storage and cache tiers
  - Propose workflow/algo for the the APIs in each tier
  - Propose flow across tiers within the microservice
    - **Most of technical design interviews have meat in this phase**
    - **Non-deterministic, every candidate will come with his or her own proposal**
    - **Changes from problem to problem**
    - **This constitutes the ‘most thinking’ portion of the interview**
    - **Most chances to flunk here unless candidate is prepared**

Data model

K-V: K: <server id, metric id, timestamp>: value: value of metric

This is timeseries data model: Insert only in increasing order of time, no updates or user deletes, TTL

Revised data model: ***K: <server id, metric id, 1 min bucket>: V: min, max, avg***

<S1, M300, 12:00:01> : V1

<S1, M300, 12:00:02>: V2

<S1, M300, 12:00 - 12:01>: min, max, avg

Organization in memory

Data model

Organization in memory

Hashmap<<Sid, Mid, Tb>, <min, max, avg>>

Hashmap of hashmap (Two hashmaps required because of two different queries)

Sid -> Mid -> Tb: value

Mid->Sid->Tb: value

Ringbuffer: 2 days of worth of 1 minute nodes: 2880: 60*24*2

SidX->MidY->Ringbuffer of 2880 elements

MidY->SidX->Ringbuffer of 2880 elements

Hashmap<Sid, Hashmap<Mid, Ringbuffer>>

Hashmap<Mid, Hashmap<Sid, Ringbuffer>>

```
Hashmap<Sid, HashMap<Mid, Ringbuffer>> sidMap;

Hashmap<Mid, RingBuffer> temp = sidMap.get(sid);

Collection<RingBuffer> collection = temp.values();
```

```
HashMap<Mid, hashMaP<SID, RINGBUFFER> MIDMAP

Hashmap<sid, RingBuffer> temp = midMap.get(mid);

Collection<RingBuffer> collection = temp.values();
```

S1, M300 -> ringbuffer X

M300, S1 -> ringbuffer X

hashmapServerc.get(Sid) = hashmapMetric<Mid, Ringbuffer>

hashmapMetricvalues() = Collection<Ringbuffer>

Data model

Organization in storage

Row oriented

[Sid, Mid, Tb1min]: min, max, avg

Maintain two secondary indexes

If an index is created in the order of (A, B, C), A in the where, A, A and B, A and B and C

Create index (sid, Tb) to serve query 3, create index (mid, Tb) to serve query 4

[A, B, C]

TTL in storage

Time partitioned data models

T_July 20, 2019 -> all the 1 min data for that date + index <Sid, Mid, Tb> : min, max, avg

T_July 21, 2019 -> all the 1 min data for that date

On July 21, 2020, drop the partition T_July 20, 2019: it is a filesystem level operation, m***uch much much much much*** faster than handling each single record

Storage Persistent

Row major storage: Write friendly and I have to see all fields

<Sid, Mid, Time Bucket> : Min, Max, Avg

Index(Sid, TimeBucket, Mid) ->

Index(A, B , C) -> query has to provide A, (A and B), (A and B and C)

Storage Persistent

Another index (Mid, Timestamp, Sid)

As the data gets cold, it can be aggregated at a much higher rate

Partition the data model by time range

T_july 2019: 1000 sids*2000mids*3 values/min*30days*24*60 min = 259 billion

T_aug 2019

Summary

1. Composite key, and query APIs involves partial key
    a. Multiple hashmaps, multiple indexes
2. If data has a lot of tags/columns, and queries are on permutations/combinations of random tags, use columnar, otherwise row
3. Ring buffer is a good model for sliding window processing, stream processing in memory
4. Data purge ,TTL is an important aspect of stream processing and IOT **TIMESERIES** data processing
    a. Handle through ring buffers in memory
    b. Handle through time range partitions in storage
    c. If you can simulate ring buffer logically in storage
        i. RRD tools do that
5. Time range based read workloads work very well with time partitioned data

Common APIs

- Data Stream APIs
  - Aggregators
  - Filters
  - Sliding Window Functions
  - Low-level joins

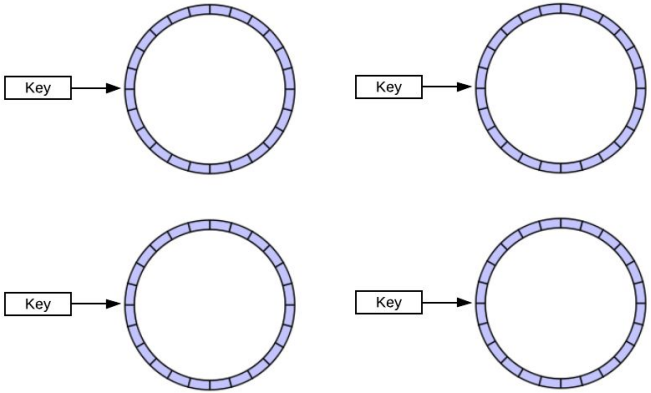
- Dataset APIs
  - Transformation APIs
  - Iterators

- Time Series Analytics

Common In-memory Data Models and Structures

- Structures
  - Hashmaps
  - Hashmap of Ring Buffers
  - Time partitioned HashMaps
  - Time Partitioned In-memory Column Stores

- Properties
  - Data Retention and TTL
  - Write into memory and storage latency fast
  - Temporal Filtering
  - Information Lifecycle Management



Common Storage Systems

- RRD Databases
  - RRDTool
    - Circular buffer based database
    - Concepts

PDP	Primary Data Point	An Event
Step	Time Interval of Data Arrival	PDP event appears every 1 Seconds
CDP	Consolidated Data Point	Aggregating Stat (Min, Max. Avg) per Minute
Window	Sliding Window	Circular buffer for CDPs for past 2 days

Common Storage Systems

- Time Series Databases
  - Apache Druid, Apache Pinot
    - Column-oriented storage
    - Native search indexes
    - Streaming and batch ingest
    - Flexible schemas
    - Time-optimized partitioning
    - Information Lifecycle Management
  - InfluxDB

Step 3: Scale Justification Category Specific

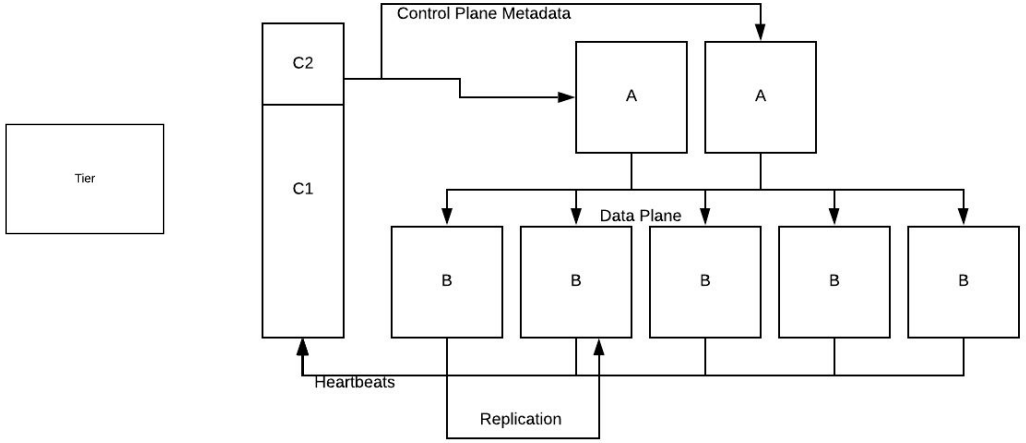
High level Performance and Scale Analysis

Scaling for Storage	Depends on TTL, sliding window, aggregation time interval
Scaling for Throughput	High throughput ingestion, typically low throughput analytics
Scaling for Latency Reduction	Typically not applicable

Deep-dive on capacity planning when explicitly asked for

Step 4: Scale the Building Block (Generic)

Template of a Scaled Architecture

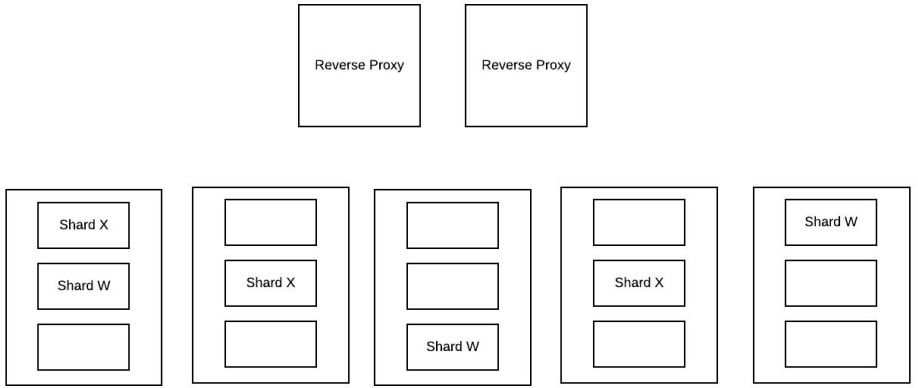


	Component
A	**Load Balancer, Reverse Proxy, Coordinator, Aggregator**
B	**Data Plane Worker**
C1	**Cluster Monitor**
C2	**Control Plane Configuration Manager**

Sharding and Replication

- Virtual partitions of data or sticky logic

- **Horizontal Partitioning**
  - Partitioning by key dimension
  - **More common**
- Vertical Partitioning
  - Partitioning by value dimension
- Hybrid



In most cases, not much changes from the dataplane side

Importance of Sharding Parameter

- Ensure sharding parameter takes care of the reasons to scale
- If API contains sharding parameter
  - Reverse proxy can enumerate the shard id
  - Through development logic
    - Hash(time)%number of shards = shard id
  - Reverse proxy caches shard directory enumerated by cluster manager
    - Shard X: Server A, C, E
  - Forwards request to appropriate server
- If API does not contain sharding parameter
  - Scatter gather or fan out

Scalable System

Horizontal: partitioning by key

If I sharded by {Sid}, Sid based query is local, Mid based is scatter gather

If I sharded by {Mid}, Mid is local but Sid query is scatter gather

Shard by function(Sid) -> Shard id

Scatter gather for Mid based query

Shard by function(Mid) - > Shard id

Scatter gather for Sid based query

Scalable System

Duplicate the data model

Table A and Table B have same data

But table A is sharded by Sid and table B sharded by Mid

Table A: sharded by Sid and then within a shard, partitioned by time

Table B: sharded by Mid and then within a shard, partitioned by time

Sidx, Midy, T, V

Shard id of Table A that will contain data for x

Shard id of Table B that will contain data for y

hash(sid) % N = id,

LinkedIn example

<Niloy, Srini>: rank of the edge, edge created in a table

Inviter: Invitee: rank - Table A

Invitee:Inviter: rank - Table B

<Niloy: Srini>: 6

<Srini:Niloy>: 6

Hash(inviter)%shards = shard id for table A

Hash(invitee)%shards = shard id for table B

Select all members that Niloy has invited

Select all members who invited Ankan

[,A B, C],

hash(A)%shards = shard id

The invariant is that all application teams need to provide the top level key part in all APIs

In Espresso, we allow data models with multiple key parts (composite keys)

K: [conversation id, message id within conversation]: V: rest of the message

Espresso (K-V store) uses hash(first key part)%shards = shard id

We ask all our application teams to put the first key part in all APIs

Sharding by time causes hotspots

Horizontal partitioning by time

[time A - time B] -> Shard 0 -> A, C, E (in storage)

[ime A - time B] -> Shard 0 -> MA, MC, ME (in memory)

Hotspots

CAP theorem

- **Consistency**, **Availability**, and Network **Partition** Tolerance cannot be achieved all together
- Network Partition Tolerance
  - Tolerate situation where a shard replica not able to communicate to another replica
- In such situation,
  - Either CP or AP
  - Either fail a update/write and be consistent
  - Or return success and be inconsistent
- **Quorum Reads and Writes** for **Strict Consistency**
  - Write into majority of replicas
  - Read from majority

Building a CP/AP System

		Config	Shard Throughput	API Latency	Availability	Ease	Strict Consistency
AP		N >= 3 R = 1 W =1					
CP	Master -Slave	N = 1 R = 1 W = 1					
CP	Quorum	N = 3 R > N/2 W > N/2					