Deep Dive Into Statistics Feature

Questions

- Who will query the number of visits?
  - Let’s assume it is used for time-based statistics.
- How often will it be queried?
  - Real-time or periodically (monthly, weekly)?
- How many read queries per second?
  - Let’s assume tens of thousands.
- What is the accepted write-to-read delay?
  - If we can count several hours later, batch data or stream processing can be considered.
- How fast data must be read?
  - If it has to be fast, the aggregation has to happen at writing instead of reading.

Non Functional Requirements

- Assumptions:
  - **scalable**: tens of thousands of visits per second.
  - **highly performant**: few tens of ms to return total view count.
  - **highly available**: survives hardware/network failure, no single point of failure.

- Cap theorem tells that if we want to have a distributed system, we have to choose between availability and consistency.
  - Based on the requirements we have to choose availability.

Functional Requirements:

- Write stats:
  - @router.get('/{hash}’)

  - count_visit_event(url_id: int)
  - count_event(url_id: int, event_type: EventType) EventType: VISIT, GENERATION
  - process_event(url_id: int, event_type: EventType, function_type: FunctionType) FunctionType: COUNT, SUM, AVG
  - process_events(events: Iterable[Event])

- Read stats:
  - @router.get('/view_count')

  - get_view_count(url_id: int, start_time: datetime, end_time: datetime)
  - get_count(url_id: int, event_type: EventType, start_time: datetime, end_time: datetime)
  - get_stats(url_id: int, event_type: EventType, function_type: FunctionType, start_time: datetime, end_time: datetime)

Storage options

url_id	timestamp
1	2022-10-05 16:10:17
1	2022-10-05 16:13:17
2	2022-10-05 16:16:43

Individual events (every click)

Pros:

- Fast writes
- Can slice and dice data however we need
- Can recalculate numbers if needed

Cons:

- Slow reads
- Costly for large scale (many events)

url_id	timestamp	count
1	2022-10-05 16:10	4
2	2022-10-05 16:10	2
1	2022-10-05 16:11	1

Aggregate data (e.g. per minute) in real time

Pros:

- Fast reads
- Data is ready for decision making

Cons:

- Can only query the way it was aggregated
- We need to pre-aggregate data in memory before storing
- Hard to fix errors

Storage options

We have to choose between the 2 based on the expected data delay.

- Raw events:
  - If several hours is ok.
  - Use batch data processing.
- Real-time aggregation
  - It should not be more than a few minutes.
  - Use stream data processing.

If complexity and cost are not a problem we can also combine the 2 options

- Let's store raw events for several days/weeks, and purge old data.
- Also, we will aggregate and store real-time so statistics are available right-away.
- By storing both we get the best of both worlds: fast reads, the ability to aggregate differently, and recalculate statistics if there are bugs or failures.

Real-time aggregation (RELATIONAL)

If a single server is not enough we have to introduce sharding (horizontal partitioning).

Each shard will store a subset of the data .

We introduce a cluster proxy server that knows about the databases and routes traffic to the correct shard.

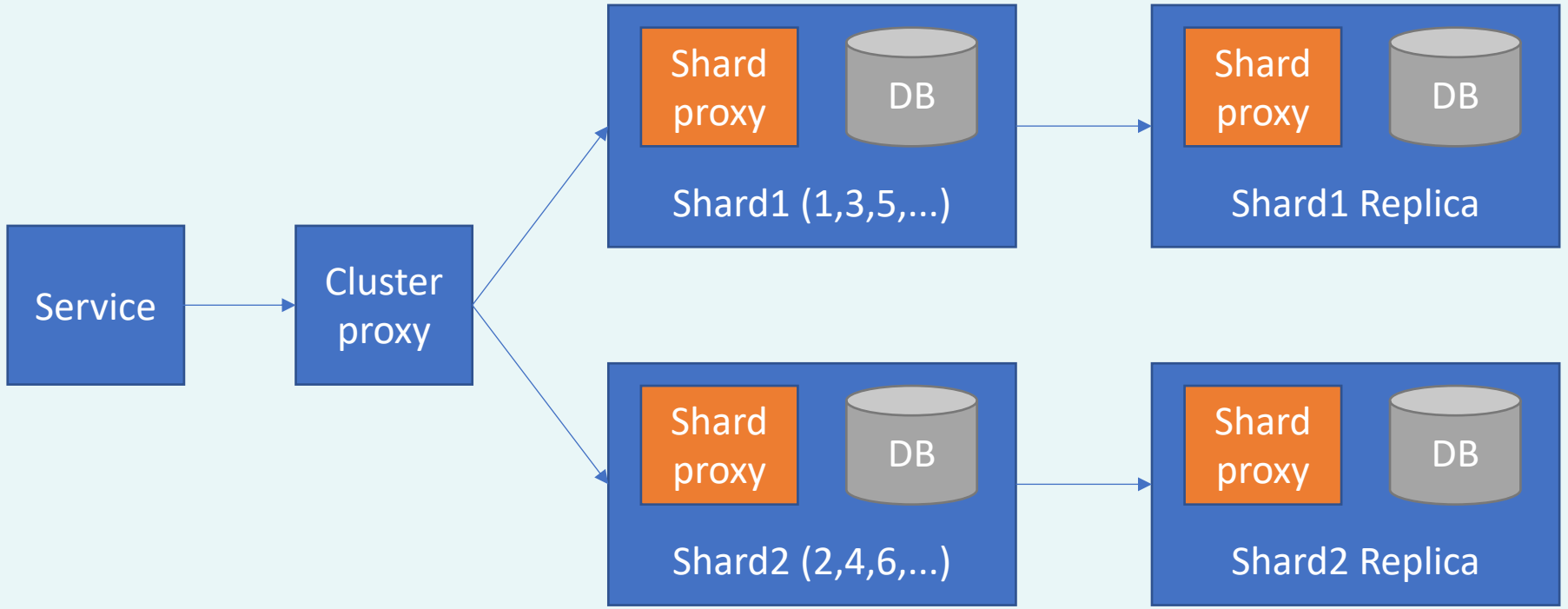
To enable the proxy server to know if a shard dies, becomes unavailable, or a new one is introduced, we have to introduce a configuration service such as ZooKeeper, which maintains a health check connection to all shards.

In front of each database, we can introduce a shard proxy that can cache query results, monitor database health, publish metrics, terminate queries that take too long to return, etc.

If a database dies we have to make sure data is not lost, so we have to replicate data. For each shard we have to introduce a replica, these replicas should be stored in a different place physically for maximum protection.

When data is received by Shard, data should be copied to a replica shard.

Real-time aggregation (RELATIONAL)



Real-time aggregation (NoSQL)

In NoSQL, We also split databases into nodes (shards), but we do not have a configuration service to monitor them, instead, nodes will talk to each other.

To reduce network load we should not connect each node to every other node.

Information is propagated through each node, using the gossip protocol.

Because all nodes are connected we do not need a clustered proxy, any of the nodes could be called and the request would be redirected to the correct one.

It makes sense to connect to the node which is closest to the client, this node will then make sure that data will be written to 2-3 different nodes to replicate data.

When reading the connected node can send multiple requests at the same time for different nodes, and return the data for which 2 other nodes returned the same data.

Real-time aggregation (NoSQL)

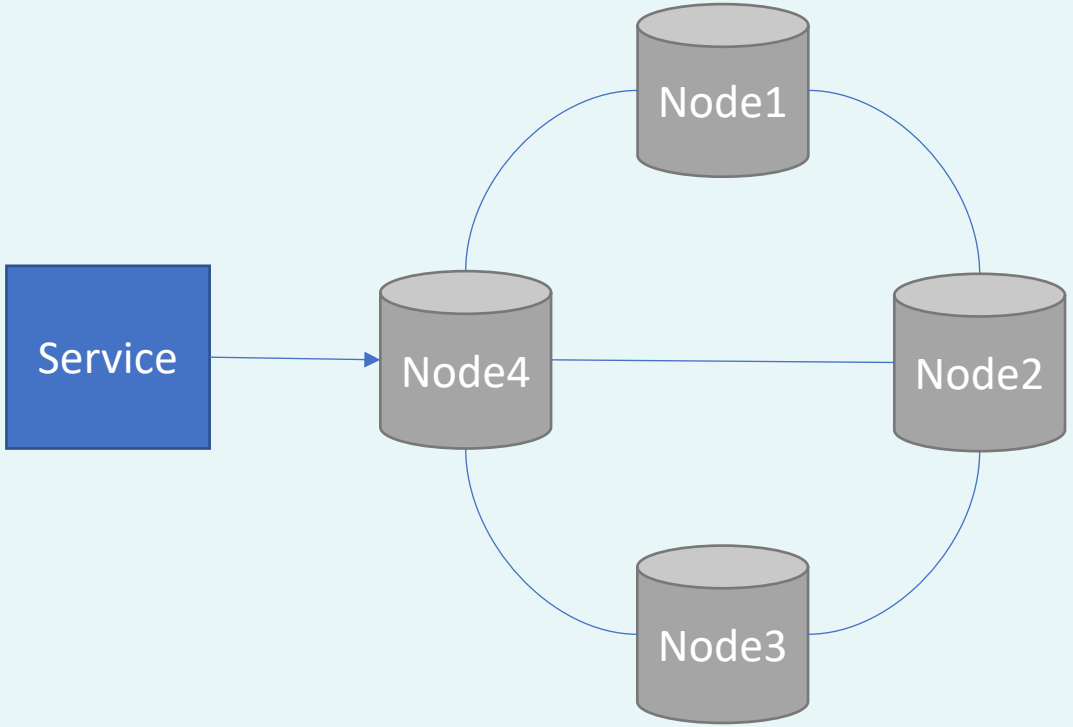


Table structure:

url_id	16:00	17:00	18:00
1	4	12	2

Improve processing

To reduce the number of writes in the database we can introduce pre-aggregation in the backend service with an in-memory counter.

Every few seconds the in-memory counter will pop its data into the database.

We can also introduce a cache here (such as Redis) which can help us achive this and also store most frequently used urls.

Improve processing

If we want to reduce the risk of losing the data if the backend service goes down, we can introduce an event store (such as Kafka) in front of the backend where the requests would be temporarily stored.

If the service goes down, we can reprocess the data.

We can also partition the event-store and for each partition we can have a separate backend service.

To enable this we have to introduce a partitioning service in front of event-store partitions.

In front of the partitioning service we can have a load balancer to evenly distribute events.

Improve processing