E100作为时序数据库的写入与查询性能测试方案

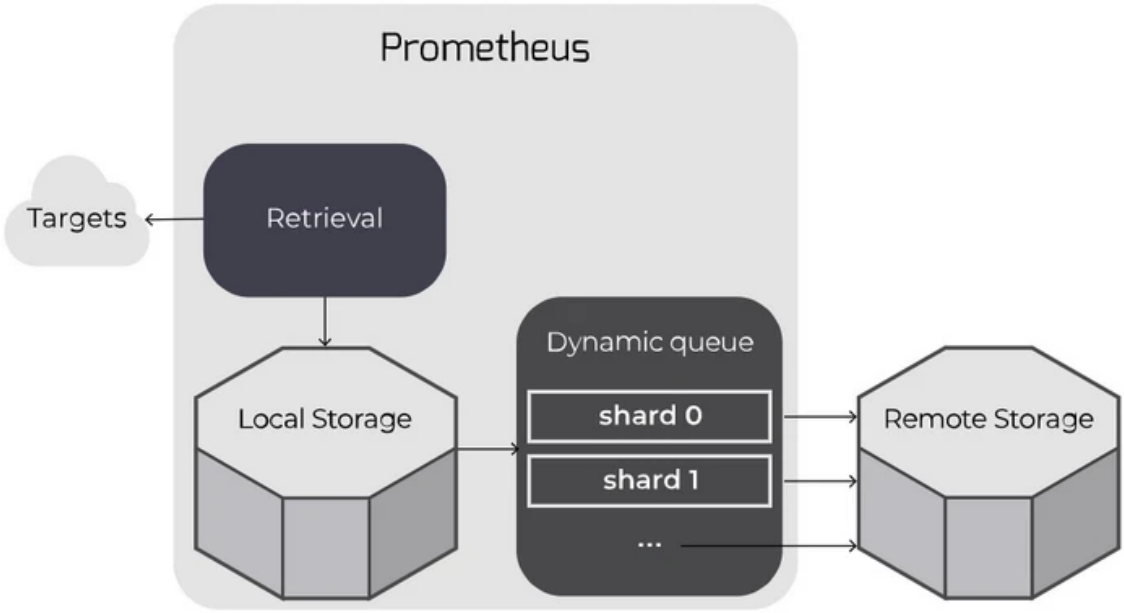
前置知识

1、配置prometheus远程时序数据库

```
remote_write:
    url: <string>

remote_read:
    url: <string>
```

Prometheus配置文件中添加remote_write和remote_read配置，其中url用于指定远程读/写的HTTP服务地址。



远程写工作是通过“定序”将时间序列样本写入本地存储，然后将它们排队以写入远程存储。

队列实际上是一组动态管理的“分片”：任何特定时间序列（即唯一指标）的所有样本最终都将位于同一分片上。

队列会自动按比例增加或减少写入远程存储的分片数量，以跟上传入数据的速率。

这样一来，Prometheus便可以在不使用远程资源的情况下，仅使用必需的资源并且以最少的配置来管理远程存储。

每个远程写目的地都启动一个队列，该队列从write-ahead log (WAL)中读取数据，将样本写到一个由（分片）shard拥有的内存队列中，然后分片将请求发送到配置的端点。数据流程如下：

```
      |-->  queue (shard_1)   --> remote endpoint
WAL --|-->  queue (shard_...) --> remote endpoint
      |-->  queue (shard_n)   --> remote endpoint
```

远程写方式

远程写是调用http请求来写的，数据格式是protobuf（一种自定义的编码格式），编码格式是snappy（一种压缩格式）；远程写通过snappy先压缩，然后protobuf编码的字节数组请求的；接收到远程写数据时是乱码，先用snappy进行解压缩，prometheus官网文档远程写提供remote.proto（包含编码和解码），remote.proto文件中依赖了types.proto和gogo.proto两个文件。

- 使用POST方式提交
- 需要经过protobuf编码，依赖github.com/gogo/protobuf/proto
- 可以使用snappy进行压缩，依赖github.com/golang/snappy

步骤：

1. 收集指标名称，时间戳，值和标签
2. 将数据转换成prometheus需要的数据格式
3. 使用proto对数据进行编码，并用snappy进行压缩
4. 通过httpClient提交数据

remote_write 配置

```
# The URL of the endpoint to send samples to.
url: <string>

# Timeout for requests to the remote write endpoint.
[ remote_timeout: <duration> | default = 30s ]

# Custom HTTP headers to be sent along with each remote write request.
# Be aware that headers that are set by Prometheus itself can't be overwritten.
headers:
  [ <string>: <string> ... ]

# List of remote write relabel configurations.
write_relabel_configs:
  [ - <relabel_config> ... ]

# Name of the remote write config, which if specified must be unique among remote
write configs.
# The name will be used in metrics and logging in place of a generated value to
help users distinguish between
# remote write configs.
[ name: <string> ]

# Enables sending of exemplars over remote write. Note that exemplar storage
itself must be enabled for exemplars to be scraped in the first place.
[ send_exemplars: <boolean> | default = false ]

# Sets the `Authorization` header on every remote write request with the
# configured username and password.
# password and password_file are mutually exclusive.
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# Optional `Authorization` header configuration.
```

```
authorization:
  # Sets the authentication type.
  [ type: <string> | default: Bearer ]
  # Sets the credentials. It is mutually exclusive with
  # `credentials_file`.
  [ credentials: <secret> ]
  # Sets the credentials to the credentials read from the configured file.
  # It is mutually exclusive with `credentials`.
  [ credentials_file: <filename> ]

# Optionally configures AWS's Signature Verification 4 signing process to
# sign requests. Cannot be set at the same time as basic_auth, authorization, or
oauth2.
# To use the default credentials from the AWS SDK, use `sigv4: {}`.
sigv4:
  # The AWS region. If blank, the region from the default credentials chain
  # is used.
  [ region: <string> ]

  # The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
  # and `AWS_SECRET_ACCESS_KEY` are used.
  [ access_key: <string> ]
  [ secret_key: <secret> ]

  # Named AWS profile used to authenticate.
  [ profile: <string> ]

  # AWS Role ARN, an alternative to using AWS API keys.
  [ role_arn: <string> ]

# Optional OAuth 2.0 configuration.
# Cannot be used at the same time as basic_auth, authorization, or sigv4.
oauth2:
  [ <oauth2> ]

# Configures the remote write request's TLS settings.
tls_config:
  [ <tls_config> ]

# Optional proxy URL.
[ proxy_url: <string> ]

# Configure whether HTTP requests follow HTTP 3xx redirects.
[ follow_redirects: <boolean> | default = true ]

# Whether to enable HTTP2.
[ enable_http2: <bool> | default: true ]

# Configures the queue used to write to remote storage.
queue_config:
  # Number of samples to buffer per shard before we block reading of more
  # samples from the WAL. It is recommended to have enough capacity in each
  # shard to buffer several requests to keep throughput up while processing
  # occasional slow remote requests.
  [ capacity: <int> | default = 2500 ]
  # Maximum number of shards, i.e. amount of concurrency.
  [ max_shards: <int> | default = 200 ]
  # Minimum number of shards, i.e. amount of concurrency.
```

```
  [ min_shards: <int> | default = 1 ]
  # Maximum number of samples per send.
  [ max_samples_per_send: <int> | default = 500]
  # Maximum time a sample will wait in buffer.
  [ batch_send_deadline: <duration> | default = 5s ]
  # Initial retry delay. Gets doubled for every retry.
  [ min_backoff: <duration> | default = 30ms ]
  # Maximum retry delay.
  [ max_backoff: <duration> | default = 5s ]
  # Retry upon receiving a 429 status code from the remote-write storage.
  # This is experimental and might change in the future.
  [ retry_on_http_429: <boolean> | default = false ]

# Configures the sending of series metadata to remote storage.
# Metadata configuration is subject to change at any point
# or be removed in future releases.
metadata_config:
  # Whether metric metadata is sent to remote storage or not.
  [ send: <boolean> | default = true ]
  # How frequently metric metadata is sent to remote storage.
  [ send_interval: <duration> | default = 1m ]
  # Maximum number of samples per send.
  [ max_samples_per_send: <int> | default = 500]
```

`capacity`

`capacity`(容量)控制在阻止从 WAL 读取之前，每个分片在内存中排队多少个样本。一旦 WAL 被阻止，就无法将样本附加到任何分片，并且所有吞吐量都将停止。

在大多数情况下，容量应足够高以避免阻塞其他分片，但是太大的容量可能会导致过多的内存消耗，并导致重新分片期间清除队列的时间更长。建议将 `capacity` 参数设置为 `max_samples_per_send` 的 3-10 倍。

`max_shards`

`max_shards`(最大分片数)配置最大分片数或并行性，Prometheus 将为每个远程写入队列使用。Prometheus 尝试不使用过多的分片，但是如果队列落后于远程写组件，则会将分片的数量增加到最大分片数以增加吞吐量。除非远程写入非常慢的端点，否则`max_shards`不能增加到默认值以上。如果有可能使远程端点不堪重负，则可能需要减少最大分片数，或者在备份数据时减少内存使用量。

`min_shards`

`min_shards`(最小分片数)配置 Prometheus 使用的最小分片数量，并且是远程写入开始时使用的分片数量。如果远程写入落后，Prometheus 将自动扩大分片的数量，因此大多数用户不必调整此参数。增加最小分片数将使 Prometheus 在计算所需分片数时避免在一开始就落后。

`max_samples_per_send`

`max_samples_per_send`(每次发送的最大样本数)可以根据使用的后端进行调整。许多系统可以通过每批发送更多样本而不会显着增加延迟来很好地工作。如果尝试在每个请求中发送大量样本，则其他后端将出现问题。默认值足够小，可以在大多数系统上使用。

`batch_send_deadline`

`batch_send_deadline`(批量发送超时时间)设置单个分片发送的最大超时时间。即使排队的分片尚未达到`max_samples_per_send`，也会发送请求。对于对延迟不敏感的小批量系统，可以增加批量发送的截止时间，以提高请求效率。

`min_backoff`

`min_backoff`(失败的最小重试时间)控制重试失败请求之前等待的最短时间。当远程端点重新连接时，增加重试时间将分散请求。对于每个失败的请求，重试时间间隔都会加倍，直到增加到`max_backoff`。

`max_backoff`

`max_backoff`(失败的最大重试时间)控制重试失败请求之前等待的最长时间。

方案设计

原理

测试数据量对性能的影响，测试E100作为时序数据库的最大写入量跟查询效率，所以需要可以定量的写入数据

方案一

由于远程存储是调用http请求来实现的，也就是可以自己手动实现prometheus远程读写的功能，模拟一下prometheus，只要发送的请求跟prometheus是一样的应该就可以了，也就是变成一个接口的性能测试。

优点：可操控性比较高，而且可以参考一些接口的性能测试方案，不用考虑prometheus内部是怎么实现的，配置完之后能不能达到生效，也不用考虑数据从哪里来，够不够发送之类的

缺点：prometheus数据格式是protobuf（一种自定义的编码格式），编码格式是snappy（一种压缩格式），尝试了一下没有实现成功，后续可能需要花点时间

方案二

通过修改prometheus的配置文件来实现，靠配置文件的指标间接计算写入数据的速度。

优点：操作起来简单，只需要改改配置文件

缺点：配置的颗粒度比较大，而且不了解prometheus内部实现机制，操作性不好，需要考虑prometheus内部是怎么实现的，配置完之后能不能达到生效，考虑数据从哪里来，够不够发送之类的，还有就是数据量的定量不太好操作