CS631 Project: Live streaming in Postgres Report

Sapan Tanted (17305R007), Akshat Garg (17305R003)

06 May 2019

Postgres version 12 is used.
Deliverable: client side api and server side postgres extension

1 Table Structure on postgres side:

1. **client_table: stores registered clients**
   (a) client_id (string) : client_id of client_side_api (limit 100 char).
   (b) connection_timestamp (timestamp) : timestamp when client_id registered.
   (c) number_of_subscriptions (integer) : number of topics client_side_api subscribed

Column	Type	Modifiers
connection_timestamp	timestamp without time zone	
client_id	varchar(100)	Not null
number_of_subscriptions	integer	

"client_table_pkey" PRIMARY KEY, btree (client_id)

2. **topic_table: stores topics that are being published and subscribed on**
   (a) **topic_id (integer)** : Id given to a topic by postgres.
   (b) **topic (string)** : topic name which client_side_api sends. In db, every topic is mapped to a integer id.
   (c) **last_msg_recv_timestamp (timestamp)** : time when the last message was received on server for this topic. (if no messages are published i.e. topic is only subscribed, then current_timestamp - 1 sec).
   (d) **relative_timeout (long)** : calculated and updated whenever a new subscriber connects or disconnects. This values helps in calculating the expiry time for a message. It is calculated as (max of all subscribers subscription_timestamp + timeout - current_timestamp) i.e. how long a message should live if it comes now.

(e) **number_of_users_subscribed (integer)**: keeps track when we can delete the topic. If 0 subscribers are there then delete it.

Column	Type	Modifiers
topic_id	integer	not null
topic	varchar(100)	
last_msg_recv_timestamp	timestamp without time zone	
relative_timeout	bigint	(default 0)

"topic_table_pkey" PRIMARY KEY, btree (topic_id)

3. **payload_table: stores new coming messages**

   (a) **client_id (string)** : client_id of publisher (foreign key from client_table).

   (b) **topic_id (integer)** : topic_id from topic table (foreign key from topic_table)

   (c) **payload (string)** : message published by client on this topic.

   (d) **payload_timestamp (timestamp)** : time when the payload is received.

   (e) **expiry_timestamp (timestamp)** : calculated by adding relative_timeout with payload_timestamp. This is the max time after which this message will be no longer needed, either it would have been sent to all subscribers or they would have timed out.

Column	Type	Modifiers
topic_id	integer	
payload	varchar(100)	
payload_timestamp	timestamp without time zone	
expiry_timestamp	timestamp without time zone	

4. **subscription_table: stores topics subscribed**

   (a) **client_id (string)** : client_id of subscriber (foreign key from client_table).

   (b) **topic_id (integer)** : topic_id which is subscribed (foreign key from topic_table).

   (c) **subscription_timestamp (timestamp)**: time when client_side_api subscribed to the topic.

   (d) **timeout (long)** : number of seconds after which subscription can be deleted if client_side_api gets disconnected and does subscribe back.

   (e) **last_ping_timestamp (timestamp)** : time when the subscriber sent last ping request so that we know that subscriber is still alive. If last_ping_timestamp + timeout is less than current_timestamp then we can disconnect the subscriber and remove its entry. Whenever a new payload is sent to a subscriber that event can also be considered as ping event.

(f) **Whenever a new entry is added or deleted from this table:** number_of_subscription is adjusted accordingly in client_table corresponding to the client_id. (if entry added then +1, if entry deleted then -1).

(g) Whenever a new entry is added in this table relative_timeout is calculated by taking max(relative_timeout, current subscriber's ({last_ping_timestamp + timeout}- current_timestamp). Whenever an entry is deleted then relative_timeout is calculated for the topic as (max of all subscribers ({last_ping_timestamp + timeout} - current_timestamp)

Column	Type	Modifiers
column$_id$	varchar(100)	not null
topic_id	integer	not null
subscription_timestamp	timestamp without time zone	
timeout	bigint	
last_ping_timestamp	timestamp without time zone	

"subscription_table_pkey" PRIMARY KEY, btree (client_id, topic_id)

2 Functions from user perspective:

1. User will call client_side_api.
2. client_side_api will give 3 functions connect, publish and subscribe.
3. While subscribing user will give callback function to client_side_api so that api can call callback for every record fetched.

3 Functions from client_side_api perspective:

1. **connect_stream(string client_id):** registers client_id in database and now client_side_api can publish as well as subscribe to any topic.
2. **publish(string client_id, string topic, string payload):**
   (a) client_side_api gives client_id and topic on which he wants to publish.
   (b) Payload is the message to be published on that topic.
   (c) All the subscribers (subscribed to that topic at that time) will get the payload. (subscribers will get notified by semaphores).
   (d) Those subscribers who got disconnected and reconnected before their timeout will also get this payload.
3. **subscribe(string client_id, string topic, long timeout):**

(a) currently thinking to make subscribe function result set as blocking and gets new data as soon as it comes but depending on feasibility we might have to shift on non-blocking subscribe function and client_side_api might need to call it every time he wants new data. (client_side_api will call subscribe function iteratively without any delay because blocking will be done at server side. if there is no record then that subscribe function will be blocked else it will return found records.)

(b) client_side_api gives client_id and topic to subscribe on (currently single topic subscription) and timeout which tell after how many seconds to remove subscription if client_id gets disconnected.

4 Functionality on postgres side:

1. **connect_stream(string client_id)**: client_side_api's client_id gets registered in client_table and number_of_subscriptions is set to 0.

2. **publish(string client_id, string topic, string payload)**: If new topic then add a new topic in topic_table. Add new entry in payload_table with the expiry_timestamp calculated from topic_table's relative_timeout field and payload_timestamp. Notify or send to all subscribers who are currently connected and subscribed.

3. **subscribe(string client_id, string topic, long timeout)**: If old subscription is not timed out, fetch all those messages where
($payload\_timestamp > last\_ping\_timestamp$)
and then send to client_side_api. If new subscription then last_ping_timestamp is set to current_timestamp. After all messages are sent to client_side_api, then client wil poll again and again for the new data.

4. A thread will be monitoring for timeout of subscribers and remove them whose timeout is reached. It will also be monitoring messages and periodically deleting the messages whose (expiry_timestamp < current_timestamp).

Command to execute: ./client {publish/subscribe} {client id} {topic} {payload/timeout}

5 Algorithm used

1. If it is the first call then add entry in topic table.

2. If entry in topic table does not exist then check if subscription entry exists, if not then add it and return the function with SRF_RETURN_DONE.

3. If subscription entry does not exists then check if
(last_ping_timestamp + timeout < current_timestamp)
if it is, then it is timed out already, therefore set last_ping_timestamp = current_timestamp and return.

4. If not timed out then keep fetching new payloads from last_ping_timestamp then once all the records are fetched set last_ping_timestamp = current_timestamp (use global array to store new rows and increment the counter by the number of rows).

6 Problems faced during the implementation:

1. **Blocking query and streaming the rows**: Tried to send the record without returning the query. We tried to use SRF functions for that, but we could not send the data in the middle of the function. The data only goes to the user when query gets completed.

2. **Blocking the subscriber by listen/notify**: Tried asynchronous listen/notify functionality of postgresql. Listen was not becoming a blocking call. We wanted the listen call to block the user there only.

3. **Using postgres semaphore for blocking**: Suppose we have two connections to postgres server and we need to share a semaphore variable between these two. But, we were not able to find a way to share the variable between them. Making them global and static was also not working.

4. **By making it as a transaction**: Tried to do the queries by making another sub-transaction. We were also not able to fetch other transaction updates until the current transaction finishes because transactions were isolated by the snapshot isolation.

5. Some other issues we faced:

    - In the transaction, we were not able fetch our own inserts.
    - Sometimes, we were having duplicate key values in the primary key.
    - Commit command was not working.

7 Contribution

Most of the work was done together.

Sapan Tanted (17305R007)

- Algorithm design
- Database schema design
- Server side extension functionality
- Tested Blocking query and streaming the rows (refer 6.1).
- Tested postgres semaphore for blocking (refer 6.3)

Akshat Garg (17305R003)

- Algorithm design
- Database schema design
- Client side api functionality
- Tested blocking the subscriber using listen/notify (refer 6.2)
- Tested by making a transaction (refer 6.4)