Background

- **Practical Connection Limit** - PostgreSQL achieves some of the fastest SQL throughput in the world, but does so through a fewer amount of connections. Typically 100 is the limit, and diminishing returns become counterproductive beyond 200. (Reference Required)
- **Multiple User Accounts** - There are a number of PostgreSQL users who wish to use PostgreSQL for more than 200 concurrent database-users. Such connections are from independent backend processes. They are often idle, but desire the ability to be NOTIFY'd on certain topics. This situation arises with Microservices Architectures, but also in situations with many Database Administrators, many Business Analysts, and DTS integrations.
- **Connection Pooling** - Is a great way to leverage the speed of PostgreSQL while keeping within a 100-connection limit. PgBouncer is one tool that helps accomplish this. Odyssey is another such tool that uses asyn-io and more than one CPU thread. These are typically used for 1 database user account, for a single database, with good benefits. A single large pool can be used.
- **Listen/Notify Pooling** - Work is being specified to enable more efficient pooling of Listen/Notify mechanisms:
  - https://github.com/yandex/odyssey/issues/365
  - https://github.com/pgbouncer/pgbouncer/issues/655

Problem

Issues

- **A single large connection pool cannot be shared across multiple Users (and Databases)**. Pools can only be practically created for related Database-User combinations. If there are 200x users, there needs to be at least 200 connections ready.
- The network connection protocol doesn't have support for changing the user on an existing connection. Instead new connections must be created specifically for the User-Database context (and old ones terminated), adding lag.
- The PostgreSQL SQL dialect doesn't have support to "authoritatively" change the user.
  - SET ROLE - is limited to a subset of roles the current session user is a member of; it can be reset using RESET ROLE. This enables the client application to gain broader privileges and impersonate other users, which is bad.
  - SET SESSION AUTHORIZATION - can only be used by a superuser to switch to another user (perhaps less privileged). `RESET SESSION AUTHORIZATION` can be sent, resetting back to being a super user. This

enables the client application to gain broader privileges and impersonate other users, which is bad.

Inadequate workarounds

- **SET SESSION AUTHORIZATION** - see above
- **Separate connections** - as above, for 200x users this requires 200x connections between the middleware and backend. This defeats the purpose of connection pooling.
- **Aggressive connection termination -** at an extreme, the middleware could create a single user connection for running a command, then close the connection. This has a performance penalty. This can be balanced with a timeout factor, and idle-detection, but that requires tuning.
- **Don't use PostgreSQL** - it seems that mySQL comfortably supports up to 200k concurrent connections. That's nice, but many systems have already chosen PostgreSQL and cannot easily change, and want to invest in its improvement.
- **Use more PostgreSQL replicated instances -** while that can work for many workloads, that's not universally efficient.
  - Consider a simple situation with 1000 users (with different database logins) who are sending 1 query every 1 hour. It's a very light load, and the server resources are underutilized. It's the concurrent connection issue that needs to be solved here. Additional nodes will work, but that's unnecessary cost, as well as new added sysadmin complexity to manage replication.
  - This proposal is particularly considering the use of Listen/Notify, which only works within a single node.
- **Use a custom middleware** - perhaps a Web API could be built that used a single shared user account
  - Assume that all of the client applications need the PostgreSQL protocol connection. (eg. PowerBI, pgAdmin, Third Party software, and more). A new API is not an option in such cases.
  - Such a Web API would need to implement authentication and authorization. That's a trap many developers fall into, but it's not one that all developers desire.

Solution

Goals

- **Fast** - any changing of username needs to be as low-overhead as possible. The middleware can be trusted (it already has the ability to connect as superuser anyway).
- **Changing the username** (not necessarily the database).
- **For use with middleware** like pgBouncer or Odyssey.

Options

The following options proposed here are backwards compatible with previous versions.

1. **Connection: Database User Impersonation** - This approach is preferred, because it is anticipated to be the most efficient with 0-RTT most of the time.
   - A new GRANT privilege to Impersonate Users. Users don't have this privilege by default. A user with the Impersonate Users privilege SHOULD NOT have any other privilege (reading any data or anything).
   - A middleware (the frontend of a direct connection to PostgreSQL) WILL use the Startup Message as usual to authenticate as a user with ImpersonateUser privilege.
   - A new protocol message format called "ImpersonateDatabaseUser" with fields for
     - user - The username is a mandatory. If the userOid is not provided, this MUST match a username that is present and active in PostgreSQL.
     - userOid - The user Oid is optional. If provided, this will be used. The middleware MAY lookup a valid oid for efficiency. The backend MUST use the provided Oid if one is set.
     - Database - The database is mandatory. This MUST match the same database that the connection was started on. In the future, this might be used to change the database.
     - Password - optional. This SHOULD be sent the first time a username is used by the middleware to validate the password. This may be empty upon subsequent uses, to reduce lookups and load.
   - A middleware MAY send commands to reset a connection state before a batch of client protocol commands.
   - A middleware WILL send ImpersonateDatabaseUser
   - A middleware WILL send one or more messages for that user (database) context.
   - A middleware MAY send commands to reset a connection state after a batch of client protocol commands
   - ImpersonateDatabaseUser will either succeed silently (0-RTT), or fail. Upon failure, no further commands will be processed until ImpersonateDatabaseUser succeeds.
     - The middleware will receive the ImpersonateDatabaseUserError message, and propagate this back to the real client. The middleware MAY cache the SimpleQuery and retry instead of propagating this message back to the real client. But that's not necessary.
2. Connection: Permit more than one **StartupMessage** -
   - Compared to [1], the difference here, is that it's the same StartupMessage with multiple RTT as usual. But this is less suitable for per-statement user-role changes.
   - Instead of only supporting the **StartupMessage** upon connection, the **StartupMessage** is expected and permitted outside of a Query request/response cycle.

3. **SQL: Database User Impersonation** - this is very similar to [1], except using new SQL SimpleQuery commands instead of Protocol Messages. Protocol Messages are preferred because they are more structured, and not SQL-standard anyway.
4. SQL: Extend `SET ROLE` with Password -
    - This would be allowed for a user role with less or no privileges to change to a privileged role. Therefore the use of `RESET ROLE` could result in losing all access.
    - Eg. `SET ROLE user5 WITH PASSWORD ‘abc123’;`
5. SQL: Extend `SET SESSION AUTHORIZATION` with Password
    - This would be allowed for a user role with less or no privileges to change to a privileged role. Therefore the use of `RESET SESSION AUTHORIZATION` could result in losing all access.
    - Eg. `SET SESSION AUTHORIZATION user5 WITH PASSWORD ‘abc123’;`
6. Multiple implementations - to enable this capability for a variety of use cases. Particularly [1] and [5] perhaps. One for Connection level, and one for SQL level.

Next Steps

- Community Consultation and Design on pgsql-hackers@lists.postgresql.org
- Test Specs
- Resourcing
    - People
    - Funding options

Design considerations, and preliminary work

- Key Reading
    - https://www.postgresql.org/docs/14/protocol-message-formats.html
    - https://wiki.postgresql.org/wiki/Backend_flowchart#tcop
        - https://www.postgresql.org/developer/backend/
    - https://www.postgresql.org/docs/devel/overview.html
- Currently focusing on Solution Option 1 [SO1].
- Design TODO: Where will ImpersonateDatabaseUser message be parsed? (Probably somewhere in tcop)
    - How will ImpersonateDatabaseUser be suppressed/delayed if there is already a query/response in progress? (It should be rejected - the client should wait for a ReadyForQuery message)
- Create a function **VerifyUserPassword**(username, password) based on CheckPasswordAuth implementation
- Design TODO: Create a new GRANT privilege OR perhaps a special “role” that permits Impersonation?
    - this is ideally cached within miscinit.c as bool **IsImpersonationPermitted** alongside CurrentUserId
- Create a function **SetSessionUserId**(username, userid, password)
    - It will check that the authenticated connection user is allowed to impersonate
    - It will lookup userid if necessary (username is supplied but not userid)

    - It will verify the password with **VerifyUserPassword**, if the password is provided
    - It will prepare a myextra structure kind of pattern - see variable.c:792
    - It will eventually call **SetSessionUserId**(userid, is_superuser=false)

Code Investigations

- According to the backend flowchart, postmaster is where the connection starts. According to code investigation, the startup finishes init in the tcop module.
- **How is the username initially set with the Start message?** Short answer: postmaster.c:2233
    - I started in the tcop module - src/backend/tcop/postgres.c
    - StartupMessage doesn't have a header byte, it's implicitly processed as the first message. See StartupMessage in https://www.postgresql.org/docs/14/protocol-message-formats.html
    - The protocol version number is distinct for this message type and should be a useful search term.
    - "FrontendProtocol" seems to be the right variable assigned
    - It looks like tcop is the wrong place to be looking for StartupMessage magic. Postmaster is where the connection startup magic occurs.
    - "ProcessStartupPacket" is looking good in /src/backend/postmaster/postmaster.c
        - This assigns StartupMessage.User to "port->user_name" on line 2233.
        - port is Port type supplied as param. Port is defined in libpq-be.h as "typedef struct Port"
        - port->user_name is limited to NAMEDATALEN (which I think is 64 chars?)
        - GSS is negotiated here if used, but the user password is not checked here
- **How does post->username get to a SQL variable like "current_user"?** Short answer: "BackendRun" function
    - Working backwards.
    - SQL Variables
        - current_user = user - this is what starts the connection
        - session_user - TODO -
    - For parsing current_user is "SVFOP_CURRENT_USER"
    - It might get resolved at: /backend/utils/adt/name.c:263.
        - CStringGetDatum(GetUserNameFromId(GetUserId(), false))
        - GetUserId resolves to CurrentUserId (OID) in miscinit.c:498
        - SetOuterUserId,**SetSessionUserId**,SetUserIdAndSecContext,SetUserIdAndContext sets to **CurrentUserId**
            - SetOuterUserId > SetCurrentRoleId > commands/variable.c - assign_role
            - **SetSessionUserId**
                - **InitializeSessionUserId** - this assigns directly to AuthenticatedUserId before calling SetSessionUserId
                - /utils/init/postinit.c - **InitPostgres** >

            - /tcop/postgres.c - **PostgresMain** > postmaster.c - **BackendRun** - this is where port->username is bound. Called by **BackendStartup** which must be the Main entrypoint.
            - ~~Postmaster.c - BackgroundWorkerInitializeConnection - "Background" seems like a deadend~~
            - ~~BackgroundWorkerInitializeConnectionByOid - probably not used~~
        - ~~InitializeSessionUserIdStandalone - we're not interested in standalone~~
        - ~~SetSessionAuthorization - likely for the SQL command~~
    - **Therefore**, Postmaster links to current_user via **BackendStartup > BackendRun** > PostgresMain > **InitPostgres** > InitializeSessionUserId > SetSessionUserId
        - And that means "SetSessionUserId" is probably the best way to change the active values.
        - InitPostgres has the logic to convert from username to Oid.
- **How is the password validated?** Short answer: InitPostgres
    - Best to start at **BackendStartup -** in postmaster.c:4235
    - AuthenticationCleartextPassword or AuthenticationMD5Password is used.
    - The connection is not valid until after, authentication, so I'm guessing that BackendRun is called after Authentication.
    - Postmaster.c has ClientAuthInProgress bool
    - (Identities are loaded during PostmasterMain)
    - BackendInitialize sets ClientAuthInProgress to true :4421
        - This calls ProcessStartupPacket :4522
        - It doesn't appear to do simple authentication.
    - InitProcess comes after BackendInitialize in BackendStartup
    - Searching for 'R'
    - pq_beginmessage(&buf, 'R'); in sendAuthRequest
        - CheckPasswordAuth calls sendAuthRequest with AUTH_REQ_PASSWORD
        - ClientAuthentication calls CheckPasswordAuth
        - PerformAuthentication calls ClientAuthentication
        - **InitPostgres** calls PerformAuthentication postinit.c:776
            - Then calls InitializeSessionUserId
- **How can we reuse a plain password function?**
    - ClientAuthentication calls CheckPasswordAuth
        - This function gets the password from sendAuthRequest, but we won't
        -
    - If password is set on ImpersonateDatabaseUser then:
        - Refer to /src/backend/libpq/auth.c:784 in CheckPasswordAuth
    - Code reference:
        - char *shadow_pass = get_role_password(port->user_name, logdetail);
        - if (shadow_pass)
        - {
        -   result = plain_crypt_verify(port->user_name, shadow_pass, passwd,

- logdetail);
- }
- //deallocations are necessary
- //result is an integer and should be STATUS_OK