PostgreSQL at Scale: Database Schema Changes with Minimal/No-Downtime

Table of Contents

Brief Overview:

Database schema migrations can be a double-edged sword. They are essential for keeping our systems up to date and in sync with evolving application requirements, but often come bundled with a set of challenges that can give even the most seasoned developers and database administrators a nightmare. There's no one tool or a process to be followed for a seamless, minimal/no-downtime and risk-free release. This session will walk-through different schema release scenarios cut across the different phases of Software Development Life Cycle (SDLC) with relevant tools and best practices for each of them with hands-on demo. The pillars for Schema release would be -

- Ease of use
- Seamless
- Rollback
- Minimal/No-Downtime
- Risk Free

Why Schemas changes are required:

- Support new features or functionality in your application
- Defect fixes
- Improve system performance by optimizing data structures
- Enhance data security through better constraints
- Adapt to evolving business requirements

Schema Changes Scenarios:

The following operations can cause downtime or errors:

1. Adding a column with a non-null default value to an existing table with minimal downtime
2. Changing the data type of a column
3. Renaming a table
4. Removing a column
5. Add a column with unique constraint
6. Running Vacuum Full with minimal downtime using pg_repack
7.Adding/Deleting an index

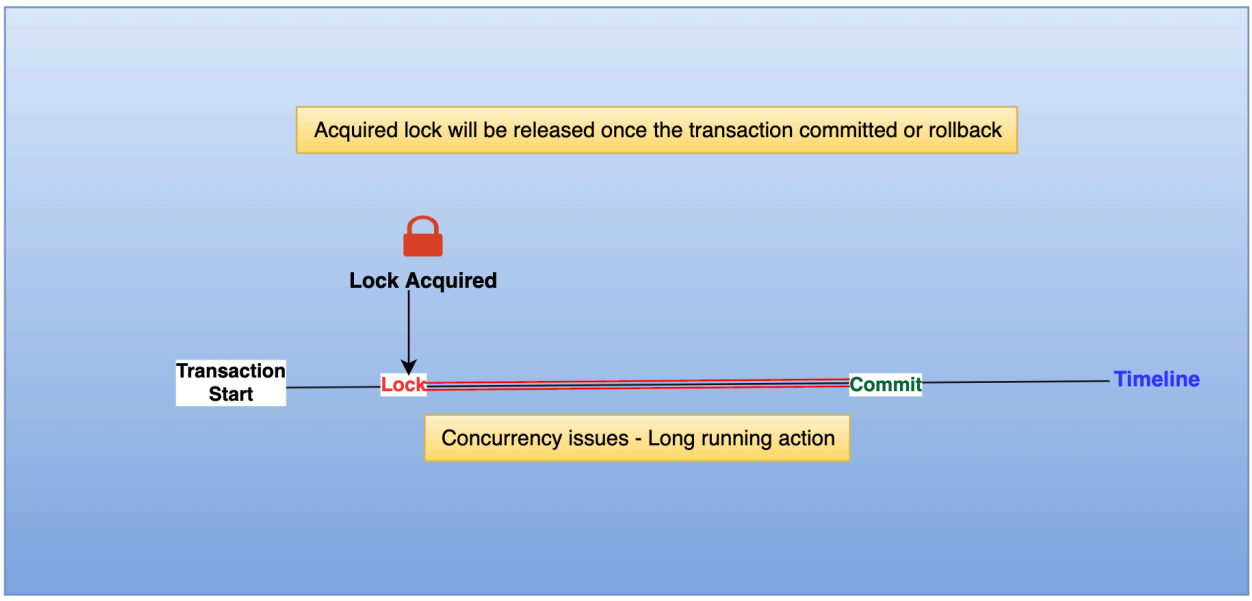
Risks involved in Schema Changes:

Locking:

```
ACCESS EXCLUSIVE
- Blocks ALL operations
- Used by: DROP TABLE, TRUNCATE, VACUUM FULL

EXCLUSIVE
- Blocks most writes and schema changes
- Used by: REFRESH MATERIALIZED VIEW

SHARE ROW EXCLUSIVE
- Blocks concurrent data changes
- Used by: CREATE TRIGGER

SHARE
- Blocks structure changes
- Used by: CREATE INDEX (non-concurrent)

ROW EXCLUSIVE
- Allows concurrent reads
- Used by: UPDATE, DELETE, INSERT
```

For OLTP workloads (such as web and mobile applications), it is important to understand object-level and row-level locks in PostgreSQL. There are several good materials that are recommended for reading:

- The official documentation is a must-read, as usual: "13.3. Explicit Locking" (do not be confused by the title – this article discusses not only explicit locks that we can add using queries like `LOCK TABLE ...` or `SELECT ... FOR UPDATE` or functions like `pg_advisory_lock(..)` but also the locks introduced by regular SQL commands such as `ALTER` or `UPDATE`)
- "PostgreSQL rocks, except when it blocks: Understanding locks" (2018) by Marco Slot – a great extension to the docs, explaining some aspects and conveniently "translating" the table provided in the docs from "lock language" to "SQL command language"



The problem of holding exclusive locks for long can be very daunting. These can be locked rows (implicitly via `UPDATE` or `DELETE` or explicitly via `SELECT .. FOR UPDATE`) or database objects (example: successful `ALTER TABLE` inside a transaction block locks the table and holds the lock until the end of the transaction). If you need to learn more about locks in Postgres, read the article "PostgreSQL rocks, except when it blocks: Understanding locks" by Marco Slot.
An abstract example of the general issue with locking:

```
begin;
alter table t1 add column c123 int8;
-- do something inside or outside of the database (or do
nothing)
commit;
```

The reason for sitting inside a transaction after lock acquisition may vary. However, sometimes it is nothing – a simple waiting with an open transaction and acquired lock. This is the most annoying reason that can quickly lead to various performance or even partial downtime: an exclusive lock to a table blocks even SELECTs to this table.

```
caution
Remember: any lock acquired in a transaction is held until the
very end of this transaction. It is released only when the
transaction finishes, with either COMMIT or ROLLBACK.
```

Every time we acquire an exclusive lock, we should think about finishing the transaction as soon as possible.

Data Integrity issues:

Systems often rely on specific schema structures to maintain data integrity. When you modify these structures, you risk creating situations where incoming data no longer meets your database's requirements. This can lead to failed insertions, constraint violations, and data quality issues.
For example, if you add a new required column without providing a default value, upstream systems might fail to insert new records. Similarly, changing data type constraints might cause previously valid data to be rejected. These issues can be particularly problematic in systems with real-time data ingestion requirements.

Schema changes can introduce subtle data inconsistencies, particularly when dealing with related tables or denormalized data structures.

Inconsistencies in Data:

For example, if you modify a column in one table but forget to update corresponding columns in related tables, you might create mismatches that negatively affect data integrity. It's bad news when inconsistencies pop up in complex systems or those relying on materialized views. The web of relationships quickly becomes overwhelming.
The ripple effects of these inconsistencies often surface in join operations, where mismatched data types or column names can lead to incorrect results or missing data. These issues might not be immediately apparent, only popping up for specific edge cases or when reconciling data across different systems.

Performance Degradation:

Tweaking indexes or data types can serious consequences on performance. Even a seemingly simple change - like adding a new column with a default value to a large table - can cause performance degradation during the modification process. In addition, there could be cascading impact that spreads far and wide. Queries that previously performed well might suddenly slow down as the query optimizer chooses different plans based on the modified schema.

a) Table-level Locking:

- ACCESS EXCLUSIVE locks blocking reads/writes
- Connection pileup due to lock waiting

- Transaction queue buildup

b) I/O Impact:

- Increased disk writes during table rewrites
- Buffer cache pollution
- WAL generation spike

Data Entry conflicts:

When schema changes aren’t properly coordinated with up/down stream data providers, you risk creating misalignments between what these systems are sending and what your database expects to receive. In systems where data pours in from multiple sources, it's trial by fire to manage the flow.

Common data entry issues include:

- Forms and applications submitting data in outdated formats
- ETL processes using incorrect field mappings
- API integrations failing to adapt to new requirements
- Batch processes sending incompatible data structures

Common data entry issues include:

- Forms and applications submitting data in outdated formats
- ETL processes using incorrect field mappings
- API integrations failing to adapt to new requirements
- Batch processes sending incompatible data structures

Mitigations

Use lineage tracking

One of the most effective ways to manage the impacts of schema changes is through data lineage tracking. Data lineage tools can automatically track these relationships, making it easier to assess the potential impact of any proposed changes.

By maintaining a clear map of table-to-table dependencies and column-level relationships, you can better understand how changes will propagate through your system.

Effective lineage tracking should go beyond documenting direct dependencies. It should capture transformation logic, business rules, and data quality requirements. This holistic view enables you to predict how schema changes might affect not just the immediate consumers of your data but also downstream processes and business operations.

Run end-to-end tests and Monitoring Performance

Testing schema changes requires a systematic approach that goes beyond simple functional verification. You need to establish a comprehensive testing strategy that includes regression testing, performance testing, and validation of business logic. This testing should occur in an environment that closely mirrors production, with realistic data volumes and usage patterns.

You can't afford to get bogged down in manual testing. Automated tools are the key to streaming your process, weeding out hiccups, and pushing through to launch.

- Performance testing under various load conditions
- Integration testing with downstream systems
- Validation of business rules and calculations
- Testing of edge cases and error conditions

These should include:

- Data quality validation to verify that transformations maintain data integrity
- Performance testing under various load conditions
- Integration testing with downstream systems
- Validation of business rules and calculations
- Testing of edge cases and error conditions

Performance monitoring should be implemented before, during, and after schema changes. Numbers to watch include query execution times, resource usage, and latency. Tracking your baseline performance metrics before making adjustments will help you instantly spot any areas that need attention.

Performance Optimization Techniques

a) Memory Configuration:

```
    # Adjust work_mem for schema changes
work_mem = 64MB
maintenance_work_mem = 256MB
```

b) Autovacuum Settings:

```
    autovacuum_vacuum_scale_factor = 0.1
autovacuum_analyze_scale_factor = 0.05
```

Implementation Strategies

a) Pre-migration Optimization:

```
    -- Analyze table before changes
VACUUM ANALYZE table_name;

-- Remove dead tuples
VACUUM FULL table_name;
```

Ensure Data Consistency

Maintaining data consistency during schema changes requires a multi-faceted approach. Before you begin, assess the connections between affected tables and columns, considering both the rules set in stone and the unwritten rules of the business. You need to be surgical when making changes to your data model to avoid downstream failures.

For example, if you modify a customer identifier format in one table, you need to make sure that this change is reflected in all related tables and views. This might involve creating temporary staging tables, implementing parallel processing strategies, or using transitional periods where both old and new formats are supported. It's time-consuming, but being thorough about proactively maintaining data quality and consistency can prevent even bigger problems down the line.

Few other Mitigations:

- Deploy changes during low-traffic window and have a rollback plan
- Use staging environments
- Implement versioning
- Communicate with stakeholders

Considerations during Schema Changes:

Plan for Rollbacks: Always have a rollback plan in place. If a migration does not go

as planned, being able to revert to the previous state quickly can save time and prevent user disruption

- Maintain old structure until verification
- Keep backup of original indexes
- Monitor application performance metrics

Monitor Performance: Schema changes can be resource -intensive, Monitoring is crucial to prevent performance issues. After deploying schema changes, monitor the performance of the application closely. Look for any signs of locking or slow queries that may indicate issues stemming from the migration.

Ensure Backward Compatibility:
When making changes, ensure that both old and new code can run simultaneously. This is crucial in environments where both versions of the application are operational, as breaking changes can lead to outages.

Other Important Considerations:

1. Always backup before schema changes
2. Test migrations in staging environment
3. Consider impact on foreign keys and constraints
4. Plan for adequate maintenance window

By following these practices and understanding the potential inconsistencies, you can minimize risks during online schema changes while maintaining data integrity and application availability.

Remember that PostgreSQL 15+ offers improved concurrent DDL operations, making some of these problems easier to handle, but careful planning is still essential for production environments.

Testing and Performance Considerations:

Query Analysis: Utilize `pg_stat_statements` to identify hot or slow queries, which can help in optimizing performance.

Load Testing: Conduct load testing preferably on a staging environment. Tools like k6 can simulate traffic from multiple users, allowing you to assess how your database performs under stress.

Traffic Surge Preparation: Anticipate surges in traffic, especially during significant launches.

Simulating Schema Changes

Below Is a schema change use-case illustrating the impacts on the system from performance and resource contention standpoint

Example-1

Add column to the transactional table which will be almost busy during the business hours.

Impacts:

Table Locking

Depending on the database system, adding a column might lock the entire table. This prevents other transactions from accessing the table and can cause application timeouts and business disruptions

Performance Impact

- Heavy CPU usage during the operation
- Increased I/O operations
- Slower response times for other queries
- Potential system-wide performance degradation

```
top - 10:49:55 up 29 days,  4:44, 10 users,  load average: 20.33, 16.56, 10.05
Tasks: 193 total,  24 running, 169 sleeping,   0 stopped,   0 zombie
%Cpu(s): 14.7 us, 11.8 sy,  0.0 ni,  4.9 id, 66.6 wa,  0.0 hi,  1.2 si,  0.8 st
MiB Mem :   3904.2 total,    516.5 free,    348.7 used,   3039.0 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3154.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1648745 postgres  20   0  381124   5480   4656 S  10.0   0.1   3:13.22 pgbench
    155 root       0 -20       0      0      0 I   4.3   0.0   0:35.52 kworker/1:1H-kblockd
     46 root       0 -20       0      0      0 I   1.7   0.0   0:27.91 kworker/0:1H-kblockd
1647125 postgres  20   0  427480  17284  14412 S   1.7   0.4   0:28.67 postgres
1648754 postgres  20   0  427252 151832 149024 R   1.3   3.8   0:19.67 postgres
1648755 postgres  20   0  427252 151876 149036 R   1.3   3.8   0:19.27 postgres
1648756 postgres  20   0  427252 151548 148744 R   1.3   3.8   0:19.29 postgres
```

Space Issues

Immediate storage requirement if adding with default values, table size increase might cause storage problems

Tools and Best Practices Process:

Liquibase

Liquibase is an open-source database schema change management and version control tool that helps track, version, and deploy database changes. It's particularly useful in DevOps environments and for teams working on applications that require database modifications.

Key Features of Liquibase:

- Version Control
- Tracks database changes through change sets
- Maintains a history of all database modifications
- Enables rollback capabilities

Format Supported

- XML
- YAML
- JSON
- SQL
- Format-specific change log files

Flyway

Flyway is a database migration tool that helps manage database schema changes

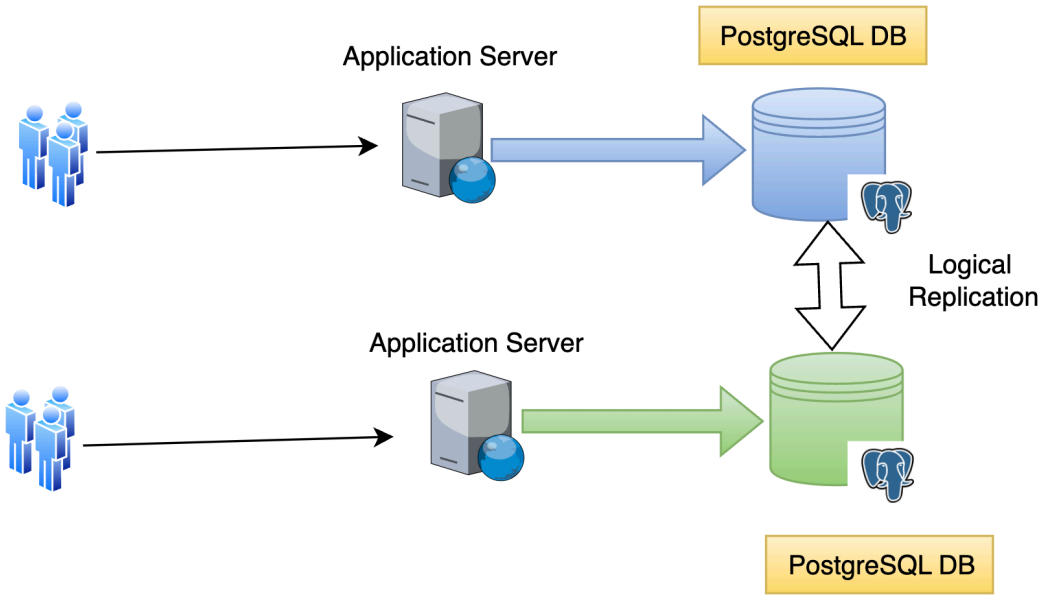
across environments. It works by applying versioned migrations in order, tracking what has been applied in a special table called "flyway_schema_history".

Key Features:

- Simple version-based migrations
- SQL and Java-based migrations
- Command-line and API support
- Easy integration with build tools
- Schema history tracking

Blue green Deployment

Blue-green deployment is an application release strategy that uses two identical production environments — referred to as “blue” (live) and “green” (standby) — to achieve reliable testing, zero-downtime releases, and instant rollbacks. At any given time, only one of these environments is live.
For example, the blue environment handles all live production traffic while the green environment remains idle. When a new application version is ready, it is deployed to the green environment for testing. Once the release passes testing, traffic is seamlessly rerouted from blue to green.



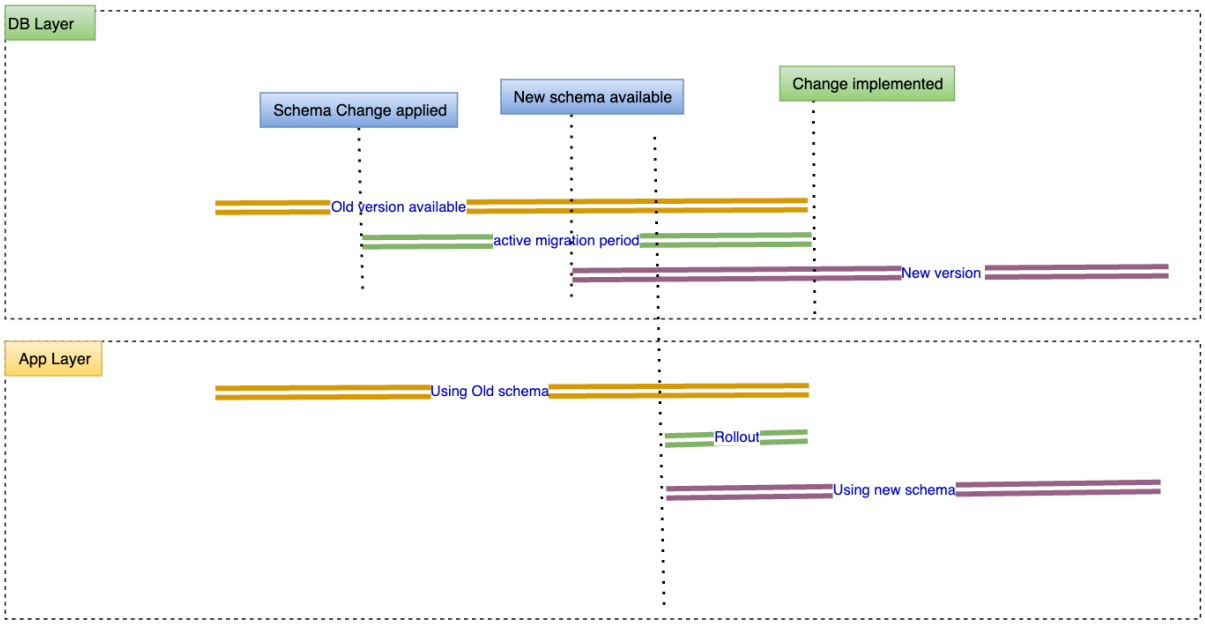
At this point, the green environment becomes the new live production environment, and the blue environment goes idle. This deployment strategy ensures that any issues

that arise during or after the transition can be easily mitigated by reverting traffic back to the previous (blue) version.

pgroll

PgRoll is a specialized database schema migration tool designed specifically for PostgreSQL, focusing on zero-downtime migrations. Created by Xata, it addresses the critical need for performing complex database changes without service interruption. Unlike traditional migration tools, pgroll implements sophisticated strategies to maintain database availability while ensuring data consistency throughout the migration process.

The tool represents a significant advancement in database DevOps, particularly for organizations running mission-critical applications where downtime is not acceptable. It combines the reliability of traditional migration tools with modern zero-downtime requirements, making it invaluable for continuous deployment environments.



How to setup pgroll

Pre-Requisites:

To install the pgroll go language should be installed in the server whose version should be 1.23 and above.

Steps to install go and pgroll

1. Download appropriate sources/binaries need compatible with your OS from below link:
    a. https://go.dev/doc/install and follow steps present in the link to install go.
2. Once the go is installed install pgroll

```
mkdir /opt/pgroll
cd /opt/pgroll
go install github.com/xataio/pgroll@latest
```

3. Check the connection with PostgreSQL and install initial schema required for pgroll

```
pgroll init --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le
```

```
[postgres@ip-172-31-17-243 ~]$ pgroll init --postgres-url postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disable
SUCCESS Initialization complete
[postgres@ip-172-31-17-243 ~]$
[postgres@ip-172-31-17-243 ~]$
[postgres@ip-172-31-17-243 ~]$
[postgres@ip-172-31-17-243 ~]$
[postgres@ip-172-31-17-243 ~]$ psql
psql (15.9)
Type "help" for help.

postgres=# \c pgconf
You are now connected to database "pgconf" as user "postgres".
pgconf=# \dn
      List of schemas
  Name   |       Owner
---------+--------------------
 pgroll | postgres
 public | pg_database_owner
(2 rows)
```

Note: In case if pgroll is not in PATH then set the PATH in .bash_profile for Postgres user.

pgroll will create its own scheme and in it there will be table by name “migrations” which will hold the information of all tables which have entry of all changes

```
pgconf=# select * from migrations  limit 1;
-[ RECORD 1 ]-----+-----------------------------------------------------------------------------------------------------------------------------
schema            | public_02_change_column_type
name              | sql_39b9eb04edb340
migration         | {"name": "sql_39b9eb04edb340", "operations": [{"sql": {"up": "drop schema public_02_change_column_type cascade;"}}]}
created_at        | 2025-01-22 07:10:19.748752
updated_at        | 2025-01-22 07:10:19.748752
parent            |
done              | t
resulting_schema  | {"name": "public_02_change_column_type", "tables": {}}
migration_type    | inferred
```

Example Schema Change Scenarios using pgroll

1. Adding a column with a not-null default value to an existing table with minimal downtime

Add column

An add column operation creates a new column on an existing table.

Structure

```
{
  "add_column": {
    "table": "name of table to which the column should be
added",
    "up": "SQL expression",
    "column": {
      "name": "name of column",
      "type": "postgres type",
      "comment": "postgres comment for the column",
      "nullable": true|false,
      "unique": true|false,
      "pk": true|false,
      "default": "default value for the column",
      "check": {
        "name": "name of check constraint",
        "constraint": "constraint expression"
      },
      "references": {
        "name": "name of foreign key constraint",
        "table": "name of referenced table",
        "column": "name of referenced column",
        "on_delete": "ON DELETE behaviour, can be CASCADE, SET
NULL, RESTRICT, or NO ACTION. Default is NO ACTION",
      }
    }
  }
}
```

The `up` SQL is used when a row is added to the old version of the table (ie, without the new column). In the new version of the table, the row's value for the new column is determined by the `up` SQL expression. The `up` SQL can be omitted if the new column is nullable or has a `DEFAULT` defined.

Default values are subject to the usual rules for quoting SQL expressions. In particular, string literals should be surrounded with single quotes.

NOTE: As a special case, the `up` field can be omitted when adding `smallserial`, `serial` and `bigserial` columns.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100)
);
```

```
ALTER TABLE users
ADD COLUMN email VARCHAR(255);
```

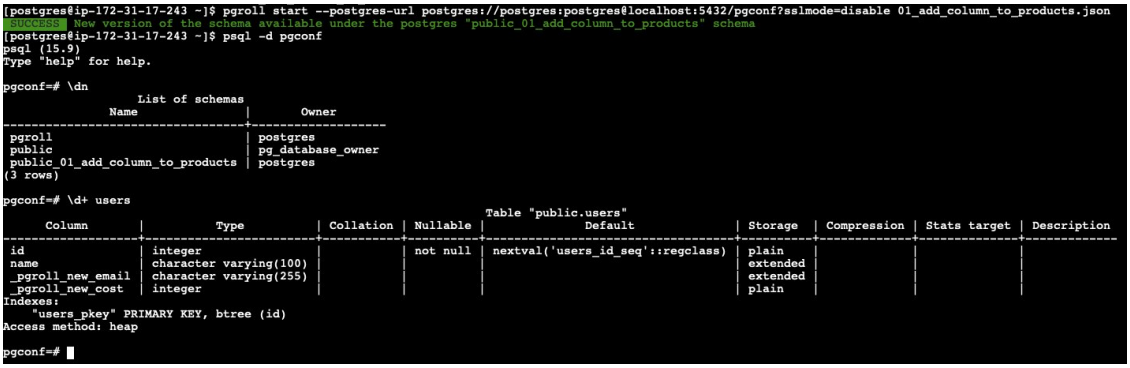
Example:

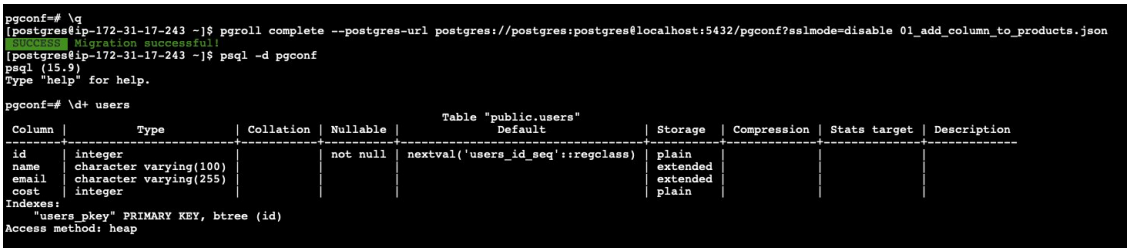
```
vi 01_add_column_to_products.json
```

```
{
  "name": "01_add_column_to_products",
  "operations": [
    {
      "add_column": {
        "table": "users",
        "column": {
          "name": "email",
          "type": "varchar(255)",
          "nullable": true
        }
      }
    },
        {
      "add_column": {
        "table": "users",
        "column": {
          "name": "cost",
          "type": "int",
          "nullable": true
        }
      }
    }
  ]
}
```

Execution of schema change

```
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 01_add_column_to_products.json
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 01_add_column_to_products.json
```





2. Changing the data type of a column

Change Datatype

A change type operation changes the type of a column.

Structure

```
{
  "alter_column": {
    "table": "table name",
    "column": "column name",
    "type": "new type of column",
```

```
    "up": "SQL expression",
    "down": "SQL expression"
  }
}
```

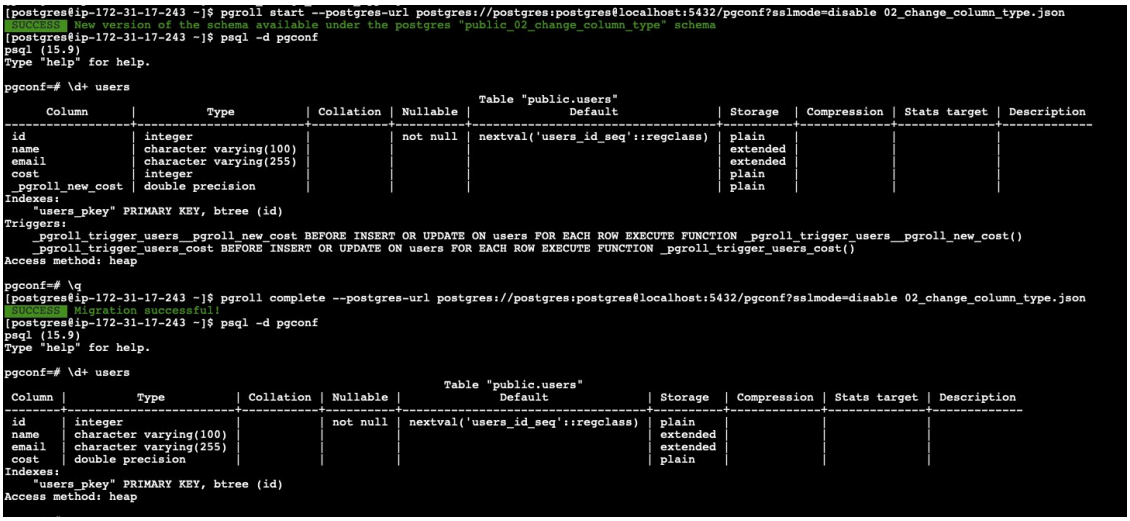
Use the up SQL expression to do data conversion from the old column type to the new type. In the old schema version, the column will have its old data type; in the new version the column will have its new type.
Use the down SQL expression to do data conversion in the other direction; from the new data type back to the old.

Example:

```
ALTER TABLE users
alter COLUMN type bigint;

vi 02_change_column_type.json

{
  "name": "02_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "users",
        "column": "cost",
        "type": "float",
        "up":"float",
        "down":"integer"
      }
    }
  ]
}
```

Execution of schema change

```
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 02_change_column_type.json
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 02_change_column_type.json
```



3. Renaming a table

Rename table

A rename table operation renames a table.

Structure

```
{
  "rename_table": {
    "from": "old column name",
    "to": "new column name"
  }
}
```

The table is accessible by its old name in the old version of the schema, and by its new name in the new version of the schema.
The table itself is renamed on migration completion.

Example:

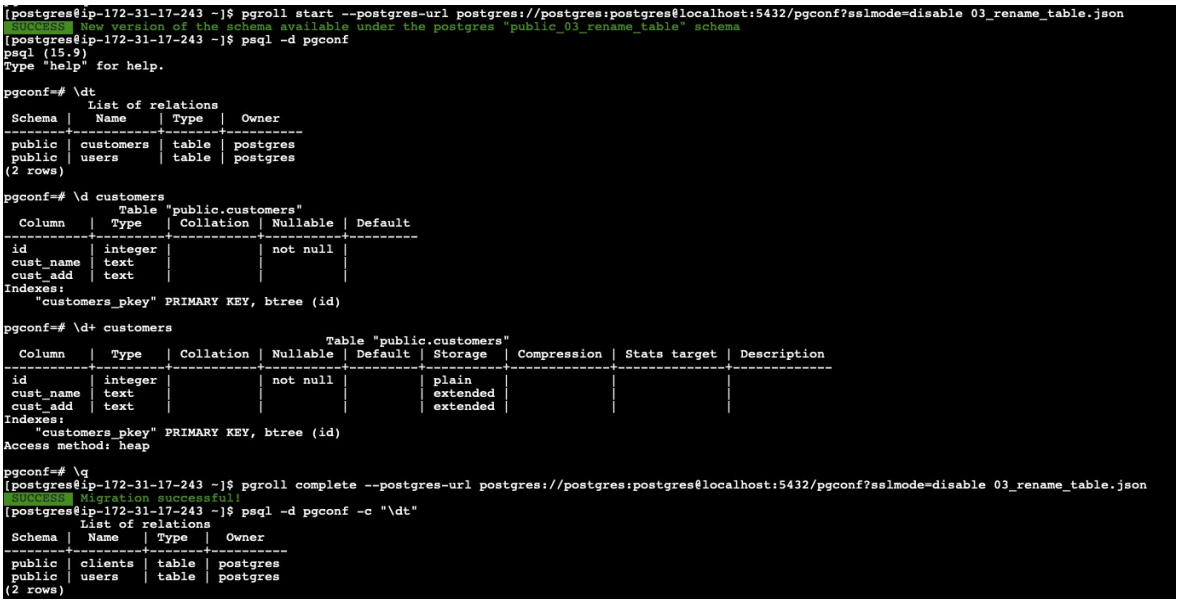
```
pgconf=# create table customers(id int primary key, cust_name
text,cust_add text);
CREATE TABLE
pgconf=# \d customers
Table "public.customers"
Column | Type | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
```

```
id | integer | | not null |
cust_name | text | | |
cust_add | text | | |
Indexes:
"customers_pkey" PRIMARY KEY, btree (id)

pgconf=# \q
[postgres@ip-172-31-17-243 ~]$
```

```
vi 03_rename_table.json
{
  "name": "03_rename_table",
  "operations": [
    {
      "rename_table": {
        "from": "customers",
        "to": "clients"
      }
    }
  ]
}
```

Execution of schema change

```
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 03_rename_table.json
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 03_rename_table.json
```



4. Removing a column

Drop column

A drop column operation drops a column from an existing table.

Structure

```
{
  "drop_column": {
    "table": "name of table",
    "column": "name of column to drop",
    "down": "SQL expression"
  }
}
```

The `down` field above is required in order to back fill the previous version of the schema during an active migration.

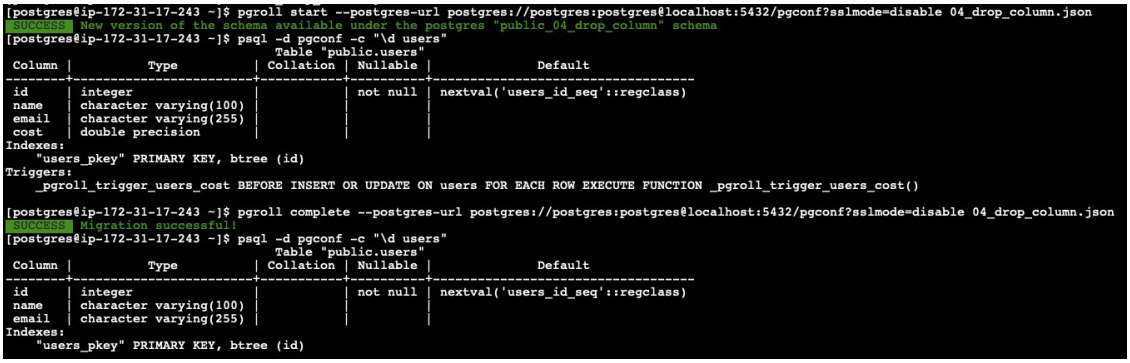
Example:

```
vi 04_drop_column.json
{
  "name": "04_drop_column",
  "operations": [
```

```
    {
      "drop_column": {
        "table": "users",
        "column": "cost",
        "down": "0"
      }
    }
  ]
}
```

Execution of schema change

```
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 04_drop_column.json
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 04_drop_column.json
```



Add a column with unique constraint

Add unique constraint

Add unique operations add a `UNIQUE` constraint to a column.

Structure

```
{
  "alter_column": {
    "table": "table name",
    "column": "column name",
    "unique": {
      "name": "name of unique constraint"
```

```
    },
    "up": "SQL expression",
    "down": "SQL expression"
  }
}
```

Use the `up` SQL expression to migrate values from the old non-unique column in the old schema to the `UNIQUE` column in the new schema.

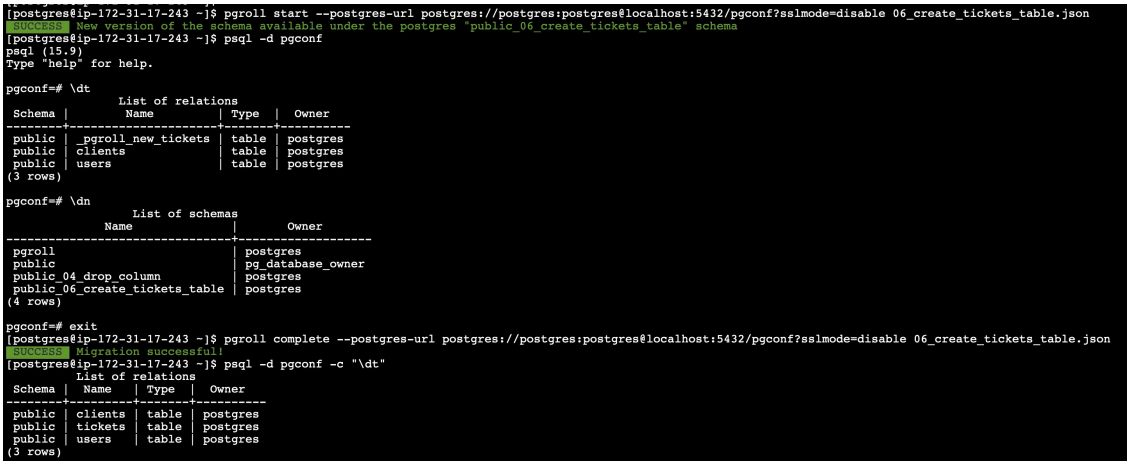
Example:

```
vi 06_create_tickets_table.json
```

```
{
  "name": "06_create_tickets_table",
  "operations": [
    {
      "create_table": {
        "name": "tickets",
        "columns": [
          {
            "name": "ticket_id",
            "type": "serial",
            "pk": true
          },
          {
            "name": "sellers_name",
            "type": "varchar(255)"
          },
          {
            "name": "sellers_zip",
            "type": "integer"
          },
          {
            "name": "ticket_type",
            "type": "varchar(255)",
            "default": "'paper'"
          }
        ]
      }
    }
  ]
}
```

Execution of schema change

```
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_create_tickets_table
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_create_tickets_table
```




```
vi 06_01_add_table_unique_constraint.json
{
  "name": "06_01_add_table_unique_constraint",
  "operations": [
    {
      "create_constraint": {
        "type": "unique",
        "table": "tickets",
        "name": "unique_zip_name",
        "columns": [
          "sellers_name",
          "sellers_zip"
        ],
        "up": {
          "sellers_name": "sellers_name",
          "sellers_zip": "sellers_zip"
        },
        "down": {
```

```
            "sellers_name": "sellers_name",
            "sellers_zip": "sellers_zip"
          }
        }
      }
    ]
}
pgroll start --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_01_add_table_unique_constraint.json
pgroll complete --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_01_add_table_unique_constraint.json
```

Other command line options are available to rollback, check the status

Rollback

Roll back the currently active migration.

Command

```
$ pgroll rollback --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_01_add_table_unique_constraint.json
```

This rolls back the currently active migration (an active migration is one that has been started but not yet completed).
Rolling back a pgroll migration means removing the new schema version. The old schema version was still present throughout the migration period and does not require modification.
Migrations cannot be rolled back once completed. Attempting to roll back a migration that has already been completed is a no-op.

Status

Show the current status of pgroll within a given schema.
Command

```
$ pgroll status --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_01_add_table_unique_constraint.json
```

The status field can be one of the following values:

- `"No migrations"` - no migrations have been applied in this schema yet.
- `"In progress"` - a migration has been started, but not yet completed.
- `"Complete"` - the most recent migration was completed.

The `Version` field gives the name of the latest schema version.
If a migration is `In progress` the schemas for both the latest version indicated by the `Version` field and the previous version will exist in the database.
If a migration is `Complete` only the latest version of the schema will exist in the database.
The top-level `--schema` flag can be used to view the status of `pgroll` in a different schema:

```
$ pgroll status --postgres-url
postgres://postgres:postgres@localhost:5432/pgconf?sslmode=disab
le 06_01_add_table_unique_constraint.json --schema public
```

For more details on pgroll

pg-osc
pg_osc, or PostgreSQL Online Schema Change – an ***open-source tool*** designed to revolutionize the execution of schema modifications without imposing substantial downtime or locking the entire database. This tool proves invaluable for making

crucial adjustments to the database structure, ranging from adding or removing columns to changing data types and creating or dropping indexes.

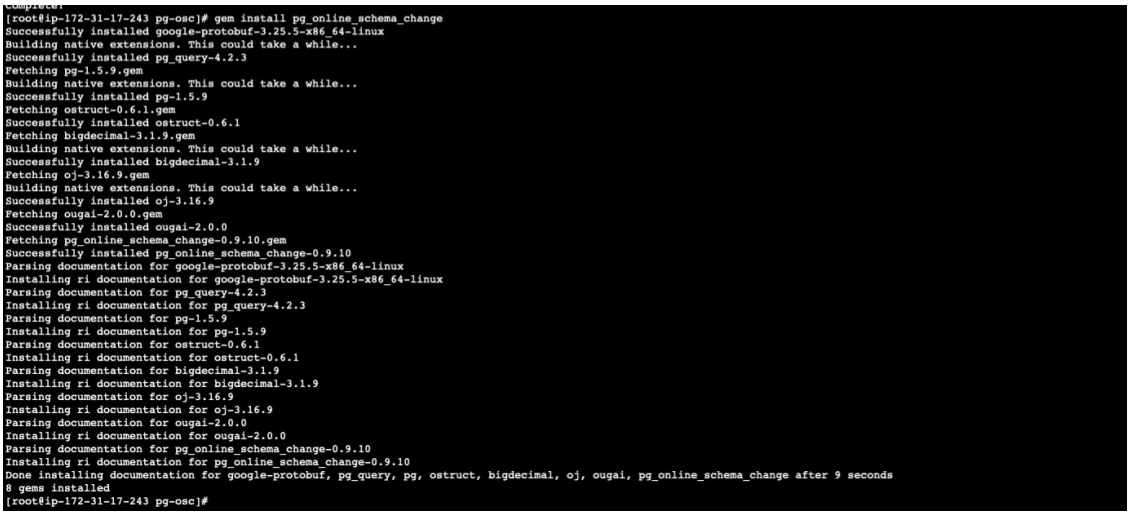
Pre-Requisites

1. Clone the pg-osc repository and build extension:

```
yum install ruby* gem* -y
gem update --system 3.6.3
```

```
git clone https://github.com/shayonj/pg-osc.git
```

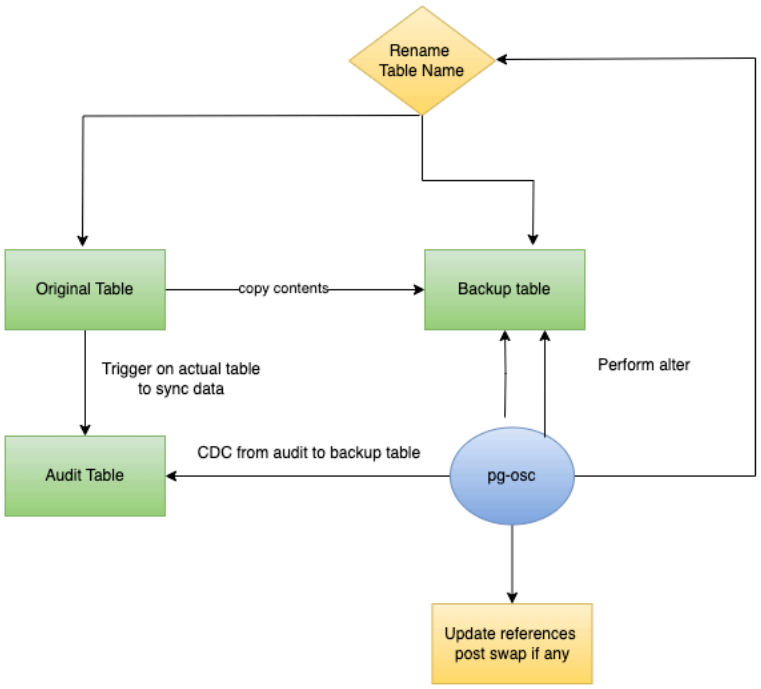
```
cd pg-osc
gem install pg_online_schema_change
```



Set path in the .bash_profiler for PostgreSQL user

How does it work

- **Primary table**: A table against which a potential schema change is to be run
- **Shadow table**: A copy of an existing primary table
- **Audit table**: A table to store any updates/inserts/delete on a primary table



Use case for pg-osc

Multiple ALTER statements

```
pg-online-schema-change perform --alter-statement 'ALTER TABLE
users ADD COLUMN "gender" BOOLEAN DEFAULT FALSE; ALTER TABLE
users ADD COLUMN "last_name" varchar(20);' --dbname "pgconf" --
host "localhost" --username "postgres" --drop
```

```
[postgres@ip-172-31-17-243 bin]$ psql -d pgconf -c "\d users"
                    Table "public.users"
 Column |          Type          | Collation | Nullable |                    Default
--------+------------------------+-----------+----------+-----------------------------------------------
 id     | integer                |           | not null | nextval('pgosc_st_users_9bd652_id_seq'::regclass)
 name   | character varying(100) |           |          |
 email  | character varying(255) |           |          |
 price  | double precision       |           |          |
Indexes:
    "pgosc_st_users_9bd652_pkey" PRIMARY KEY, btree (id)

[postgres@ip-172-31-17-243 bin]$ pg-online-schema-change perform --alter-statement 'ALTER TABLE users ADD COLUMN "gender" BOOLEAN DEFAULT FALSE; ALTER TABLE users ADD COLUMN "last_name"
varchar(20);' --dbname "pgconf" --host "localhost" --username "postgres" --drop > /tmp/multiple_alter.log 2<&1
[postgres@ip-172-31-17-243 bin]$ psql -d pgconf -c "\d users"
                    Table "public.users"
  Column   |          Type          | Collation | Nullable |                    Default
-----------+------------------------+-----------+----------+-----------------------------------------------
 id        | integer                |           | not null | nextval('pgosc_st_users_27958a_id_seq'::regclass)
 name      | character varying(100) |           |          |
 email     | character varying(255) |           |          |
 price     | double precision       |           |          |
 gender    | boolean                |           |          | false
 last_name | character varying(20)  |           |          |
Indexes:
    "pgosc_st_users_27958a_pkey" PRIMARY KEY, btree (id)
```

Renaming the column

```
pg-online-schema-change perform --alter-statement 'ALTER TABLE
```

```
users RENAME COLUMN "price" TO "cost"' --dbname "pgconf" --host
"localhost" --username "postgres"  --drop
```

Difference between all above listed tools:

This is each tool's high-level features, use cases, and considerations to improve an application's operational performance and availability. We don't recommend one tool over another; you should choose the best tool depending on your use case.

Monitoring

Online schema changes are very critical and have high impact on the live system, it's very important to have a robust monitoring system. Below are few areas that should be should be taken care during schema changes

Feature/Tool	pg-osc	pgroll	Flyway	pg_migrate	Liquibase	Blue-Green
Primary Purpose	Online schema changes	Zero-downtime schema changes	Database migration version control	Database migration	Database migration version control	Deployment strategy
Database Support	PostgreSQL only	PostgreSQL only	Multiple DBs	PostgreSQL only	Multiple DBs	Any
Language/Platform	Python	Go	Java	Node.js	Java	N/A
Configuration Format	SQL	YAML	SQL	JavaScript/SQL	XML, YAML, JSON, SQL	N/A
Learning Curve	Moderate	Moderate	Low	Low	High	Moderate
Best For	Large table alterations	Modern PostgreSQL apps	Java projects	Node.js projects	Enterprise apps	Any application
Version Control	No	Yes	Yes	Yes	Yes	N/A
Rollback Support	Yes	Yes	Limited	Yes	Yes	Yes
Zero-Downtime	Yes	Yes	No	No	No	Yes
Enterprise Features	Limited	Yes	Basic	Limited	Extensive	N/A
Community Support	Moderate	Growing	Large	Moderate	Large	N/A
Release Year	2015	2023	2010	2014	2006	N/A
Scripting Support	Limited	Yes	Yes	Yes	Yes	N/A
Documentation Quality	Good	Good	Excellent	Good	Excellent	N/A
CI/CD Integration	Limited	Good	Excellent	Good	Excellent	Excellent
Cost	Free	Free	Free/Commercial	Free	Free/Commercial	N/A

Locks:

Since, locks are the most critical element in schema changes, ensure you've the most robust lock monitoring system/tools in place.

- "Active Session History in PostgreSQL: blocker and wait chain" by Bertrand Drouvot – this post describes the recursive CTE query `pg_ash_wait_chain.sql` that is useful for those who use the pgsentinel extension. The query is inspired by Tanel Poder's script for Oracle.
- `locktree.sql` – query to display a tree of blocking sessions based on the information from `pg_locks` and `pg_stat_activity`, by Victor Yegorov.

Query

```
with recursive activity as (
  select
    pg_blocking_pids(pid) blocked_by,
    *,
    age(clock_timestamp(), xact_start)::interval(0) as tx_age,
    -- "pg_locks.waitstart" - PG14+ only; for older
versions:  age(clock_timestamp(), state_change) as wait_age
    age(clock_timestamp(), (select max(l.waitstart) from
pg_locks l where a.pid = l.pid))::interval(0) as wait_age
  from pg_stat_activity a
  where state is distinct from 'idle'
), blockers as (
  select
    array_agg(distinct c order by c) as pids
  from (
    select unnest(blocked_by)
    from activity
  ) as dt(c)
), tree as (
  select
    activity.*,
    1 as level,
    activity.pid as top_blocker_pid,
    array[activity.pid] as path,
    array[activity.pid]::int[] as all_blockers_above
  from activity, blockers
  where
    array[pid] <@ blockers.pids
    and blocked_by = '{}'::int[]
  union all
  select
    activity.*,
    tree.level + 1 as level,
    tree.top_blocker_pid,
    path || array[activity.pid] as path,
    tree.all_blockers_above || array_agg(activity.pid) over ()
as all_blockers_above
  from activity, tree
  where
    not array[activity.pid] <@ tree.all_blockers_above
    and activity.blocked_by <> '{}'::int[]
    and activity.blocked_by <@ tree.all_blockers_above
)
select
```

```
  pid,
  blocked_by,
  case when wait_event_type <> 'Lock' then replace(state, 'idle
in transaction', 'idletx') else 'waiting' end as state,
  wait_event_type || ':' || wait_event as wait,
  wait_age,
  tx_age,
  to_char(age(backend_xid), 'FM999,999,999,990') as xid_age,
  to_char(2147483647 - age(backend_xmin), 'FM999,999,999,990')
as xmin_ttf,
  datname,
  usename,
  (select count(distinct t1.pid) from tree t1 where
array[tree.pid] <@ t1.path and t1.pid <> tree.pid) as blkd,
  format(
    '%s %s%s',
    lpad('[' || pid::text || ']', 9, ' '),
    repeat('.', level - 1) || case when level > 1 then ' ' end,
    left(query, 1000)
  ) as query
from tree
order by top_blocker_pid, level, pid

\watch 10
```

CPU/Memory:

For schema changes scenarios involving huge tables might be resource intensive operation especially with high CPU/Memory consumption. These kinds of operations might choke the system and be detrimental to the overall health of the system. It's highly recommended to have a robust monitoring system to monitor the CPU/Memory usage, the common metrics to observe are CPU Utilization, Load Average amongst the top OS processes.

Disk Space:

In few of the schema changes operations/task, there would be a need to create a duplicate table, hence it's important to keep a close eye on the available disk space using your existing monitoring setup/tool.

Live Demo

Customer Case Studies

Adding check constraint on the existing column

Orders Table Structure

```
load=# \dt
          List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | customers     | table | postgres
 public | order_items   | table | postgres
 public | orders        | table | postgres
(3 rows)

load=# \d orders
                                        Table "public.orders"
      Column      |            Type             | Collation | Nullable |                  Default
------------------+-----------------------------+-----------+----------+-----------------------------------------
 order_id         | integer                     |           | not null | nextval('orders_order_id_seq'::regclass)
 customer_id      | integer                     |           |          |
 order_date       | timestamp without time zone |           |          | CURRENT_TIMESTAMP
 total_amount     | numeric(10,2)               |           |          |
 payment_method   | character varying(20)       |           |          |
 status           | character varying(20)       |           |          |
 shipping_address | text                        |           |          |
 delivery_date    | date                        |           |          |
 created_at       | timestamp without time zone |           |          | CURRENT_TIMESTAMP
 updated_at       | timestamp without time zone |           |          | CURRENT_TIMESTAMP
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id)
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
Referenced by:
    TABLE "order_items" CONSTRAINT "order_items_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(order_id)

load=#
```

Orders data

```
load=# ;
 order_id | customer_id |        order_date         | total_amount | payment_method |   status   |    shipping_address    | delivery_date |        created_at         |        updated_at
----------+-------------+---------------------------+--------------+----------------+------------+------------------------+---------------+---------------------------+---------------------------
        1 |           1 | 2025-02-15 09:03:08.912413 |       299.99 | credit_card    | pending    | 123 Main St, City      | 2025-02-18    | 2025-02-15 09:03:08.912413 | 2025-02-15 09:03:08.912413
        2 |           2 | 2025-02-15 09:03:08.912413 |       159.50 | paypal         | processing | 456 Oak Ave, Town      | 2025-02-17    | 2025-02-15 09:03:08.912413 | 2025-02-15 09:03:08.912413
        3 |           3 | 2025-02-15 09:03:08.912413 |       499.99 | credit_card    | delivered  | 789 Pine Rd, Village   | 2025-02-14    | 2025-02-15 09:03:08.912413 | 2025-02-15 09:03:08.912413
        4 |           4 | 2025-02-15 09:03:08.912413 |       499.99 |                | delivered  | 789 Pine Rd, Village   | 2025-02-14    | 2025-02-15 09:03:08.912413 | 2025-02-15 09:03:08.912413
(4 rows)
```

Observe payment_method for the 4 order_id its null value is added it is difficult to understand which is payment method?

How to resolve this issue?

1. Add default value (if anything is not added then default should be cash)or
2. Add check constraint (apart from given values anything comes then reject it along with null).

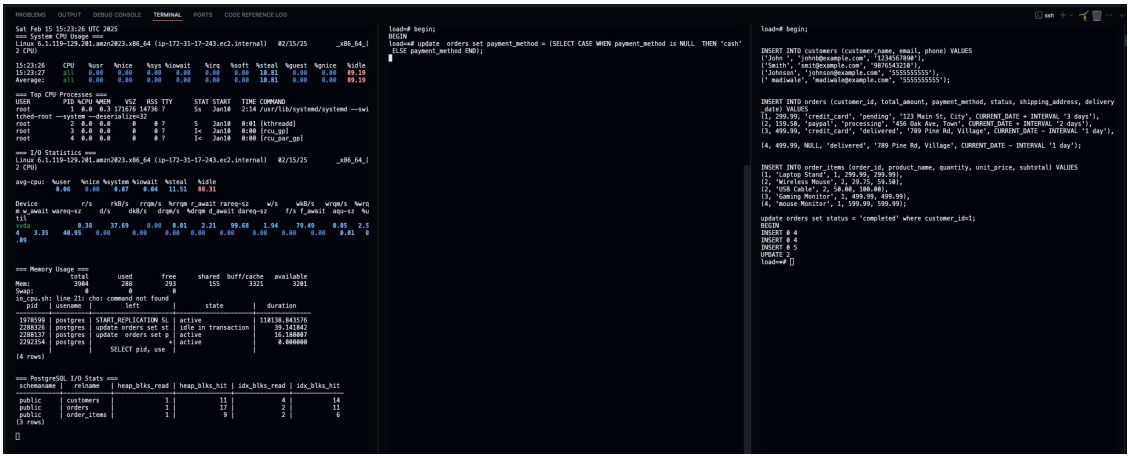
Consider you have application which needs to validate the new corrected data and also you need validation to avoid unknown values. To do this during business hours we need to do UPDATE to existing NULL to default values i.e. Cash and ALTER to add check constraint. Once it is done application should be able to see new value.

What is impact of doing it during peak hours

1. Tables may get lock/query performance.
2. CPU impact.
3. Validation of application (if values are migrated properly or not).

Test-1 Without schema change tool

1. Customer, Orders tables were occupied in one transaction.
2. Update executed on orders table to update NULL status to cash and in waiting state
3. Alter table to add constraint
4. In parallel generating load using pgbench (which was locked and also alter was locked)



Blocked:

Once transaction completed it released the lock, altered the table and pgbench is running.

Test-2 With Schema change tool

1. Json file(to update/add check constraint)
2. Execution of migration
3. pgbench running update (which is locked for some very less time)

```
[postgres@ip-172-31-17-243 ~]$ cat 101_add_table_check_constraint.json
{
  "name": "101_add_table_check_constraint",
  "operations": [
    {
      "create_constraint": {
        "type": "check",
        "table": "orders",
        "name": "chk_valid_payment_method",
        "columns": [
          "payment_method"
        ],
        "check": "payment_method IN ('credit_card', 'paypal', 'bank_transfer', 'cash') AND payment_method is not null",
        "up": {
          "payment_method": "SELECT CASE WHEN payment_method is NULL  THEN 'cash' ELSE payment_method END"
        },
        "down": {
          "payment_method": "payment_method"
        }
      }
    }
  ]
}
```

Orders table (new column added by pgroll):

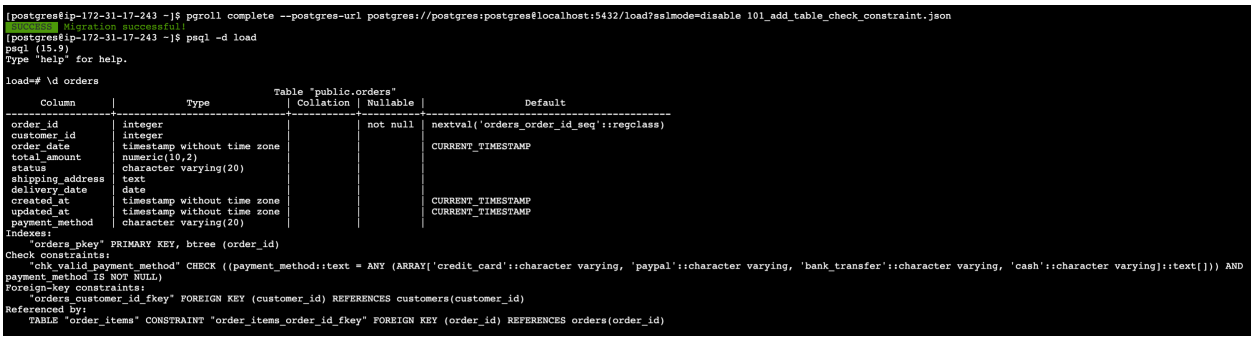
```
load=# \d orders
                                        Table "public.orders"
          Column           |            Type             | Collation | Nullable |               Default
---------------------------+-----------------------------+-----------+----------+--------------------------------------
 order_id                  | integer                     |           | not null | nextval('orders_order_id_seq'::regclass)
 customer_id               | integer                     |           |          |
 order_date                | timestamp without time zone |           |          | CURRENT_TIMESTAMP
 total_amount              | numeric(10,2)               |           |          |
 payment_method            | character varying(20)       |           |          |
 status                    | character varying(20)       |           |          |
 shipping_address          | text                        |           |          |
 delivery_date             | date                        |           |          |
 created_at                | timestamp without time zone |           |          | CURRENT_TIMESTAMP
 updated_at                | timestamp without time zone |           |          | CURRENT_TIMESTAMP
 _pgroll_new_payment_method | character varying(20)      |           |          |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id)
Check constraints:
    "chk_valid_payment_method" CHECK ((_pgroll_new_payment_method::text = ANY (ARRAY['credit_card'::character varying, 'paypal'::character varying, 'bank_transfer'::
character varying, 'cash'::character varying]::text[])) AND _pgroll_new_payment_method IS NOT NULL) NOT VALID
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
Referenced by:
    TABLE "order_items" CONSTRAINT "order_items_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(order_id)
Triggers:
    _pgroll_trigger_orders__pgroll_new_payment_method BEFORE INSERT OR UPDATE ON orders FOR EACH ROW EXECUTE FUNCTION _pgroll_trigger_orders__pgroll_new_payment_meth
od()
    _pgroll_trigger_orders_payment_method BEFORE INSERT OR UPDATE ON orders FOR EACH ROW EXECUTE FUNCTION _pgroll_trigger_orders_payment_method()
```

New column added by pgroll

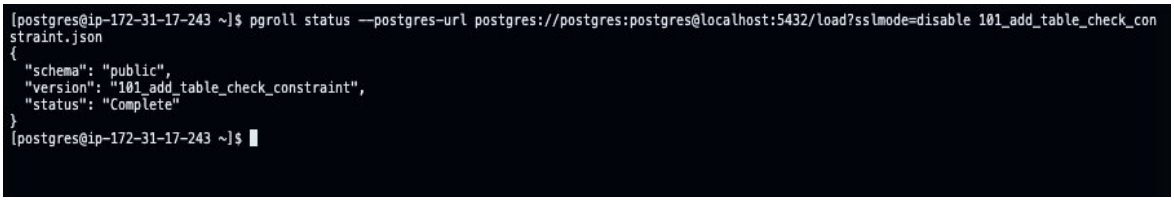
```
load=# select payment_method,_pgroll_new_payment_method from orders;
 payment_method | _pgroll_new_payment_method
----------------+----------------------------
 paypal         | paypal
 credit_card    | credit_card
                | cash
 paypal         | paypal
 credit_card    | credit_card
                | cash
 credit_card    | credit_card
 credit_card    | credit_card
(8 rows)
```

```
load=# \dn
                    List of schemas
                Name                |       Owner
------------------------------------+--------------------
 pgroll                             | postgres
 public                             | pg_database_owner
 public_101_add_table_check_constraint | postgres
(3 rows)

load=# \d+ public_101_add_table_check_constraint.orders
                        View "public_101_add_table_check_constraint.orders"
      Column      |            Type             | Collation | Nullable | Default | Storage  | Description
------------------+-----------------------------+-----------+----------+---------+----------+-------------
 order_date       | timestamp without time zone |           |          |         | plain    |
 customer_id      | integer                     |           |          |         | plain    |
 total_amount     | numeric(10,2)               |           |          |         | main     |
 delivery_date    | date                        |           |          |         | plain    |
 payment_method   | character varying(20)       |           |          |         | extended |
 shipping_address | text                        |           |          |         | extended |
 status           | character varying(20)       |           |          |         | extended |
 order_id         | integer                     |           |          |         | plain    |
 created_at       | timestamp without time zone |           |          |         | plain    |
 updated_at       | timestamp without time zone |           |          |         | plain    |
View definition:
 SELECT orders.order_date,
    orders.customer_id,
    orders.total_amount,
    orders.delivery_date,
    orders._pgroll_new_payment_method AS payment_method,
    orders.shipping_address,
    orders.status,
    orders.order_id,
    orders.created_at,
    orders.updated_at
   FROM orders;
Options: security_invoker=true
```

After complete migration:



Status of last migration



Best Practices

- **Splitting the work into batches -** Consider splitting the work into batches, each one being a separate transaction. If you're working in the OLTP context (mobile or web apps), the batch size should be determined appropriately
- **Take care or Tune VACUUMing -** Tune autovacuum and/or consider using explicit VACUUM calls after some number of batches processed.
- **Use extended lock analysis:** As explained in the monitoring section
- Use **CONCURRENTLY** when adding indexes, but that doesn't work inside transactions.
- When adding a constraint like NOT NULL, Postgres needs to first check that there are no NULLs in the table, and it has to do that while holding the lock. You can work around it by adding a **CHECK CONSTRAINT with NOT VALID**, which means that the constraint is applied to new rows but not to old ones. Then you can run VALIDATE later, which doesn't need the ACCESS EXCLUSIVE lock because it assumes new rows are respecting the constraint.
- **Tune checkpoint -** If not well tuned checkpoints may occur very often during such a massive operation**.**as an extra protection measure, tune the

checkpointer so that even if a massive change happens, our database's negative effect is not so acute "Basics of Tuning Checkpoints

- Use Lock-retry methodology at the SQL level - **lock_timeout,** to acquire a lock on the DB objects that are subject to change and do not implement some kind of retry logic.
- **DML never should go after DDL** unless they both deal with some freshly created table, It is usually wise to split DDL and DML activities into separate transactions / migration steps
- Increase **Statement_timeout** parameter value, if you have both Production and Non-Production environments using same values of statement_timeout, it's quite possible that the smaller tables are, the faster queries are executed. This can easily lead to a situation when a timeout is reached only on production.

Conclusion:

Online schema changes in PostgreSQL require careful planning and execution to minimize downtime and maintain data integrity. The most effective approach combines multi-phase deployments with backward-compatible changes, utilizing PostgreSQL's built-in features like transactional DDL and concurrent index creation. While some operations like adding nullable columns and creating indexes concurrently are relatively safe, others such as changing column types or adding NOT NULL constraints require special attention and potentially more complex migration strategies. Success depends on following best practices: implementing changes incrementally, maintaining backward compatibility, thorough testing with production-scale data, and having clear rollback plans. Tools like Liquibase and Flyway can help manage these changes, but understanding the underlying implications of each schema modification remains crucial for maintaining system reliability during migrations.

Disclaimer:

RECOMMENDATION: Always test thoroughly in non-production environments first and engage qualified database administrators for critical production changes.

References:

Lock Types
Pgroll
Pg_OSC
Common DB Schema changes mistakes
PostgreSQL Locks