MS Access to PostgreSQL

Replicator and data sanitiser program

Table of Contents

ronohara@duck.com

ronohara@duck.com

Overview:

This project has the objective of replicating any MS Access database into the equivalent PostgreSQL database with as little configuration as possible.

In the AI research world, there is almost no development aimed at integrating with user data stored in MS Access databases. There is however significant support for PostgreSQL database, so by creating a replica of the user data, with the option to update that replica at will, it becomes possible to connect AI platforms to the data normally being managed using MS Access.

This document describes the 'pg_replicator' program – not the research tools and processes that might use the replicated data.

The AI open context layer that we want to use the replicated database is WrenAI

MS Access is a very effective UI and data management tool, but has strong limitations related to capacity and robustness. Microsoft recommends MS SQL Server for users where those limits become problem. A consequence of that is that as of 2026, Microsoft has marked a number of re-distributable support libraries for Access database as 'no longer supported'. That does not mean MS Access is unsupported, but given that the UI component of Access can use MS SQL Server as its database, and MS SQL Server has an 'Express' which has higher capacity limits than MS Access and is free for use, you can deduce that the MS Access database component is being strongly de-emphasized.

I would estimate that many many users of the original MS Access tool have only modest transaction and storage capacity requirements, and very limited budgets for software migrations, so a free tool like this allows them to then use the amazing amount development in the AI community for analysing data.

ronohara@duck.com

The use case for a different technology as a replica

This replicator is a bare bones equivalent of MS SQL Server Migration Assistant for a target PostgreSQL database, which is then accessible from many AI integration platforms.

However is the intended use not the same.

The replicator is just a normal program. But its functions allow you to easily maintain a replica of your data in a database system that is much better (directly) supported in a wide range of AI platforms.

And PostgreSQL is platform neutral - Windows, MacOS or Linux ... so you are not forced to consider a platform migration just to make your data accessible to an AI system.

We did experiment with Microsoft SQL Server Migration Assistant, but that is aimed at migrating away from MS Access rather than maintaining a replica - and it had lots of problems with some elements of the Access system we had as input – the same problems we had. Primarily with stored queries It did not convert any of them, and we do not replicate them either.
Microsoft SQL Server Migration Assistant is an interactive migration assistance tool rather than a production process.

A big point is to leave your existing Access application(s) untouched. As a 'single source of truth' ... and the replication process becomes an automated workflow with zero manual steps to avoid that source of errors.

Lots of people are extracting Excel or CSV files and feeding them into a AI platform. That works of course, but is normally a manual approach - fine for experiments, but not so good for regular operation.

ronohara@duck.com

Environments:

The original database that is fed into this replicator is a Microsoft Access database managed via Microsoft Access

The whole concept of 'replicating' a database it not as simple as it seems. New rows should be added, existing rows should be updated, but that only works for tables with some kind of unique key .. tables without keys, can not be updated, only appended to - both in MS Access and PostgreSQL ... so the replicator needs to deal with these subtleties.

The target PostgreSQL database can be created in a Windows, MacOS or Linux system. But the examples in this document show everything operating in a Windows system.

In reality, since the AI systems almost all work in MacOS or Linux systems, the PostgreSQL server would normally operate in one of those environments.

ronohara@duck.com

Component software needed

1. Python language – and assorted Python library modules
2. PostgreSQL database manager

This selection of software is very robust and (except for the DAO drivers) is Open Source.

The selection ensures ongoing zero cost updates indefinitely, and a platform neutral design that can be have the PostgreSQL server operating on Windows, MacOS or Linux systems interchangeably, preventing supplier lock-in issues that are common.

After installing the PostgreSQL server, the examples in the document assume you have used the ‘psql’ utility that gets installed with the sever to create as replicator user and passwords.... using a command like this:

```
CREATE USER replicator WITH CREATEDB PASSWORD 'xxxxxxxx';
```

Obviously you can user a different username and you definitely should use a better password.

Adjust the replicatorconfig.yaml to match the credentials you select.

You can test the config and your database access with a command like this:

```
C:\Users\ronoh>python pg_replicator.py -c test.yaml --network --tpassword xxxxxxx
Replication processing starts
2026-05-21 05:41 - INFO - ====== testing network connections ======
2026-05-21 05:41 - INFO - Connected to MS Access database: C:\Users\ronoh\Test.accdb
2026-05-21 05:41 - INFO - MS Access connection: SUCCESS
2026-05-21 05:41 - ERROR - PostgreSQL connection failed: connection to server at "localhost"
(::1), port 5432 failed: FATAL:  password authentication failed for user "replicator"

2026-05-21 05:41 - ERROR - Check that:
2026-05-21 05:41 - ERROR - 1. PostgreSQL server is running
2026-05-21 05:41 - ERROR - 2. Host and port are correct
2026-05-21 05:41 - ERROR - 3. Username and password are correct
2026-05-21 05:41 - ERROR - 4. Database exists
2026-05-21 05:41 - ERROR - PostgreSQL connection: FAILED - connection to server at
"localhost" (::1), port 5432 failed: FATAL:  password authentication failed for user "replicator"

2026-05-21 05:41 - INFO - ====== network test completed =====

Replication processing completed – successful
```


ronohara@duck.com

If you configure or supply the correct password when the PostgreSQL database has not been created (which you could do before any of this, using the psql command line tool)

```
C:\Users\ronoh>python pg_replicator.py -c test.yaml --network --tpassword xxxxxxxx
Replication processing starts
2026-05-21 05:44 - INFO - ====== testing network connections ======
2026-05-21 05:44 - INFO - Connected to MS Access database: C:\Users\ronoh\Test.accdb
2026-05-21 05:44 - INFO - MS Access connection: SUCCESS
2026-05-21 05:44 - ERROR - PostgreSQL connection failed: connection to server at "localhost"
(::1), port 5432 failed: FATAL:  database "test" does not exist

2026-05-21 05:44 - ERROR - Check that:
2026-05-21 05:44 - ERROR - 1. PostgreSQL server is running
2026-05-21 05:44 - ERROR - 2. Host and port are correct
2026-05-21 05:44 - ERROR - 3. Username and password are correct
2026-05-21 05:44 - ERROR - 4. Database exists
2026-05-21 05:44 - ERROR - PostgreSQL connection: FAILED - connection to server at
"localhost" (::1), port 5432 failed: FATAL:  database "test" does not exist

2026-05-21 05:44 - INFO - ====== network test completed =====

Replication processing completed – successful
```

But because the user you created is allowed to create databases, you can have the replicator do this by using the –schema option (remove the –network test option)

```
    python pg_replicator.py -c test.yaml --schema --tpassword xxxxxxxx

Replication processing starts
2026-05-21 05:48 - INFO - Dropped database: test
2026-05-21 05:48 - INFO - Created database: test
2026-05-21 05:48 - INFO - Connected to MS Access database: C:\Users\ronoh\Test.accdb
2026-05-21 05:48 - INFO - Connected to PostgreSQL database: test
2026-05-21 05:48 - INFO - Discovered 4 foreign keys from MS Access (after exclusion filtering)
2026-05-21 05:48 - INFO - Configured 0 foreign keys from YAML
2026-05-21 05:48 - INFO - Total foreign keys: 4
2026-05-21 05:48 - INFO - No 'tables' section in configuration - auto-discovering all non-system
tables
2026-05-21 05:48 - INFO - Auto-discovered 4 non-system tables
2026-05-21 05:48 - INFO - Created table: ecgs
Created table: ecgs
2026-05-21 05:48 - INFO - Created table: events
Created table: events
2026-05-21 05:48 - INFO - Created table: investigations
Created table: investigations
2026-05-21 05:48 - INFO - Created table: patients
```


ronohara@duck.com

```
Created table: patients
2026-05-21 05:48 - INFO - Created index: patientsevents on events
2026-05-21 05:48 - INFO - Created index: lab_id on investigations
2026-05-21 05:48 - INFO - Created index: patientsinvestigations on investigations
  ….. lots more
2026-05-21 05:48 - INFO - Processing foreign key: fk_patients_investigations
2026-05-21 05:48 - INFO - Direction analysis for patients(['patient_number']) <->
investigations(['patient_number'])
2026-05-21 05:48 - INFO -   patients: PRIMARY KEY=True, UNIQUE=False
2026-05-21 05:48 - INFO -   investigations: PRIMARY KEY=False, UNIQUE=False
2026-05-21 05:48 - INFO -   Decision: patients has PRIMARY KEY -> Parent, investigations ->
Child
2026-05-21 05:48 - WARNING -   Foreign key fk_patients_investigations direction is
INCORRECT - needs swapping
2026-05-21 05:48 - INFO -   Original: patients(['patient_number']) ->
investigations(['patient_number'])
2026-05-21 05:48 - INFO -   Corrected: investigations(['patient_number']) ->
patients(['patient_number'])
2026-05-21 05:48 - INFO - Processing foreign key: fk_patients_investigations
2026-05-21 05:48 - INFO - Direction analysis for investigations(['patient_number']) <->
patients(['patient_number'])
2026-05-21 05:48 - INFO -   investigations: PRIMARY KEY=False, UNIQUE=False
2026-05-21 05:48 - INFO -   patients: PRIMARY KEY=True, UNIQUE=False
2026-05-21 05:48 - INFO -   Decision: patients has PRIMARY KEY -> Parent, investigations ->
Child
2026-05-21 05:48 - INFO -   Foreign key fk_patients_investigations direction is correct
2026-05-21 05:48 - INFO - Direction analysis for investigations(['patient_number']) <->
patients(['patient_number'])
2026-05-21 05:48 - INFO -   investigations: PRIMARY KEY=False, UNIQUE=False
2026-05-21 05:48 - INFO -   patients: PRIMARY KEY=True, UNIQUE=False
2026-05-21 05:48 - INFO -   Decision: patients has PRIMARY KEY -> Parent, investigations ->
Child
2026-05-21 05:48 - INFO -   Given direction investigations -> patients is VALID
2026-05-21 05:48 - INFO - Created foreign key: fk_patients_investigations
2026-05-21 05:48 - INFO - Schema replication only - data copy skipped
2026-05-21 05:48 - INFO - Connections closed

Replication processing completed - successful
```


ronohara@duck.com

Processes:

Replication and sanitising data

The replicator program by default will replicate all the tables it finds in the source MS Access database into the target PostgreSQL database. If the configuration file specifies table name(s) to replicate it will replicate only those tables.

Most of the time you should not have a list of table names in the configuration file.. the program will ***automatically replicate ALL tables, indexes and foreign key constraints*** *in the database*

The table names will be the same, the field names will be the same, and data types are mapped to their equivalent data type.

The details of the data type mapping are found in the pg_datatype_mapping.md file.

Table names, index names and column names may be transformed to all lower case but always in accordance with PostgreSQL rules. As follows:

> Unquoted identifiers (safe default): letters (including non-ASCII), digits (0–9) and underscore (_); must start with a letter or underscore; dollar sign ($) is allowed by PostgreSQL but not SQL standard; unquoted names are folded to lower case; maximum length is 63 bytes by default.
>
> Quoted identifiers (double quotes "like this"): can contain any character except NUL (U+0000); a double quote inside the name is written as two double quotes (""); retain case exactly; still limited to 63 bytes by default.
>
> Notes: SQL keywords may be used only when quoted; identifiers longer than the limit are truncated.

The Primary key of each table tin he PostgreSQL database is created to match the Primary key in the MS Access database.

All other keys of each table in the PostgreSQL database are created to match the original keys in the MS Access database.

If foreign key constraints are given in the configuration file they will be applied to the PostgreSQL tables. But if none are configured [the normal setup] they will be discovered from the Access database and those constraints will be applied to the PostgreSQL database tables.

ronohara@duck.com

** Limitations **

Foreign keys of each table the PostgrSQL database are *automatically* created to match the data You can define the foreign keys that you want created by entering their details in the configuration file, but this is optional. The default action is to create Foreign key constrains that match the MS Access database.

The replicator apples the PostgreSQL identifiers rules to table names, index names and column names

By default, the tables created in the target database have

- the same name as the tables source database
- the same column names as the tables have in source database
- the same primary key as the tables have in the source database
- the same foreign key structures as the tables have in the source database

But these names will be translated to lower case unless they need to be “quoted” to conform with the PostgrSQL rules (mixed case names with special characters or spaces)

and read access (SELECT) is set to Anyone … if you need different access permissions you can change that using PostgreSQL directly or by using a nice GU administrator program like PgAdmin or Webmin

At the moment, by default, the replicator does not replicate deletions in the source MS Access database. This is supported by the command line option –sync-deleted, but it implies a second pass that reviews the PostgreSQL tables and looks up each row in the MS Access database, then removes rows in the PostgreSQL database that it can not find in the MS Access database …

A second pass is slower than simply running the replicator with the –full-refresh command line option, which achieves the same result – though it makes the PostgresSQL database incomplete while it runs, until the run completes.

See the –sync-deleted section of this document

Another optimisation is available for tables where there is only infrequent changes made by adding or deleting rows --nonvolatile This optimisation can be used even on volatile changes where only the columns for rows are change and there is not a lot of need for a timely update of that type of change. The data changes in rows will ‘catch up’ the next time a row is added or deleted.

See the –nonvolatile section of this document

ronohara@duck.com

MS Access Relationships are not quite Foreign keys

During testing on a sample database from the Kaggle community website, I encountered some odd messages from the fairly defensive code in the replicator.

2026-05-21 01:56 - INFO - Creating foreign keys...

2026-05-21 01:56 - INFO - Processing foreign key: fk_dimtarih_factsatis

2026-05-21 01:56 - INFO - Direction analysis for DimTarih(['TarihKod']) <-> FactSatis(['SatisTarihi'])

2026-05-21 01:56 - INFO - DimTarih: PRIMARY KEY=False, UNIQUE=False

2026-05-21 01:56 - INFO - FactSatis: PRIMARY KEY=False, UNIQUE=False

2026-05-21 01:56 - WARNING - Decision: Neither table has PRIMARY KEY or UNIQUE constraint

2026-05-21 01:56 - INFO - Direction analysis for DimTarih(['TarihKod']) <-> FactSatis(['SatisTarihi'])

2026-05-21 01:56 - INFO - DimTarih: PRIMARY KEY=False, UNIQUE=False

2026-05-21 01:56 - INFO - FactSatis: PRIMARY KEY=False, UNIQUE=False

2026-05-21 01:56 - WARNING - Decision: Neither table has PRIMARY KEY or UNIQUE constraint

2026-05-21 01:56 - WARNING - Foreign key 'fk_dimtarih_factsatis' cannot be created: neither table has PRIMARY KEY or UNIQUE constraint

So I asked DeepSeek: In MS Access your can define relationships between tables, but these do not seem to match SQL foreign keys because they don't seem to need indexes on the tables

The answer was interesting – shown in the next section,

BUT not having the useful indexes is simple to resolve in Access itself -- just add a primary key to each table and other needed indexes ... the replicator runs fine even if you don't do that, but may not create foreign keys which you might like to have.

This was previous just a documented limitation .. with a simple correction possible by using Access to adjust the table definition(s). Add Primary keys or unique keys that are needed in relationships defined to your Access database.

In the current version, the replicator will attempt to create enough primary keys or unique indexes to satisfy implementing the MS Access relationships as PostgreSQL foreign keys.

ronohara@duck.com

🔗 MS Access relationships vs. SQL foreign keys

Why Access table relationships don't *require* indexes — and how that differs from true FK constraints

🎯 **Your sharp observation:** “In MS Access you can define relationships between tables, but these do not seem to match SQL foreign keys because they don’t seem to need indexes on the tables.”

✅ You are absolutely right: a basic Access relationship (just a line) is **not** the same as a SQL foreign key. It acts more like a **query design hint** rather than an integrity rule. Indexes become optional depending on whether you enforce referential integrity.

📁 MS Access relationship (simple link)

When you drag a line between two fields in the Relationships window **without** checking `Enforce Referential Integrity`:

- ✅ It’s purely **informational / visual** for the query designer.
- ✅ Access will **auto-join** tables in queries, but no physical rule is stored.
- ✅ Orphan records **are allowed** (you can add an order with a missing CustomerID).
- ✅ **No index required** on the foreign-key side — Access can use hash or loop joins in memory.

⚙️ Design-time only · No ON DELETE CASCADE · No automatic index creation

💡 **key point** Without enforcement, the "relationship" is just metadata — it doesn't behave like an FK constraint.

🧠 SQL foreign key (true constraint)

In PostgreSQL, MySQL, SQL Server, or Oracle, a foreign key creates a **physical integrity rule**:

- ✅ Prevents orphan rows (child records must reference an existing parent).
- ✅ Supports `ON DELETE CASCADE / SET NULL`.
- ✅ **Requires an index** on the foreign key column to maintain referential integrity efficiently.
- ✅ The index is often created automatically by major DBMS (or mandatory for performance).

```
ALTER TABLE orders ADD CONSTRAINT fk_customer
FOREIGN KEY (customer_id) REFERENCES customers(id);
```

📌 **critical** Without an index, FK checks would cause full table scans, so SQL engines strongly require/auto-create indexes.

🔐 The missing piece: “Enforce Referential Integrity” changes everything

If you double-click the relationship line in Access and check **“Enforce Referential Integrity”**, Access *behaves almost exactly like a SQL foreign key*. At that moment:

- ❌ You can’t create orphan records (child must have a matching parent).

- ❌ You can’t delete a parent record if child records exist (unless you cascade).
- 📈 **Index requirement:** Access will either prompt you to create an index or **implicitly creates a hidden index** on the foreign key column. You just don’t see it in the table design grid.

👉 So why doesn't a simple (non-enforced) relationship need an index? Because it doesn’t enforce anything — it’s just a visual & query auto-join hint. No constraint = no need for the engine to constantly verify child rows.

📊 Feature comparison table

Prevents orphan records	❌ No	✅ Yes	✅ Yes
Requires / uses index on FK side	❌ Not required (optional for query speed)	✅ Yes (implicit or explicit index required)	✅ Required / auto-created
Can you see the index in table design?	No, unless manually added	Sometimes hidden, but engine relies on it	Visible in schema (index on FK column)
Query auto-join behaviour	✅ Yes (designer hint)	✅ Yes	Not relevant (physical constraint)
ON DELETE CASCADE option	No	✅ Optional	✅ Optional
Performance overhead for inserts/updates	Minimal (no check)	Moderate (referential check & index update)	Moderate–High (index + constraint verification)

⚡ Why doesn't a basic Access relationship require indexes?

Because the Jet/ACE database engine is file-based, not server-based. For modest row counts (< 100k–200k), Access can perform **hash joins or nested loop joins in memory** faster than it can traverse a disk-based index. But the moment you *Enforce Referential Integrity* or need high performance on millions of rows, you **must** add an index. The rule is: *no constraint = no forced index; constraint = index needed*. That's why simple relationships feel different from SQL foreign keys.

🛠️ Practical demonstration (Mental model)

📌 **Non-enforced relationship**
✅ No index required
✅ Allows orphan records
⚠️ Joins might be slower on large data
✅ **Enforced RI (acts like FK)**
🔍 Hides/creates index behind the scenes
❌ Rejects invalid foreign keys
🧠 Closest to SQL foreign key
⚙️ **SQL foreign key**
🗄️ Mandatory index (explicit)
🔒 ACID + constraint strictness
🚀 Designed for large workloads

🗂️ When are indexes actually required in Access relationships?

- **Simple line (no RI):** Index is optional. Helps join performance on large tables but not forced.
- **Enforced RI:** Access will either **demand an index** or create one implicitly. If you try to enforce RI without an index, Access often shows the warning: *“Do you want to create an index on these fields?”*
- **Join queries:** Even without relationships, an index on foreign key column drastically improves speed.

🔎 Therefore, your original statement holds true: *basic relationships do not 'match' SQL foreign keys because they lack forced referential integrity and mandatory indexes.* The moment you enable enforcement, Access transforms the relationship into something functionally equivalent — including hidden reliance on indexes.

🧩 Final answer – why the mismatch feels confusing

Many data professionals assume that drawing a relationship line in Access creates a **full foreign key constraint** like in SQL. It does not — unless you explicitly **Enforce Referential Integrity**. Without that checkbox, Access treats the relationship as a lightweight instruction for query design, not a database integrity rule, which is why indexes aren’t mandatory. That’s the root of the difference you observed.

✅ **Bottom line:** For strict data integrity + index-backed foreign key behavior in MS Access, always check “Enforce Referential Integrity” and verify that an index exists (or let Access create it). Otherwise, you just have a visual guide – not a real foreign key.

⚡ Insight based on Access Jet/ACE engine & standard SQL behavior · your question highlights one of the most common misconceptions among SQL users transitioning to Access.

Handling MS Access Relationships During PostgreSQL Migration

When migrating a Microsoft Access database to PostgreSQL, a common challenge arises from the fundamental differences in how each system enforces data integrity. Microsoft Access allows you to define relationships in its Relationship Window, even if the underlying tables do not have properly indexed columns to support a formal foreign key constraint in a SQL database.

This document describes the program's behavior and technical requirements when converting Access relationships into PostgreSQL foreign keys, using standard SQL parent/child terminology. The "parent" table is the primary table in a relationship (the "one" side), and the "child" table is the referencing table (the "many" side).

This behaviour can be suppressed by using the –no-auto-index command line option. If that option is used you will see messages about any table that is affected … where the equivalent foreign key was not created and the MS Access relationship has not been replicated.

The Core Challenge: Indexing

In PostgreSQL (and most relational database management systems), a foreign key constraint, which defines a parent/child relationship, requires a unique index on the referenced columns of the parent table. This is a strict technical requirement.

By contrast, Microsoft Access is more flexible. You can create a relationship in the Relationship Window between a parent and a child table, even if the parent table's primary key field lacks a unique index. For example, if the parent table’s key field allows duplicate values, Access may still accept the relationship definition, but its enforcement is less rigorous.

Program Behaviour During Replication

When the program analyses a parent/child relationship from MS Access, it performs a specific validation check to determine if it can be safely converted into a PostgreSQL foreign key.

Analysis of the Parent Table: The program identifies the column in the parent table that the relationship is based upon.

ronohara@duck.com

Index Validation: It checks if this parent column has a unique index.

Decision Point:

- If a unique index exists: The program proceeds. It will create the corresponding foreign key constraint in PostgreSQL using the standard ALTER TABLE command. This ensures data integrity is maintained at the database level.

- If a unique index does NOT exist: The program cannot create a standard foreign key, as this would violate PostgreSQL's core rules.

Handling Unsuitable Parent Keys

When the parent table lacks a unique index, the program employs a strategy to avoid errors and preserve the logical link.

The Relationship is Skipped: The program will not create a formal FOREIGN KEY constraint in the PostgreSQL schema. This prevents a migration failure that would otherwise occur if it attempted to create an invalid constraint.

Integrity Enforcement is Shifted: The responsibility for maintaining the parent/child relationship is not dropped, but it is shifted from the database to the application layer.

- The PostgreSQL Database: Will manage the two tables as independent entities. It will not automatically check that every value in the child's foreign key column has a matching value in the parent's referenced column.

- The Application Layer: The program recommends or relies on the application using the database to enforce the relationship logic. This is often done through application code or, in the case of a front-end like a migrated Access project, through queries and business logic rules.

Summary – Foreign key issues

The program prioritizes a successful and stable migration. It enforces the strict requirement for a unique parent index. If this requirement is not met, it safely skips creating the formal constraint, leaving the relational logic to be handled by the application to prevent a migration failure.

ronohara@duck.com

Sample console messages for --no-auto-index

2026-05-21 23:30 - INFO - Processing foreign key: fk_dimaltkategori_dimurun

2026-05-21 23:30 - INFO - Direction analysis for DimAltKategori(['AltKategoriKod']) <-> DimUrun(['AltKategoriKod'])

2026-05-21 23:30 - INFO - DimAltKategori: PRIMARY KEY=False, UNIQUE=False

2026-05-21 23:30 - INFO - DimUrun: PRIMARY KEY=False, UNIQUE=False

2026-05-21 23:30 - WARNING - Decision: Neither table has PRIMARY KEY or UNIQUE constraint

2026-05-21 23:30 - INFO - Direction analysis for DimAltKategori(['AltKategoriKod']) <-> DimUrun(['AltKategoriKod'])

2026-05-21 23:30 - INFO - DimAltKategori: PRIMARY KEY=False, UNIQUE=False

2026-05-21 23:30 - INFO - DimUrun: PRIMARY KEY=False, UNIQUE=False

2026-05-21 23:30 - WARNING - Decision: Neither table has PRIMARY KEY or UNIQUE constraint

==========

⚠️ FOREIGN KEY SKIPPED (--no-auto-index enabled):

Foreign key name: fk_dimaltkategori_dimurun

Child table (base_table) : DimAltKategori

Parent table (reference) : DimUrun

Child columns : AltKategoriKod

Parent columns : AltKategoriKod

This foreign key requires a PRIMARY KEY or UNIQUE constraint on the child

table columns, but automatic creation is disabled.

The foreign key will NOT be created.

==========

ronohara@duck.com

** data sanitising / transforming **

You may wish to sanitise the data before it is replicated into the PostgreSQL data system, individual data items may be identified as not to be replicated, or edited according to a transformation rule before being replicated

During replication, existing rows in the target table(s) are updated to match any revised data.

New rows that are not yet in the replicated target system are simply inserted after transformation rules are applied.

Transformation of fields is based on configuration file entries that specify individual field names in a table that should have their data altered to preserve the security of the application data,

The AI research can often function properly even if some of the data is obscured

Note: having new fields specified as ‘Drop’ will not clear any data from any prior runs of the replicator. In this case, you should use the –full-refesh command line option to force previously copied data to be removed. After that you can resume normal operation for incremental updates.

The transformations possible are:

MMH3	replace a field with the MurmurHash3 hash of it – normal used to obscure any integer
YearOnly	replace a full date with 1st of January of the same year
Drop	Don’t copy this field

An example of the configuration file entries

```
# Transformations - MUST be a LIST of dictionaries
transformations:
  - table: "Dimindirim"
    columns:
      - name: "Baslangic"
        transform: "YearOnly"
```


ronohara@duck.com

Command line switches and their usage

There are a number of situations where you may want to use the optional command line switches, and this part of the document tries to guide when each switch could be used.

Minimum configuration file

The basic use of the replicator is to have a configuration file (replicatorconfig.yaml) with the connection information needed for the Access input database, and the PostgreSQL target database.

```
DAO:
  driver: "Microsoft Access Driver (*.mdb, *.accdb)"
  database: "ContosoTR.accdb"


# Target PostgreSQL connection
postgresql:
  host: "localhost"
  port: 5432
  database: "contosotr"
  user: "replicator"
  password: "xxxxxxx"


# Global settings
global:
  debug: false
  managerid: "Replicator"
  verbose: true
```

Then the command could be: **python pg_replicator.py**

If the configuration file does not have a *tables:* entry (or it is empty) the default action is to create a complete copy of the tables found in the Access database, into the PostgreSQL database. A compatible schema for each Access table with columns, indexes and foreign key constraints is created in the PostgreSQL database (if not already created), and then the data for every table is copied or updated (with optional transformations) into the PostgreSQL database.

ronohara@duck.com

You can use the same configuration file without a *tables:* entry with the **–list** command line option to get exactly that. A list of the table names in your database.

ronohara@duck.com

MS Access tables without keys

The replicator can and will copy tables without keys – append only. If the PostgreSQL database is empty or you run it with the –full-refresh command line option this is fine.

But if you re-run it normally, the tables with keys get updated, but the tables without keys get all the data appended to the output table **again..** almost certainly not what you would like.

There is a command line option to correct this situation but **WARNING** **it changes the MS Access database structure**. This is the only replicator function that writes to your input MS Access database so you should take a backup before you start.

It is also the only command line option that does not require valid PostgreSQL configuration data, So it can operate independently on the MS Access database. **WARNING** since it adds an AutoNumber field (if needed) to your tables, and if the tables have a large number of rows this will be a very slow, one time operation. If a table has an AutoNumber field, but no Primary key or unique key, the replicator will not add anew column. It will change the Access database to use that existing AutoNumber field as a primary key. That field **should** be valid as a primary key – but MS Access has been known to have duplicate values in an AutoNumber field … and there will be an error message given if that happens. Your data will need to be corrected manually to fix this. A simple fix would be to change that column to Long Integer rather than AutoNumber – but that could cause problems with your Access application.

The option is called **--adjust-ms-access**

Running the replicator with this option is like having a stand alone MS Access table editing function.

ronohara@duck.com

--adjust-ms-access

DESCRIPTION

This option modifies the schema of an MS Access database by adding an AutoNumber primary key column to every user table that does not already have a primary key.

The operation is performed directly on the Access database file specified in the configuration file or via the --source parameter. No connection to PostgreSQL is required or attempted when this option is used.

PURPOSE

Many MS Access tables lack explicit primary keys, which can cause issues when replicating data to PostgreSQL or when attempting to establish relationships. This function ensures that every user table has a unique, auto-incrementing identifier column that can serve as a primary key.

PROCESSING DETAILS

1. The program connects to the MS Access database using DAO.
2. It iterates through all non-system user tables.
3. For each table without a primary key:
   - A new column is added with the name: {table_name}_replicator_id
   - The column is created as an AutoNumber (Long Integer) type, which automatically generates sequential numbers for existing rows.
   - A PRIMARY KEY constraint is created on this new column.
   - The column name is sanitised (lowercase, spaces replaced with underscores, non-alphanumeric characters removed) and truncated

ronohara@duck.com

to 64 characters if necessary.

4. Tables that already have a primary key are skipped.
5. Tables that already contain a column with the same name are skipped.
6. System tables (names starting with MSys or ~) are excluded from processing.

OUTPUT

For each table processed, the program displays:

- ⊘ Table name: already has PRIMARY KEY - skipping
- ➕ Table name: adding AutoNumber primary key column 'column_name'
  √ Successfully added primary key on 'column_name'
- ⚠️ Table name: column 'column_name' already exists - skipping
- ✗ ERROR on table name: (error description)

At the end of processing, a summary is displayed showing:

- Tables modified
- Tables skipped
- Errors encountered

EXIT CODES

The program exits with code 0 on successful completion.
Exits with non-zero code if the Access database cannot be opened or if a fatal error occurs.

USAGE EXAMPLES

```
# Basic usage with configuration file
python pg_replicator.py --adjust-ms-access
```


ronohara@duck.com

```
# Specify configuration file and source database
python pg_replicator.py --adjust-ms-access -c myconfig.yaml -s C:\data\database.accdb
```


ronohara@duck.com

```
# Verbose output
python pg_replicator.py --adjust-ms-access -v
```

NOTES

- This option is mutually exclusive with --schema, --list, --network, --dump, and --full-refresh.
- PostgreSQL connection parameters are ignored when this option is used.
- The Access database must not be open exclusively by another user.
- Existing rows automatically receive sequential numbers when the AutoNumber column is added.
- The new primary key is named "PrimaryKey" in Access (default naming).
- This operation modifies the source Access database. It is recommended to backup your database before running this command.

RELATED OPTIONS

--source, -s	Specify the Access database file path
--config, -c	Specify the configuration file (default: replicatorconfig.yaml)
--verbose, -v	Display detailed progress information

TYPICAL OUTPUT LOG

```
2026-05-22 02:03 - INFO - Connected to MS Access database: C:\Users\ronoh\Test.accdb
2026-05-22 02:03 - INFO - Connected to PostgreSQL database: test
2026-05-22 02:03 - INFO - Connected to MS Access database: C:\Users\ronoh\Test.accdb
2026-05-22 02:03 - INFO - adjust_ms_access_schema: DAO connection successful
2026-05-22 02:03 - INFO - adjust_ms_access_schema: Found 5 user tables
2026-05-22 02:03 - INFO - adjust_ms_access_schema: ecgs already has PK, skipping
2026-05-22 02:03 - INFO - adjust_ms_access_schema: events already has PK, skipping
```


ronohara@duck.com

2026-05-22 02:03 - INFO - adjust_ms_access_schema: investigations already has PK, skipping

2026-05-22 02:03 - INFO - adjust_ms_access_schema: patients already has PK, skipping

2026-05-22 02:03 - INFO - adjust_ms_access_schema: Adding PK to some-log with column some_log_replicator_id

2026-05-22 02:03 - INFO - adjust_ms_access_schema: Successfully added PK to some-log

2026-05-22 02:03 - INFO - Connections closed

2026-05-22 02:03 - INFO - adjust_ms_access_schema completed: 1 modified, 4 skipped, 0 errors

~

~

ronohara@duck.com

--sync-deleted

Purpose

The --sync-deleted option synchronizes record deletions from the source Microsoft Access database to the target PostgreSQL database. During normal replication, the program copies new and updated records from Access to PostgreSQL but does not remove records that have been deleted from the source. This option addresses that gap by identifying and removing records from PostgreSQL that no longer exist in Access.

How It Works

When --sync-deleted is enabled, the program performs the following operations after the main data copy phase is complete:

1. Row Count Comparison – For each table being replicated, the program compares the total number of rows in the Access source against the row count in the PostgreSQL target.

2. Selective Processing – By default (fast mode), only tables with differing row counts are processed. Tables with matching row counts are skipped, as no deletions are needed.

3. Key-Based Row Identification – For each table requiring synchronization, the program identifies which rows in PostgreSQL should be deleted by:

- Using the table's PRIMARY KEY if one exists
- Falling back to the first available UNIQUE constraint if no PRIMARY KEY is defined
- If neither exists, the table is skipped with a warning (deletion synchronization requires a way to uniquely identify each row)

4. Batch Processing with Keyset Pagination – To avoid memory issues with large tables, the program processes rows in batches of 1,000 records. For each batch:

- Rows are retrieved from PostgreSQL in sorted order by the key columns

ronohara@duck.com

- Each row's key value is checked against the Access database using a SQL COUNT query
- If the key value is not found in Access, the corresponding PostgreSQL row is marked for deletion
- Marked rows are deleted from PostgreSQL using a batched DELETE statement

5. Progress Reporting – When --verbose is enabled, the program displays:

- A progress bar showing completion percentage
- Estimated time remaining (ETA) for the current table
- Number of rows deleted so far
- Final count of deleted rows for each processed table

6. Validation – After deletion synchronization completes, the program re-validates row counts and displays a summary showing which tables were processed and how many rows were deleted.

The --slow Option

When --slow is specified together with --sync-deleted, the program changes its behaviour in the following way:

```
| Mode | Behaviour |
|------|----------|
| Fast Mode (default) | Only tables with differing row counts between Access and PostgreSQL are processed |
| Slow Mode (--slow) | Every table is processed regardless of whether row counts match |
```

When to use Slow Mode:

Slow mode is useful in the following scenarios:

- Silent data corruption – When row counts match but individual records may have been deleted and reinserted with different key values, leaving orphaned records with different identifiers

ronohara@duck.com

- Foreign key cascade testing – When verifying that cascade deletions are properly propagating through related tables
- Recovery from interrupted syncs – When a previous sync-deleted operation was interrupted and the row counts no longer accurately reflect the state of deletions
- Debugging and validation – When performing thorough integrity checks on the synchronization process

Performance Note: Slow mode significantly increases processing time because every table is fully scanned regardless of whether deletions are needed. It should only be used when the additional thoroughness is required.

Important Requirements

For --sync-deleted to function correctly:

- Each table must have a PRIMARY KEY or at least one UNIQUE constraint defined in PostgreSQL
- Foreign key relationships should be defined with ON DELETE CASCADE where appropriate to maintain referential integrity
- The Access and PostgreSQL databases must be otherwise synchronized (run a normal replication first)

Example Usage

Standard deletion synchronization (fast mode):

```
python pg_replicator.py --config myconfig.yaml --sync-deleted --verbose
```

Slow mode (process all tables):

```
python pg_replicator.py --config myconfig.yaml --sync-deleted --slow --verbose
```


ronohara@duck.com

Typical Output

```
============================================================
SYNC DELETED RECORDS
============================================================
Fast mode - will only process tables where row counts differ.

  Patients: Access=3090, PostgreSQL=3094
    → Counts differ - processing Patients
      Syncing Patients: PostgreSQL has 3094 rows, Access has 3090 rows
      Processing in batches of 1000 rows...
      Progress: Batch 4, Processed approx 4000/3090 rows
[████████████████████████████████████████] 100.0% - Deleted 4 so far
      √ Completed Patients: Deleted 4 rows (errors: 0) (elapsed: 01s)

============================================================
SYNC DELETED SUMMARY:
  Tables processed: 1
  Tables skipped:   21
============================================================
```

Limitations

- Tables without a PRIMARY KEY or UNIQUE constraint cannot be synchronized and will be skipped with a warning
- The process is row-by-row for existence checking, which may be slower than native database replication tools
- Very large tables (millions of rows) may still require significant processing time even in fast mode

ronohara@duck.com

--nonvolatile

Purpose

The --nonvolatile option provides an optimization for tables that rarely change (non-volatile tables). When enabled, the program skips copying tables that are marked as non-volatile in the configuration file, provided their row counts match between the source Access database and the target PostgreSQL database. This reduces processing time by eliminating unnecessary copy operations for stable tables.

Configuration

To use this feature, add a nonvolatile: section to the replicatorconfig.yaml file listing the tables to be treated as non-volatile:

nonvolatile:

- Suppliers
- Insurance
- InsuranceCodes

How It Works

When --nonvolatile is enabled, the program performs the following steps for each table during the copy phase:

1. Row Count Retrieval – The program queries both the Access source and PostgreSQL target to obtain the current row count for the table.

ronohara@duck.com

2. Non-Volatile Check – If the table name appears in the nonvolatile: section of the configuration file:

- Row counts match – The program skips copying the table entirely. A diagnostic message is logged and displayed, and the table is marked as validated without any data transfer.

- Row counts differ – The program proceeds with the normal copy operation despite the non-volatile designation, logging a message explaining that copying is required due to count mismatch.

3. Normal Processing – If the table is not listed in the nonvolatile: section, or if the --nonvolatile flag is not specified, the program copies the table as usual.

```
Behavior Summary

+-------------------------+--------------------+------------------+---------------------------
-+
| Table in nonvolatile list? | --nonvolatile flag? | Row counts match? | Action
|
+-------------------------+--------------------+------------------+---------------------------
-+
| No                      | Either             | Any              | Normal copy
|
| Yes                     | No                 | Any              | Normal copy
|
| Yes                     | Yes                | Yes              | Skip copy (optimization)
|
| Yes                     | Yes                | No               | Normal copy (with diagnosi
s)|
+-------------------------+--------------------+------------------+---------------------------
```

Command Line Usage

Enable non-volatile optimization:

python pg_replicator.py --config replicatorconfig.yaml --nonvolatile --verbose

ronohara@duck.com

Combine with other options (example: deletion synchronization):

python replicator.py --config replicatorconfig.yaml --nonvolatile --sync-deleted --verbose

Output Messages

When a non-volatile table is successfully skipped:

Table BloodTestType (non-volatile): row counts match (Access: 8, PostgreSQL: 8) - skipping copy (--nonvolatile enabled)

When a non-volatile table requires copying due to count mismatch:

Table Insurance (non-volatile): row counts differ (Access: 36, PostgreSQL: 35) - copying despite --nonvolatile

Validation Summary Indicator

In the final validation summary, tables that were skipped due to the --nonvolatile optimization are marked with [NV SKIPPED] in the dump_internal_data() output for debugging purposes.

Use Cases

The --nonvolatile feature is particularly useful in the following scenarios:

- Reference tables – Lookup tables such as test types, insurance codes, or category lists that change infrequently

ronohara@duck.com

- Stable historical data – Tables containing archival records that are never modified after initial import

- Large static tables – Tables that are time-consuming to copy but rarely change, where the row count check provides a fast alternative to full table scanning

- Frequent replication runs – When running the replicator multiple times per day on the same database, non-volatile tables can be quickly verified rather than fully copied

Important Notes

- The optimization relies entirely on row count comparison. If a non-volatile table has the same number of rows but different content (e.g., updated records without change in row count), the update will not be detected and the table will be incorrectly skipped.

- For tables that may have row count-preserving updates (e.g., correcting values in existing rows), if you include them in the nonvolatile: list their data will not be updated until the next time the row counts differ.

- The --nonvolatile flag has no effect unless the nonvolatile: section exists in the configuration file and it only affects tables listed in that section

- The optimization applies only to the copy phase. Other operations such as foreign key creation and deletion synchronization are unaffected.

ronohara@duck.com

```
Comparison with --sync-deleted

+-------------------+------------------------+------------------------+
| Feature           | --nonvolatile          | --sync-deleted         |
+-------------------+------------------------+------------------------+
| Purpose           | Skip copying stable    | Remove deleted records |
|                   | tables                 |                        |
| Phase             | Copy phase             | Post-copy phase        |
| Detection method  | Row count comparison   | Key-based existence    |
|                   |                        | checking               |
| Data modification | None (skips copy)      | Deletes rows           |
| Default behavior  | Off                    | Off                    |
+-------------------+------------------------+------------------------+
```


ronohara@duck.com

--simple-names Capability

Purpose

The `--simple-names` option creates PostgreSQL tables and columns with simple, lowercase, underscore-separated identifiers instead of quoted identifiers that preserve spaces, mixed case, and special characters from MS Access. This makes the resulting database more compatible with external tools that cannot handle quoted identifiers or special characters.

Usage

The `--simple-names` option can only be used with `--schema`:

`python pg_replicator.py --schema --simple-names -c config.yaml`

When the database is created with `--simple-names`, all subsequent replication runs automatically detect and use simple naming mode via the `internal_replicator_data` metadata table. No additional flags are needed for normal replication or view creation.

What It Does

Table Names: Converted to lowercase, spaces replaced with underscores, special characters replaced with descriptive words.

Column Names: Same rules as table names. Name collisions are handled by appending `_01`, `_02`, etc.

Index and Constraint Names: Remain as lowercase underscore-separated names (unchanged from default behavior).

ronohara@duck.com

Views: View names always use the `view_{sanitised_name}` format regardless of mode. View column aliases use the same sanitisation rules.

Special Character Replacements

Character	Replacement	Example
Space	`_`	`"First Name"` → `first_name`
`%`	`_percent`	`"nadir%"` → `nadir_percent`
`$`	`_dollar_` (embedded), `dollar_` (leading), `_dollar` (trailing)	`"$cost"` → `dollar_cost`
`#`	`_hash_`, `hash_`, `_hash`	`"#tag"` → `hash_tag`
`@`	`_at_`, `at_`, `_at`	`"user@domain"` → `user_at_domain`
`&`	`_amp_`, `amp_`, `_amp`	`"procter&gamble"` → `procter_amp_gamble`
`*`	`_star_`, `star_`, `_star`	`"select*from"` → `select_star_from`
`+`	`_plus_`, `plus_`, `_plus`	`"a+b"` → `a_plus_b`
`-`	`_minus_`, `minus_`, `_minus`	`"a-b"` → `a_minus_b`
`(`	`_lbrk_`, `lbrk_`, `_lbrk`	`"(test)"` → `lbrk_test_rbrk`
`)`	`_rbrk_`, `rbrk_`, `_rbrk`	`"(test)"` → `lbrk_test_rbrk`

Additional Rules

- If the result starts with a digit, an underscore is prefixed (e.g., `"4THR"` → `_4thr`)
- Trailing underscores from special character replacement are preserved (e.g., `"nadir%"` → `nadir_percent` retains the trailing underscore from `%` replacement)
- Consecutive underscores are collapsed into a single underscore

Examples

Original MS Access Names:

- Table: `Patient Visits`
- Columns: `P ID`, `Visit Date`, `4THR`, `nadir%`, `user@domain`

With `--simple-names` (PostgreSQL):

- Table: `patient_visits`
- Columns: `p_id`, `visit_date`, `_4thr`, `nadir_percent`, `user_at_domain`

ronohara@duck.com

Compatibility Notes

- This mode is only applied during schema creation (`--schema`). Existing databases without this flag continue to use quoted naming.
- Normal replication (`replicate_tables`) automatically detects the naming mode from the database metadata.
- View creation (`--create-views`) also auto-detects the naming mode.
- The `--adjust-ms-access` option is unaffected by `--simple-names`.

ronohara@duck.com

--create-views Capability

Purpose

The `--create-views` option creates "sane views" in an existing PostgreSQL database. These views provide simplified, lowercase, underscore-separated aliases for all columns in each table, making the data more accessible to external tools that cannot handle quoted identifiers, spaces, or mixed-case column names. Some external tools can not cope with many valid table and column names – especially mixed case names and names with embedded spaces. If the can program can use names in the **simple names** format described earlier the replica datbase can retain the original MS Access names. If not, then a new replica using –simple-names may be a better answer.

Usage

Create views on an existing database (no schema changes, no data copy):

`python pg_replicator.py --create-views -c config.yaml`

Create schema and views in one operation (use with `--schema`):

`python pg_replicator.py --schema --create-views -c config.yaml`

What It Does

For each table in the MS Access database (subject to exclusions), the program creates a view with the following characteristics:

View Naming: `view_{sanitised_table_name}` — the table name is sanitised to lowercase with underscores replacing spaces and special characters.

ronohara@duck.com

Column Aliases: Every column in the original table is aliased with a sanitised name (lowercase, underscores instead of spaces, special characters replaced with descriptive words).

Name Collisions: If two different original column names sanitise to the same alias, suffixes `_01`, `_02`, etc., are appended.

Creation Method: Uses `CREATE OR REPLACE VIEW` so existing views are replaced without error.

Schema: Views are created in the `public` schema.

Automatic Naming Mode Detection

The program automatically detects whether the target database was created with `--simple-names` by reading the `internal_replicator_data` metadata table. It then uses the appropriate naming convention for table lookups:

- **Quoted mode (default):** Original table names are quoted to preserve case and spaces (e.g., `"Patient Visits"`)
- **Simple names mode:** Table names are unquoted, lowercase, with underscores (e.g., `patient_visits`)

No additional flags are required to handle either naming mode.

ronohara@duck.com

Examples

Original MS Access Table: `Patient Visits`

Original Columns: `P ID`, `Visit Date`, `4THR`, `nadir%`

Created View: `view_patient_visits`

View Definition (simplified):

- `"P ID"` AS `p_id`
- `"Visit Date"` AS `visit_date`
- `"4THR"` AS `_4thr`
- `"nadir%"` AS `nadir_percent`

Requirements

- The target PostgreSQL database must already exist (created by a previous `--schema` run)
- The MS Access database must be accessible
- Tables must exist in PostgreSQL (views are only created for tables that exist in both databases)
- Excluded tables are skipped

Relationship with --schema

- When used alone: creates views only, assumes schema already exists
- When used with `--schema`: first recreates the schema (dropping and recreating the database), then creates views

Compatibility Notes

- This option is mutually exclusive with `--list`, `--network`, `--dump`, `--full-refresh`, and `--adjust-ms-access`
- Can be used with `--verbose`, `--debug`, `--trace`, and other non-conflicting options
- Views remain in the database after creation and can be queried like any other table

ronohara@duck.com

Other ways to use the minimum configuration file

You can also use the same configuration file without a *tables:* entry with the **–output {filename}** command line option to generate a different yaml format configuration file with all the discovered schema structure in listed in the new configuration file. This is handy if you want to move some table names that were discovered into the ***excluded:*** section to suppress the replication of them

If run the replication with a minimum configuration file that specifies an Access database, (and does not have a *tables* section:) with the **–output {filename}** command line option . It will generate a yaml format configuration file with all the discovered schema structure in listed in the new configuration file. That will include the definitions of the foreign keys .. this new yaml configuration file can be edited - usually to move some table names into the *excluded* section

--verbose – puts more messages on the console and in the log. If you are watching the process you probably want –verbose so that you get a progress indicator and a completion time estimate as each table is processed.

--schema – **Deletes the existing PostgreSQL** database and recreates the tables in it, but does not replicate the data. Useful if you wish to start again from a clean position

--full-refresh … the same as using –schema and then running it again to replicate the data

--no-auto-index suppresses the program behaviour needed to replicate MS Access relationships

--sync-deleted runs a second pass to remove rows from the PostgreSQL tables that have been deleted in the MS Access tables. As an optimisation it assumes that if the number of rows in table is the same in both databases, then that table does not need to be scanned (true unless you have deleted and inserted the same number of rows)

--slow runs the –sync-deleted logic on all tables even if the row counts for the source and target are the same.

The following command line options are only for developer use

--debug

--trace

--dump

ronohara@duck.com

Maximum performance choices

You can tune your replication setup for an automatic maximum speed optimisation with only a small trade off in the delay of data updates for individual rows in tables, If that trade of is acceptable, then the following setup can usually decrease the replication run time dramatically.

configure all tables as nonvolatile

This makes every table a candidate for the optimisation logic. It does not really mean that they are subject many add, change or delete operations in MS Access

The simplest way to achieve this is to generate a new config.yaml file using the –**output {filename]** option, then making one very simple edit to this new configuration file.

```
python ms_replicator.py –output fastupdate.yaml
```

Then edit ‘fastupdate.yaml’ and find the tables line

```
tables:
```

changed it to:

```
nonvolatile:
```

Then you can use the –config command line option or copy ‘fastupdate.yaml’ over your existing ‘replicatorconfig.yaml’ file

run the replicator with the row count optimisations enabled

```
python ms_replicator.py –config fastupdate.yaml –nonvolatile –sync-deleted –verbose
```

This setup will automatically update all tables where add or delete rows has occurred. Any rows where only data has changed will be updated automatically if an add or delete row triggers the processing of a table.

In the rare situation of data updates occurring but a table never changing the row count, you should remove the table name from the ‘nonvolatile:’list in the config file

And as an alternative, just run the replicator at intervals that suit your needs without the optimisations enabled. Like this:

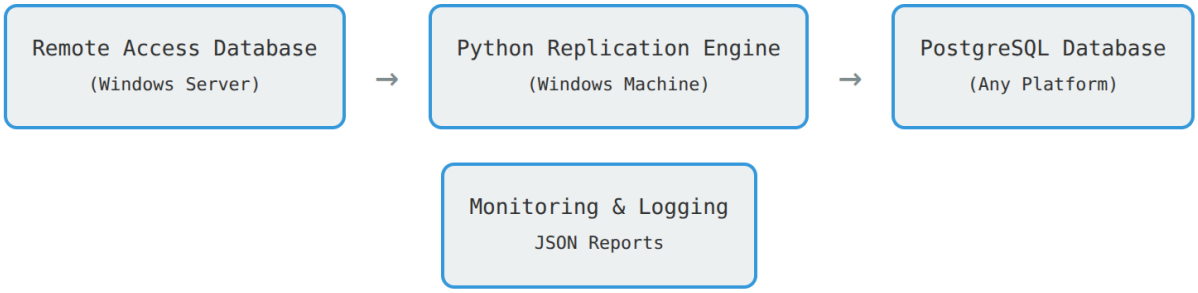
```
python ms_replicator.py –config fastupdate.yaml –sync-deleted --slow –verbose
```


ronohara@duck.com

Appendix 1 – Software for the replicator

The replicator tool runs in a Widows machine and is a Python program that uses DAO driver to let it read the MS Access database that holds your data.

The program also connects to a PostgreSQL database server running in the target machine (either MacOS , Linux or Windows Pro) using the Python library module for PostgreSQL client access.



Software required in the machine running the replicator

Mandatory

1. It Access is not installed, DAO/ODBC/Jet driver for MS Access: ODBC download (free)
2. Python 3.x (free)
3. modules installed using ‘pip’ execsql pywin32 mmh3 schedule dotenv pyyaml psycopg2-binary pandas (free)
4. the replicator program - pg_replicator.py (free)
5. replicator configuration file – replicatorconfig.yaml (adjust to suit DAO database name, PostgreSQL serve IP address, database name and credentials)

Recommended

6. UniGetUI 3.3.6 (from GitHub) - global package manager (free)

ronohara@duck.com

The replicator program checks its configuration file and decides which type of connection to used depending on which format of parameters is present the file.
The configuration file also holds the details of the PostgreSQL server and the access credentials needed. The default configuration file name is “**replicatorconfig.yaml**”

```
# Source DAO connection
DAO:
  driver: "Microsoft Access Driver (*.mdb, *.accdb)"
  database: "ContosoTR.accdb"

# Target PostgreSQL connection
postgresql:
  host: "localhost"
  port: 5432
  database: "contosotr"
  user: "replicator"
  password: "xxxxxx"

# Global settings
global:
  debug: false
  managerid: "Replicator"
  verbose: true

# Tables to replicate (commented out - will auto-discover from database)
#tables:
#   - "DimKanal"
#   - "FactSatis"

excluded:
  - "Events"

# Primary keys for tables (optional - will auto-discover)
# primarykeys:
#   xxxxxs:
#     - "customer_number"
#   Visits:
#     - "VisitID"

# Transformations - MUST be a LIST of dictionaries
transformations:
  - table: "Dimindirim"
    columns:
      - name: "Baslangic"
        transform: "YearOnly"

# Foreign keys (optional - will auto-discover)
# foreignkeys:
#   fk_patients_investigations:
#     base_table: "DimCheck"
#     reference_table: "Dimindirim"
#     base_columns:
#       - "Urunkod"
#     reference_columns:
#       - "Urunkod"
```

Transformations are of course optional, and the production settings of the desired transformations will need discussion about the actual data privacy needs.

ronohara@duck.com

You can obviously have multiple different configuration files. This allows the replication to be targeted at multiple research systems if needed..... just run the replicator once for each target system that needs to be updated, and specify the configuration file on the command line like this:

python pg_replicator.py –config secondconfig.yaml

This the default behaviour of this replicator program is that it is a generic MS Access database to PostgreSQL database converter. It is not tied to the specific data structures of the input database.

Normally processing messages and error messages are displayed in the command window during execution.

In addition most messages are also logged to a file called ‘pg-replicator.log’ .The log file is automatically rotated when it exceeds 10 megabytes in size. A limit of 10 generations of historical log files is retained. The historical log files have a “.1”. “2” or similar suffix appended to the file name.

ronohara@duck.com

Installing the replicator

There are 5 files that need to be placed in the users home directory on the Windows system that holds the Access database to be replicated.

- pg_replicator.py

- run_replicator_normal.bat

- run_replicator_refresh.bat

- run_replicat0r_help.bat

- replicatorconfig.fyaml

For ease of operation you will want to create a New->Shortcut on your desktop for each of the .bat files.

Install python (Video instructions)

In addition to installing python you need also open a CMD window and issue the following command to install the extra python modules used by the pg_replicator.py code

pip install execsql pywin32 psycopg2-binary mmh3 schedule dotenv pyyaml psycopg2-binary pandas

ronohara@duck.com

**** configuration file notes ****

The configuration file is what you normally use to control the replicator, and for most uses the only thing it needs to contain, is the Access database file name, the access credentials and and PostgreSql connection details. The ‘global’ section can also be set up if you like.

You would normally only use the command line options for things the configuration file does not support (like –schema, --debug, --trace)

The replicator copies (or updates) and transforms the tables without foreign keys first before copying tables with foreign keys that depend of data already being held in another table.

replicator command line usage

python pg_replicator.py –help

```
usage: pg_replicator.py [-h] [-V] [-c CONFIG] [-s SOURCE] [--thost THOST]
                        [--tport TPORT] [--tdatabase TDATABASE]
                        [--tuser TUSER] [--tpassword TPASSWORD] [-v]
                        [-o OUTPUT] [--debug] [--trace] [-a] [--sync-deleted]
                        [--slow] [--nonvolatile] [-S | --adjust-ms-access |
                        -l | -n | --dump | --full-refresh]

MS Access to PostgreSQL Replication Tool

options:
  -h, --help            show this help message and exit
  -V, --version         Show version and exit
  -c, --config CONFIG   Path to configuration file
  -s, --source SOURCE   MS Access database file name
  --thost THOST         PostgreSQL server host name or IP
  --tport TPORT         PostgreSQL server port number
  --tdatabase TDATABASE
                        PostgreSQL database name
  --tuser TUSER         PostgreSQL user name
  --tpassword TPASSWORD
                        PostgreSQL password
  -v, --verbose         Print informational messages
  -o, --output OUTPUT   Output file for generated YAML configuration
  --debug               Enable SQL debugging output
  --trace               Enable trace logging to file
  -a, --no-auto-index   Suppress automatic creation of indexes/constraints for
                        foreign keys (skip FK if needed)
  --sync-deleted        Synchronize deleted records from Access to PostgreSQL
  --slow                Use slower but safer deletion method (only valid with
                        --sync-deleted)
  --nonvolatile         Skip copying non-volatile tables when row counts match
  -S, --schema          Drop and recreate database, then replicate schema ONLY
  --adjust-ms-access    Adjust MS Access schema (add AutoNumber primary key to
                        tables without PK)
  -l, --list            List table names and exit
  -n, --network         Test both source and target connections
  --dump                Dump internal program data
  --full-refresh        Perform full refresh (drop and recreate all tables)
```


ronohara@duck.com

Appendix 3 – Development notes

Development and initial testing is being done within a fairly powerful HP Spectre notebook which is running Linux Mint. The test environment is a Windows 11 Pro system installed and running as a Virtual Machine created and operated using the Oracle Virtual Box software as a Type 2 Hypervisor

The Linux Mint computer can run the same (identical) software that would operate in a MacOS environment, so it is a suitable development environment.

There is also a Windows 11 Pro environment available as a Virtual Machine which in addition to having a full PostgreSQLl server installed has Access 2024 installed to allow creation of test data.

All the code development is done in the Linux machine, and is made available to the Windows VM systems (either of them) as a shared drive Z: using the capabilities of the Virtual Box software.

Very simple version control of the development output is done using the RCS version control software.

Project output is created using:

LibreOffice Writer – documents

LibreOffice Draw - diagrams

Spectacle – Linux screen shots

Kate editor – internal notes and work log

Vi editor – program editing

Greenshot – Windows screen shots

ronohara@duck.com

Evolution of the development

During development it became clear that the existing input databases can be the source of a number of problems.

Support for the .mdb database format has been progressively downgraded by Microsoft.

- Access 2024 no longer allows you to create .mdb databases

- Access 2024 no longer allows you to import .mdb database

Other issues.

- one table I used for testing had over 109,000 rows … but with the .mdb file already over 1Gb in size, the “select *” ran out of space when trying to copy it. Attempts to process 1 row at a time via ODBC which should not create any temporary table as a result set, just do not work.

- and that database is locked into presenting a form at startup for the application users … but you can compact it by running MSAccess.exe from the command line with the /compact switch at the end of the command. That reduced the size of the .mdb file to 135,020,544 bytes … clearly some housekeeping was in order. The MSAccess.exe can be installed in a variety of places, but from a CND prompt to locate the MSAccess executable

```
cd C:\
dir /s *access*exe
```

On Windows 11 Pro my Access 2024 command to compact the database was:

C:\Program Files\Microsoft Office\root\Office16\MSAccess.exe" "C:\Users\ronoh\xxxxxx.mdb" /compact

The ‘huge table’ problem encountered while using the ODBC driver as the basis for the replicator was resolved this way.

However, after revising the replicator to use the DAO library rather than ODBC, the ‘huge table’ problem no longer exists and the replicator works properly against the original database.

ronohara@duck.com

Test data

A really small test database

Available with the replicator is a very small MSAccess database called “Test.accdb” and a replicator configuration file called “test.yaml” that assumes you are running the PostgreSql under Windows in the same machine. You need to edit that to set the credentials (user and password) to the user you set up for this process. Any PostgreSQL user with the ‘createdb’ authorisation will do.

You can start verifying the installation by working in a CMD window .. in a the directory with the ‘pg_replicator.py’ program, the ‘Test.accdb’ file and the ‘test.yaml’ configuration file.

#1 check the MS Access database is accessible

```
python pg_replicator.py -c test.yaml –list
```

#2 check that your network connection to PostgreSQL is ok

```
python pg_replicator.py -c test.yaml –network
```

Then you can get the replication done

#3 convert the MS Access database schema to PostgreSQL

```
python pg_replicator.py -c test.yaml –schema
```

#4 replicate the MS Access data to PostgreSQL

```
python pg_replicator.py -c test.yaml –verbose
```

You should review the ‘pg_replicator.log’ to see how well your data was replicated. There may be problems reported. Especially for tables that have MS relationships without suitable indexes and with ‘enforce referential integrity’

Access tables without any keys (primary key OR unique key) do not replicate properly …. if you are able to revise your MS Access database to correct this please do so .. only a DB analyst can properly assess the impact of any changes made. The replicator has an option –adjust-ms-access which can make adjustments to the Access schema, but it can not assess the potential negative performance impacts.

ronohara@duck.com

Other test databases.

The website Kaggle has a huge library of dataset that can be useful for testing

One particular MS Access dataset that was helpful in testing the replicator was the ContosoTR dara.

It is a very *large* database, with table sizes well in excess of the commonly recommended maximum of 100,000 rows.

This is the config file for that database: (replicator-ConosoTR.yaml)

```
# replicatorconfig.yaml
# Configuration for ODBC to PostgreSQL Database Replication System

# Source DAO connection
DAO:
  driver: "Microsoft Access Driver (*.mdb, *.accdb)"
  database: "ContosoTR.accdb"

# Target PostgreSQL connection
postgresql:
  host: "localhost"
  port: 5432
  database: "contosotr"
  user: "replicator"
  password: "badpassword"

# Global settings
global:
  debug: false
  managerid: "Replicator"
  verbose: true

# Tables to replicate (commented out - will auto-discover from database)
#tables:
#   - "DimKanal"
#   - "FactSatis"

excluded:
```


ronohara@duck.com

```
  - "Events"

# Primary keys for tables (optional - will auto-discover)
# primarykeys:
#   xxxxxs:
#     - "customer_number"
#   Visits:
#     - "VisitID"

# Transformations - MUST be a LIST of dictionaries
transformations:
  - table: "Dimindirim"
    columns:
      - name: "Baslangic"
        transform: "YearOnly"

# Foreign keys (optional - will auto-discover)
# foreignkeys:
#   fk_patients_investigations:
#     base_table: "DimCheck"
#     reference_table: "Dimindirim"
#     base_columns:
#       - "Urunkod"
#     reference_columns:
#       - "Urunkod"
```


ronohara@duck.com