Using PostgreSQL for Data Privacy and Security

September 19, 2019
PostgresConf SV

Agenda

1. Intro
2. Foreign Data Wrappers
3. Row and Column Based Access
4. Roles & Purposes
5. Handling Privacy
    1. De-identification (Masking)
    2. K-anonymization
    3. Differential Privacy
    4. Identifying Private Data
6. Auditing/Logging

whoami

- Mason Sharp
- Worked on public and private PostgreSQL forks such as Postgres-XL
  - Including a geographically distributed database with differing regional regulations
- Work at Immuta
  - Data Governance Platform

Data Privacy

Privacy Regulations

- GDPR - General Data Protection Regulation (Europe)
- CCPA - California Consumer Privacy Act
  - Becomes effective January 1, 2020
- HIPAA - Health Insurance Portability and Accountability Act

Privacy Regulations

- GDPR - General Data Protection Regulation (Europe)
- CCPA - California Consumer Privacy Act
  - Becomes effective January 1, 2020
- HIPAA - Health Insurance Portability and Accountability Act

If you are not doing anything about privacy now, you may be doing so soon

Privacy Challenges

- Comply with local regulations and internal policies
- Make data useful
  - Gain insight
    - Better healthcare, better products and services at a lower cost
  - Make data available for Data Scientists
    - Machine Learning models need detailed data - a privacy risk
    - Differential Privacy techniques add noise to data sets
      - Too much noise, not useful; too little, privacy risk
- Provide access to data quickly
  - Long ETL processes, developer resources
  - Burdensome approval processes

Can PostgreSQL Help?

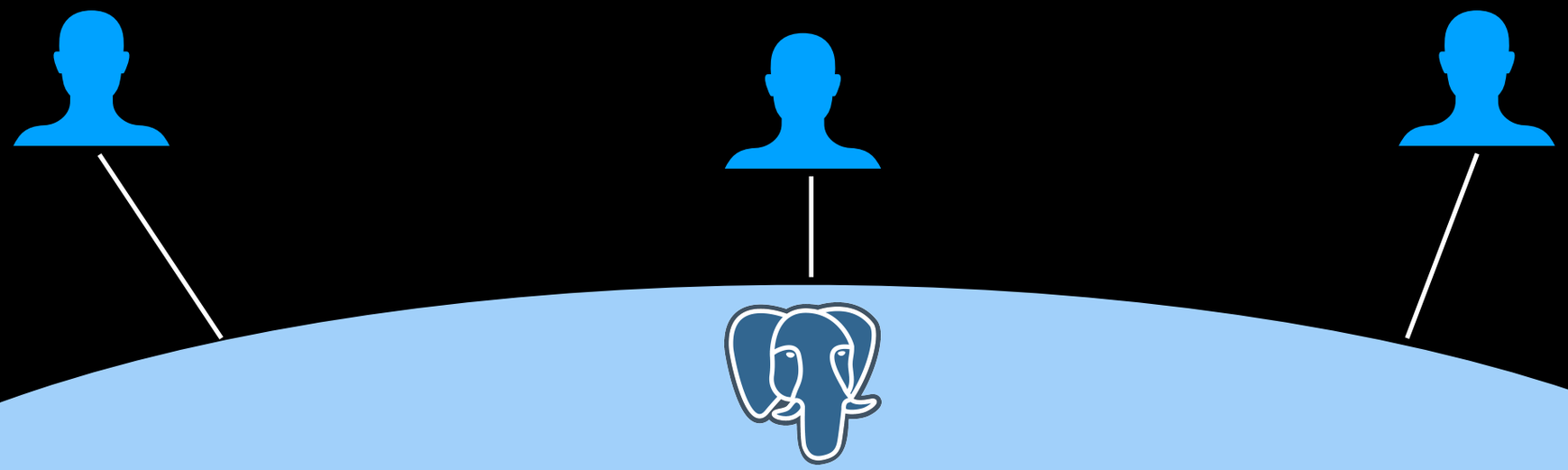
Can PostgreSQL Help?

Yes

Disclaimer: to be clear, the examples presented here are not necessarily how Immuta is implemented, the examples just provide some ideas for basic functionality for protecting data privacy

Foreign Data Wrappers

PostgreSQL as Data Hub



Roles

Users

Purposes

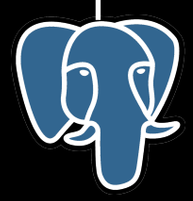
Views

Foreign Sources

Materialized Views

Foreign Sources

Foreign Sources



Foreign Data Wrappers

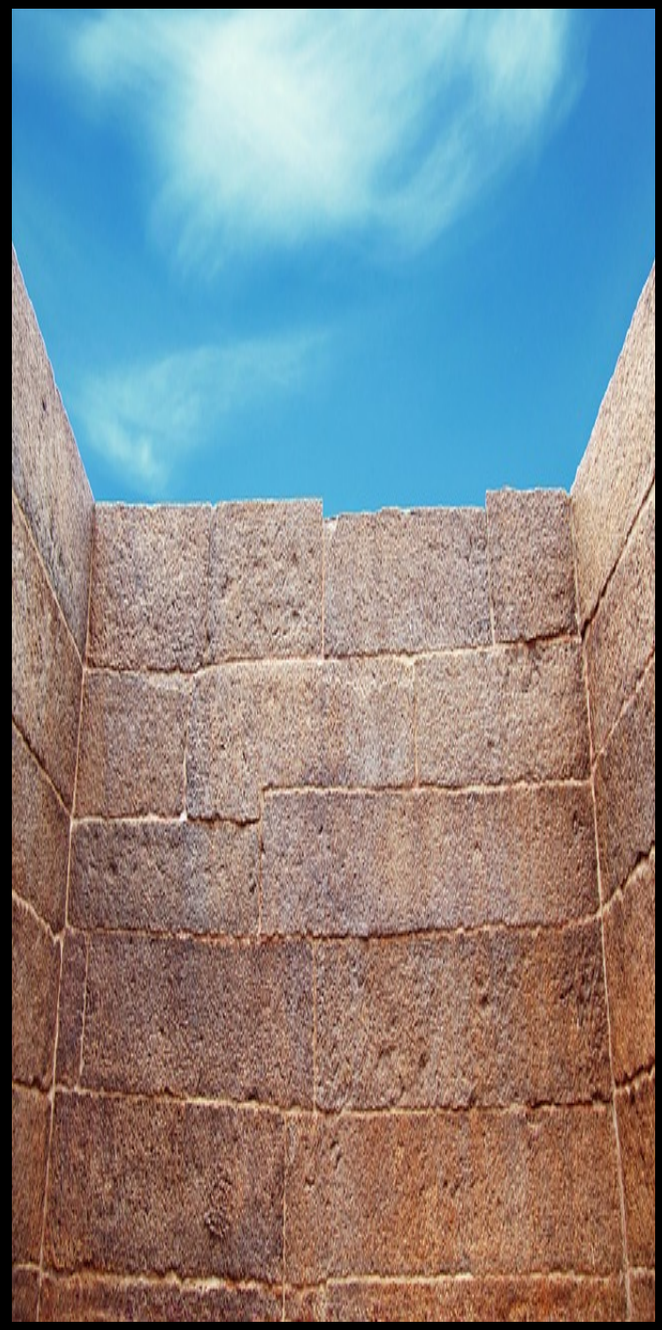
- Allows for querying external data sources, not just PostgreSQL
  - https://wiki.postgresql.org/wiki/Foreign_data_wrappers
- SQL based
  - PostgreSQL, MySQL, Oracle, MS SQL Server, Sybase, SQLite, MonetDB, ClickHouse
  - odbc_fdw
- NoSQL based
  - Need to give structure to unstructured (JSON) data
  - MongoDB, CouchDB, Redis, Riak
- File based
  - CSV, Fixed, JSON
- Etc
  - Hadoop (HDFS via Hive), HDFS

FDW Advantages

- Manage access to external data all in one place
- Join data against multiple external data sources at the same time (Data Federation)
- Potentially avoid complex ETL processes
- Potentially avoid building a Data Lake
    - PostgreSQL acts as a virtual Data Lake

FDW Limitations

- Perfomance
- What cannot be “pushed down” in the SQL statement is done at the PostgreSQL level
  - Sometimes not much can be pushed down- varies by FDW
  - May or may not include aggregates, joins, functions
  - Example: If a join cannot be pushed down, it may bring back millions+ of rows from both tables to join locally in PostgreSQL
  - BUT, that **may** be ok if this is just used for occasional read-only queries hitting production replicas, not high volume TPS



odbc_fdw

- Supports any ODBC data source
- Slower than native connectors, but good for when none available
- Pushdown is very limited
- Requires installing corresponding ODBC drivers on server where PostgreSQL is running

postgres_fdw

- Good push down
  - (Some) joins, aggregates, groups
- Also can be used with Greenplum, Redshift

Views

- Can reference foreign tables
- Allows control of what data can be accessed
- Big Caveat: may lead to an explosion of views based on what users are able to access
    - Maintenance headache

Materialized View

- Keep a copy of remote data in PostgreSQL, periodically refresh
- Run slow queries less often, use the view instead
- Also may help against privacy attacks because more difficult to get deltas of data source within short time frame
- If masking based on role, need a base materialized view, then a conditional masked view on top of that

Table Access

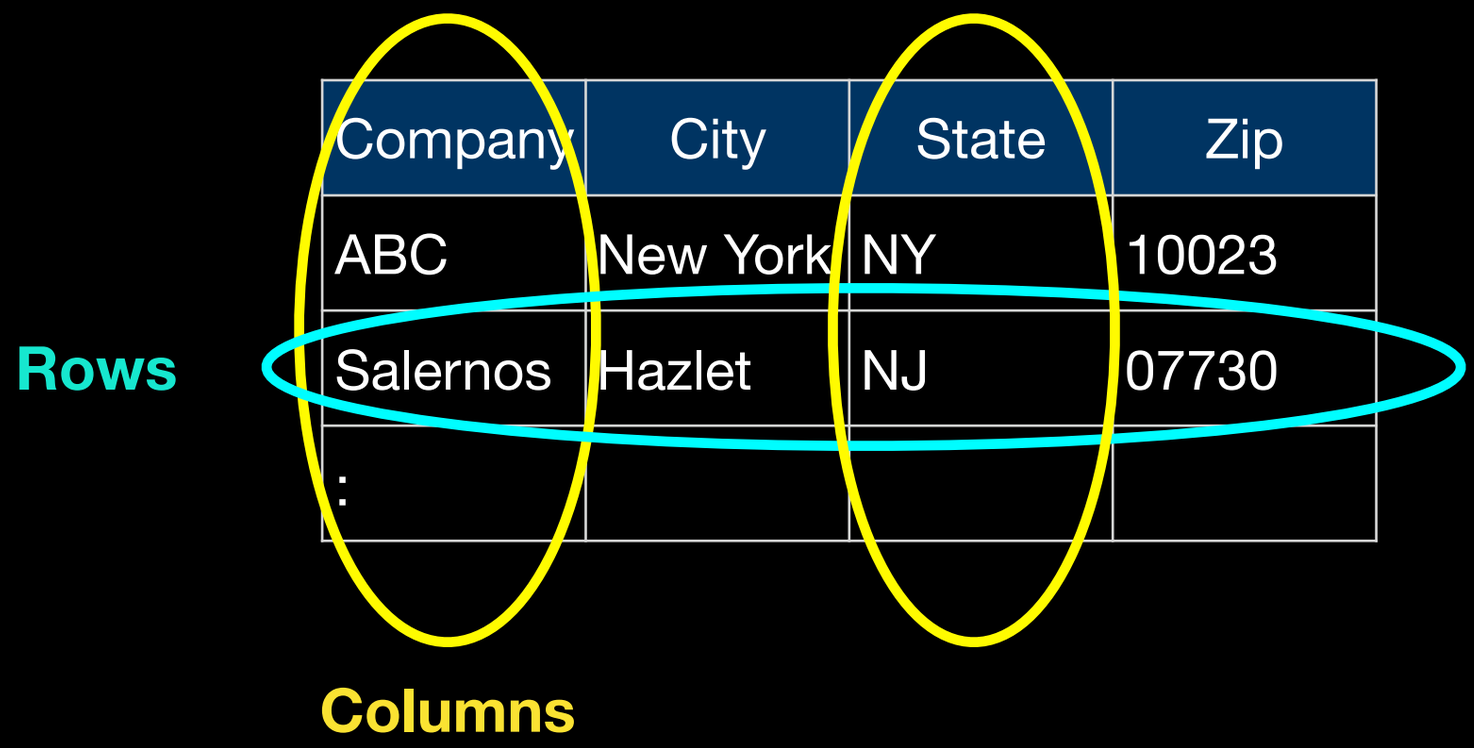
Table Access

- Via GRANT and REVOKE
  - CREATE ROLEs for users

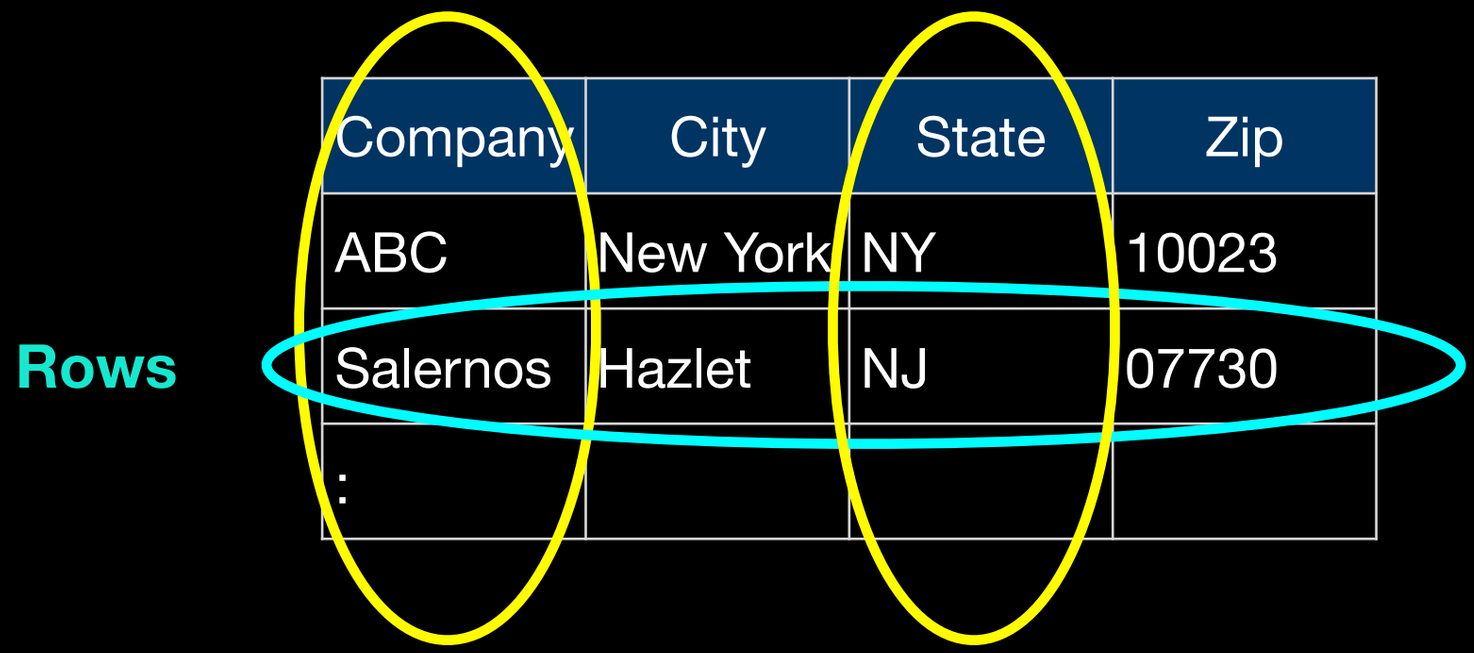
Row and Column Access

Company	City	State	Zip
Luigis	Aberdeen	NJ	07747
Salernos	Hazlet	NJ	07730
:			

Row and Column Access



Row and Column Access



- Limit row based access
  - Via Row Level Security
  - Via views

- Limit column based access
  - Via native column-level permissions
  - Via views
  - Via masking

ROW LEVEL SECURITY

- `CREATE POLICY name ON table_name`

  `[ TO { role_name | PUBLIC | CURRENT_USER |`

  `SESSION_USER } [, ...] ]`

  `[ USING ( using_expression ) ]`

- `ALTER TABLE`
  - `DISABLE/ENABLE ROW LEVEL SECURITY`
    - “If enabled and no policies exist for the table, then a default-deny policy is applied. Note that policies can exist for a table even if row level security is disabled - in this case, the policies will NOT be applied and the policies will be ignored.”

Row Level Policies

```
CREATE ROLE user1 LOGIN PASSWORD
'password';


ALTER TABLE company ENABLE ROW LEVEL
SECURITY;


CREATE POLICY company_nj_access ON
company
  TO user1
  USING (state = 'NJ');

GRANT SELECT ON company TO user1;
```

Row Level Policies

```
CREATE ROLE user1 LOGIN PASSWORD
'password';


ALTER TABLE company ENABLE ROW LEVEL
SECURITY;


CREATE POLICY company_nj_access ON
company
  TO user1
  USING (state = 'NJ');

GRANT SELECT ON company TO user1;

SELECT * FROM company;
 company  |   city   | state |  zip
----------+----------+-------+-------
 ABC      | New York | NY    | 10023
 Salernos | Hazlet   | NJ    | 07733
```

```
— user1
SELECT * FROM company;
 company  |   city   | state |  zip
----------+----------+-------+-------
 Salernos | Hazlet   | NJ    | 07733
```

Row Level Policies with Column Restrictions

```
CREATE ROLE user1 LOGIN PASSWORD
'password';


ALTER TABLE company ENABLE ROW LEVEL
SECURITY;


CREATE POLICY company_nj_access ON
company
  TO user1
  USING (state = 'NJ');

GRANT SELECT ON company TO user1;

SELECT * FROM company;
 company  |   city   | state |  zip
----------+----------+-------+-------
 ABC      | New York | NY    | 10023
 Salernos | Hazlet   | NJ    | 07733
```

```
REVOKE SELECT ON company FROM user1;

GRANT SELECT (company, state) ON
company TO user1;
```

```
– user1

SELECT * FROM company;

ERROR:  permission denied for table
company


SELECT company, state FROM company;
 company  | state
----------+-------
 Salernos | NJ
```

Using Views for Row & Column Restrictions

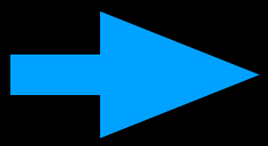
```
CREATE VIEW v_company AS
SELECT company, state
  FROM company
 WHERE state = 'NJ';

GRANT SELECT ON v_company TO user1;
```

```
— user1
SELECT * FROM v_company;
 company  | state
----------+-------
 Salernos | NJ
```

Roles and Purposes

Purposes

- For what purpose is the data being accessed?
- A user may have access to see more of the data, but depending on the purpose of why the data is being accessed, only the needed data for the purpose should be accessible
- Consider allowing only one purpose at a time for users
- Strict adherence to this for compliance



Can be achieved via ROLES

Roles

- CREATE ROLE and CREATE USER
  - Very similar, but CREATE USER implies LOGIN privileges
- A role may be given multiple other roles

```
CREATE ROLE cfo WITH IN ROLE accounting, payroll
```

Or

```
CREATE ROLE cfo

ALTER ROLE accounting WITH ROLE cfo

ALTER ROLE payroll WITH ROLE cfo
```

Roles

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

CREATE ROLE WITH NOINHERIT & SET ROLE

- WITH INHERIT (the default)
    - “automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of”
- WITH NOINHERIT
    - Only can act in the role via SET ROLE

      SET ROLE accounting;

      SET ROLE payroll;
    - Forces the user to access data for a specific purpose
    - View of the data may change depending on current role (purpose), even though user may have access to multiple roles

Protect Data via Roles & Logins

1. Create roles for users with login privilege and WITH NOINHERIT, without access to any tables

2. Create roles without login privileges to access table data based on purpose

3. Grant purpose-based roles to user roles

4. Users may use only one role at a time

Handling Privacy

Handling Privacy:
De-identification (Masking)

Why Mask?

- Just block/hide access to a column instead?
- Sometimes identifiers are mapped to another value so rows can be later identified while not revealing the original value
  - Example: Social security number
    - Could use reversible masking where users may request to have a specific data element unmasked from data governor
- Organizations sometimes have pre-canned queries and reports that include sensitive columns
  - Masking (if practical) allow reports to be used for users who should not have access to the raw values in these columns, and see the masked value instead

Masking Types

Type	
NULL	
Constant	Example: <REDACTED>
Hash	Example: AB3D07F3169CCBD0ED6C4B45DE21519F9F938C72D24124998AAB949CE83BB51B
Regular Expression	Example: 800-655-0982 -> 800-xxx-xxxx
Reversible Encryption	Can request value to be decrypted by data governor
Format Preserving Encryption	Data “looks” real, but it is encrypted Example: 800-655-0982 -> 415-555-1212
Grouping (Rounding)	Example (round month): date_of_birth 1999-04-12 -> 1999-04-01

```
— Mask column if user does not have SELECT privileges
CREATE OR REPLACE FUNCTION mask_text(
    table_name VARCHAR, column_name VARCHAR,
    column_value VARCHAR, masked_value VARCHAR)
RETURNS VARCHAR AS $$
    BEGIN
        RETURN CASE
            WHEN pg_catalog.has_column_privilege
                (CURRENT_USER, table_name, column_name, 'select')
                THEN column_value
            ELSE masked_value
        END;
    END;
$$ LANGUAGE PLPGSQL;
```

```
CREATE OR REPLACE VIEW v_lineitem AS
SELECT mask_int('lineitem', 'l_orderkey', l_orderkey, NULL) AS l_orderkey,
    mask_int('lineitem', 'l_partkey', l_partkey, NULL) AS l_partkey,
    mask_int('lineitem', 'l_suppkey', l_suppkey, NULL) AS l_suppkey,
    mask_int('lineitem', 'l_linenumber', l_linenumber, NULL) AS l_linenumber,
    mask_numeric('lineitem', 'l_quantity', l_quantity, NULL) AS l_quantity,
    mask_numeric('lineitem', 'l_extendedprice', l_extendedprice, NULL) AS l_extendedprice,
    mask_numeric('lineitem', 'l_discount', l_discount, NULL) AS l_discount,
    mask_numeric('lineitem', 'l_tax', l_tax, NULL) AS l_tax,
    mask_text('lineitem', 'l_returnflag', l_returnflag, NULL) AS l_returnflag,
    mask_text('lineitem', 'l_linestatus', l_linestatus, NULL) AS l_linestatus,
    mask_date('lineitem', 'l_shipdate', l_shipdate, NULL) AS l_shipdate,
    mask_date('lineitem', 'l_commitdate', l_commitdate, NULL) AS l_commitdate,
    mask_date('lineitem', 'l_receiptdate', l_receiptdate, NULL) AS l_receiptdate,
    mask_text('lineitem', 'l_shipinstruct', l_shipinstruct, NULL) AS l_shipinstruct,
    mask_text('lineitem', 'l_shipmode', l_shipmode, NULL) AS l_shipmode,
    mask_text('lineitem', 'l_comment', l_comment, NULL) AS l_comment
FROM lineitem;
```

Performance of Masking

- Table lineitem has 6 million rows

```
dbt3=# select count(*) from lineitem where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 754.323 ms
```

Performance of Masking

- Table lineitem has 6 million rows

```
dbt3=# select count(*) from lineitem where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 754.323 ms

dbt3=# select count(*) from v_lineitem where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 333254.483 ms (05:33.254)
```

Overhead of masking function is high, time went from < 1 second to 5 and a half minutes!

New View

- Get privileges just once, use CROSS JOIN

```
CREATE VIEW v_lineitem2 AS
SELECT CASE
          WHEN acc_l_quantity THEN l_quantity
          ELSE NULL
       END l_quantity
  FROM lineitem
       CROSS JOIN
       (SELECT pg_catalog.has_column_privilege(CURRENT_USER,
               'lineitem', 'l_quantity', 'select') acc_l_quantity)
       access_perms;
```

Performance of Masking

- Table lineitem has 6 million rows

```
dbt3=# select count(*) from lineitem where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 754.323 ms
```

```
dbt3=# select count(*) from v_lineitem where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 333254.483 ms (05:33.254)

dbt3=# select count(*) from v_lineitem2 where l_quantity > 10;
  count
---------
 4802275
(1 row)
Time: 4109.049 ms (00:04.109)
```

Down to 4.1 seconds

Effects of Masked Columns

- WHERE
  - View uses functions, so if lastname is masked to a constant for a user, a query like

```
SELECT *
FROM v_patient
WHERE lastname = 'Jones'
```

returns no rows

Effectively:

```
WHERE '<REDACTED>' = 'Jones'
```

- GROUP BY
  - Constant masking: all rows for the grouped column go into the same group

Masked Joins

- WARNING: permitting linking data may expose identity!
    - Example: age is one table, street name in other table
- If masked to a constant, joins will return all rows
- If masked to NULL, joins will return no rows
- Could consider creating views that include the join columns unmasked, but masks SELECT-ed columns
    - Maintenance headache to create many views of all combinations of masked joins
        - table1 - table2
        - table2 - table3
        - table1 - table2 - table3
        - Etc.

Handling Privacy:

K-anonymity

Aggregate Views

- Create views that only allow viewing aggregate data and hide detailed information
- However, information can be gleaned by periodically repeating the query
    - If you know John Smith was just added, rerunning the aggregate where the count increased by one will expose information by comparing aggregate function results before and after
    - Using materialized views will add some protection
        - Result is cached, updated less frequently
        - Repeated queries return the same result until refreshed
        - Better performance
- Which groups of attributes to use?

K-anonymity

- Reveal level of information such that each row cannot be distinguished from k other rows in the data set
- Which columns to anonymize to satisfy? Think of data elements as being in three groups:
  - Identifiers
    - Example: first name, last name, SSN
    - Always suppress or mask
  - Quasi-identifiers
    - Example: zip code, gender, occupation
    - Target these for k-anonymity— if we know enough with a small number of rows we can identify someone
  - Other Target Interesting Data
    - With the other data elements protected, ok to release
    - Example: Blood enzyme levels

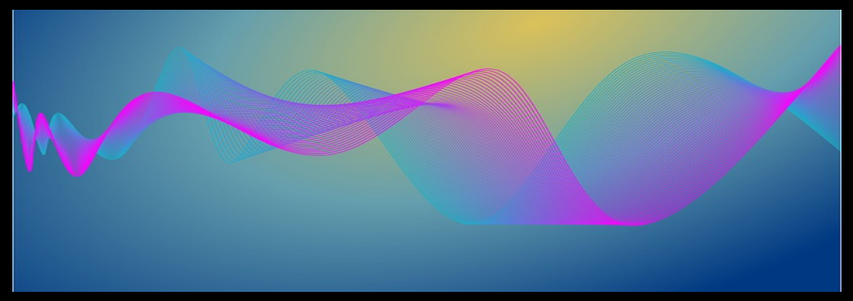
K-anonymity

- It may be helpful to break up some column values into ranges
  - Example: age: 20 - 29, 30 - 39, etc.
  - Example: zipcode: (substr(zipcode, 1, 2))
- Ideally, one could examine all data and generate groups to ensure k-anonymity that are as small as possible
- If there is less than k rows for the group, use NULL

```
SELECT * FROM v_patient_anon;

 firstname  |  lastname  |   city   |  county   | state_cd | has_cancer | has_diabetes
------------+------------+----------+-----------+----------+------------+--------------
 <REDACTED> | <REDACTED> |          | Middlesex | NJ       | f          | t
 <REDACTED> | <REDACTED> |          | Middlesex | NJ       | f          | f
 <REDACTED> | <REDACTED> |          | Middlesex | NJ       | f          | t
 <REDACTED> | <REDACTED> |          |           | NJ       | t          | f
 <REDACTED> | <REDACTED> |          |           | NJ       | t          | t
 <REDACTED> | <REDACTED> |          |           | NJ       | t          | f
 <REDACTED> | <REDACTED> |          |           | NJ       | t          | t
 <REDACTED> | <REDACTED> | New York | New York  | NY       | t          | t
 <REDACTED> | <REDACTED> | New York | New York  | NY       | t          | f
 <REDACTED> | <REDACTED> | New York | New York  | NY       | t          | f
:
```

Adding Noise to Queries

Adding Noise to Queries



- Numbers:

```
SELECT SUM(new_net_paid)
   FROM (SELECT
            ws_net_paid * (1 + 0.1*(random()-0.5))
         FROM web_sales) AS new_net_paid;
```

- random() is <u>not</u> deterministic here

- If rerunning multiple times we can take an average to get at the real value

Adding Noise to Queries

Either:

- Use deterministic own random number (cryptographic hash + salt)
- Seed deterministically with setseed() function

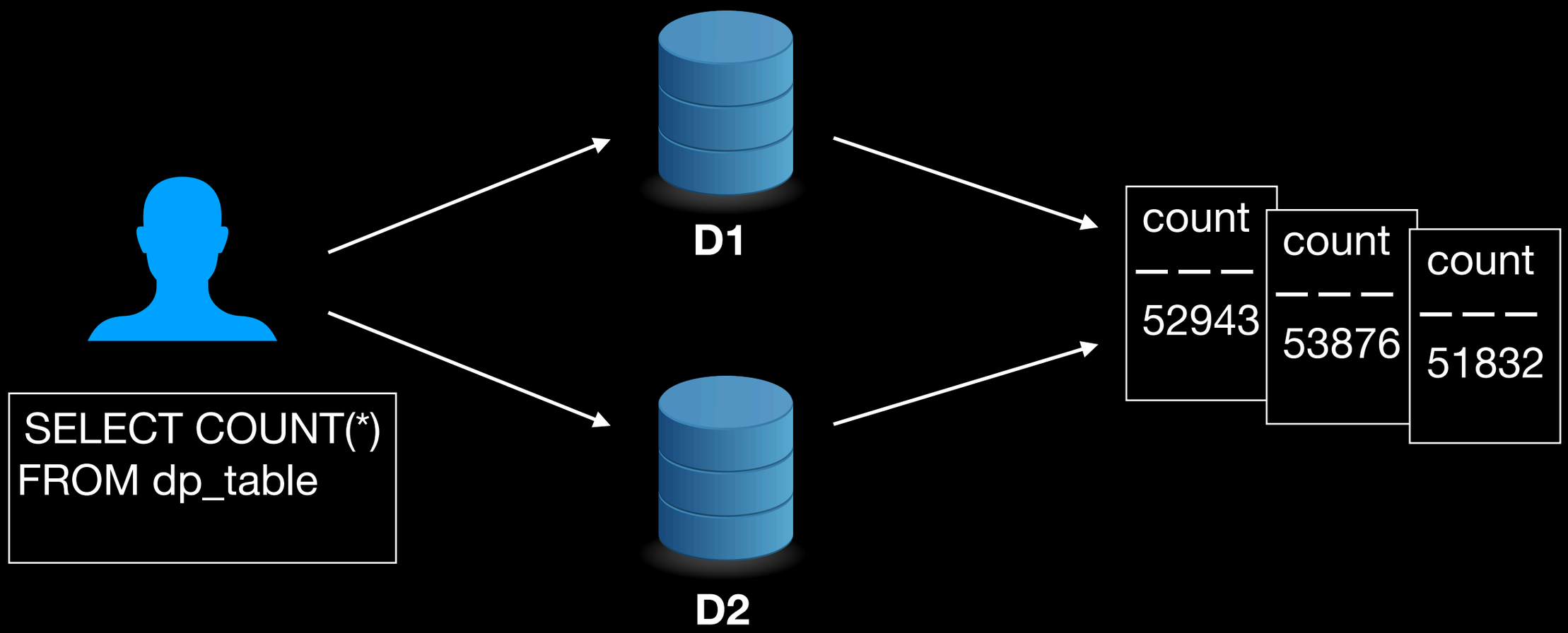
```
SELECT ws_order_number, SUM(ws_net_paid), SUM(new_net_paid)
  FROM (SELECT ws_order_number,
               setseed((ws_bill_customer_sk + 12345)/
                       power(2,64)),
               ws_net_paid,
               ws_net_paid*(1 + 0.1*(random()-0.5)) new_net_paid
  FROM web_sales) as new_sales
GROUP BY ws_order_number;
```

- setseed() is applied for each row based on customer id
  - Nitpick: brute force risk
- Hide in view, do not allow users to see query plan

Differential Privacy

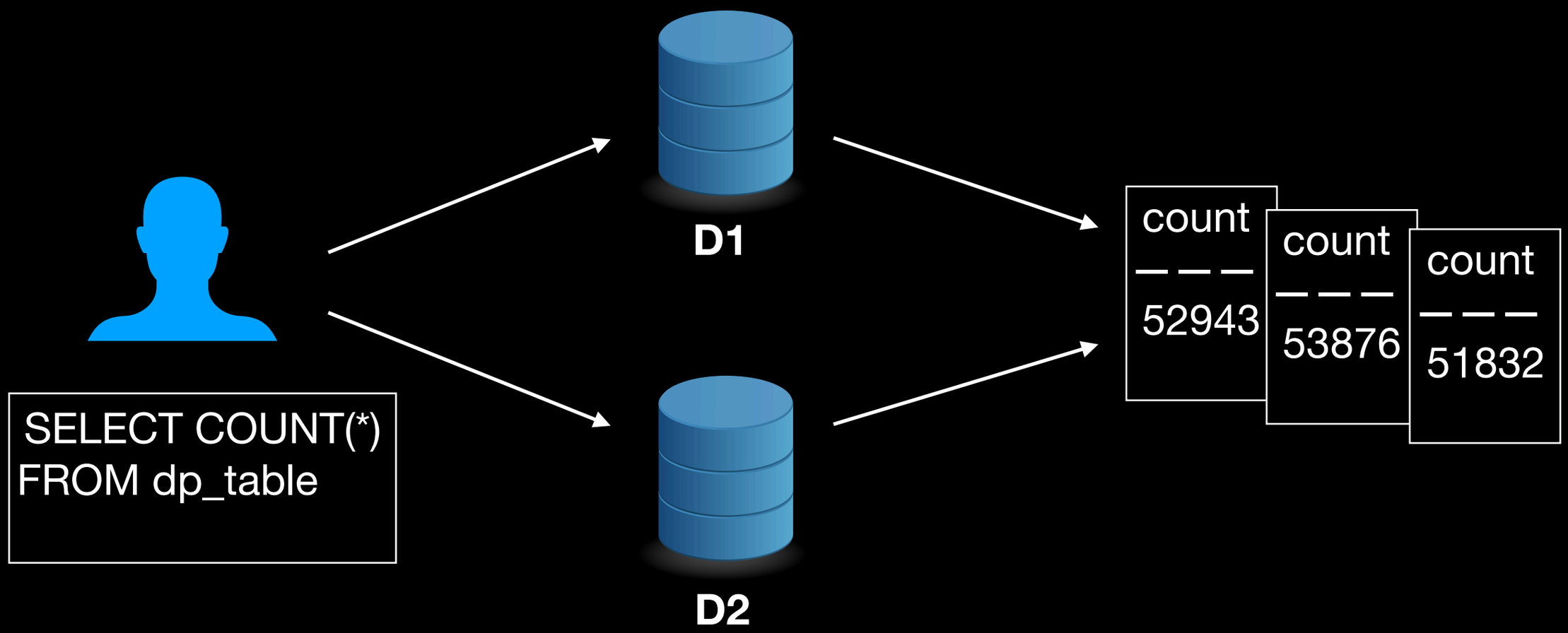
Differential Privacy

- Given two databases D1 and D2 with one additional row in D2, add enough noise such that querying either will return a result indistinguishable from the other
  - User unable to determine if row exists in data set



Differential Privacy

“Differential privacy prevents an interested third party from being able to infer, with statistical confidence, whose record might be among the input, given the query output and unlimited external knowledge”



Differential Privacy

- Differential Privacy is just a definition, not an algorithm
- Mechanisms
  - Laplace Mechanism
  - Exponential Mechanism
  - Posterior Sampling
  - Randomized Response

Differential Privacy

- Instead of plain random noise shown earlier, use ε (epsilon) differential privacy
- Mathematically ensures the same privacy if an individual's data is present or removed
- Caveats:
  - Repeating the query often enough may yield the true value
    - Ideally would want the ability to track a privacy budget
    - Query caching or materialized views can return the same result
      - May be able to be defeated by slightly modifying the query
    - Auditing can show that a user repeated query
- Not enough noise - private data revealed
  - Fewer rows requires more noise
- Too much noise - data set not useful
  - Analytics fail to gain insight
  - Machine Learning models ineffective

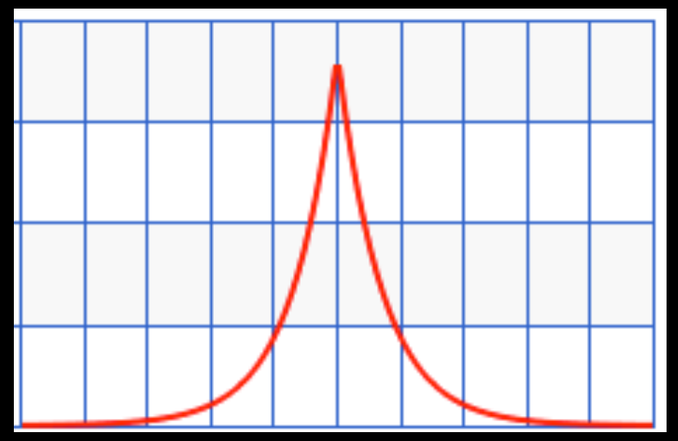
Laplace Mechanism

- ε (epsilon): the higher epsilon is, more noise and more privacy (privacy vs utility trade off)
- sensitivity: how much a column value can change for additional row
  - For COUNT(), sensitivity should be one (A row is in the data set or not)
  - For SUM(), sensitivity should be greatest(abs(max_value), abs(min_value))
- u = random sample [0,1]
- s = u - 0.5
- Added noise

  SELECT agg(some_column) + sensitvity/epsilon * sign(s) * ln(1-2*abs(s)) FROM..

  Example:

  SELECT COUNT(*) + 1/2 * sign(s) * ln(1-2*abs(random() - 0.5)) FROM dp_table



Laplace Distribution

Randomized Response

- Used for a set of possible answers
    - Example: cancer_found: true | false
- Flip a coin
    - If heads, return actual value
    - If tails, flip a coin again
        - Return true for heads, false for tails
- 2 * (Reported Trues - 0.25 * count) =~ Actual Trues
    - 0.25: 50% of the answers will be random, 50% of those true
- Individuals get plausible deniability, can state the answer was random

Anonymous Functions anon_func PostgreSQL Extension

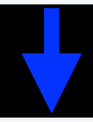
- Google announced a PostgreSQL extension (Sept 4, 2019)

  https://github.com/google/differential-privacy/tree/master/differential_privacy/postgres

- Defines new anonymous functions
  - ANON_COUNT, ANON_SUM, ANON_AVG, ANON_VAR, ANON_STDDEV, ANON_NTILE
- Requires careful manual rewriting of queries
- Could create views and restrict access to raw table
- Could access remote tables via FDWs, may pull over more rows

Anonymous Functions
anon_func PostgreSQL Extension

```
SELECT fruit, COUNT(fruit)
FROM FruitEaten
GROUP BY fruit;
```



```
SELECT result.fruit, result.number_eaten
FROM (
  SELECT per_person.fruit,
    ANON_SUM(per_person.fruit_count, LN(3)/2) as number_eaten,
    ANON_COUNT(uid, LN(3)/2) as number_eaters
    FROM(
      SELECT * , ROW_NUMBER() OVER (
        PARTITION BY uid
        ORDER BY random()
      ) as row_num
      FROM (
        SELECT fruit, uid, COUNT(fruit) as fruit_count
        FROM FruitEaten
        GROUP BY fruit, uid
      ) as per_person_raw
    ) as per_person
  WHERE per_person.row_num <= 5
  GROUP BY per_person.fruit
) as result
WHERE result.number_eaters > 50;
```

Manually Rewritten Query

(also accounting for more than one row per person)

Identifying Private Data

Identify Private Data

Tagging Services:

- Microsoft Presidio
- Google DLP API (Data Loss Prevention)
- Amazon Macie

InfoType	Description
`AGE`	An *age* measured in months or years.
`ALL_BASIC`	A special type that triggers all of the most common infoType detectors. Some infoTypes are not included, as they are specialized or tend to generate more noise when used generally. `ALL_BASIC` is good for initial testing and exploration, but for production use you should choose and configure the individual types that are applicable to your needs.
`CREDIT_CARD_NUMBER`	A *credit card number* is 12 to 19 digits long. They are used for payment transactions globally.
`CREDIT_CARD_TRACK_NUMBER`	A *credit card track number* is a variable length alphanumeric string. It is used to store key cardholder information.
`DATE`	A *date*. This includes most date formats, as well as the names of common world holidays.
`DATE_OF_BIRTH`	A *date of birth*.
`DOMAIN_NAME`	A *domain name* as defined by the DNS standard.
`EMAIL_ADDRESS`	An *email address* identifies the mailbox that emails are sent to or from. The maximum length of the domain name is 255 characters, and the maximum length of the local-part is 64 characters.
`ETHNIC_GROUP`	A person's *ethnic group*.

Google DLP API

- Caveat: Must submit private data for tagging

Auditing

Auditing

Try to proactively restrict what users can see..

- But, also capture user queries
    - Who ran what queries?
    - Which tables were accessed?
    - When were the queries run?
        - Is a user repeatedly running the same or similar queries in a short time frame?

Auditing - logging queries

- postgresql.conf logging options:

```
logging_collector = on
log_directory = log
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_statement = 'all'  # none,ddl,mod,all
log_line_prefix = '%m %u %d %p '
 #(timestamp, user, database, process id)
```

What about pg_stat_statements?

- Collects statistics about statements run
- Useful for performance tuning
- May provide some insight, but
  - Aggregates together unique statements
  - Only tracks a maximum number of unique statements

pgAudit

- Free open source extension available for PostgreSQL
- Allows for more fine grained capturing of user activity
    - Example: each time a table was accessed, not just the entire query
- Harder to fool compared to generating dynamic SQL EXECUTE statements

pgAudit - reported fields

Output Field	Description
AUDIT_TYPE	SESSION or OBJECT
STATEMENT_ID	Unique statement ID for session
SUBSTATEMENT_ID	Sequential ID for each sub-statement
CLASS	e.g. READ, WRITE, DDL
COMMAND	e.g. ALTER TABLE, SELECT
OBJECT_TYPE	TABLE, VIEW, etc.
OBJECT_NAME	e.g. public.account
STATEMENT	Statement executed
PARAMETER	Parameters if pgaudit.log_parameter is set

pgAudit - sample output

```
2019-08-21 16:07:38.325 EDT msharp test1 84870
LOG:  AUDIT: SESSION,1,1,READ,SELECT,TABLE,
public.company1,select * from company1 limit
1;,<none>
```

- Timestamp, user, database, and process id (or others like application) are not in pgAudit fields since they can be specified via `log_line_prefix`.

- Output can be on multiple lines, careful if parsing output

pgAudit - use CSV logging?

- Alternatively:

```
log_destination = csvlog
```

https://www.postgresql.org/docs/11/runtime-config-logging.html#RUNTIME-CONFIG-LOGGING-CSVLOG

- But the “AUDIT” info appears as a string, will need to pull out info

Table for audit data

```
CREATE TYPE pgaudit_type AS ENUM ('SESSION', 'OBJECT');

CREATE TYPE pgaudit_class AS ENUM
  ('READ', 'WRITE', 'FUNCTION',
   'ROLE', 'DDL', 'MISC', 'MISC_SET');

CREATE TABLE pgaudit_log (
    db_timestamp timestamp,
    username varchar,
    db_name varchar,
    pid int,
    audit_type pgaudit_type,
    statement_id bigint,
    substatement_id int,
    class pgaudit_class,
    command varchar,
    object_type varchar,
    object_name varchar,
    statement varchar,
    parameter varchar);
```

audit_csv.py - formatting pgAudit output

```
import csv
import sys
# csv.reader handles multi-line CSV input
reader = csv.reader(sys.stdin)
for row in reader:
    # get last word of first column
    if 'AUDIT: ' in row[0]:
        # PostgreSQL prefix is in element 0
        first = row[0].split(' ')
        db_timestamp = first[0] + ' ' + first[1] + ' ' + first[2]
        user = first[3]
        dbname = first[4]
        process = first[5]
        audit_type = first[-1]   # SESSION or OBJECT
        print('{0},"{1}","{2}",{3},{4},{5},{6},{7},"{8}","{9}","{10}","{11}","{12}"'
           .format(db_timestamp, user, dbname, process, audit_type,row[1],row[2],row[3],
           row[4],row[5],row[6], row[7].translate(str.maketrans({"\"": r"\""})),row[8]))
```

Load individual files into pgaudit_log

```
cat $LOGFILE |

python audit_csv.py |

psql -c "COPY pgaudit_log FROM STDIN WITH
(format csv, quote '\"', escape '\\')"
$DEST_DB
```

Query pgaudit_log Table

```
# Query must frequently accessed table by user

SELECT username, object_type, object_name, COUNT(*)
  FROM pgaudit_log
 WHERE object_type = 'TABLE'
   AND object_name != 'public.pgaudit_log'
   AND db_timestamp BETWEEN '2019-08-01' AND '2019-08-22'
GROUP BY 1,2,3
ORDER BY count(*) DESC;
```

```
 username  | object_type |    object_name    | count
-----------+-------------+-------------------+-------
 baduser   | TABLE       | public.visits     |  4300
 msneaky   | TABLE       | public.patient    |  1211
 :
```

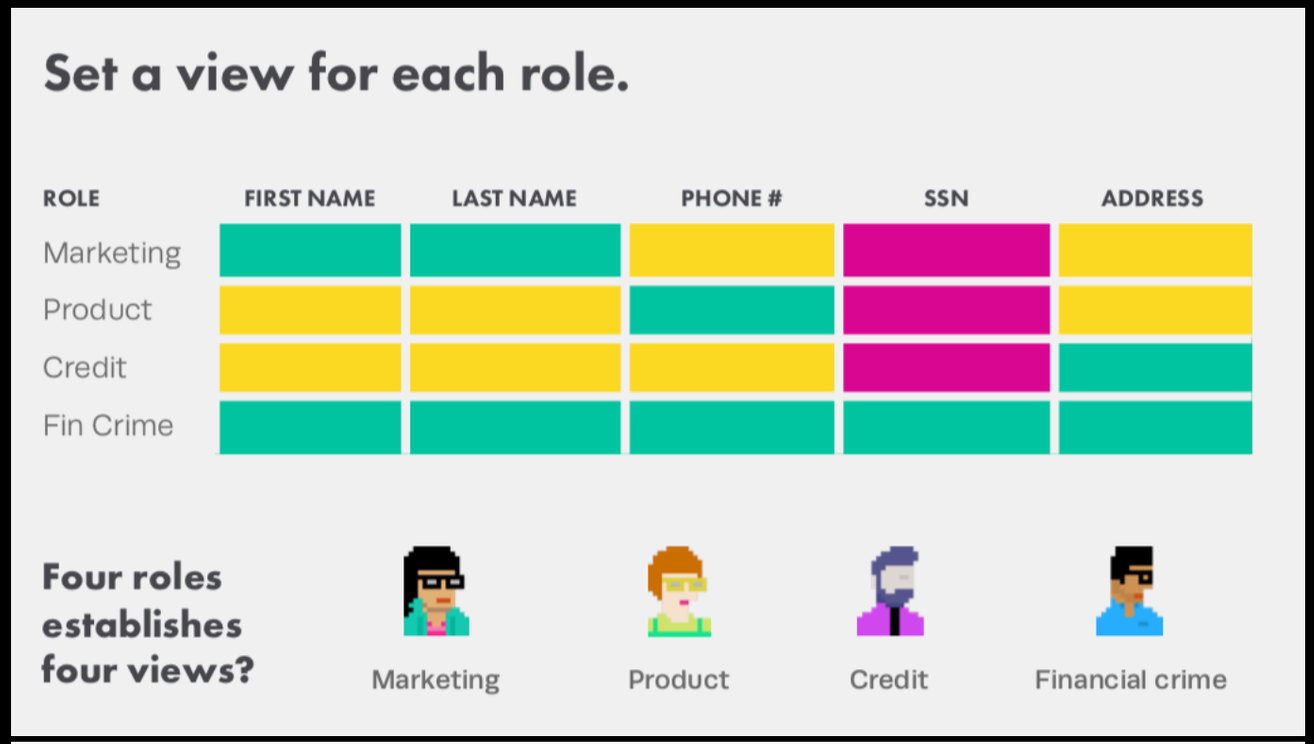
Summary

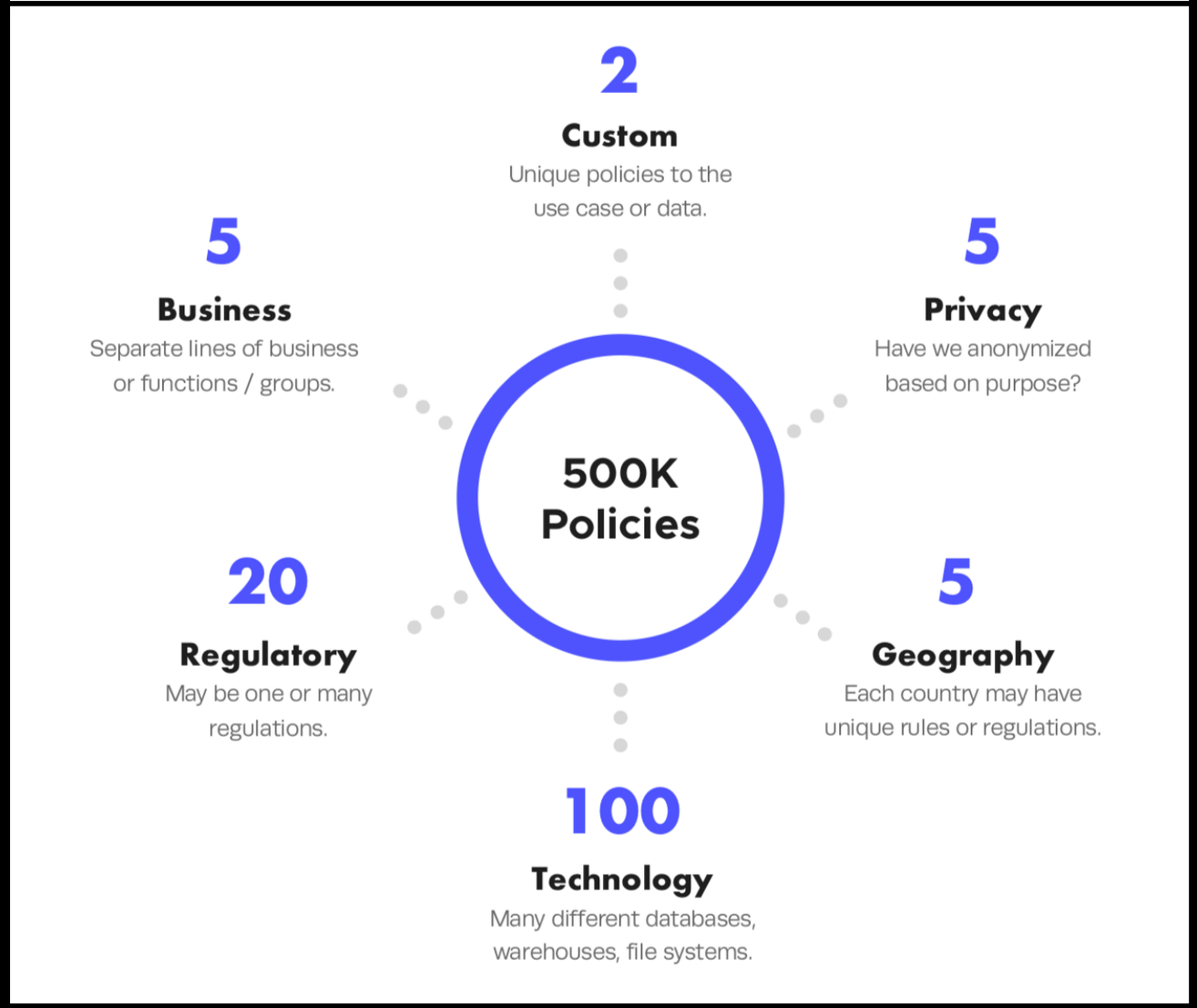
Putting It All Together

- Avoid having multiple copies of data internally
- Use PostgreSQL for managing read-only access centrally
- Use Foreign Data Wrappers to make PostgreSQL a data hub
- Identify private data (Google DLP API, Microsoft Presidio, Amazon Macie)
- CREATE ROLEs (no login permission) based on purpose
- Force (most) users only to be able to act under one role at a time
- Use row level security policies to limit access to rows
- Use column level permissions and/or masking to limit access to columns
- Use differential privacy and k-anonymity when appropriate
- CREATE VIEWs as needed, MATERIALIZED views if forcing the data to be cached
- Use pgAudit to track query activity

View Approach

- Maintaining views may become unmanageable
- Just “Marketing” as a purpose may not be enough
- Views may also need to take into account
  - Geography
  - Business unit
  - Regulations





..Or do it the easy way

aws marketplace

Sign in or Create a new account

Categories | Delivery Methods | Solutions | Migration Mapping Assistant | Your Saved List | Partners | Sell in AWS Marketplace | Amazon Web Services Home | Help

IMMUTA

About Immuta

The award-winning Immuta automated data governance software platform creates trust across security, legal, compliance, and business teams so they can work together to ensure timely access to critical business data with minimal risks. Its self-service, automated, scalable, no code approach makes it easy for users to access the data they need, when they need it, while protecting sensitive data and ensuring data privacy. For more information, visit https://www.immuta.com or email contact@immuta.com.

Immuta

Visit the Immuta Website

Immuta Products (1)

showing 1 - 1

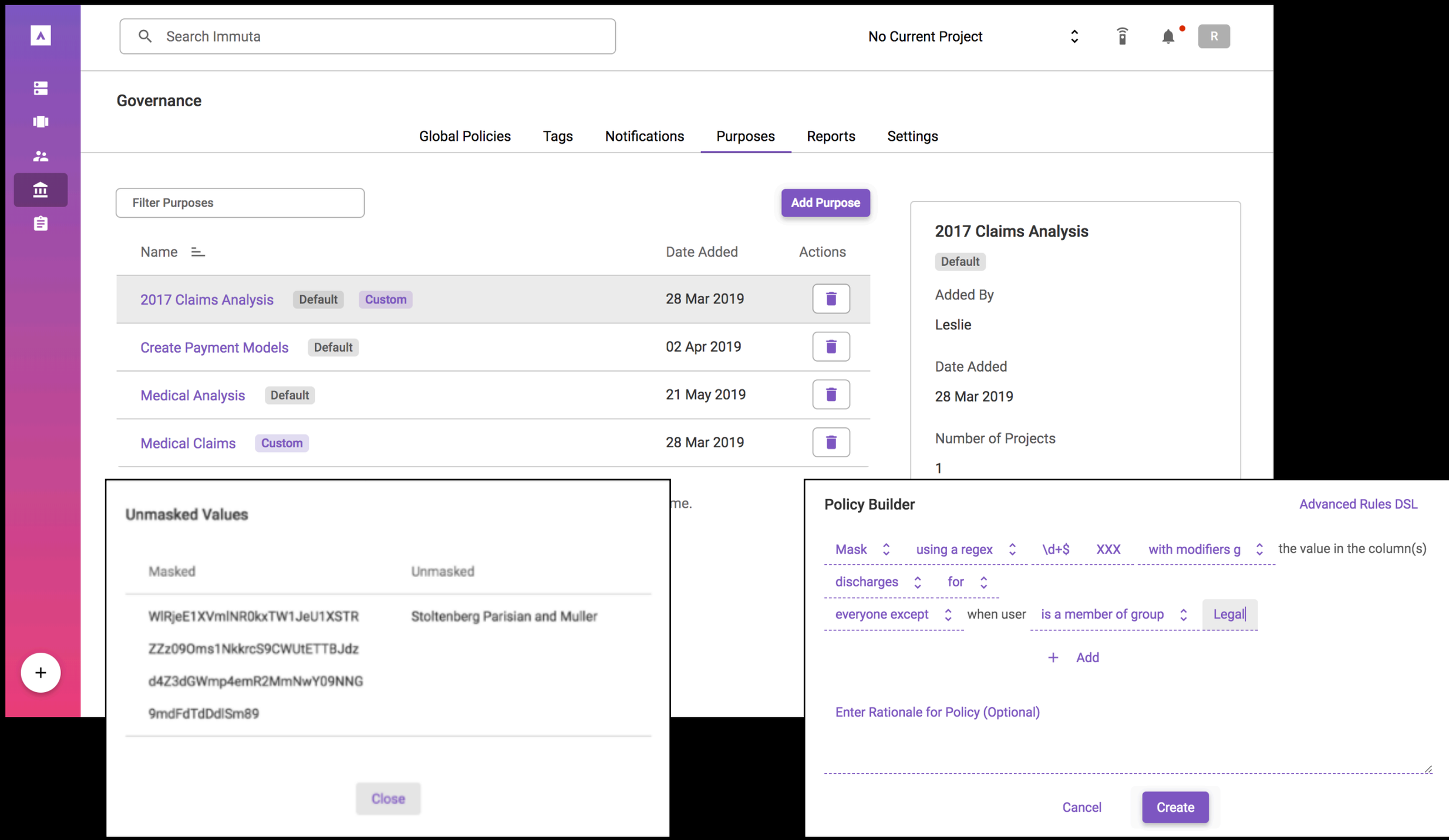
Immuta Professional

★★★★★ (0) | Version 1 | Sold by Immuta

https://immuta.com/ Immuta enables construction and enforcement of complex data policies, allowing you to control and monitor data access, view policies at work across your data environment, and ensure that compliance requirements are met. The Immuta Policy Engine simplifies the creation of data policies...

showing 1 - 1

IMMUTA™



- UI-Driven Data Governance
- Connects to 20+ different RDBMS
- Create Policies, Purposes & Projects
- Masked Joins
- Differential Privacy

- Access Approval Workflow and Alerting
- Query Auditing
- Intelligent Query Pushdown
- Spark Integration

Thank You