“A Little Respect”

Erasure of Personal Data in Distributed Systems

Neville Samuell, VP Eng @ Ethyca

ethyca

BIO

@nsamuell

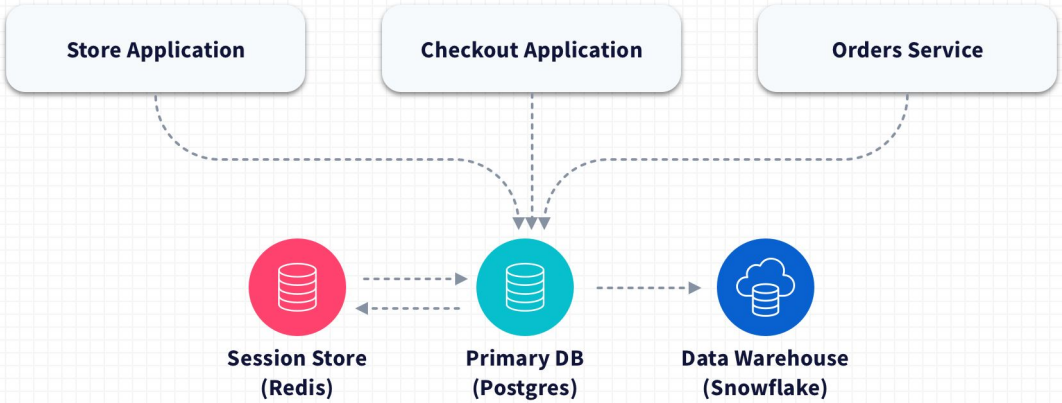
I'm **Neville Samuell**. As VP of Engineering at Ethyca, I lead a distributed team building Ethyca's data privacy automation platform, which helps businesses of all sizes efficiently honor privacy rights for their customers.

At Ethyca, our vision is to be the trust infrastructure of the Internet. We're building software to help engineers create more respectful systems throughout the SDLC, both with tools to better design and annotate data during development, and with our production platform that integrates with deployed applications.

AGENDA

1. Key Concepts
2. Basic Example: Application DB
3. Traversing The Tree
4. Connecting Across Datastores
5. Designing Respectful Systems

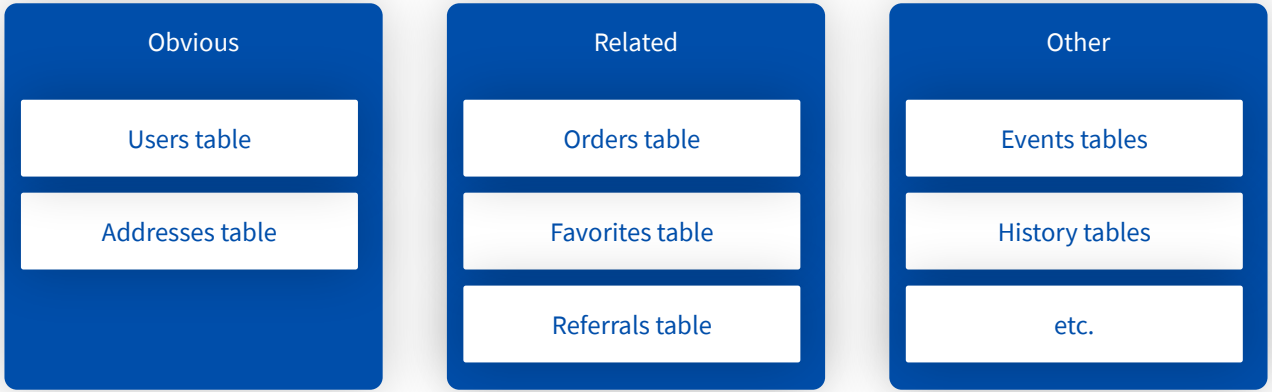
OUR EXAMPLE



- Store and Checkout web applications for users
- Order service for backend management
- Postgres database for primary application data
- Redis session store
- Snowflake data warehouse for analytics

ethyca

WHERE DOES PERSONAL DATA LIVE?



ethyca

KEY CONCEPT: RIGHT TO FORGET

- Fundamental feature of respectful systems and most regulations
- Canonical example: **GDPR Article 17**
- Requires you to give data subjects (ie. users) the right to erase all their personal data from your systems
- GDPR allows 30 days to respond, CCPA is similar but allows 45 days
- Many restrictions & exceptions… but not for this talk!

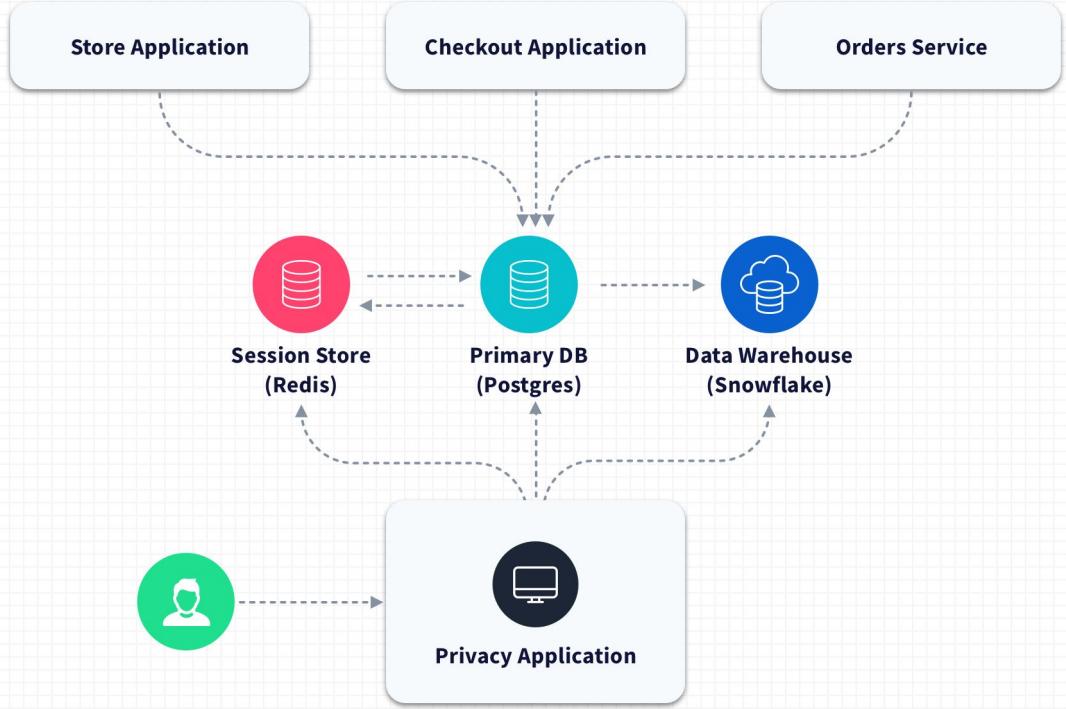
KEY CONCEPT: ERASING ENTIRE ROWS

- On the surface, erasing sounds simple:
  - DELETE FROM users WHERE users.email = ‘subject@example.com’
- Practically, however this rarely works:
  - Personal data tends to be copied and spread around
  - Leaves orphaned records with personal data (addresses, orders…), which breaks referential integrity of database
  - Deletes non-personal data that is stored alongside
  - Can’t natively CASCADE out to other datastores
  - etc.

KEY CONCEPT: ERASING INDIVIDUAL FIELDS

- Erasure requires forgetting all personal data about a subject
- However, erasing entire rows will also remove anonymous data (aggregate historical user counts, product usage data, etc.)
- Instead, can selectively erase individual fields, if you're careful
- This preserves aggregate row counts, maintains referential integrity, etc.
- In other words:
  - Use DELETE statements to erase entire rows where possible
  - Use UPDATE statements to erase personal data fields otherwise

BASIC ALGORITHM



1. Receive request from identified user (e.g. email)
2. Collect matching row IDs
3. For each row, collect all related row IDs
4. Repeat until all rows are identified
5. Erase personal data in all the collected rows

ethyca

EXAMPLE CODE: USERS TABLE

id	email	name
1	test@example.com	Example Name
2	another@example.com	Another Name
3	more@example.com	Repeated Name
4	unique@example.com	Repeated Name

EXAMPLE CODE: ERASE USER BY EMAIL

```python
def erase_user_by_email(email: str = "test@example.com") -> int:
    """
    Erase all personal data for a given email address. Returns number of affected rows
    """
    user_ids: List[int] = magicsql.run("SELECT id FROM users WHERE email = ?", email)
    row_count = 0
    for user_id in user_ids:
        row_count += erase_user_row_by_id(user_id)
    return row_count
```

1. Receive request from identified user (e.g. email)
2. Collect matching row IDs
3. For each row, erase all personal data fields
4. Repeat

EXAMPLE CODE: ERASE USER ROW BY ID

```python
def erase_user_row_by_id(user_id: int, should_delete: bool = False) -> int:
    """
    Erase all user data for a given ID. Returns number of affected rows
    """
    if not should_delete:
        user: User = magicsql.run("SELECT * FROM users WHERE id = ?", user_id)
        hashed: User = compute_hashed_user_values(user)
        row_count = magicsql.run(
            "UPDATE users SET email = ?, name = ? WHERE id = ?",
            hashed.email,
            hashed.name,
            user_id,
        )
        return row_count
    else:
        row_count = magicsql.run("DELETE FROM users WHERE id = ?", user_id)
        return row_count
```

EXAMPLE CODE: COMPUTE HASHED USER VALUES

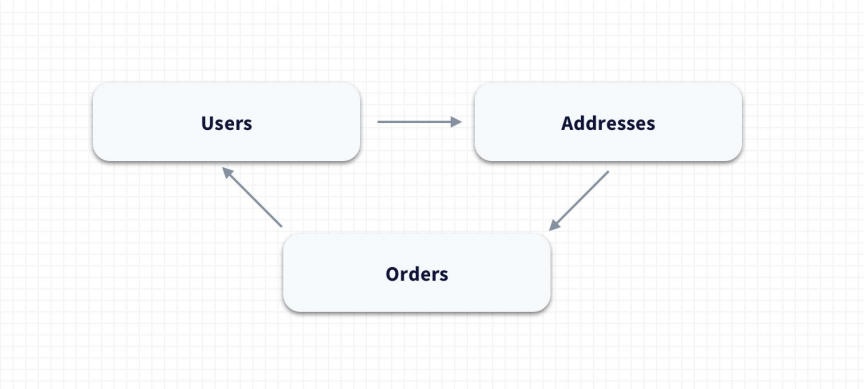
```python
def compute_hashed_user_values(user: User, salt: str = magicsha.salt()) -> User:
    """
    Computes hashed values for all personal data for a User. Returns the hashed copy
    """
    hashed: User = user.copy()
    hashed.email = magicsha.compute_secure_hash(user.email, salt)
    hashed.name = magicsha.compute_secure_hash(user.name, salt)
    return hashed
```

- Remember: secure hashing isn't trivial, use a real library!
- Don't forget a salt! If you're careful, use the same salt for the entire erasure request for consistent hashes across tables
- Notice that erasure strategy varies by field, based on:
  - Data types (strings, numbers, datetimes, etc.)
  - Data categories (emails, locations, etc.)
  - Application constraints (uniqueness, validations, etc.)

EXAMPLE CODE: USERS TABLE (ERASED)

id	email	name
1	4144754cc22727a70dcb...	70e722de552ffcea6688be...
2	another@example.com	Another Name
3	more@example.com	Repeated Name
4	unique@example.com	Repeated Name

TRAVERSING RELATIONSHIPS



- Apply the same basic strategy for all related tables. For example:
  - user “has-many” addresses
  - user “has-many” orders
- For each related table:
  1. Start from the user_id
  2. Collect matching row IDs
  3. For each row, erase personal data
  4. Repeat

EXAMPLE CODE: ADDRESSES TABLE

id	user_id	name	street	city	state	zip
1	1	Home	123 Example St	New York	NY	10011
2	1	Office	456 Imaginary Ln	Dallas	TX	75001
3	4	Home	100 W 100 St	Salt Lake City	UT	84190

- Use the user_ids we collected to find address IDs
- Erase all fields in each related address row
- Note that hashing might not be desirable; use static values instead (NULL, empty strings, etc.)

EXAMPLE CODE: ADDRESSES TABLE

```python
def erase_address_by_user_id(user_id: int) -> int:
    """
    Erase all personal data in address rows for user_id. Returns number of affected rows
    """
    address_ids: List[int] = magicsql.run(
        "SELECT id FROM addresses WHERE user_id = ?", user_id
    )
    row_count = 0
    for address_id in address_ids:
        row_count += erase_address_row_by_id(address_id)
    return row_count
```

- Use the user_ids we collected in the first phase to collect IDs
- Erase all fields in each related address row

EXAMPLE CODE: ADDRESSES TABLE

```python
def erase_address_row_by_id(address_id: int, should_delete: bool = False) -> int:
    if not should_delete:
        row_count = magicsql.run(
            "UPDATE addresses SET name = ?, street = ?, city = ?, state = ?,"
            " zip = ? WHERE id = ?",
            None,  # name
            None,  # street
            None,  # city
            None,  # state
            None,  # zip
            address_id,
        )
        return row_count
    else:
        row_count = magicsql.run("DELETE FROM addresses WHERE id = ?", address_id)
        return row_count
```

EXAMPLE CODE: ADDRESSES TABLE

id	user_id	name	street	city	state	zip
1	1	NULL	NULL	NULL	NULL	NULL
2	1	NULL	NULL	NULL	NULL	NULL
3	4	Home	100 W 100 St	Salt Lake City	UT	84190

- Preserves the fact that User #1 had two addresses
- Be careful that your anonymized data is truly anonymous!

EXAMPLE CODE: ORDERS TABLE

id	user_id	address_id	order_email	state	amount
1	1	1	test@example.com	NY	100.00
2	1	2	test+TX@example.com	TX	500.00
3	4	3	accounts@example.com	UT	150.00
4	4	3	unique@example.com	UT	50.00

- Example where deleting entire row may be impossible!
- Must preserve the state and amount for tax purposes
- However, can still erase email address
- And, potentially, unlink the user & address IDs

EXAMPLE CODE: ORDERS TABLE

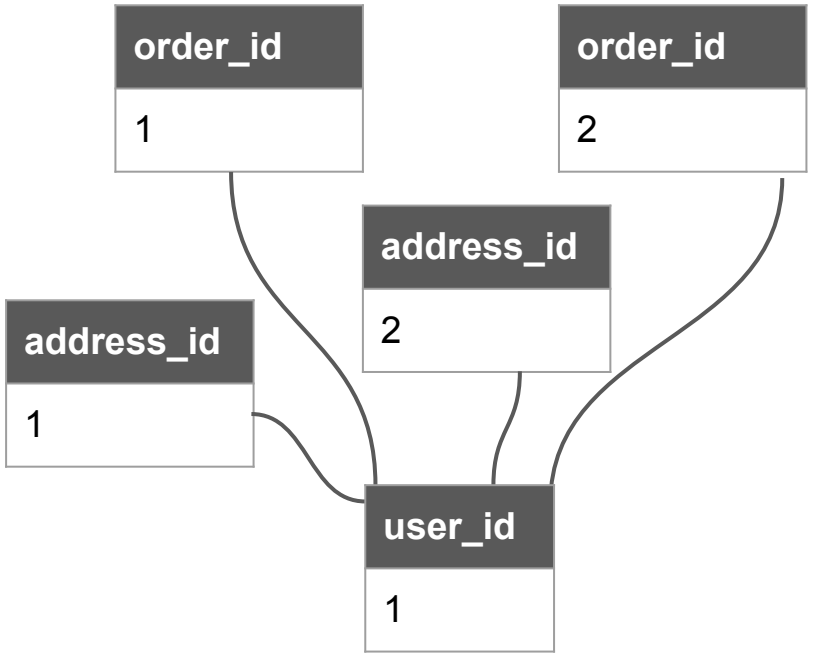
```python
def compute_erased_order_values(order: Order) -> Order:
    """
    Computes erased values for all personal data for a order. Returns the erased copy
    """
    erased: Order = order.copy()
    erased.order_email = None
    # erased.state must be preserved for tax purposes
    # erased.amount must be preserved for tax purposes

    # NOTE: this "unlinking" may not be possible for your application!
    erased.user_id = None
    erased.address_id = None
    return erased
```

EXAMPLE CODE: ORDERS TABLE

id	user_id	address_id	order_email	state	amount
1	NULL	NULL	NULL	NY	100.00
2	NULL	NULL	NULL	TX	500.00
3	4	3	accounts@example.com	UT	150.00
4	4	3	unique@example.com	UT	50.00

TRAVERSAL ORDER



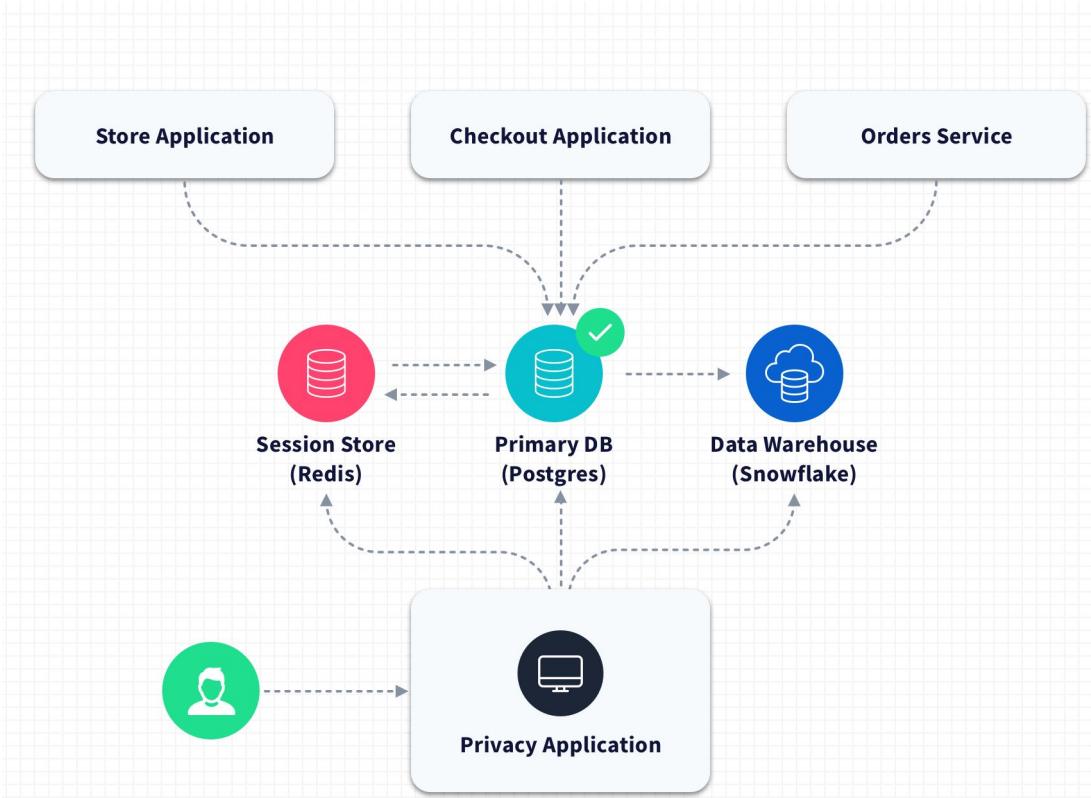
- Order of these erasures matters!
- Think of it like a tree, and start at the “leaf nodes” and work down towards the root:
  - Orders → Addresses → Users
- Allows the erasure to be interruptible; integrity is maintained throughout

EXAMPLE CODE: FULL ERASURE

```python
def erase_user_by_email(email: str = "test@example.com") -> int:
    """
    Erase all personal data for a given email address. Returns number of affected rows
    """
    user_ids: List[int] = magicsql.run("SELECT id FROM users WHERE email = ?", email)
    row_count = 0
    for user_id in user_ids:
        # Collect all related address & order ids
        address_ids: List[int] = get_address_ids_for_user(user_id)
        order_ids: List[int] = get_order_ids_for_user(user_id)

        # Erase in reverse order
        for order_ids in order_ids:
            row_count += erase_order_row_by_id(order_id)
        for address_ids in address_ids:
            row_count += erase_address_row_by_id(address_id)
        row_count += erase_user_row_by_id(user_id)
    return row_count
```

CHECKPOINT



- Collected IDs for users, addresses, orders, etc.
- Erased all the primary application data
- Now, use the collected IDs to move downstream:
  - Data Warehouse
  - Session Store

ethyca

DATA WAREHOUSES

- Overall, use the same algorithm as a OLTP database
- However, some key differences:
  - Tend to have chains of related tables (via DAGs)
  - Often include denormalized tables for analytics (lots of copies)
  - Rarely indexed by user_id, instead partitioned by date, etc.
- This means they are slow to erase
- Therefore, the best way to erase personal data from a warehouse is to avoid storing it in the first place

DENORMALIZED EVENTS TABLE

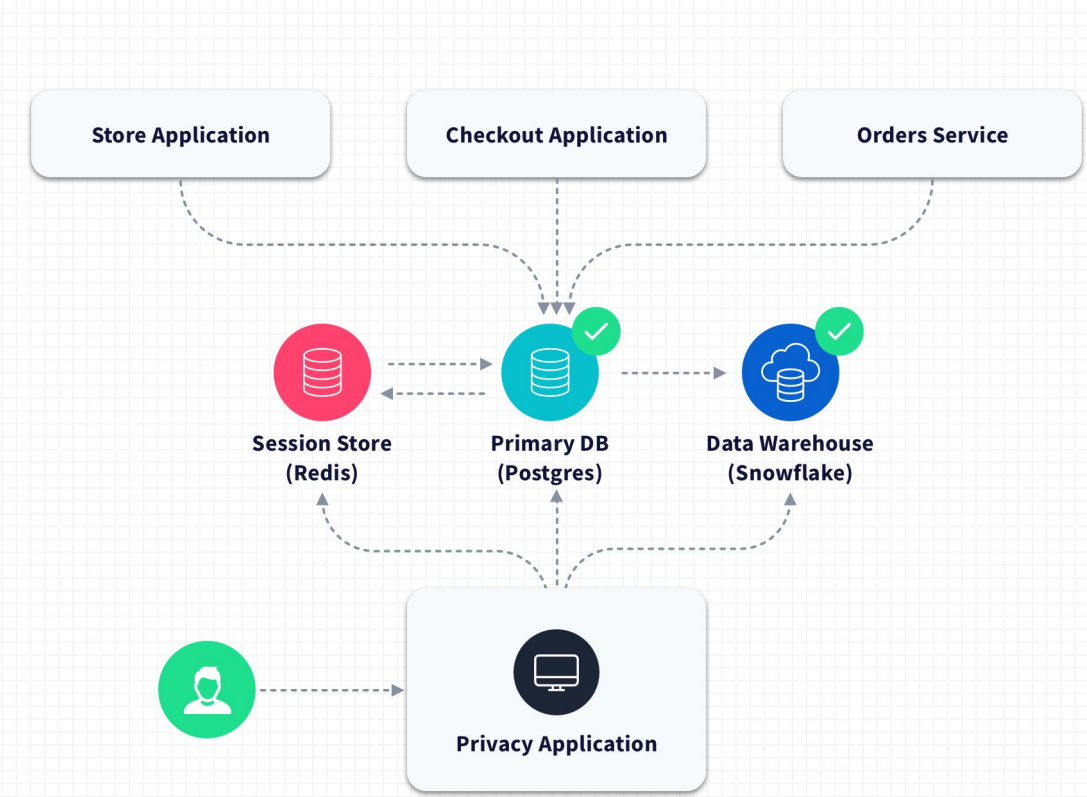
id	user_email	user_name	event	timestamp
100	test@example.com	Example Name	logged_in	2021-01-01 12:00
101	another@example.com	Another Name	searched	2021-01-01 12:01
102	test@example.com	Example Name	searched	2021-01-01 12:02
103	test@example.com	Example Name	logged_out	2021-01-01 12:05

- May requires visiting and erasing 1000s of rows, often without the ability to index...

BULK UPDATES

- “Row-wise” queries tend to be slow, depending on your warehouse
- Two potential improvements:
  1. Group updates together for a single user that have the same values (i.e. not hashes) and bulk update
  2. Group updates together for multiple users that leverages your partitions (e.g. dates)

CHECKPOINT



- Erased all the event data in our warehouse
- Lastly, need to erase our session store

ethyca

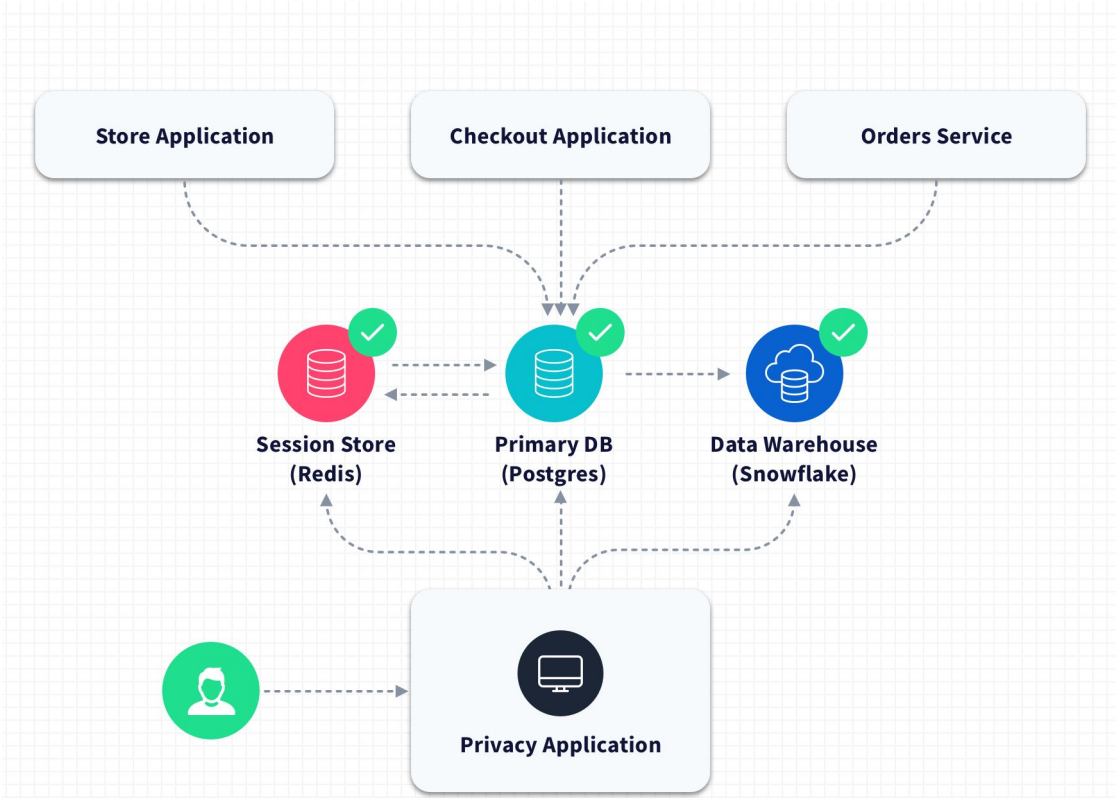
NOSQL ERASURE

- At this point, you understand the essential algorithm:
  - Use the user ID to query documents based on your schema
  - Collect the document IDs and run erasure on each
- NoSQL data stores need to be traversed and updated differently, but the same general principle (leaf nodes first) applies

CACHE EXPIRATION

- However: what about a potential alternative?
- Assume this Redis store is simply a cache with expiry dates on all keys...
- If the expiration is less than 30 days, you can reliably let this “self-erase”
- Why stop there? Design the system to expire every 60 minutes!
- Some questions to ask when designing respectful systems:
  - How much of the user data is really needed permanently?
  - Can you design your systems to proactively erase data once used?

CHECKPOINT



- Let our cache expire...
- Erasure request complete!

ethyca

REALITY CHECK

- Reminder: this example was not realistic at all!
- Many applications will already have 100s or 1000s of datasets that contain personal data, each of which needs to be customized
- Furthermore, personal data stored in unstructured blob storage is even harder to query and erase

RESPECTFUL SYSTEMS

- Why is something so simple in theory, so laborious in practice?
- Systems are not designed respectfully, they are designed for convenience
- It's convenient to:
  - Capture more personal data than you have a use for
  - Copy personal data to many locations
- It's respectful to:
  - Capture minimal amounts of personal data
  - Avoid creating copies to minimize possibilities of breach
  - Maintain metadata about what categories of data are collected, why they are needed, where they are stored...

SUMMARY

- Map out your systems, where data is stored, and how they are connected
- Query the primary store based on the user identity and build a relationship tree between tables and dependent systems
- Traverse (in reverse order) and erasing personal data from each
- Preserve anonymous data along the way, especially financial, tax, etc.
- For analytics warehouses, consider bulk updates for performance
- If you avoid storing personal data in the first place, or automatically erase it (like a cache!) you can achieve this goal proactively
- Design respectful systems!

Thanks!

Questions? Follow @nsamuell on Twitter, or neville@ethyca.com

ethyca