SIMPLE

Data Engineering Architecture at Simple

SIMPLE

```
$ whoami
```

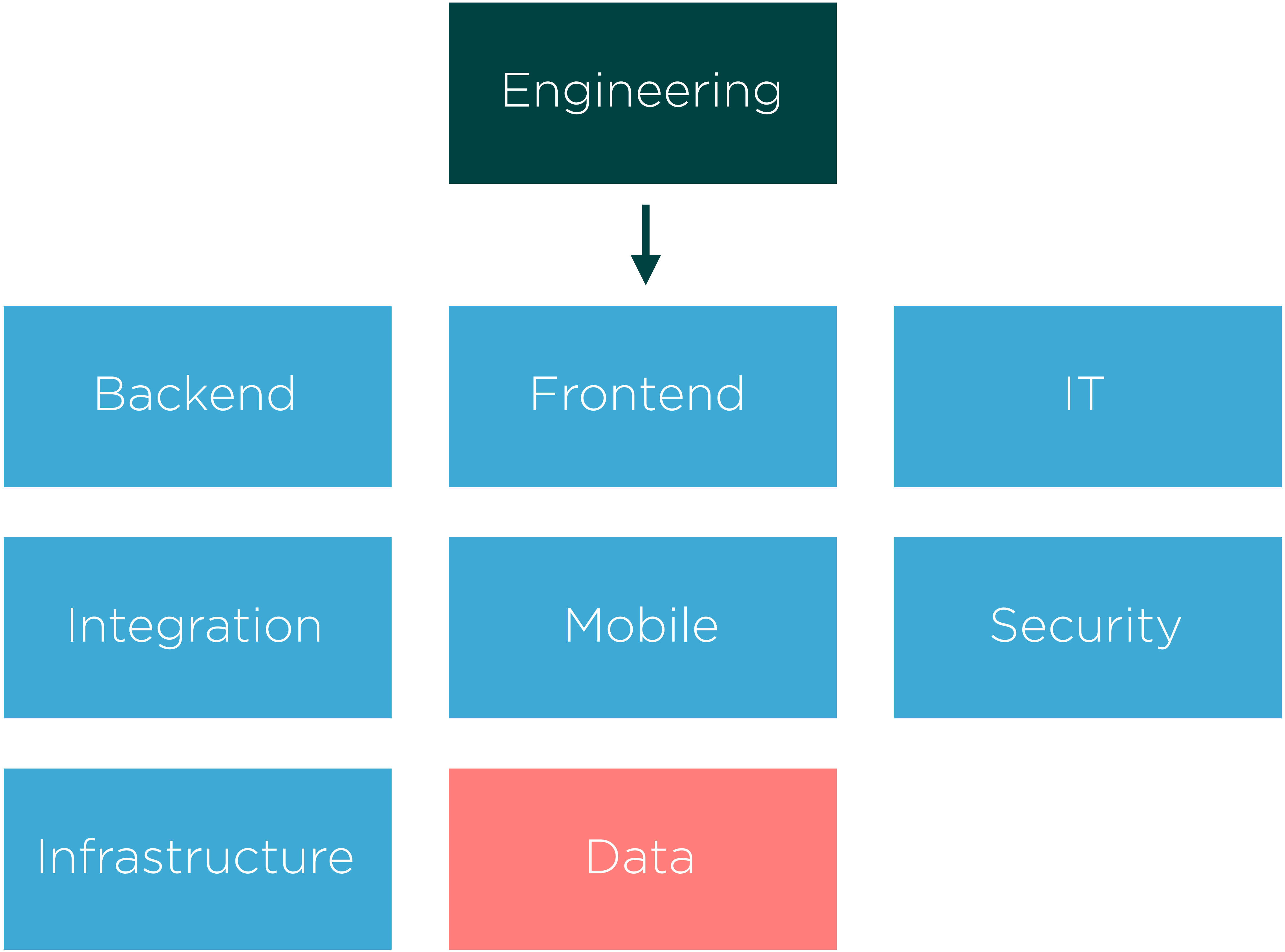
Rob Story

Senior Data Engineer

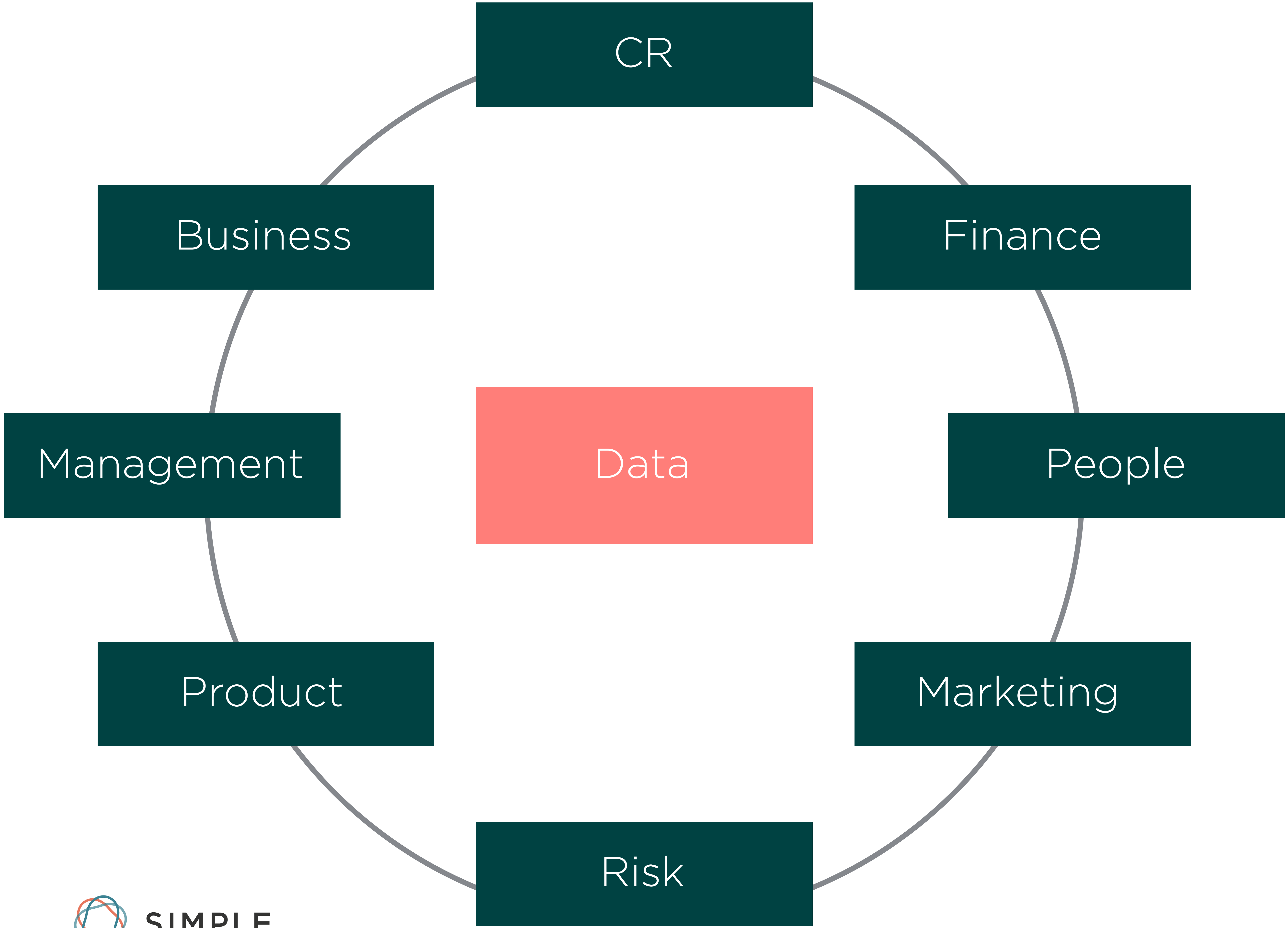
@oceankidbilly

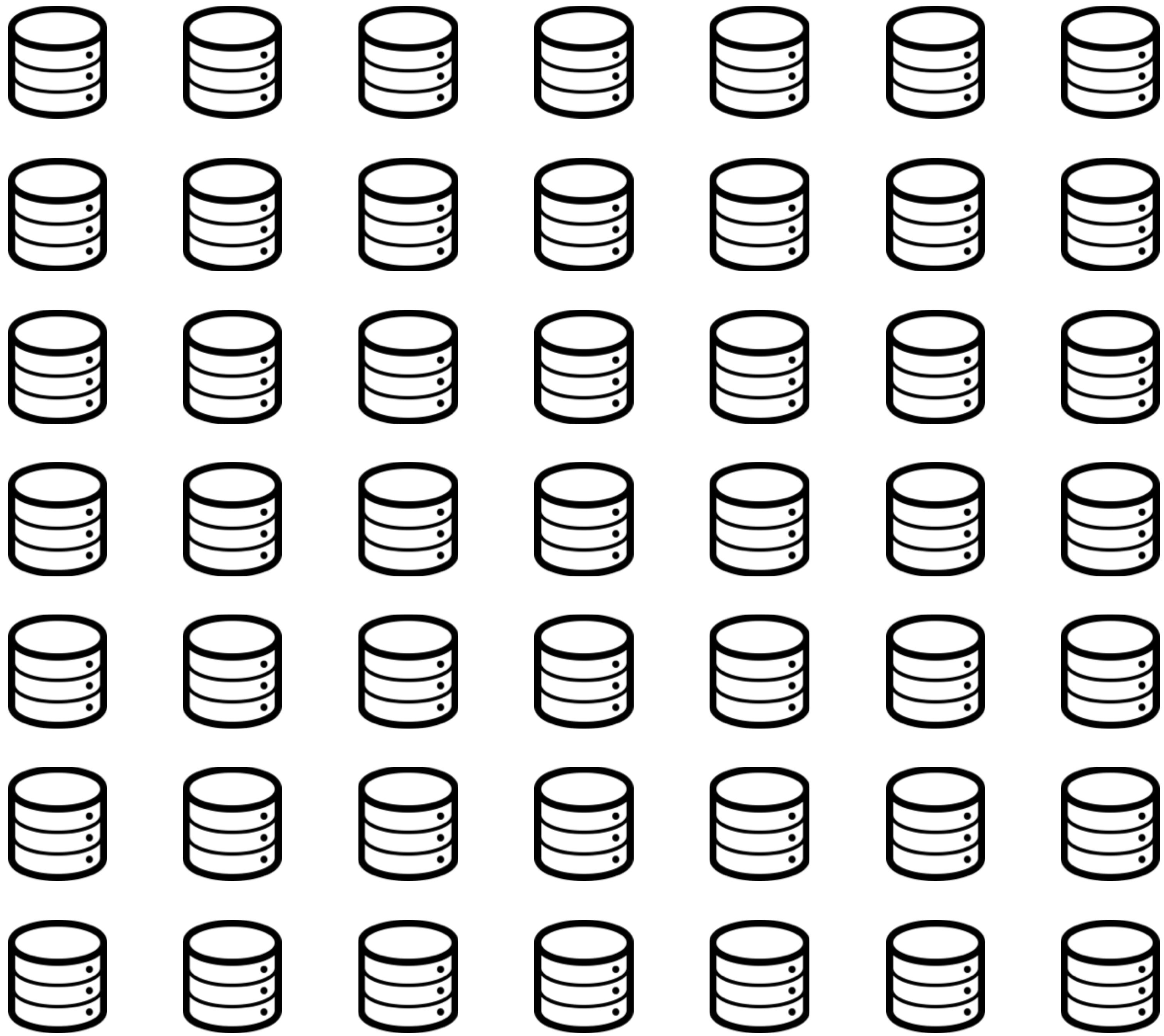
SIMPLE

Why do we have a Data Team?


Engineering
Backend
Frontend
IT
Integration
Mobile
Security
Infrastructure
Data

SIMPLE


CR
Business
Finance
Management
Data
People
Product
Marketing
Risk
SIMPLE



SIMPLE

Find all users who have made a transaction > $1 in the past week.

SIMPLE

No Data Warehouse

```
psql -d SCARY_PRODUCTION_DATABASE                    \
     -c "COPY (SELECT user_id,                       \
                      email,                         \
                      first_name,                    \
                      last_name                      \
               FROM user_data)                       \
         TO user_data.csv DELIMITER ',' CSV;"        \

psql -d VERY_VERY_SCARY_PRODUCTION_DATABASE                                  \
     -c "COPY (SELECT user_id                                                \
               FROM transactions                                             \
               WHERE (txn.amount / 10000) > 1                                \
               AND when_recorded > CURRENT_DATE - INTERVAL '7 days';)        \
         TO txn_data.csv DELIMITER ',' CSV;"                                 \
```

Then use R or Python to read_csv and perform the join

SIMPLE

Data Warehouse: Redshift

```sql
SELECT email, first_name, last_name
FROM transactions txn
JOIN user_data u using (user_id)
WHERE (txn.amount / 10000) > 1
AND when_recorded > CURRENT_DATE - INTERVAL '7 days';
```

SIMPLE

Data Warehouse

35+ DBs

↓

1 DB

Analysts

Engineers

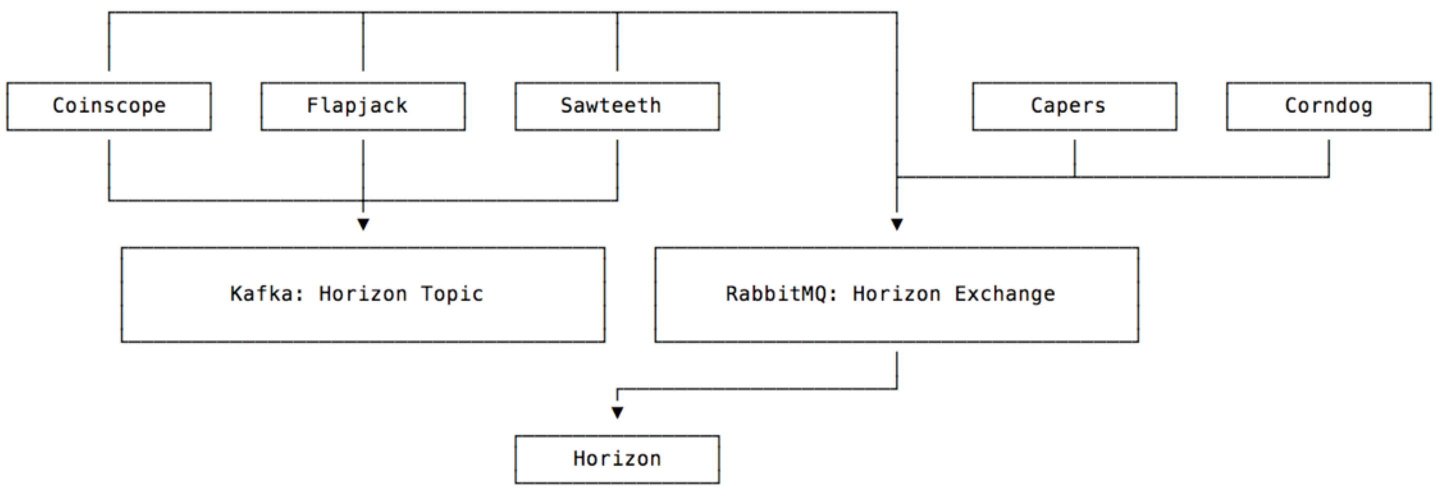
Dashboards

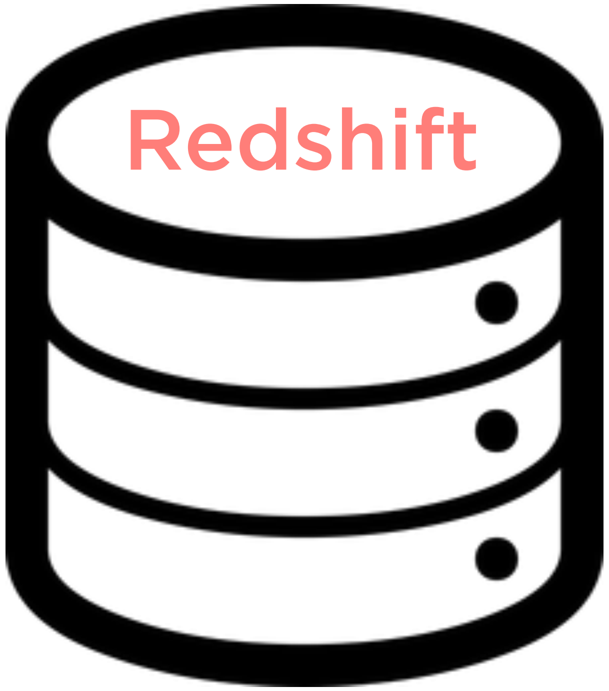
SIMPLE

Data Engineering

What do we do?

Build the tools and systems to make data available to all who need it.





SIMPLE

Redshift

Why Redshift?

SQL!

Fast Loads from S3

Fast Query Times*

Security

Simple runs on AWS

*what's the catch?

SIMPLE

"An Amazon Redshift data warehouse is an **enterprise-class relational database query and management system***...*

Amazon Redshift achieves efficient storage and optimum query performance through a combination of **massively parallel processing***,* **columnar data storage***, and* **very efficient, targeted data compression encoding schemes***. "*

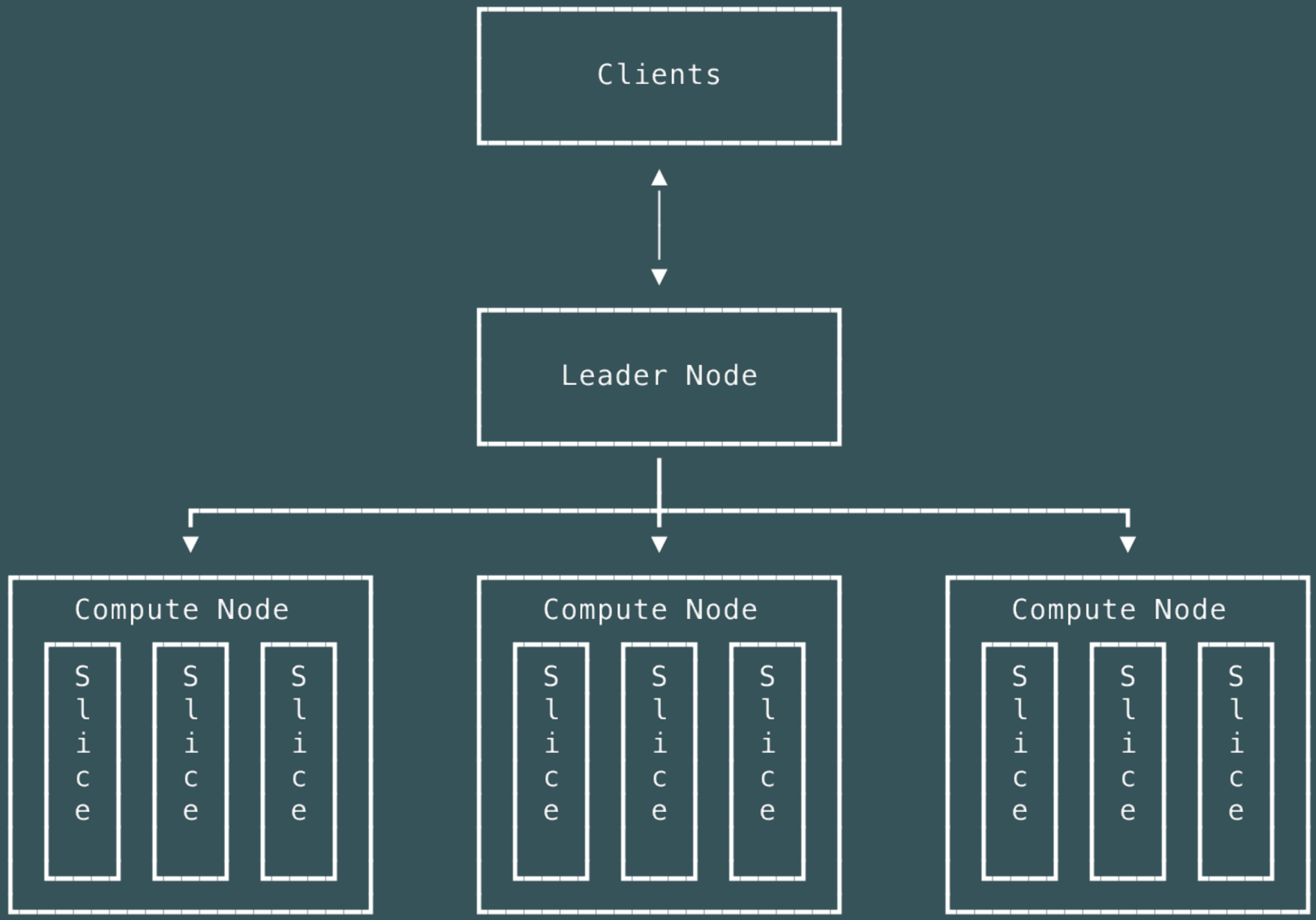
- Amazon Redshift Documentation Writer

SIMPLE

"...enterprise-class relational database query and management system..."

- Quacks like a SQL database
- Forked from Postgres 8.2
- JDBC + ODBC connectors all work
- Query language looks a *little* different from Postgres, but it's pretty close

"...massively parallel processing..."



SIMPLE

DISTKEY and SORTKEY

```
CREATE TABLE user_data (
    user_id char(36) encode lzo,
    email varchar(2048) encode lzo,
    first_name varchar(2048) encode lzo,
    last_name varchar(2048) encode lzo
)
DISTKEY (user_id)
SORTKEY (when_recorded);
```

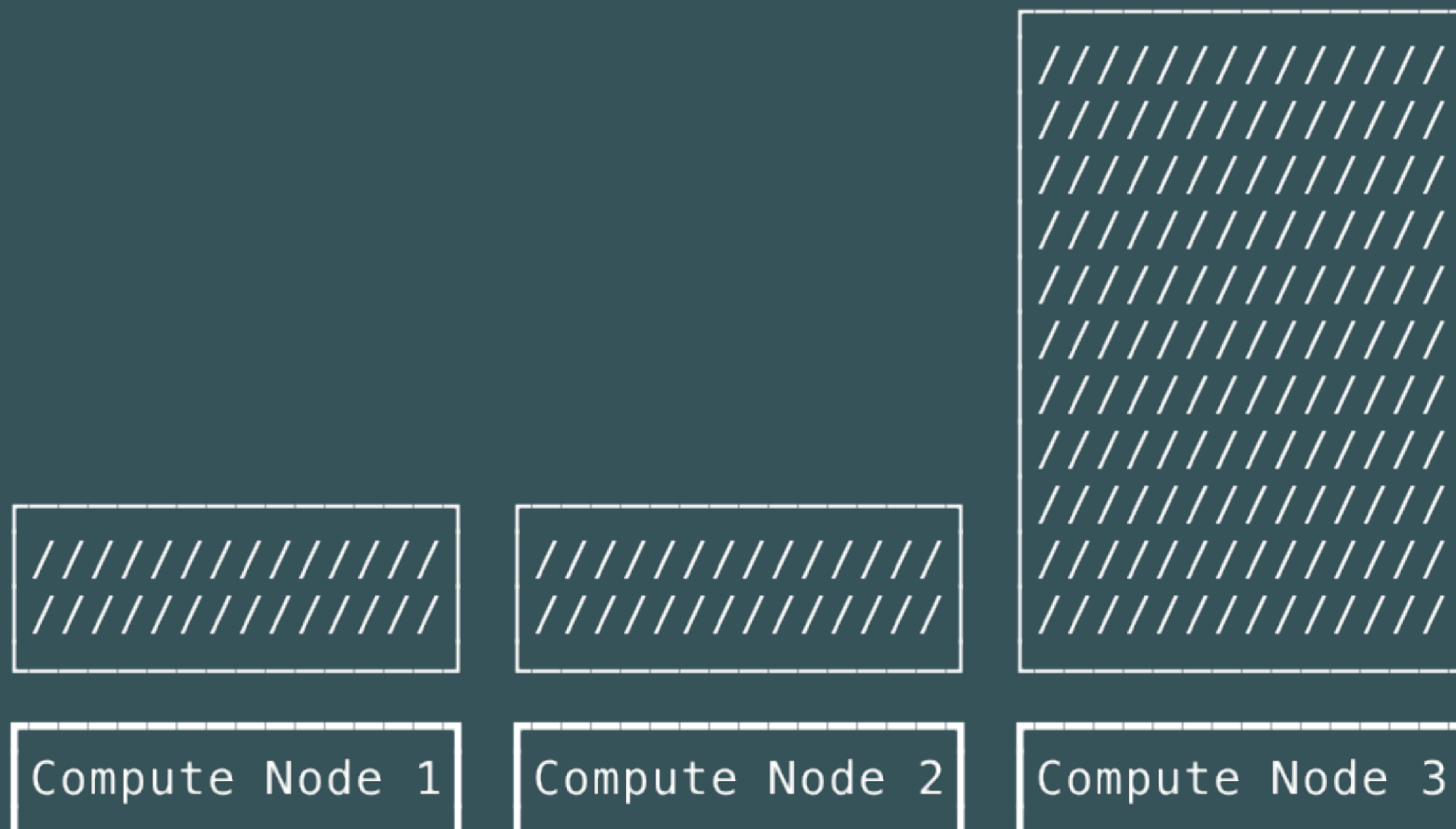
Redshift emphasizes the use of `DISTKEY` and `SORTKEY` to specify data distribution and sortedness during table creation.

If these statements are omitted or specify the incorrect columns, query performance will decay as the tables grow.

SIMPLE

Distkey Example:

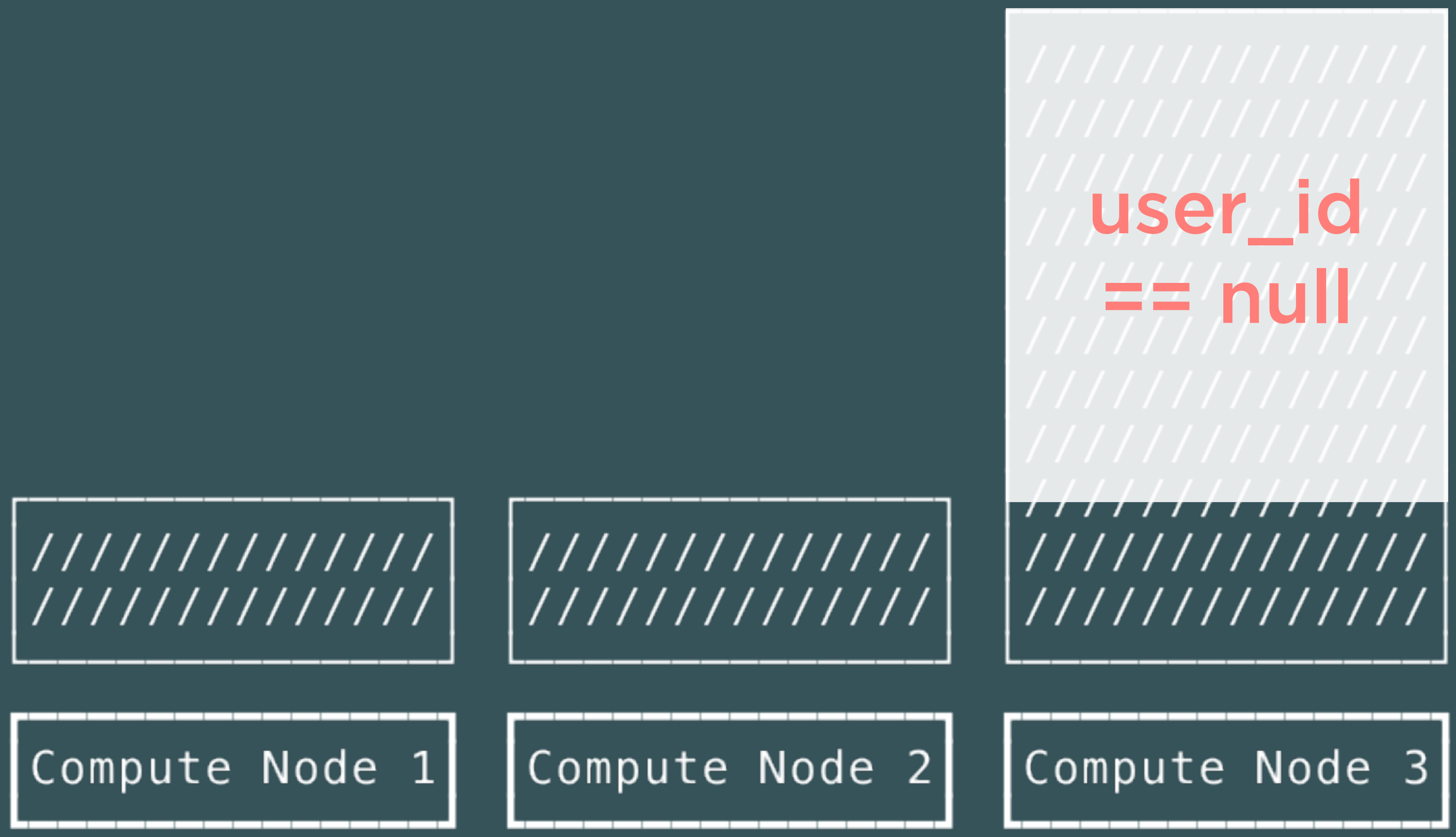
`hash(user_id) % N_Worker_Nodes`



SIMPLE

Distkey Example:

`hash(user_id) % N_Worker_Nodes`



SIMPLE

Solution

```
CREATE TABLE user_data (
    user_id char(36) encode lzo,
    email varchar(2048) encode lzo,
    first_name varchar(2048) encode lzo,
    last_name varchar(2048) encode lzo
)
DISTKEY EVEN
SORTKEY (when_recorded);
```

Unfortunately, now joins on *user_id* require more network shuffles, and range queries touch all nodes!

Sortkey Example

```
EXPLAIN
SELECT t.user_id,
       t.transaction_id,
       g.id AS goal_id
FROM transactions t
JOIN goals g ON (g.userid = t.user_id);
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 XN Hash Join DS_DIST_INNER  (cost=251434.43..638084091578.82)
   Inner Dist Key: g.userid
   Hash Cond: (("outer".user_id)::character(36) = "inner".userid)
   ->  XN Seq Scan on transactions t  (cost=0.00..1385828.00)
   ->  XN Hash  (cost=201147.54..201147.54)
         ->  XN Seq Scan on goals g  (cost=0.00..201147.54)
```

SIMPLE

A hash table can spill to disk.

A hash table can fill all available disk space.

Redshift query timeouts can prevent runaway queries from claiming all available resources.

Sortkey Example

```
CREATE TEMP TABLE temp_transaction_ids
DISTKEY (user_id)
SORTKEY (user_id)
AS (SELECT user_id::char(36) AS user_id,
           transaction_id
    FROM transactions);

CREATE TEMP TABLE temp_goals
DISTKEY (user_id)
SORTKEY (user_id)
AS (SELECT userid AS user_id,
           id AS goal_id
    FROM goals);
```

SIMPLE

Sortkey Example

```
EXPLAIN
SELECT tt.user_id,
       tt.transaction_id,
       tg.goal_id
FROM temp_transaction_ids tt
JOIN temp_goals tg USING (user_id);
```

```
                    QUERY PLAN
---------------------------------------------------------------------
 XN Merge Join DS_DIST_NONE  (cost=0.00..459002950.30)
   Merge Cond: ("outer".user_id = "inner".user_id)
   ->  XN Seq Scan on temp_transaction_ids tt  (cost=0.00..1409302.24)
   ->  XN Seq Scan on temp_goals tg  (cost=0.00..204838.78)
```

SIMPLE

"…columnar data storage…"

A good practice for querying columnar databases:

Try to avoid

```
SELECT * FROM
```

Specify as few individual columns as possible:

```
SELECT A, B FROM
```

```
SELECT A, B
FROM table;
```

Postgres

A	B	C	D
1	Foo	2016-01-01	True
17	Bar	2016-01-02	False
9	Baz	2016-01-03	True

Redshift

A	B	C	D
1	Foo	2016-01-01	True
17	Bar	2016-01-02	False
9	Baz	2016-01-03	True

"…very efficient, targeted data compression encoding schemes…"

- **Redshift will recommend a compression to use with** `ANALYZE COMPRESSION`*
- **Compress, Compress, Compress: it improves both query performance and storage overhead**
- **Lightning Talk on Columnar Compression: https://github.com/wrobstory/ds4ds_2015**

*acquires table lock

SIMPLE

One last thing:
Redshift supports fully serializable isolation.

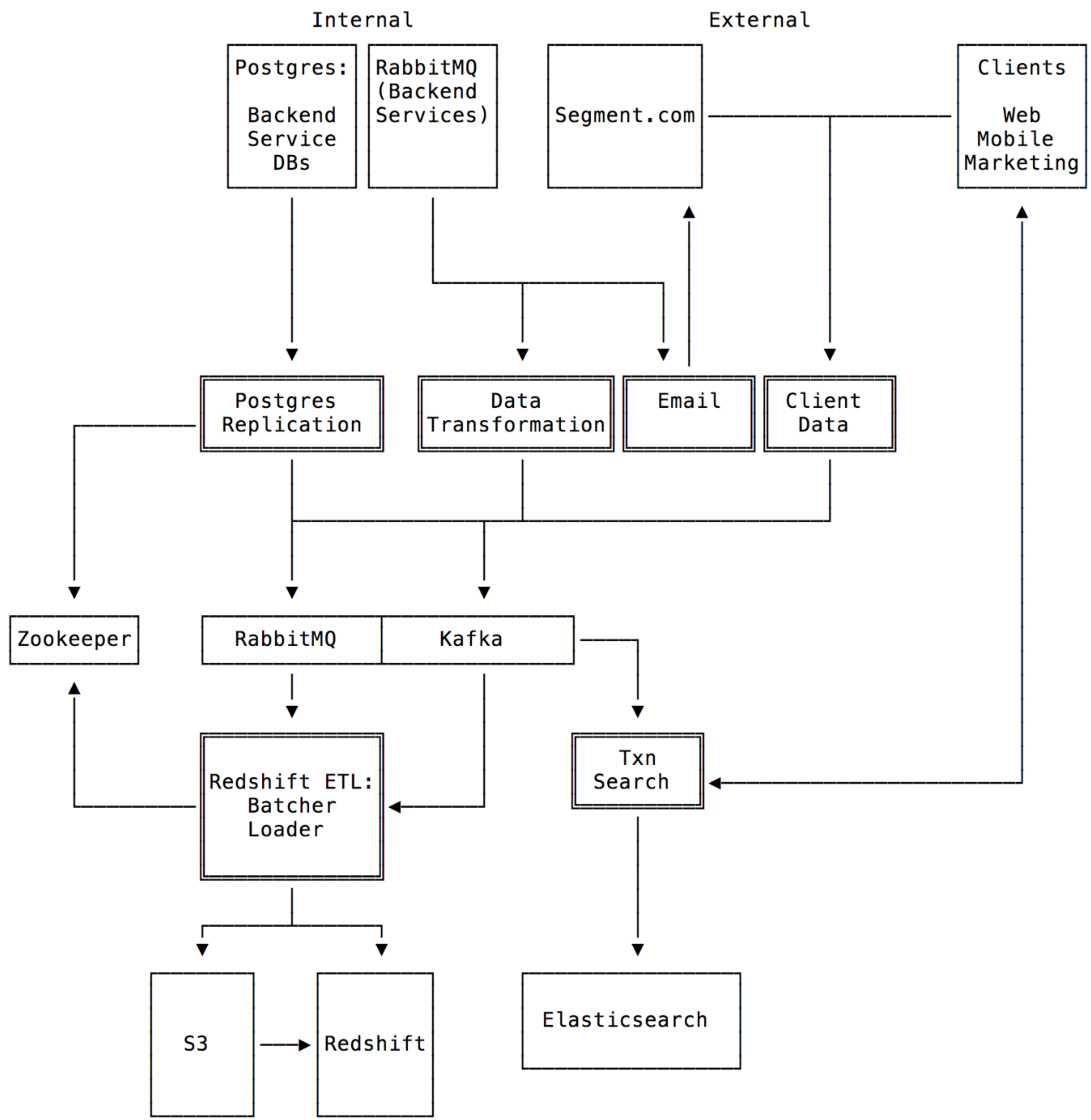
What does that mean operationally?
Transactions are expensive. The commit queue will back up.

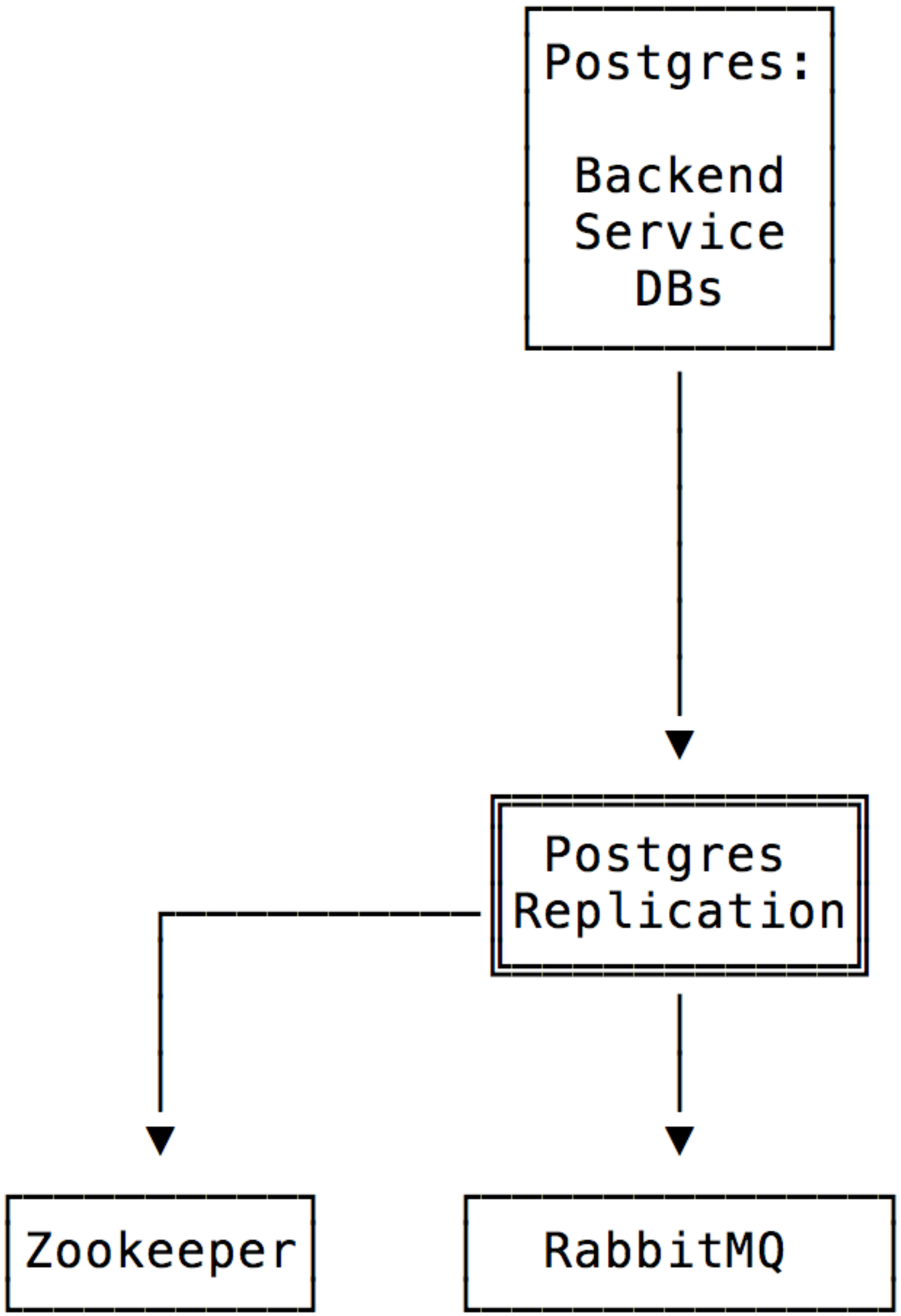
SIMPLE

Queries look like Postgres.
Redshift forked from Postgres.
But it’s not Postgres.

Keeping its distributed nature in mind is important for performance when building tables and queries.

Pipelines

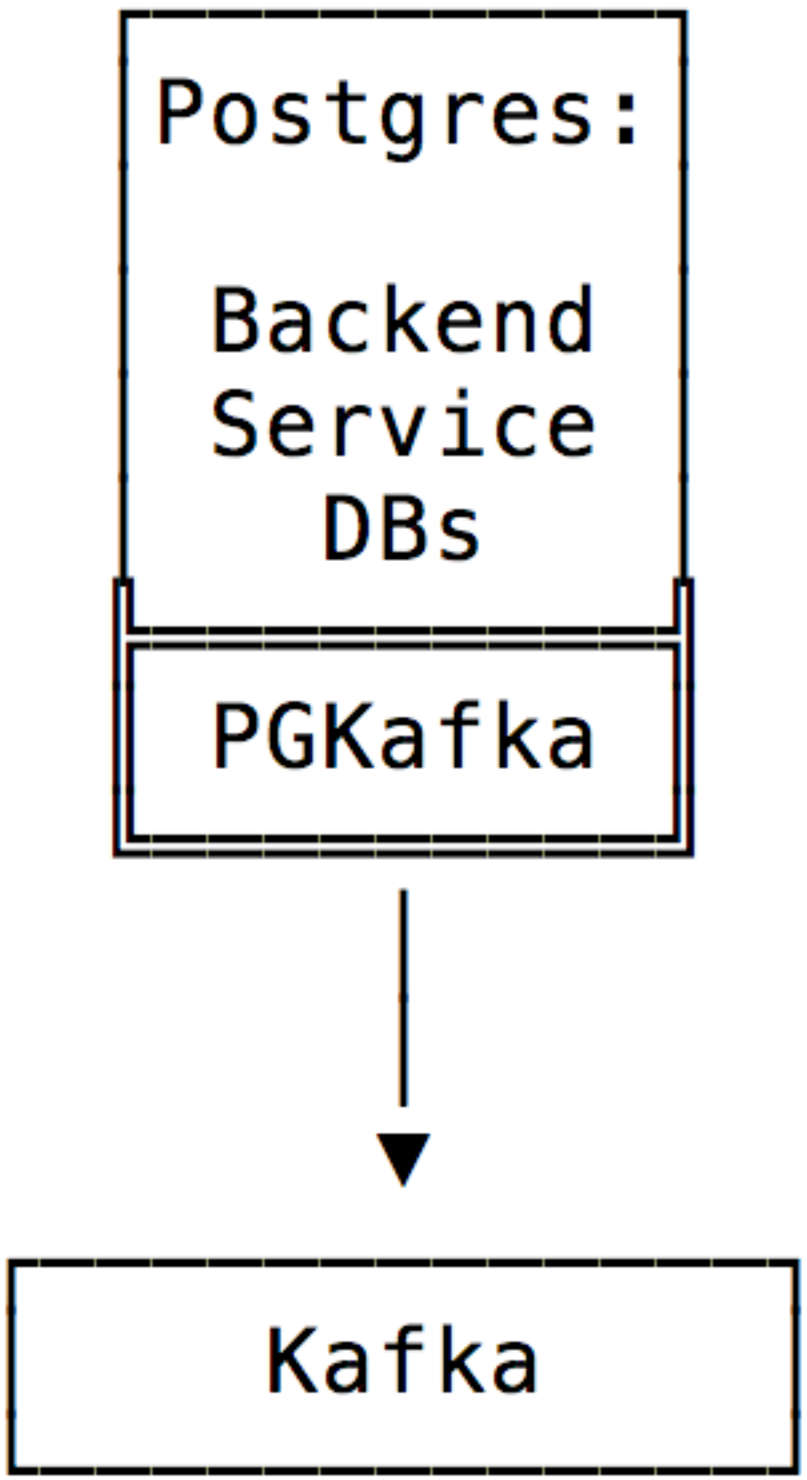

Internal
External
Postgres:
Backend
Service
DBs
RabbitMQ
(Backend
Services)
Segment.com
Clients
Web
Mobile
Marketing
Postgres
Replication
Data
Transformation
Email
Client
Data
Zookeeper
RabbitMQ
Kafka
Redshift ETL:
Batcher
Loader
Txn
Search
S3
Redshift
Elasticsearch



"Sawtooth": Postgres Replication

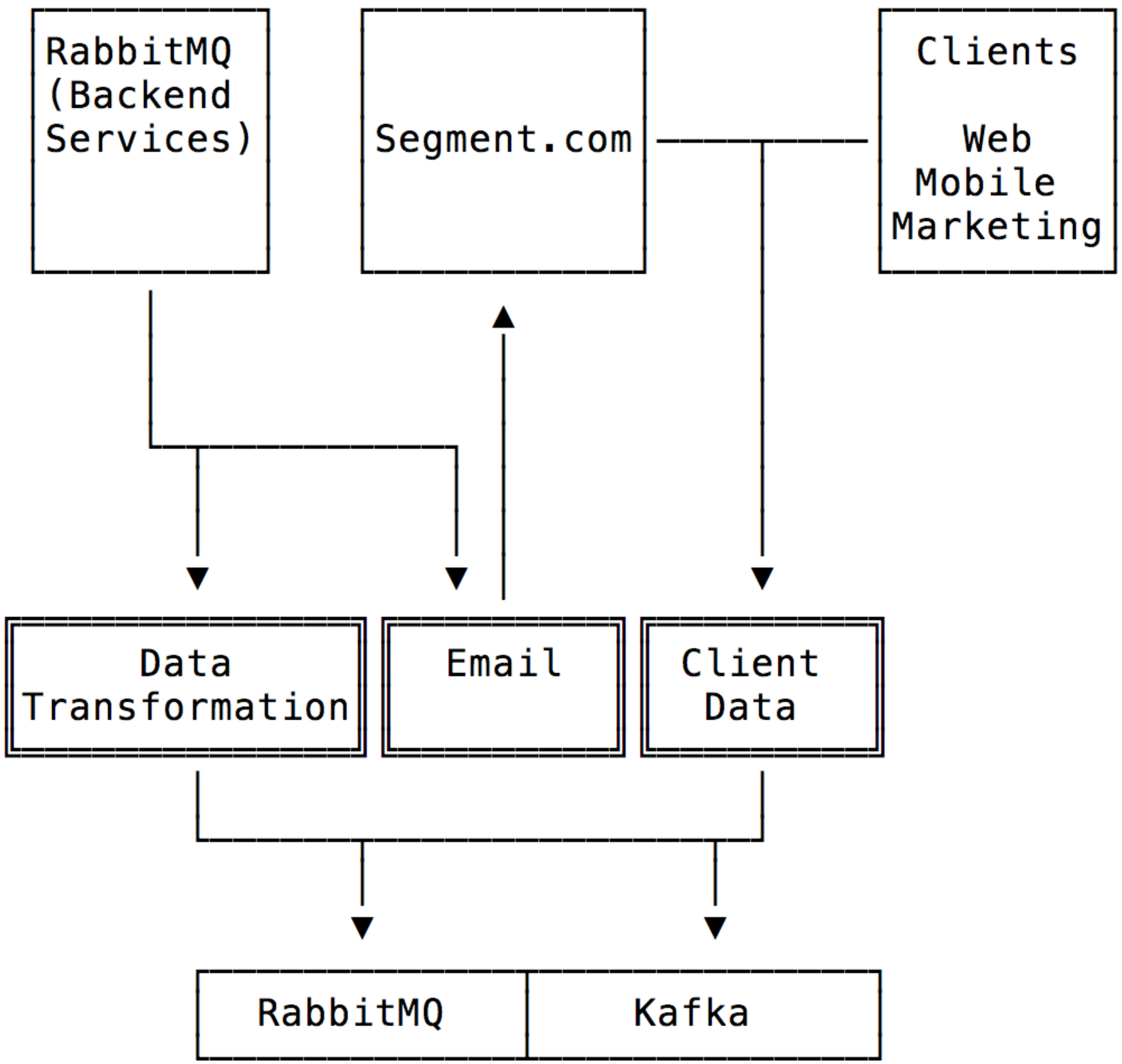
- Crawls Postgres on a schedule
- Uses Zookeeper to store timestamps for last crawl
- Writes to RabbitMQ
- Currently the source of truth for most of our tables

SIMPLE



PGKafka

- Postgres Logical Replication Slot
- Streams *every* change in the DB to Kafka. More or less a real-time stream of the WAL.



"Coinscope"

Data Transformation Service....Kafka Streams?

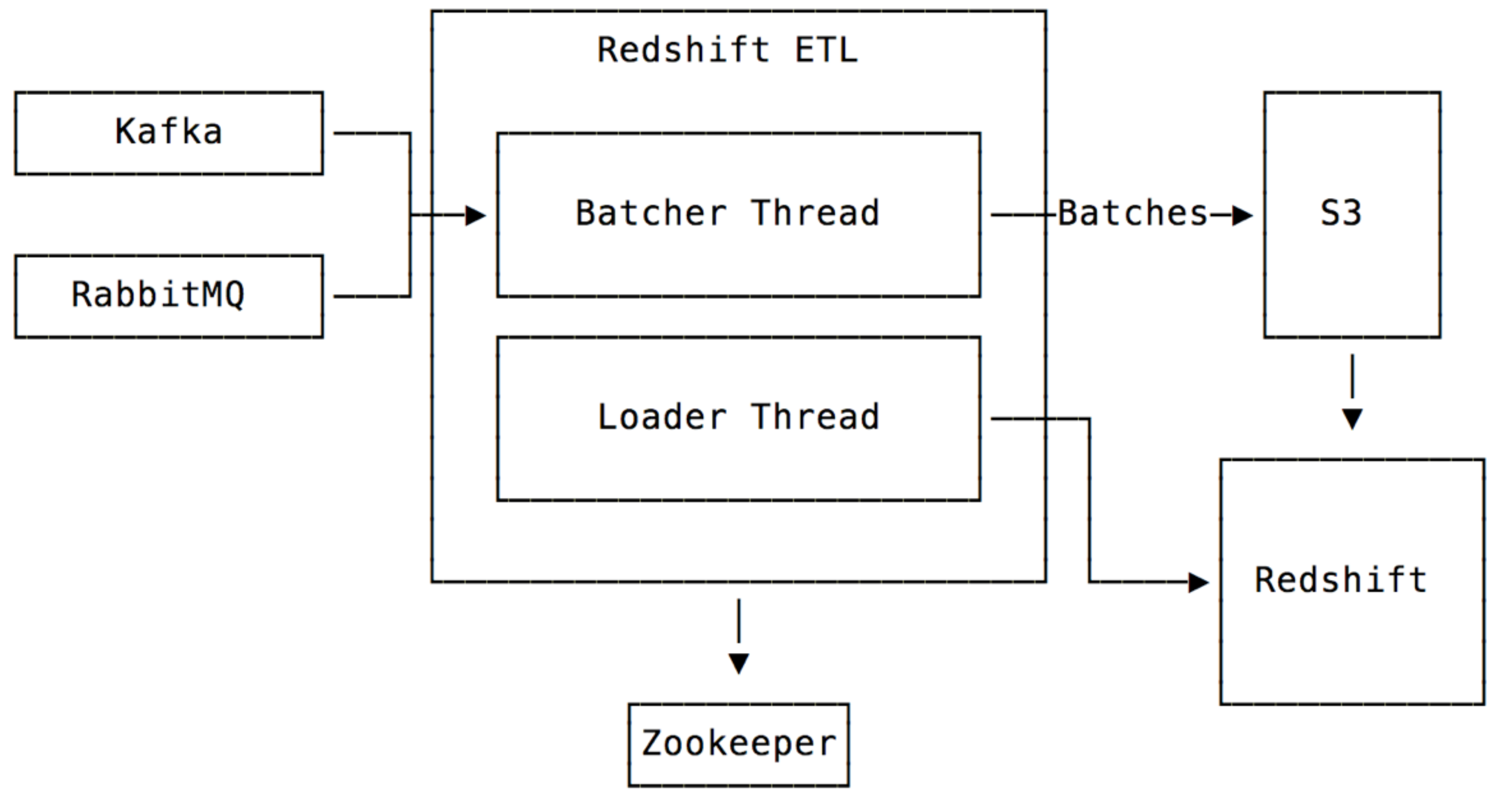
"Herald"

Data handling for email

"Flapjack"

User events from clients

SIMPLE



“Horizon”: Redshift ETL

Batcher: Read from Kafka/RabbitMQ, batch messages, put batches to S3.

Loader: Load batches from S3 to Redshift. Bisect batches for bad msgs.

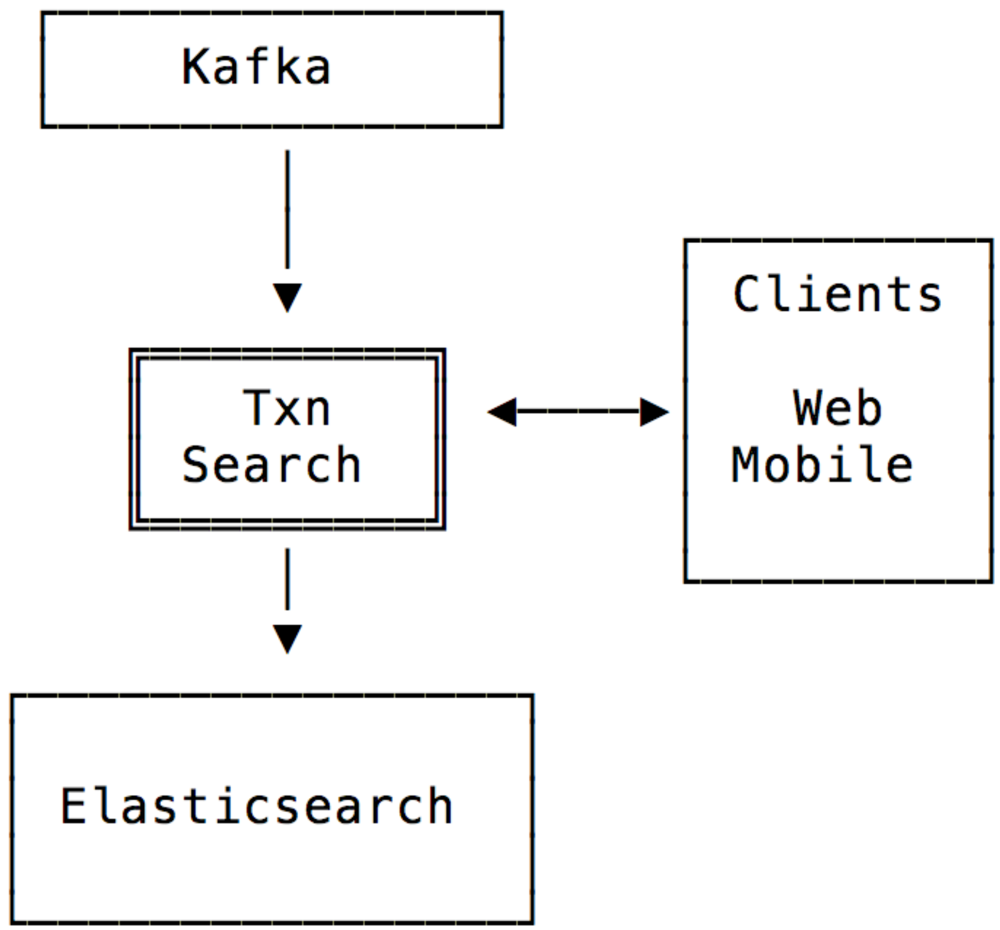
Zookeeper: With multiple instances of Horizon running, we need to keep locks on given tables when loading

Horizon Data Schema

```
{
  "table_name": "some_table_in_redshift",
  "datum": {
    "column_name_1": "value",
    "column_name_2": "value"
  }
}
```

Horizon polls Redshift every N minutes for table names: no need to change service when you want to send data to a new table

SIMPLE



“Pensieve”

Transaction Metadata Search

- Read from Kafka
- Index to Elasticsearch
- Serve client search queries

Those are the pipes.

What has the Data team learned?

SIMPLE

RabbitMQ —> Kafka

RabbitMQ was in place before most of the Data infra was build. Now transitioning to Kafka.

Why?

Offsets

Network Partition Tolerance

Offsets make backfills and error handling easy. RabbitMQ does not play nicely with network partitions.

SIMPLE

Metrics and Healthchecks!

SIMPLE

Use Postgres....

It is a very, very good database. Whether you need a relational store, JSON store, or key-value store, Postgres is often a great solution.

...except where Elasticsearch might be a better fit

ES is a more powerful full-text search engine because of Lucene. The scaling story for Postgres full-text search wasn't clear, and the query capabilities are more limited.

The flexibility of Horizon's data model has made scaling our Redshift schema easier.

DB table migration tools make schema management easier.

(we use Flyway)

(Alembic is nice for Python)

Celery (the task queue) is nice.

Because we're streaming data into the warehouse, our ETL jobs are mostly around view materialization.

If you don't need a Directed Acyclic Graph (DAG) for your ETL, it might not be worth the complexity.

SIMPLE

Elasticsearch Data Migrations are painful.

Do you have a strict schema (**ES won't let you write unknown fields**) and want to add a new field?

Just reindex everything you've ever indexed.

Dropwizard is great.

Languages

JVM for Services.

Python* for analysis, one-off-ETL, DB interactions...

**(and maybe a little shell scripting...)*

SIMPLE

Java for service libraries.

Scala, Clojure, or Java for HTTP services.

One place for critical shared code be reviewed by everyone on the team.

Includes Healthchecks, Metrics, an S3 wrapper, and Kafka Producers

Having the flexibility to write in different languages is great, but we needed to think about the team.

Questions?