




Data Engineering Architecture at Simple



\$ whoami

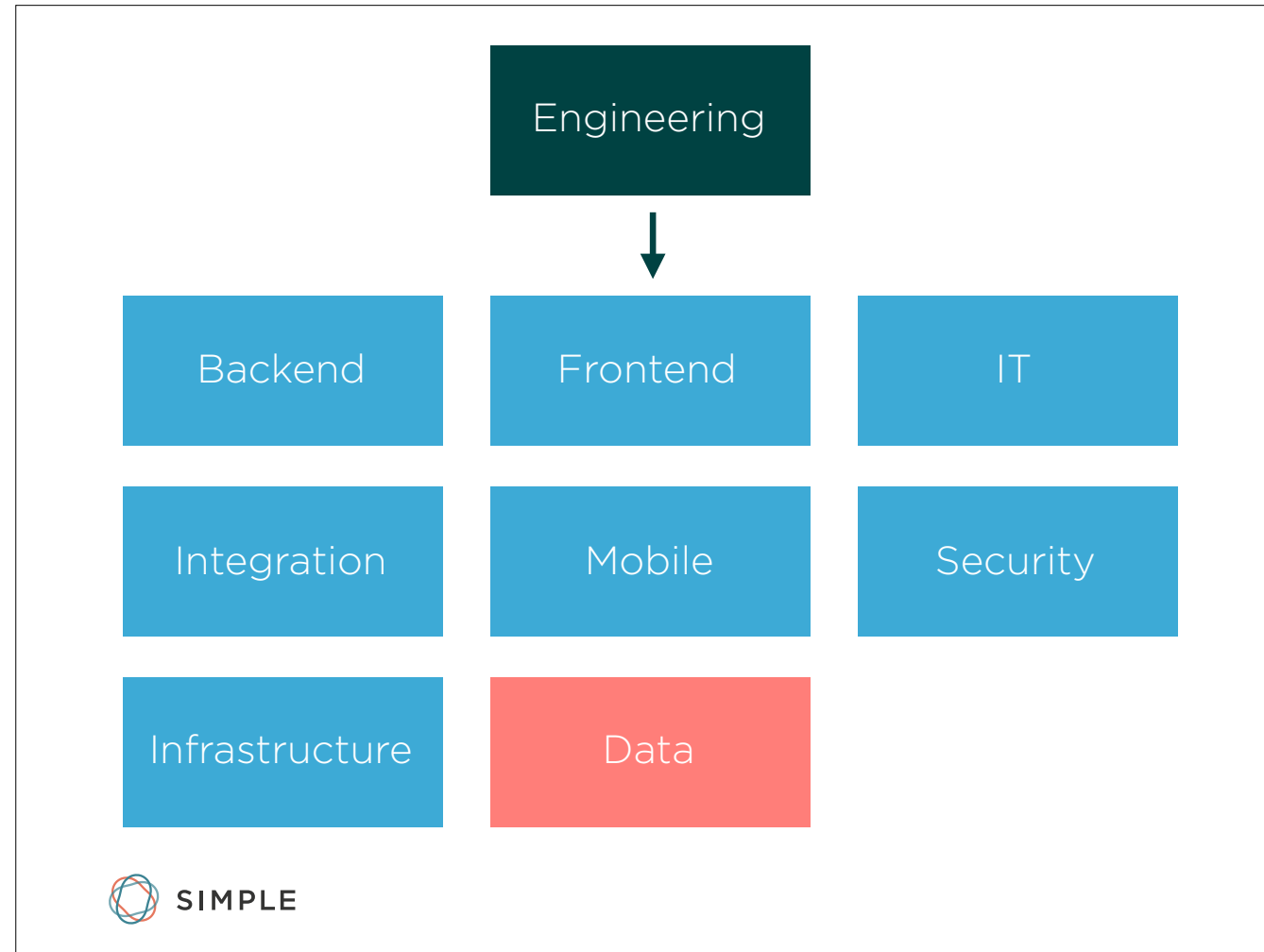
Rob Story
Senior Data Engineer
 **@oceankidbilly**



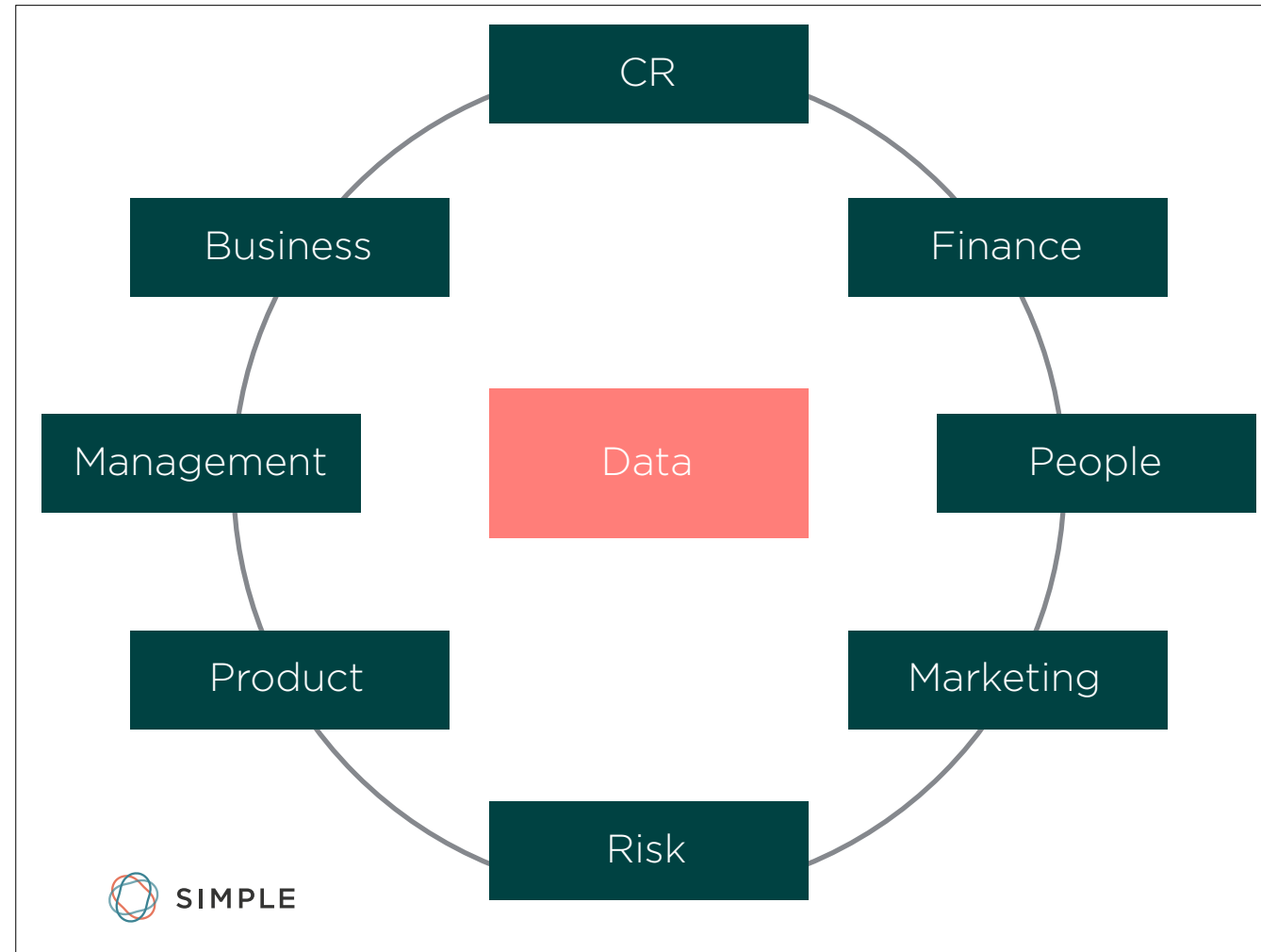
Why do we have a Data Team?



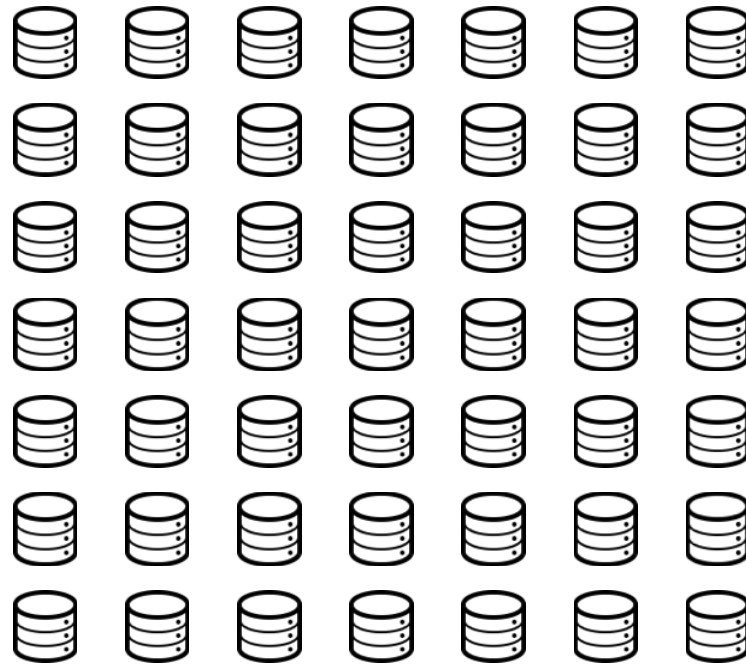
Why does Simple even have a data team? What role does it serve within the company?



Each of the Engineering teams has a pretty clear job: Backend builds core banking services; Frontend and Mobile build clients; IT keeps our tools, hardware, and networks functioning well; Integration manages and plans projects; Security keeps us safe; Infrastructure builds our operational platform (they used to be called Ops). So where does Data fit in?



Our role: all of the other groups in the company eventually need data from engineering. Everyone wants to know about some facet of customer behavior, be it Marketing & Product trying to figure out how customers interact with Simple or CR trying to serve our customers more effectively with data. The Data Team exists to help the rest company make decisions and tell stories based on the data we gather.



How do we get this data? Simple runs on over 35+ separate services, and most of those have their own Postgres database instance. So in a world without the data team or a data warehouse, what did it look like to answer a simple business question about our customers?

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



Example: let's say that the marketing team wants a list of all of the users who have made a transaction $> \$1$ in the past week so that we can send them a thank you for using 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



What does it look like to do this when the user data and transaction data are in two separate databases? Well, assuming that your Ops/DBA folks even allow you access to production data, you need to fetch both datasets into a format that you can read with R/Python/language of choice and then join them in-memory.

Data Warehouse: Redshift

```
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';
```



Now let's assume the data is in a warehouse that lets you use SQL. What does it look like now to answer your query?

Data Warehouse

35+ DBs



1 DB



Analysts



Engineers



Dashboards

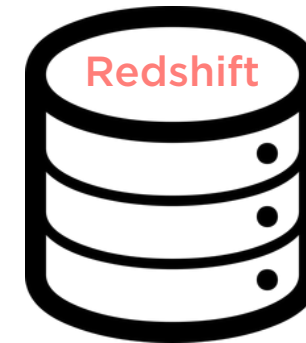
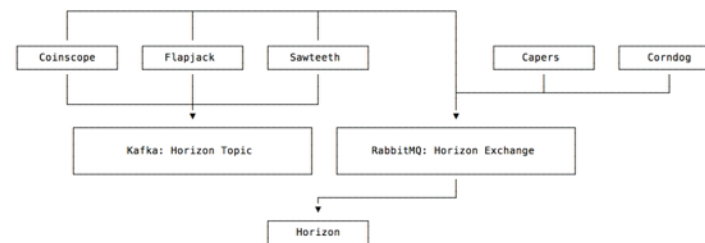


The Data Warehouse (Redshift in our case) lets us collapse 35+ Postgres DBs into a single DB that analysts and engineers can query and build dashboards against.

Data Engineering

What do we do?

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



So what does Data Engineering do? We build the tools and systems to make data available to all who need it. This includes building data pipelines to move data from the backend Postgres DBs to Redshift as well as managing Redshift itself. We also build some services to send data to 3rd party vendors and (recently) provide some data-centric services to clients such as transaction search.



Redshift

Let's talk about Redshift!

Why Redshift?

SQL!

Fast Loads from S3

Fast Query Times*

Security

Simple runs on AWS

*what's the catch?



Why Redshift? Most analysts and engineers already know SQL. It loads from S3 **very** quickly. The query times are fast (more on this later). Security story is straightforward and can leverage IAM (and is doing more in this direction). The rest of Simple runs on AWS, and Redshift is a thing on AWS. So what's the catch?

“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



Redshift is a distributed system! Let's start with the stock quote from the Redshift docs about what exactly Redshift is, and walk through each of these pieces one at a time to show how these snippets are meaningful in how you use Redshift.

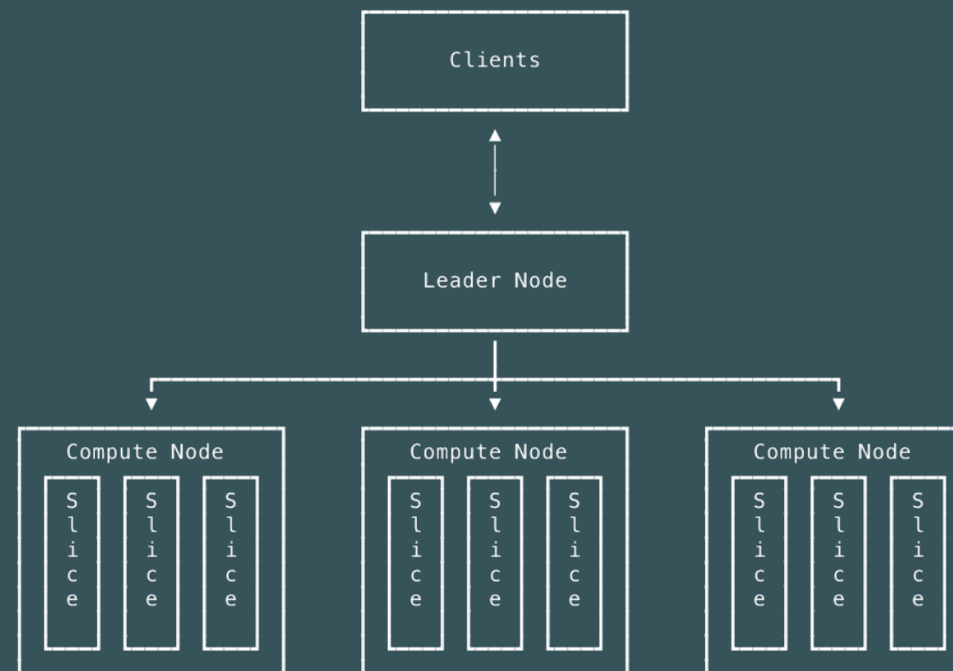
“...enterprise-class relational database query and management system...”

- 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



Redshift looks like a regular SQL database from the query side. If your analysts are familiar with SQL, they'll be able to start using it immediately. The query language is only a little different, and the docs are good.

“...massively parallel processing...”



However, Redshift is parallel: your data is split amongst many compute nodes, which further splits it into “slices”, where each slice is allocated a portion of the node’s memory and disk space.

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);
```



DISTKEY and SORTKEY: Redshift allows you to decide how you want to distribute and sort your data. In this case, we are distributing by user_id, and sorting it by when_recorded (to speed up date-based range queries)

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

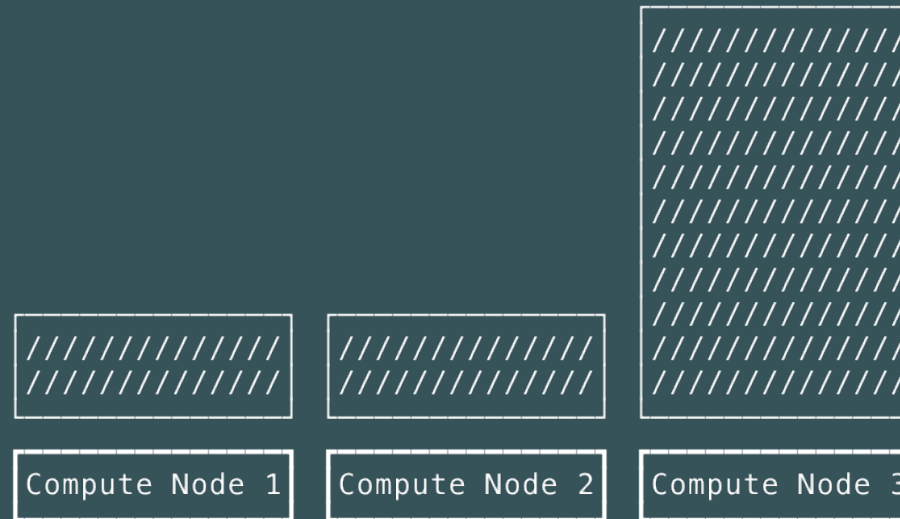
If these statements are omitted or specify the incorrect columns, query performance will decay as the tables grow.



The DISTKEY tells Redshift how you want your data to be distributed between all of the worker nodes. The SORTKEY tells Redshift what column you want to keep the columns sorted by to feed into the query optimizer. If you don't put some consideration into these two things, performance will suffer.

Distkey Example:

`hash(user_id) % N_Worker_Nodes`

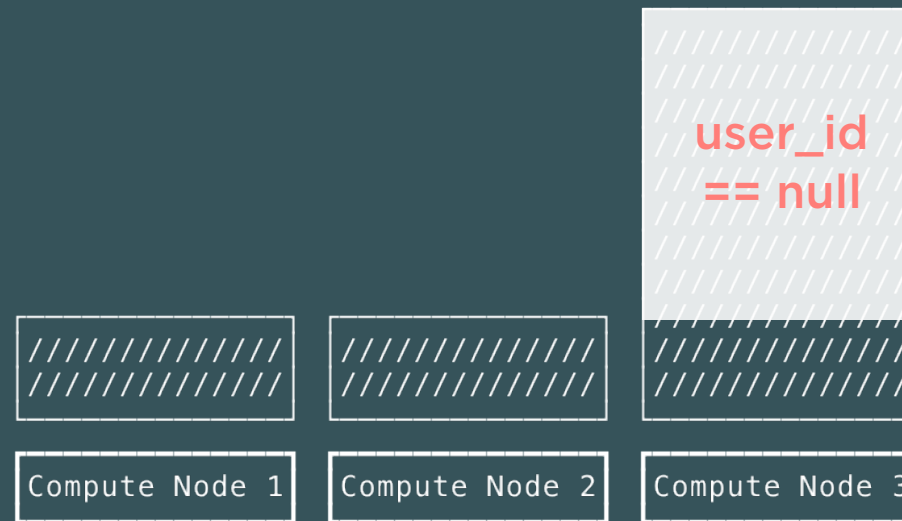


As we load data into the table, Redshift will use hash-based partitioning to hash the `user_id`, modulo it by the number of worker nodes, and send it to the appropriate node.

But one node is getting more data than the other. What's happening?

Distkey Example:

`hash(user_id) % N_Worker_Nodes`



`user_id == null`! A bunch of messages came through with a `user_id` of null, and now we have a LOT more data on one node. That particular disk is filling up.

The reason I'm using this particular example is because it happened to us. We didn't realize so much (valid) data was going to come through with no `user_id` attached, and needed to redistribute the data accordingly.

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!



The solution: distribute all data evenly across nodes.

But we've made a HUGE tradeoff here: now our joins on *user_id* require a shuffle across nodes, and our range queries on *user_id* touch all nodes. Bummer. If we could partition **all** of our tables on *user_id*, joins could be calculated by a single node.

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)
```



This is a naive join on two tables on user_id. Using EXPLAIN (try this command if you never have!) tells us that Redshift is going to perform a Hash Join. It's going to build a Hash Table from the goals table, and then probe it with a full scan through transactions.

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.



Both of these things have happened to us in the course of query execution. Luckily you can usually track these things in the Redshift dashboard and cancel the query before things get out of control. You can also avoid your disk filling by specifying timeouts on all queries in the cluster. This is **not** set by default.

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);
```



What if we create two temp tables with the distkey and sortkey we're going to be joining on? Temp tables are pretty cheap to create. Note that I had to be really careful with the types here to ensure compatibility on the join key.

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)
```



Neat- we get a merge join, which can be performed in a streaming manner. No hash tables blowing up disk space, and generally *extremely* performant.

“...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 * when you only need results from a few columns will usually result in poorer performance than specifying only those columns.

Postgres

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

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

Redshift

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



Let's look at an example: we want to select two columns from a given table. With Postgres, we're going to do a full table scan when we perform this query, because the data is stored in a row-wise fashion. With Redshift, we only need to pull the two columns of interest from disk, because each column is stored separately.

*“...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



Compress-all-the-things. Redshift has helpers to let you know how best to compress your data. Warning: Analyze Compression will acquire a full table lock.

One last thing:
Redshift supports fully
serializable isolation.

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



SIMPLE

If you're thinking about streaming data into Redshift, note that transactions can be expensive because of Redshift's isolation levels and two-phase locking. Simple has had to back off how often we can load data into the warehouse.

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

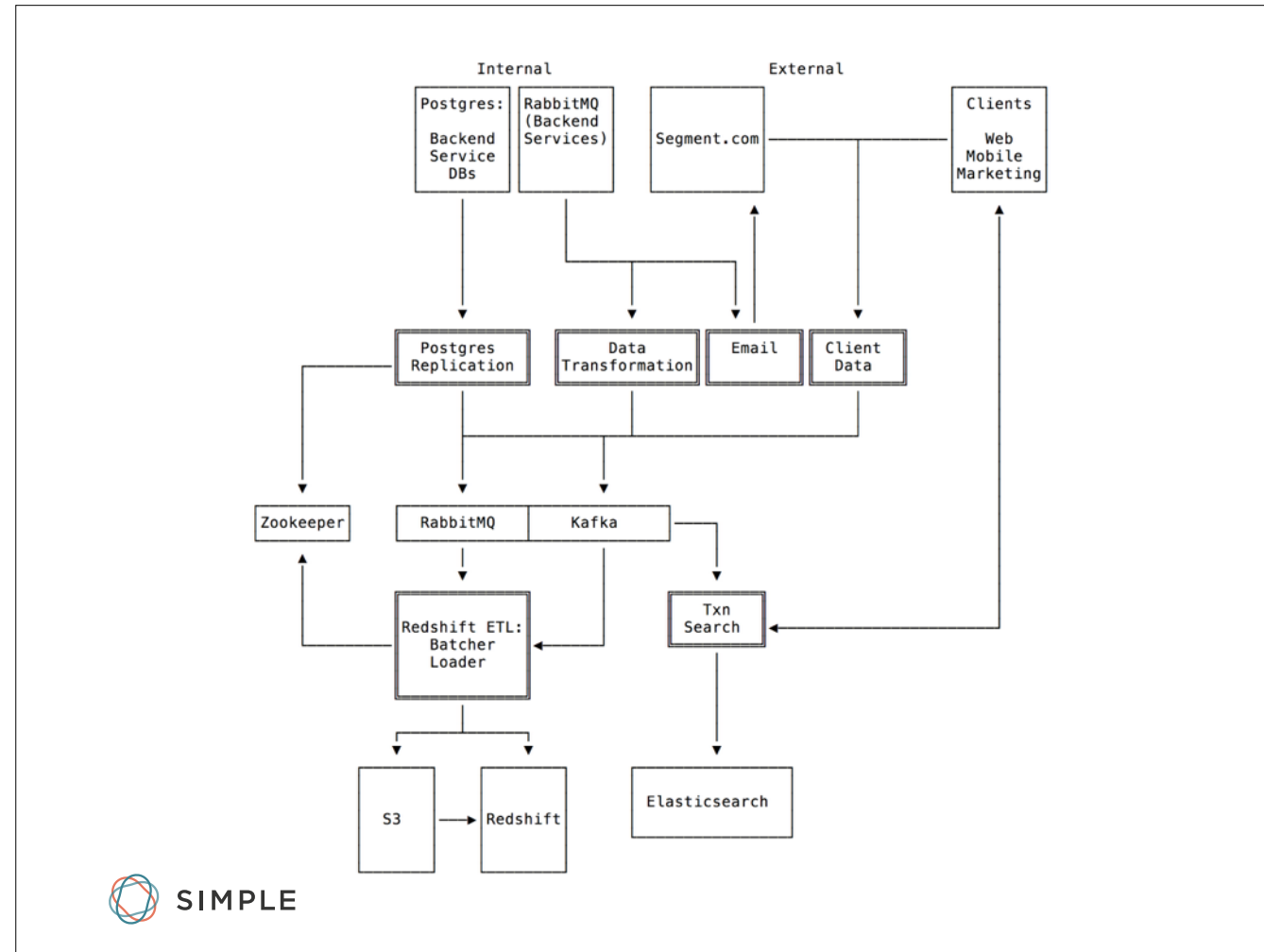
Keeping its distributed nature in
mind is important for performance
when building tables and queries.



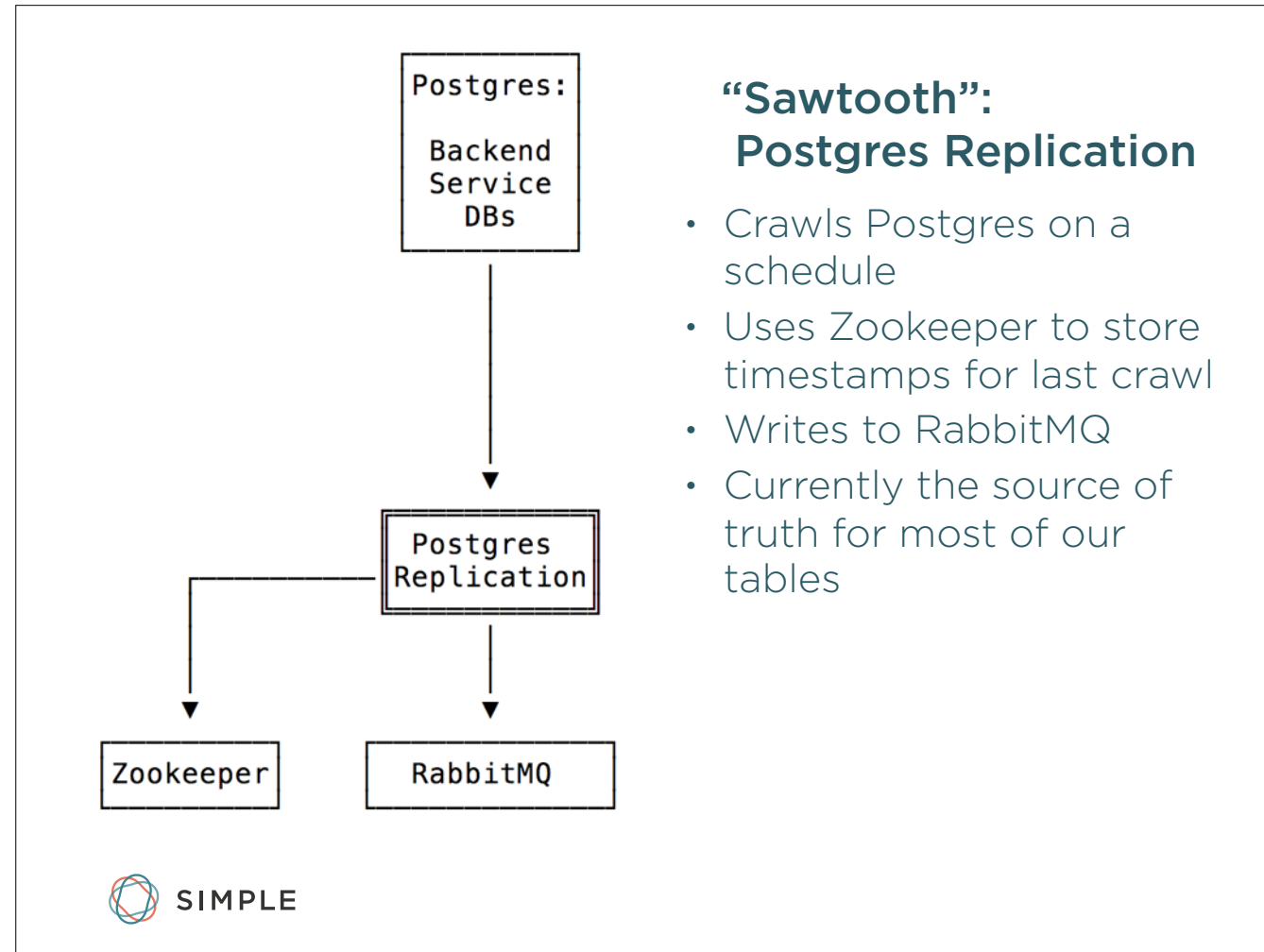
You can't quite think about Redshift like you can Postgres when you're designing schemas and building queries.



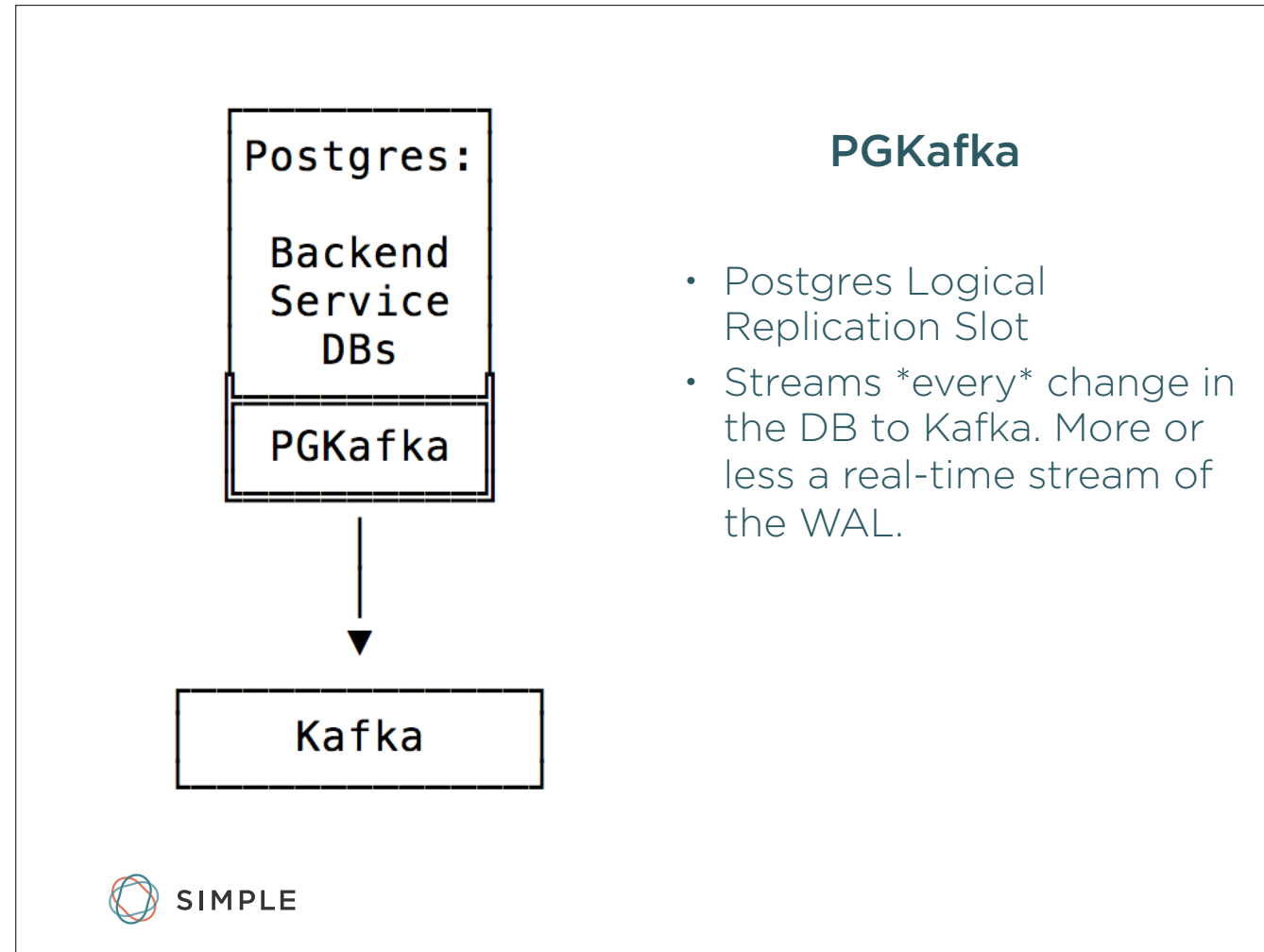
Pipelines



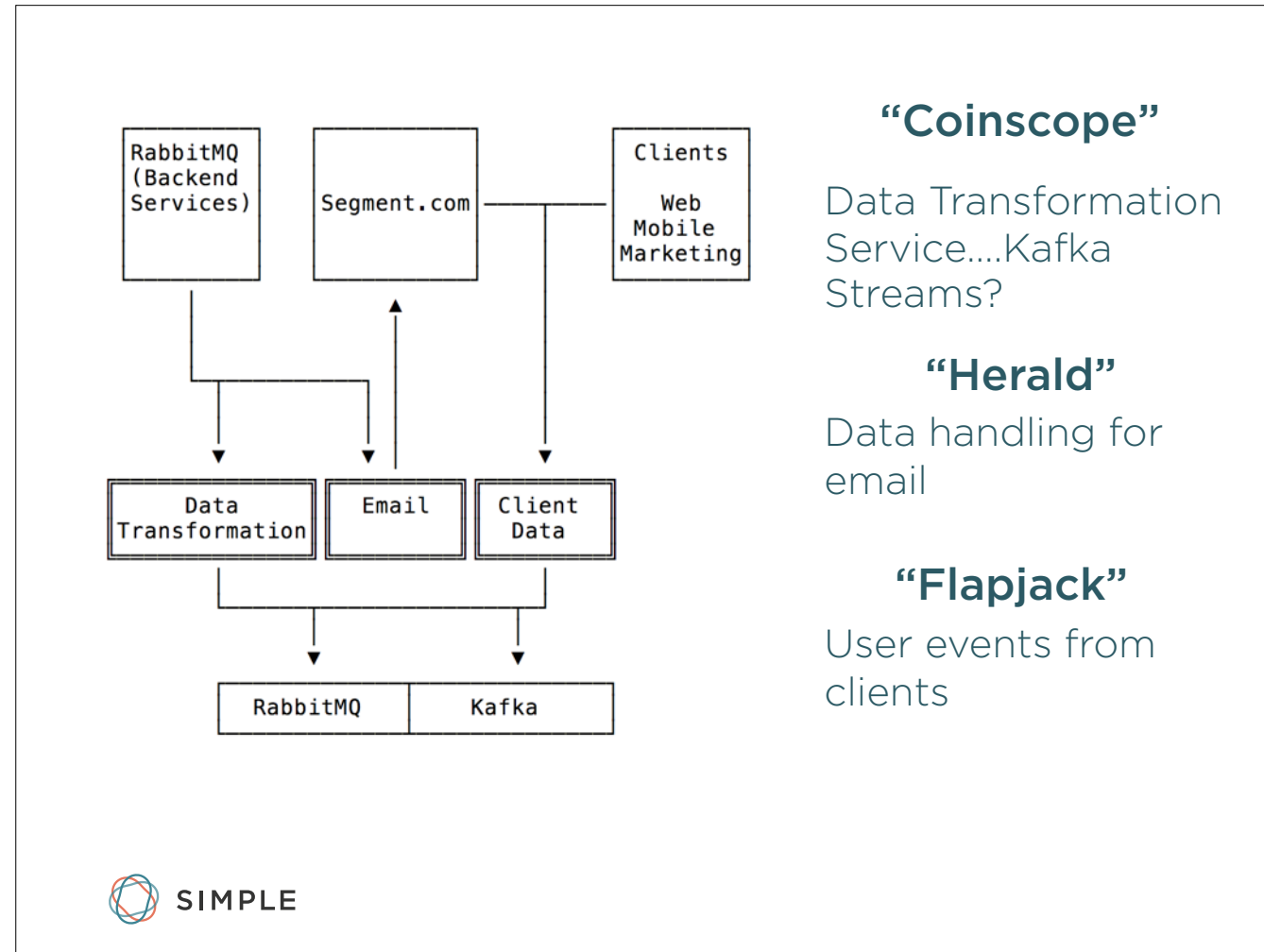
This is the entirety of the Simple Data Pipeline. Boxes with emphasized borders are our services- Postgres replication, Data transformation, Email, Client Data, Redshift ETL, and Txn Search. Everything else is connective tissue: Kafka and RabbitMQ as message queues, external services and clients making requests, Zookeeper, S3, Redshift, and Elasticsearch holding some state. Let's walk through each of these one at a time.



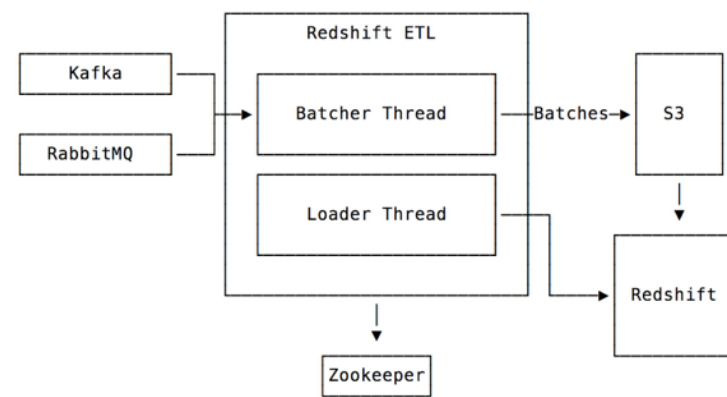
Sawtooth is a Scala service that scrapes our backend Postgres tables for data, forwarding it to RabbitMQ on a path to Redshift. This is the old way of doing things, and is used for our DBs that haven't been upgrade to Postgres 9.4. The future is...



PGKafka is our future. A logical replication slot in Postgres allows us to stream every change from the DB to Kafka: inserts, updates, deletes, you name it.



A quick rundown of some of our other services: Coinscope acts as a data transformation service, reading data from RabbitMQ, transforming it, and putting it back to RabbitMQ/Kafka. Herald is a proxy for backend data and our email providers. Flapjack handles all client data via HTTP endpoints. As business needs grow, the Data team ends up tackling a lot of these data integration problems.



“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 is the workhorse of our data pipeline. It reads from both Kafka and RabbitMQ, batching messages and putting the batches to S3. A separate loader thread reads those batches, attempts a NOLOAD COPY into Redshift. If it fails, it bisects the batch until a COPY works, then puts the bad messages to S3 for later investigation/processing.

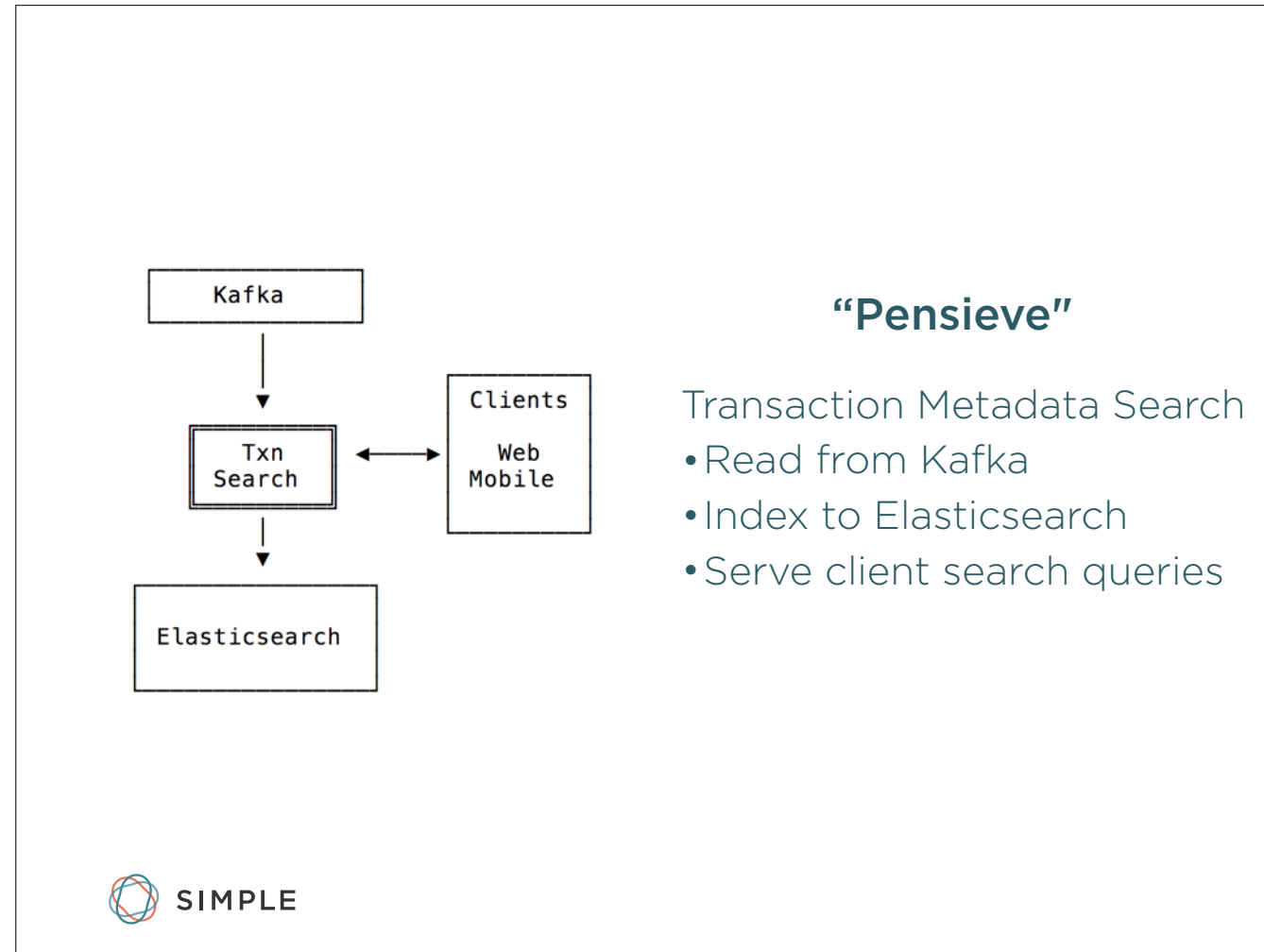
Horizon Data Schema

```
{  
  "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



IMO, the Horizon data schema is one of the best decisions made when the service was built. It takes a table name and datum that maps columns to values, batches by table name, and writes the batches to Redshift. Horizon polls Redshift for tables every N minutes, so if you want to send a new table to Horizon, you don't need to change the service at all. It will eventually pick up the new table and start loading data to it.



Pensive is our transaction metadata search service. It indexes transaction data into Elasticsearch to enable full-text search of our transaction data. Not serving end users yet, but hopefully coming online soon.

Those are the pipes.

What has the Data team learned?



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.



Most of Simple was built on RabbitMQ, but now the Data team is transitioning as much of our infra to Kafka as possible. Why? Kafka's model of being able to read from a given offset allows for easy data backfills and error handling, in the event a service crashes and needs to replay data. Additionally, Kafka has a better story around network partition handling. Straight from the RabbitMQ docs: "RabbitMQ clusters do not tolerate network partitions well."

Metrics and Healthchecks!



There are a lot of moving parts to keep track of. If you don't have your services properly instrumented with both metrics and healthchecks, deploying will be scary, and you won't know when your stuff is broken.

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.



If you're looking for a database, Postgres is probably the answer. It's widely used, flexible, and serves Simple's needs well. We used Elasticsearch for the Pensieve service because it's a more capable full-text search engine, and the Postgres full-text scaling story wasn't clear. Talked a bunch of folks who use ES a LOT, and despite mixed feedback on operational stuff, it serves them well for billions of records.

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



By polling the Redshift schema for new tables and allowing any producers to send new table data to Horizon without a change to the service, it's been very easy to build our schema.

DB table migration tools make schema management easier.

(we use Flyway)

(Alembic is nice for Python)



As the data warehouse gets bigger, managing migrations becomes more important. We use Flyway to add handle migrations, and haven't had any problems with it. Adding or modifying a table is as simple as adding a .sql file and calling the command line invocation.

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.



There are lots of ETL tools out there that allow you run jobs based on a Directed Acyclic Graph (DAG). However, we don't have any jobs like that right now. So we chose celery, a python task queue. Defining tasks is as easy as creating python functions, and it runs on a cron-like scheduler. It's great.

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.



Elasticsearch allows you to define a mapping, or schema, in two different ways. 1. You can say the schema is strict, which means it will not accept a message that doesn't adhere to the provided mapping of types 2. You can let ES infer types and allow for new fields whenever they are present. This option is a path to mayhem. Instead, you need to create a new index, duplicate writes to both index, backfill from the first index, atomically switch the index aliases...

Dropwizard is **great**.



All of our services, including our Clojure ones, run on the Dropwizard framework. It has a lot of things like health checks and metrics built in, and we're really happy with it.



Languages

JVM for Services.

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

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



All of our services run on the JVM, be they in Scala, Java, or Clojure. The JVM is performant and Simple Engineering is good at running JVM services. But for all of those one-off jobs where we might need to backfill data, perform analysis, pull a few rows from the DB, etc? We use Python. We even have an OSS lib: <https://github.com/SimpleFinance/shiftmanager>

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



We use Java for shared service libraries, and HTTP services can be built on your JVM language of choice (yes, Kotlin has been discussed). Having shared code in a single modularized library lets us reuse critical application code such as Kafka consumers and health checks on message volume.

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



Before writing the Pensieve service in Clojure, we had a discussion amongst the team about whether or not we wanted to write a major data service in the language. Adding languages requires more cognitive overhead for the team. Folks down the road have to manage these services and be able to review PRs and fix bugs. Before adding a new language, it's important to make sure everyone is onboard.



Questions?