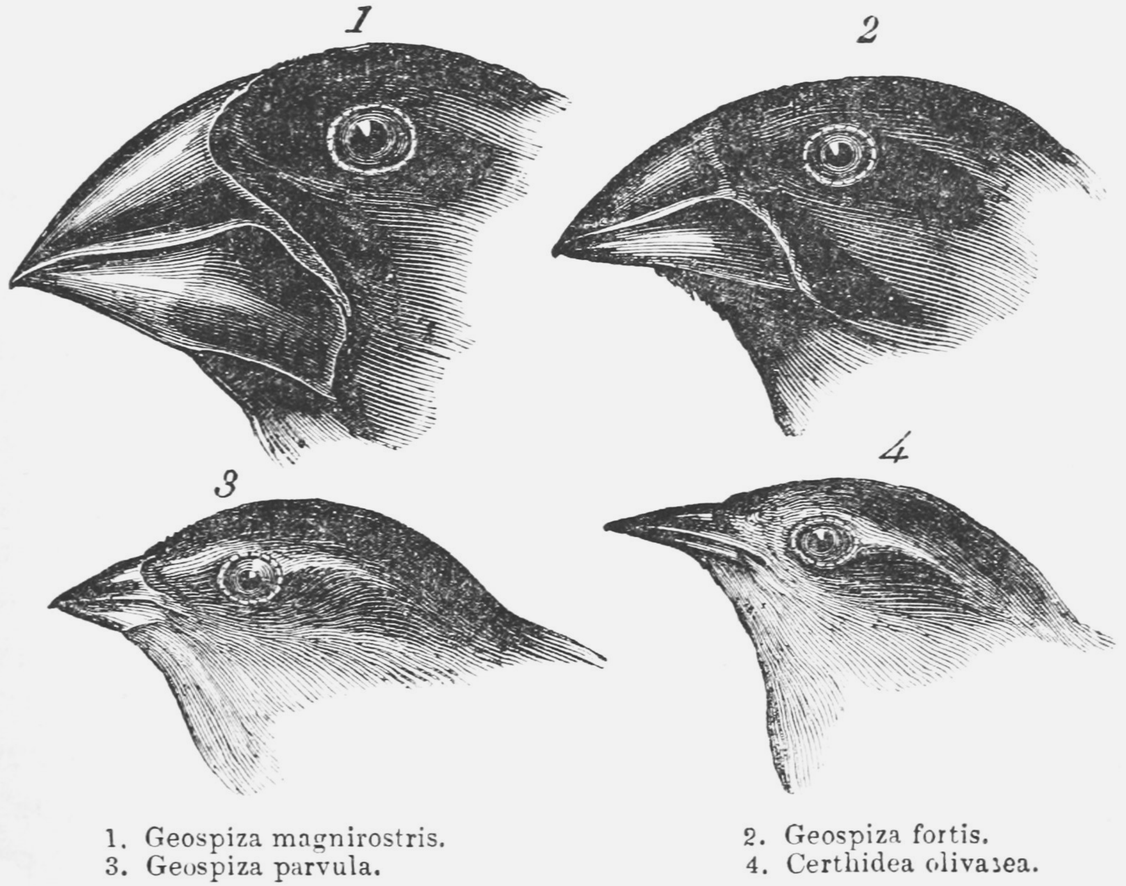
Schema Evolution Patterns

Alex Rasmussen
alex@bitsondisk.com



John Gould (14.Sep.1804 - 3.Feb.1881) [Public domain]

Hi, I'm Alex!

Twitter, GitHub, etc: @alexras

alex@bitsondisk.com

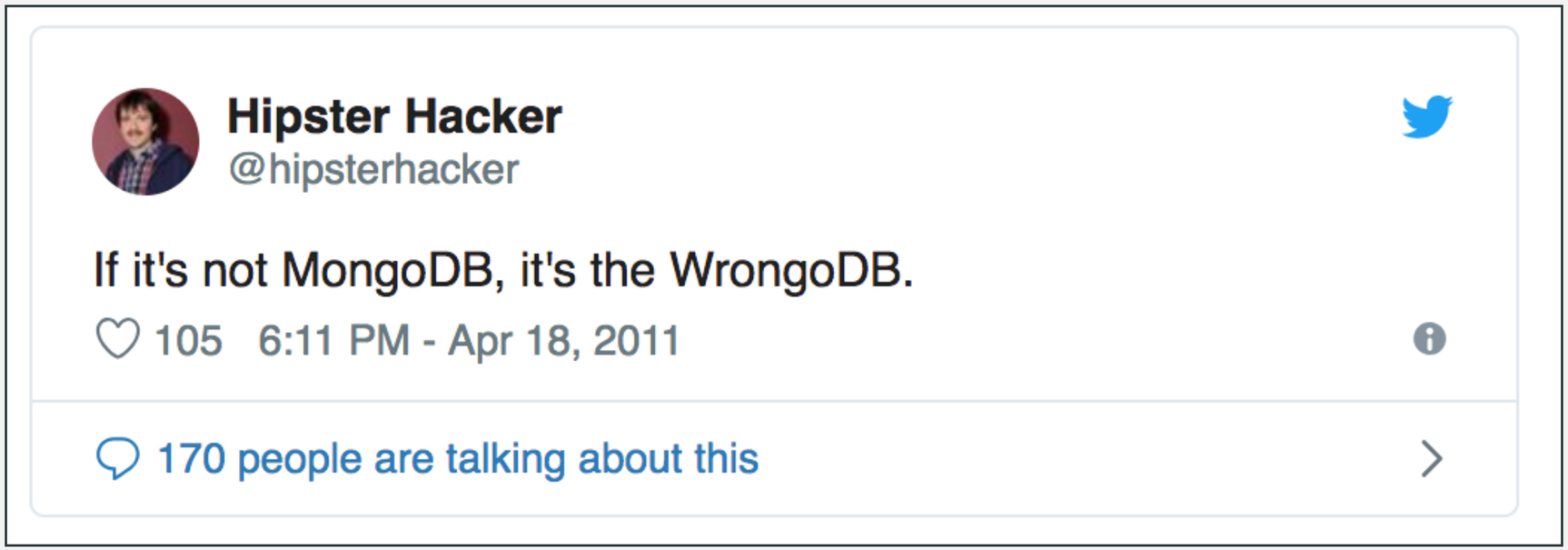
LA-based Data Engineering Consultant

BITS ON DISK

https://www.bitsondisk.com/

Data Migrations Are Hard

- How should I migrate? When?
- Will the old code still work?
- Can I migrate without downtime?
- Can I roll back? How? What if I can't?



Should I just use MongoDB?

How do I decide?

API Versioning Is Hard

- When should I bump versions?
- Will old callers break?
- Should I just switch to GraphQL?

THESE HARD PROBLEMS ARE THE SAME PROBLEM!

Schema Evolution

- The structure of your data will change
- What does that mean for the data?
- How should our systems respond?
- How does this impact how we build?

Goals of This Talk

- Mental toolkit for reasoning about schema evolution that applies across technologies and domains
- How this looks in practice

References: https://github.com/bitsondisk/velocity-sj-2019

1. WHAT ARE SCHEMAS?
2. SCHEMA COMPATIBILITY
3. CROSSING COMPATIBILITY GAPS
4. WRAPPING UP / TAKEAWAYS

1. WHAT ARE SCHEMAS?

2. SCHEMA COMPATIBILITY

3. CROSSING COMPATIBILITY GAPS

4. WRAPPING UP / TAKEAWAYS

Schemas

- From the Greek skhēma, meaning “form” or “figure”. The data’s shape.

```
Person {
    name: string,
    age: integer,
    favorite_color: enum(RED, BLUE, …),
    . . .
}
```

What/Where Is the Schema?

- **Explicit**: SQL tables, Thrift, Avro*
- **Implicit**: CSV, Excel, JSON
- **Separate**: JSON Schema, schema registries, an ex-employee's brain

When Are Schemas Enforced?

- On Write: e.g. in RDBMS tables
- On Read: enforce after reading
- On Need: dynamic, late-binding
- Can enforce one schema, or many

1. WHAT ARE SCHEMAS?

2. SCHEMA COMPATIBILITY

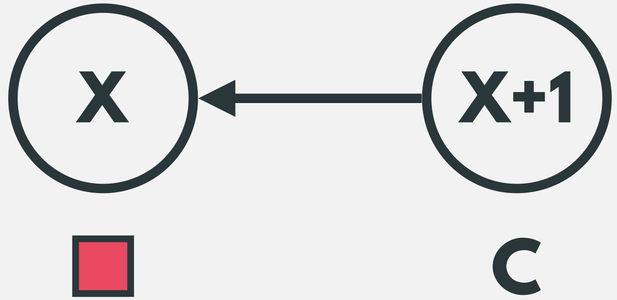
3. CROSSING COMPATIBILITY GAPS

4. WRAPPING UP / TAKEAWAYS

Schema Compatibility

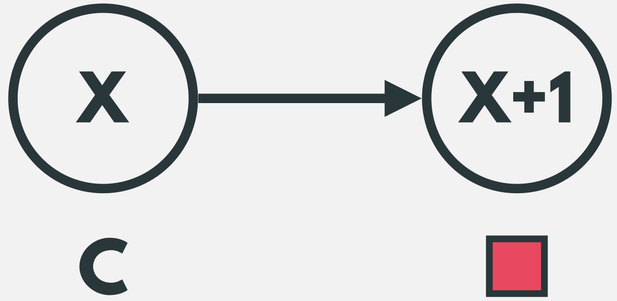
- If two schemas are compatible, evolving from one schema to another can be done automatically on read*
- Readers can be oblivious to schema change**
- Two directions: backwards and forwards

Backwards-Compatible



Data written with old schema
readable by clients with new schema

Forwards-Compatible



Data written with new schema
readable by clients with old schema

Add a Field With a Default

```
name: string,
age: integer,
is_admin: boolean (default: false)
```

```
name: “Bob Jones”,
age: 42
```

```
name: “Bob Jones”,
age: 42,
is_admin: false
```

Backwards: substitute default in older data

Forwards: ignore field in newer data

Remove a Field With a Default

```
name: string,
age: integer,
is_admin: boolean (default: false)
~~pto_days_left: int (default: 0)~~
```

```
name: “Alice Smith”,
age: 29,
is_admin: true
pto_days_left: 16
```

```
name: “Alice Smith”,
age: 29
is_admin: true
```

Backwards: ignore field in older data

Forwards: substitute default in newer data

Other Types of Changes

- Without defaults:
  - Adding a field breaks backwards-compatibility
  - Removing a field breaks forwards-compatibility
- Field type changes: typecasting rules apply(?)
- Renaming and reordering: it depends

In Practice - Protocol Buffers

```
message Person {
    required string name = 1;
    required int32 age = 2;
    optional bool is_admin = 3
        [default = false];
}
```

- **Field numbers can’t change or be reused**
- **In proto3, no** `required` **or** `optional`

In Practice - Avro

```
record Person {
  string name;
  int age;
  boolean is_admin = false;
}
```

- **Reorders are OK, but not renames**
- **Compatibility determined by rules**

In Practice - GraphQL

```
type Person {
  name: String!,
  age: Int!,
  is_admin: Boolean
}
```

- **Every field is nullable by default**
- **Only return queried fields (bounded enforcement)**
- **Hide deprecated fields; deletes tricky**

In Practice - Wix

- Adding fields
  - Non-indexed fields: JSON blob column (schema-on-read!)
  - Indexed fields: new table + join
- Removing fields
  - Don’t. Just stop using them (similar to GraphQL)

Recap

- Compatibility makes evolution easier
- Changes can be backwards-compatible, forwards-compatible, both, or neither
- Ease-of-compatibility drives many decisions behind “rules” of format design

1. WHAT ARE SCHEMAS?

2. SCHEMA COMPATIBILITY

3. CROSSING COMPATIBILITY GAPS

4. WRAPPING UP / TAKEAWAYS

Crossing Compatibility Gaps

- Not all schema changes are compatible
- Not all incompatibilities are simple
- Not all compatible changes are practical
- How do we deal with that?

Complex Changes

```
~~name: string,~~
first_name: string,
last_name: string,
age: integer,
is_admin: boolean (default: false)
```

- Not obvious how to split
- Client code changes (likely) required

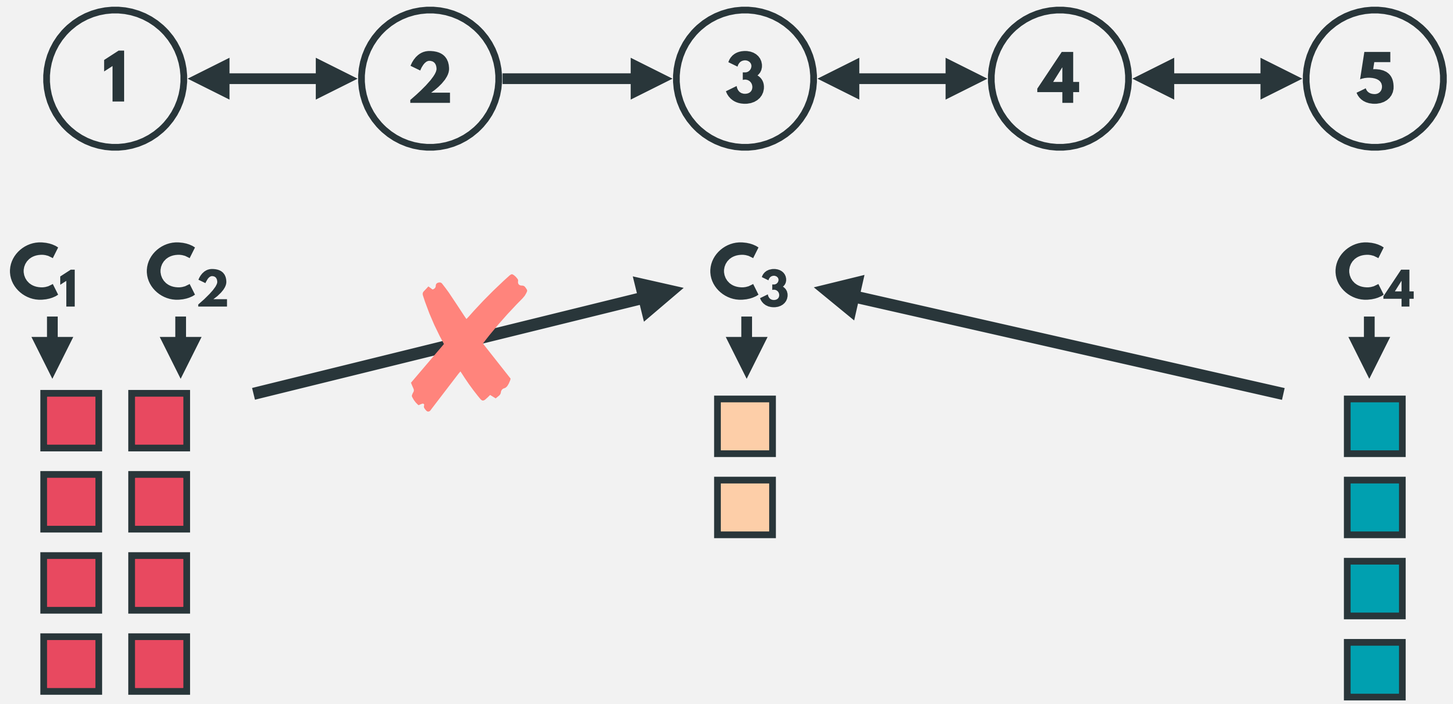
Impractical Changes

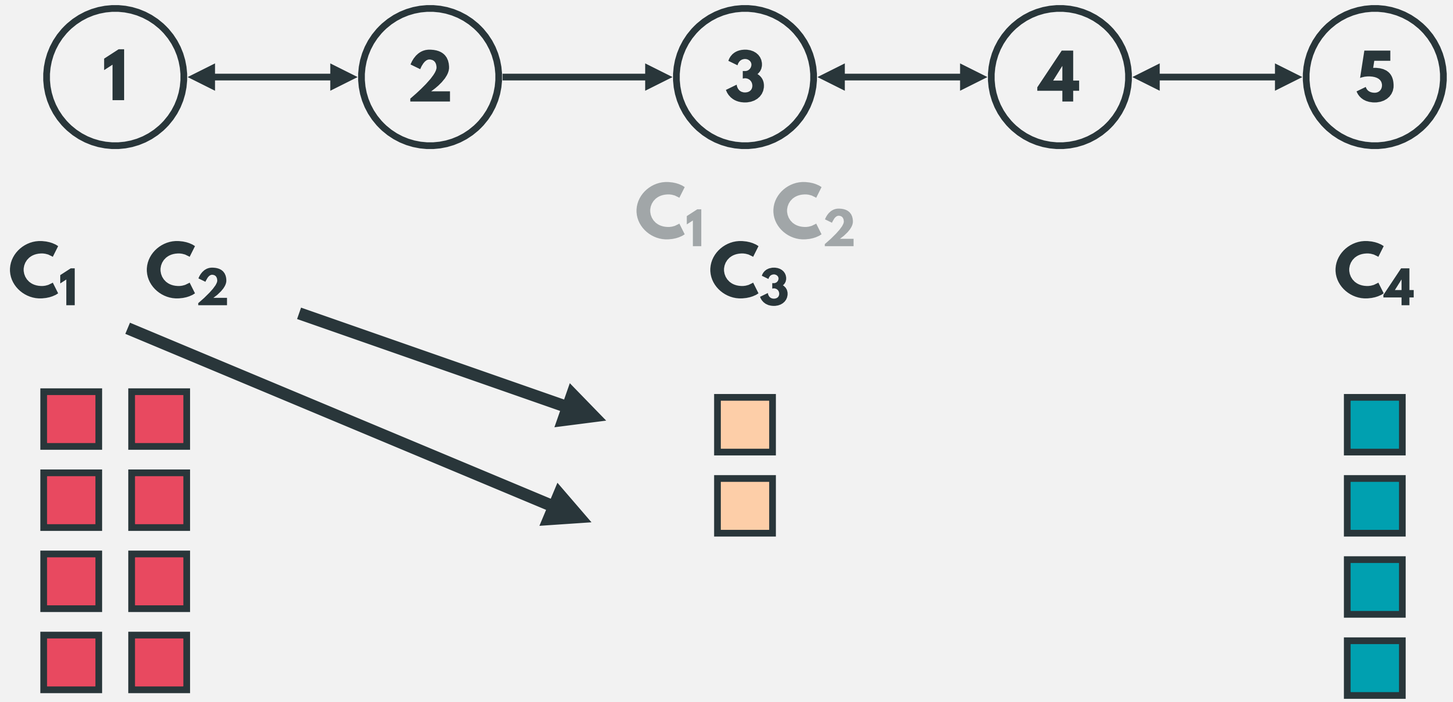
- e.g. Adding a column in MySQL requires locking/copying the table
  - Days to weeks not unheard of

Crossing Compatibility Gaps

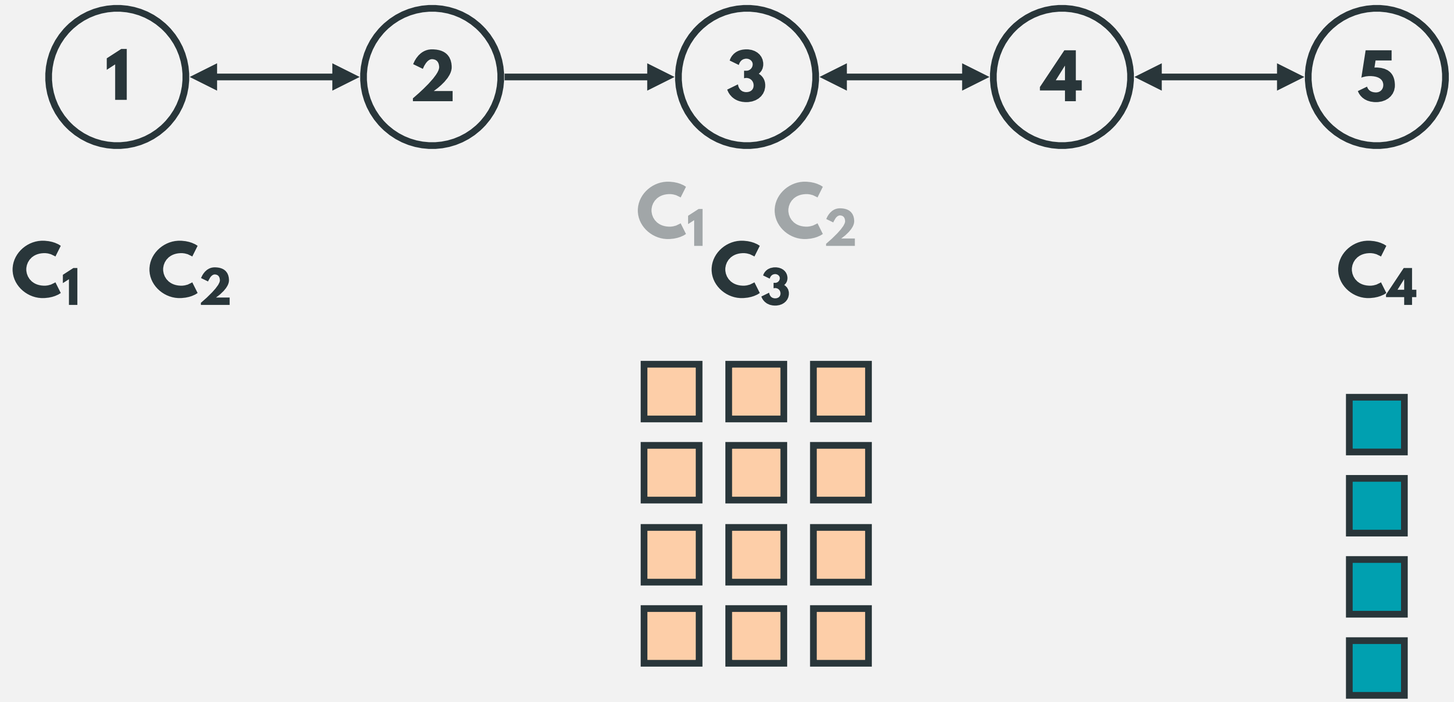
- We'll look at:
  - Multi-schema stores (Kafka, et al)
  - Single-schema stores (RDBMS, et al)

Multi-Schema Stores

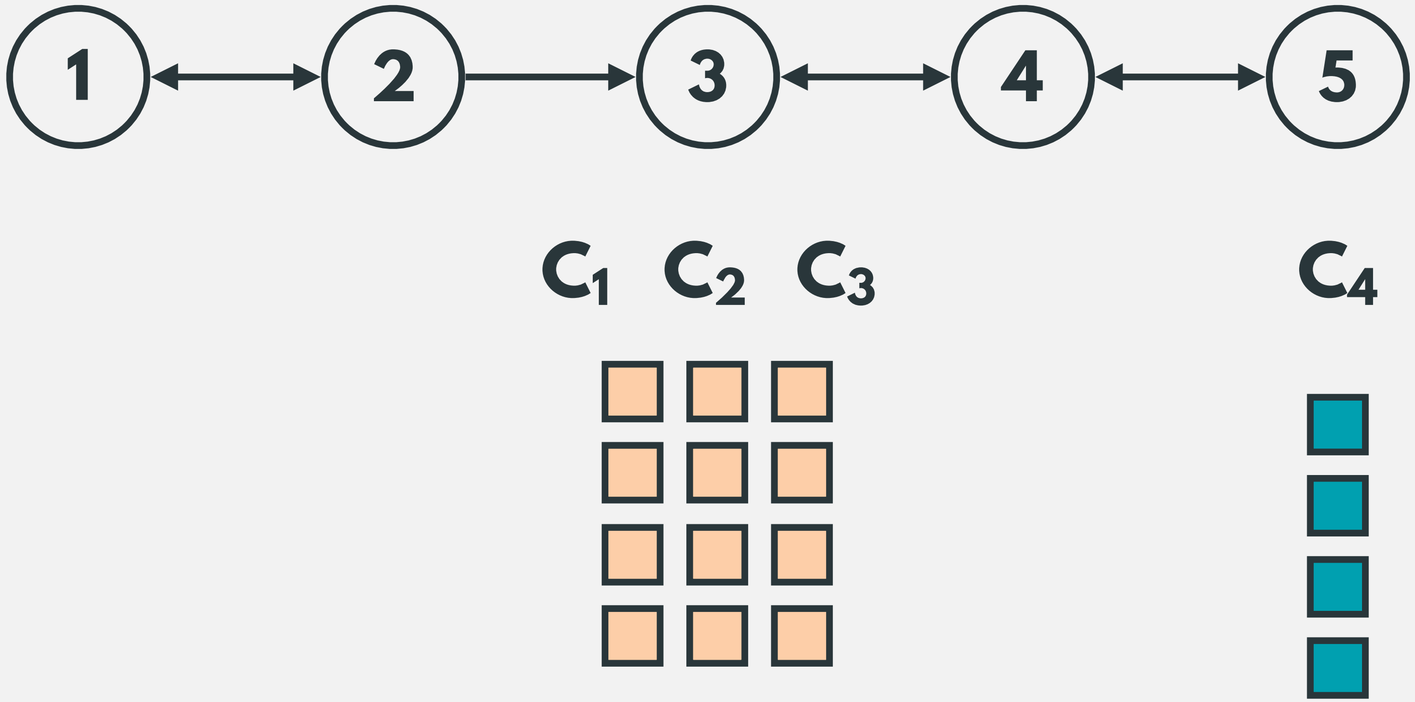




Step 1: Old clients write with new schema



Step 2: Migrate old data to new schema



Step 3: Migrate old clients to new schema

In Practice: Kafka (Confluent)

- Stores Avro schemas in schema registry with compatibility check built-in
- Backwards-compatible changes: update readers first
- Forwards-compatible changes: update writers first

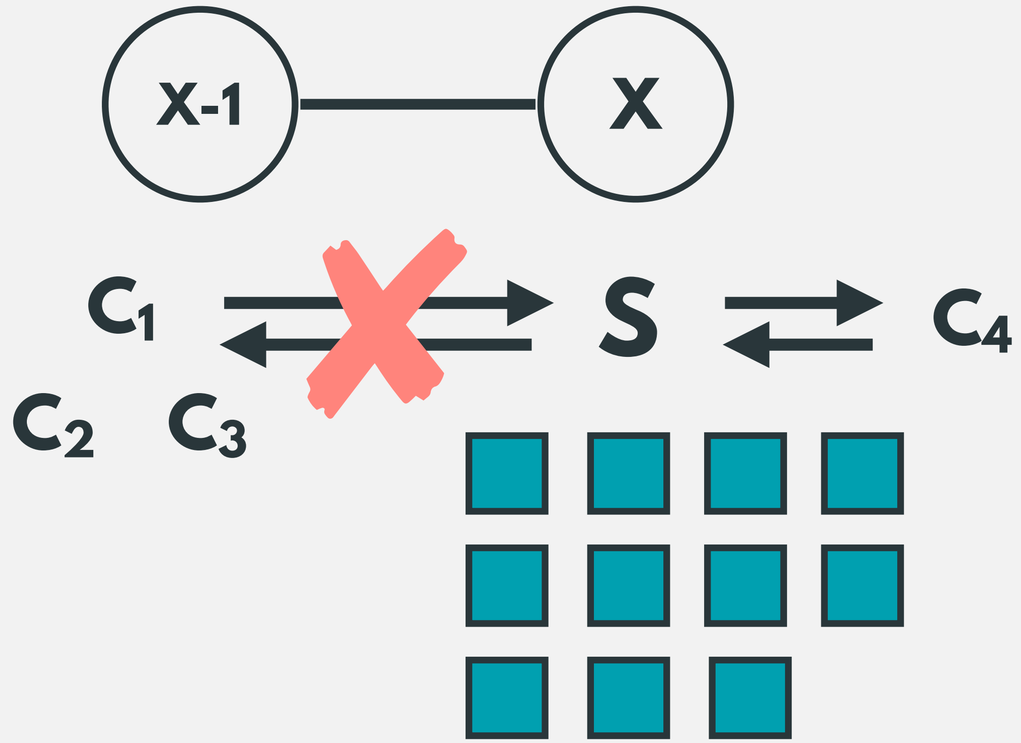
In Practice: Stripe

- **Goal**: (backwards-incompatible) API responses that work for old clients, forever
- Version change modules applied backwards from present to client's version
- They admit: this is **hard**, can't last forever

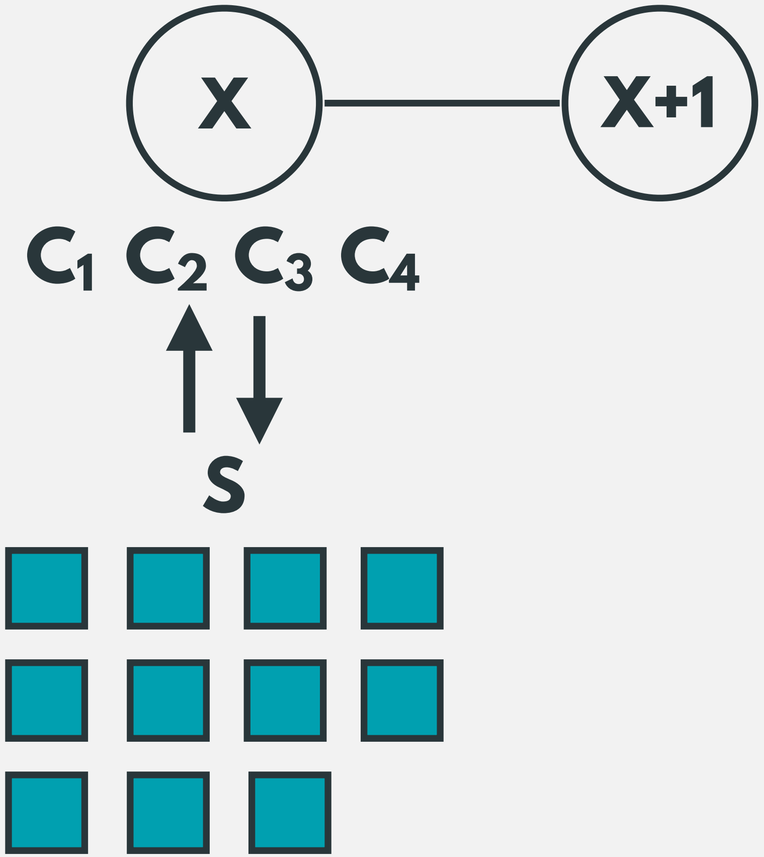
Recap

- In multi-schema stores:
  - Migrate client writes/reads
  - Migrate data
  - Migrate client reads/writes

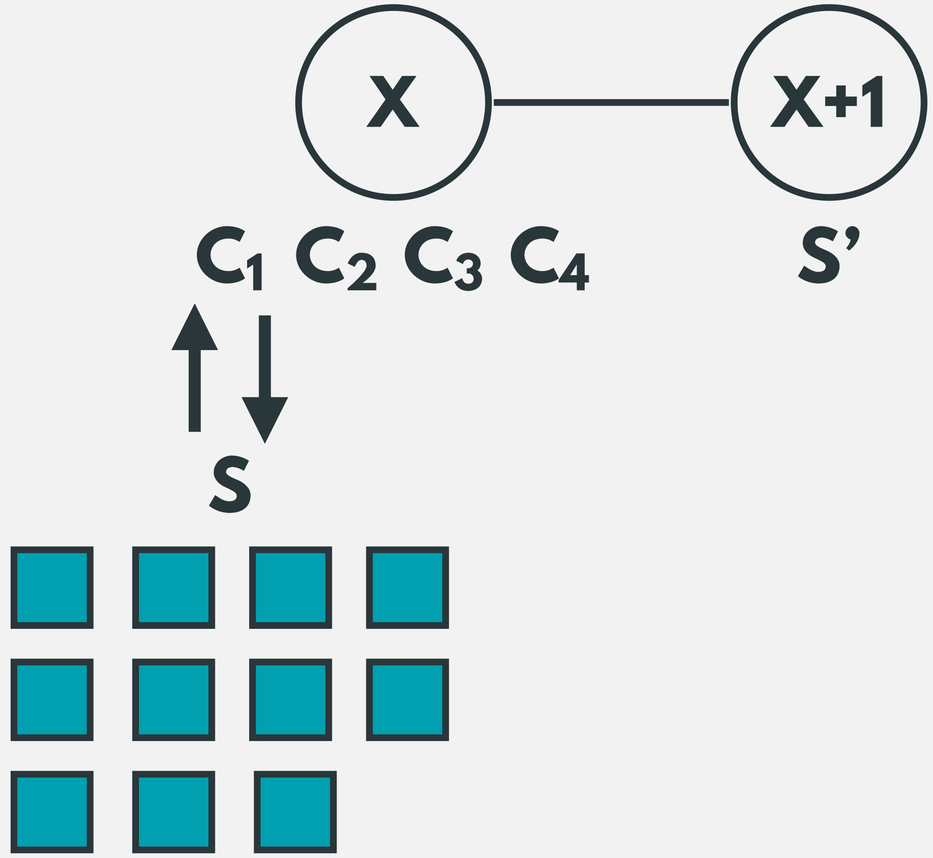
Single Schema Stores



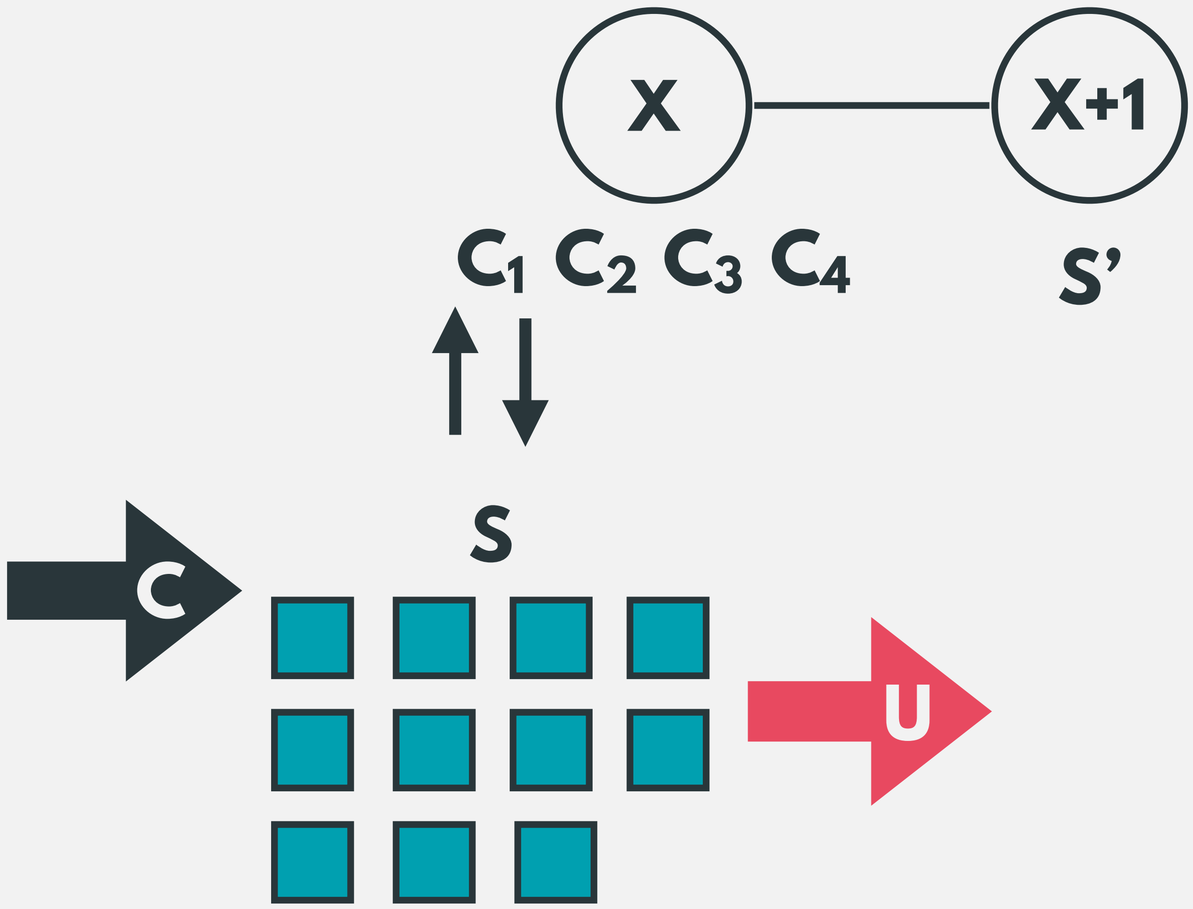
Single-Schema Migration



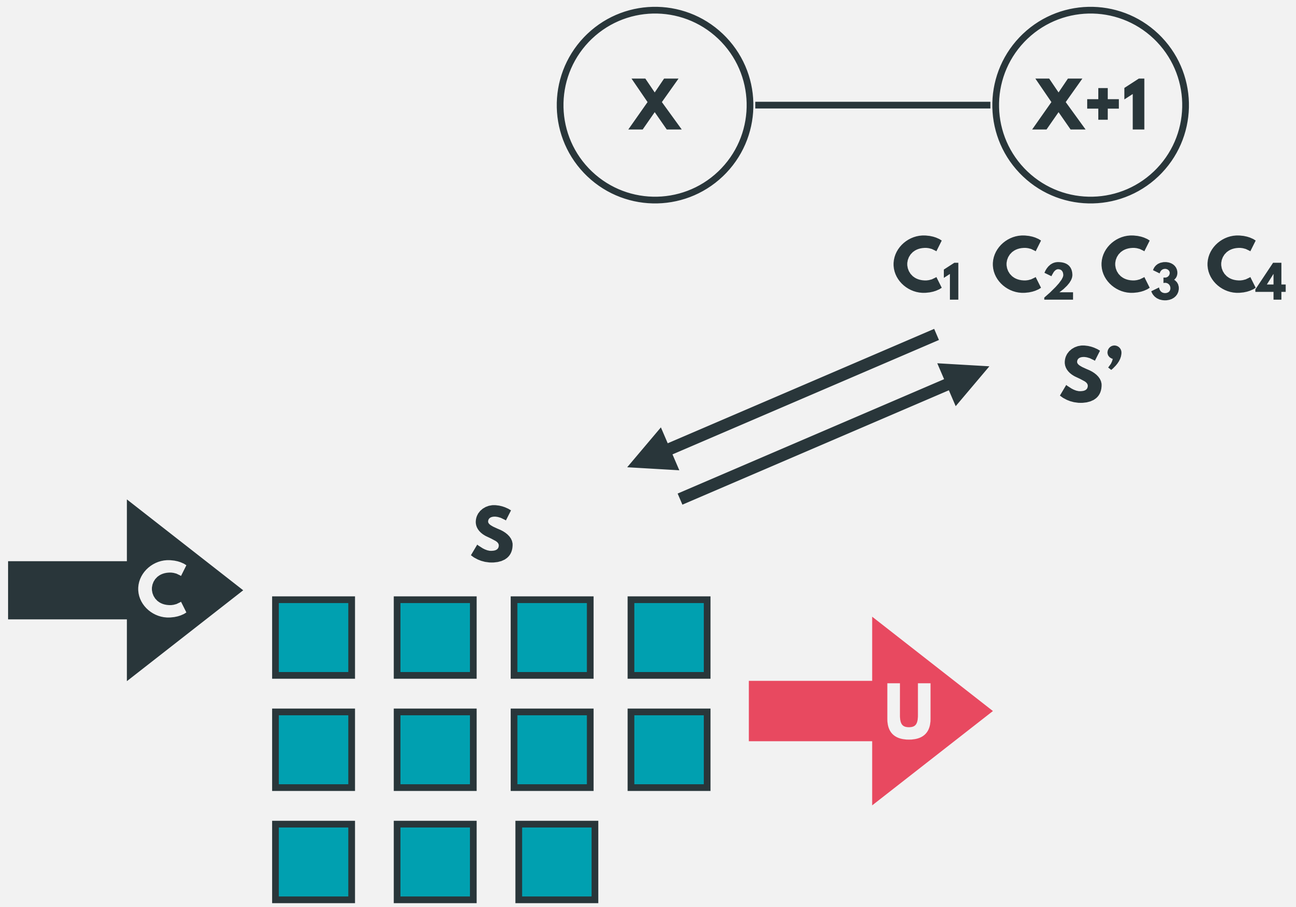
Move from X
to (incompatible) X + 1
without downtime



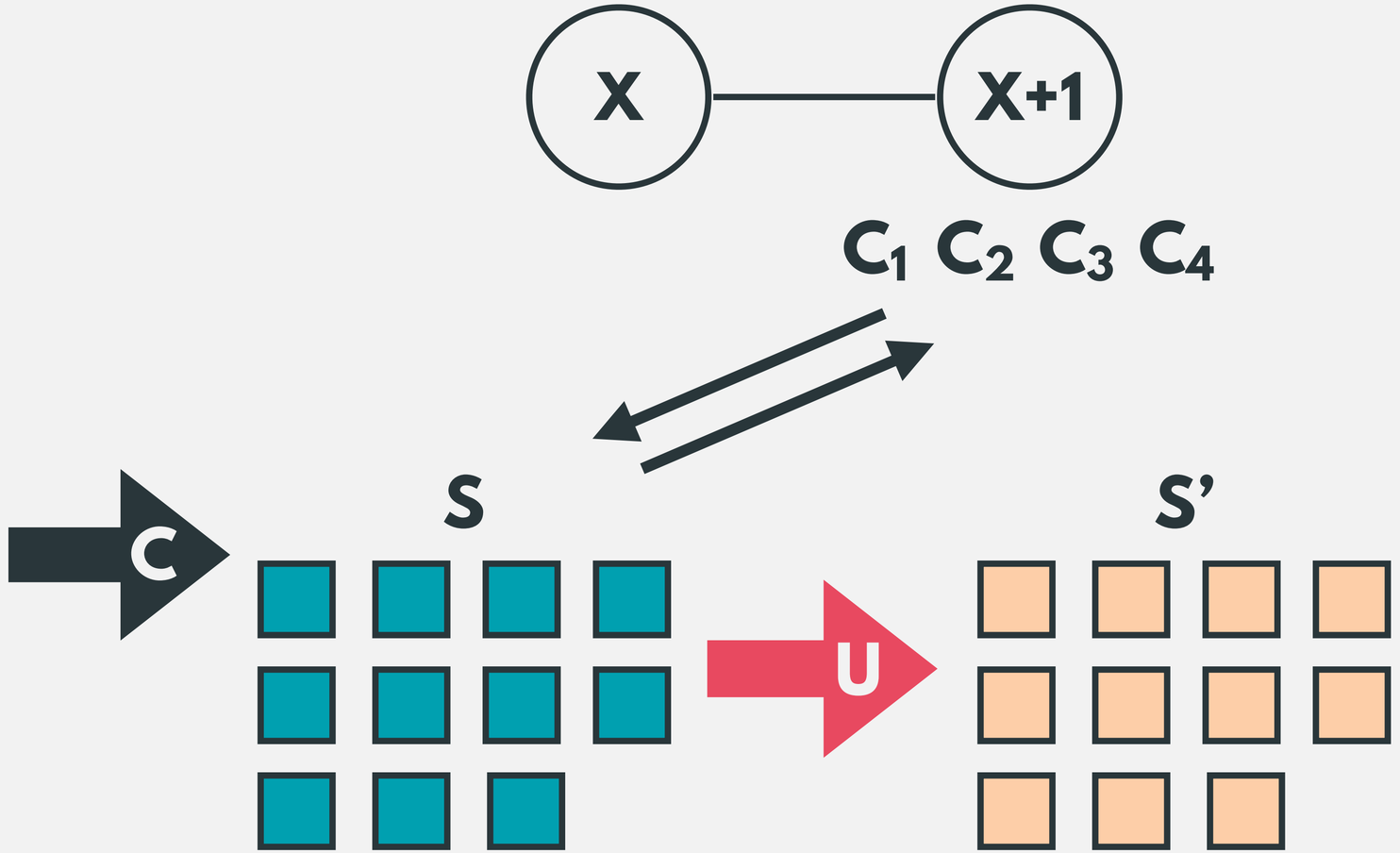
Step 1: Create and migrate temporary store S'



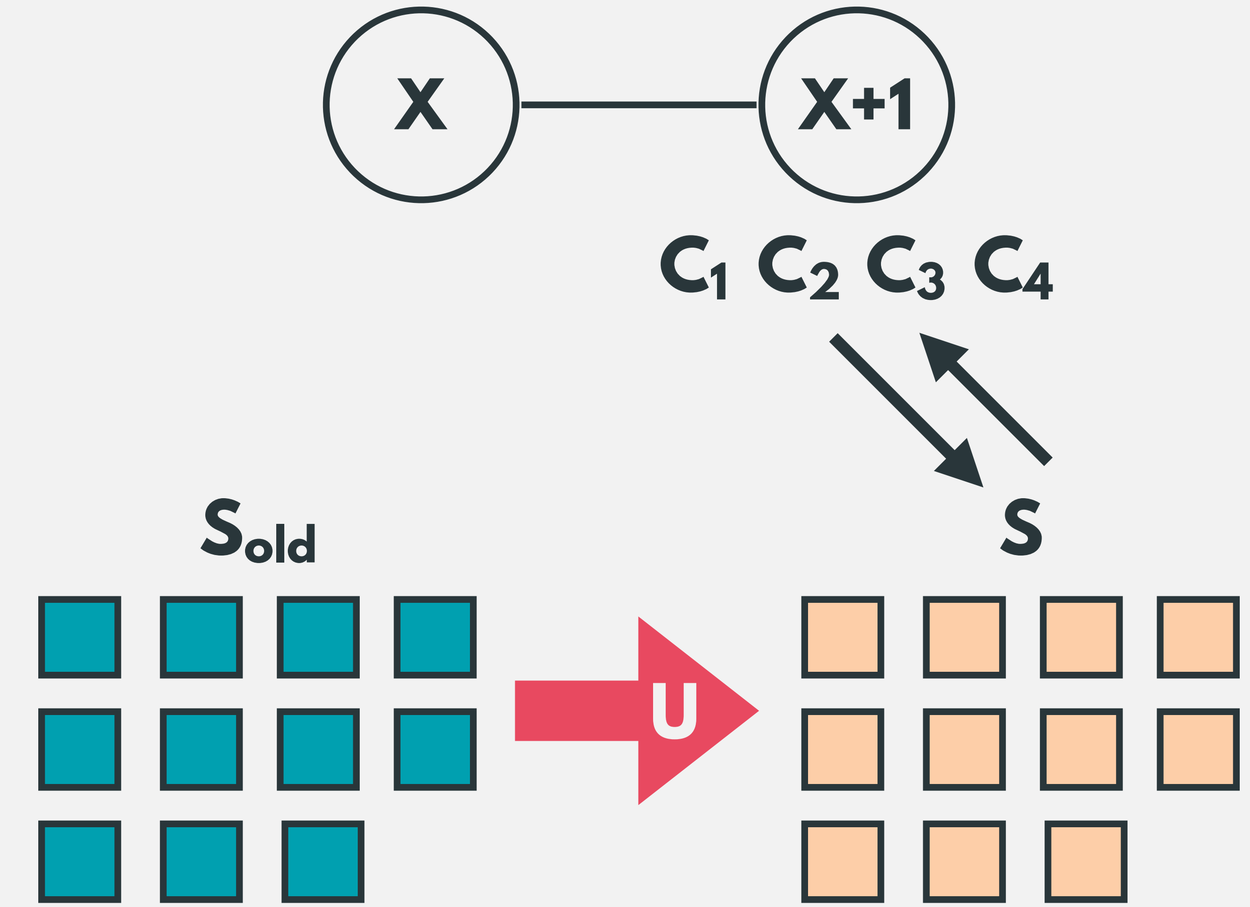
Step 2: Create a copier and an updater



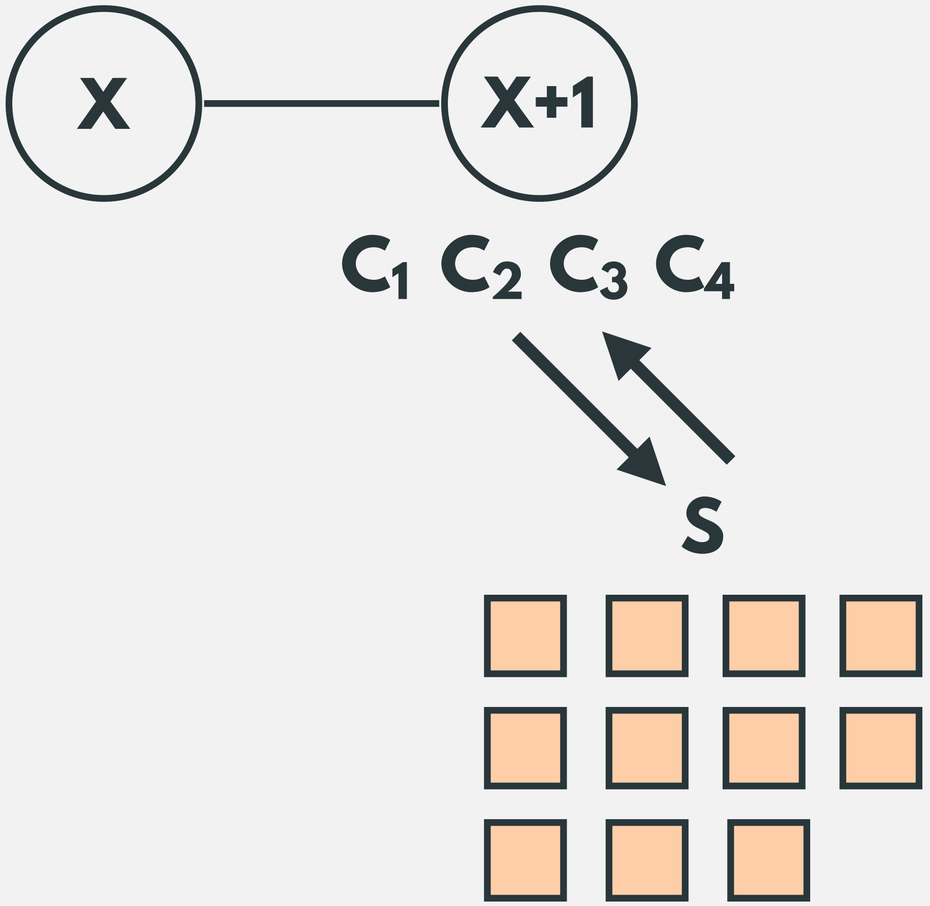
Step 3.1: Move clients over to new schema



Step 3.2: Copy data, record / apply updates



Step 4: Cutover - S' becomes S



Step 5: Drain updater, delete S_{old}

In Practice - Percona

- `pt-online-schema-change`
  - **Copier: scan/copy in timed chunks**
  - **Updater: synchronous table triggers**
  - **Cutover:** `RENAME TABLE`

In Practice - Facebook

- OSC (Online Schema Change)
  - Copier: dump/load snapshot chunks
  - Updater: triggers + changelog
  - Cutover: lock, replay, rename, unlock

In Practice - GitHub

- gh-ost
  - Copier: chunked reads/writes
  - Updater: read binlog, interleave copies
  - Cutover: 2-step blocking swap

In Practice - Wix

- Updater client-side, controlled with feature toggles
  - Phase 1: write to *S* and *S'*
  - Phase 2: read from *S'*, fallback to *S*
  - Phase 3: write only to new
- Copy phase eagerly after phase 3

Recap

- In single-schema stores:
  - Migrate clients, retaining old schema
  - Create data w/ new schema, over time
  - When in sync, cut over

1. WHAT ARE SCHEMAS?
2. SCHEMA COMPATIBILITY
3. CROSSING COMPATIBILITY GAPS
4. WRAPPING UP / TAKEAWAYS

Wrapping Up

- Concepts covered here cut across RDBMSs, APIs, document stores, data lakes, etc.
- Reason about schema evolution up-front to guide your architecture choices
- Your schemas will change. Be prepared.

A Few Takeaways

- Make your changes compatible if you can
- Compatibility informs migration strategy
- Schema-on-read/need won't save you, but it helps sometimes
- One size does not fit all

Thank You! Questions?

Contact Me

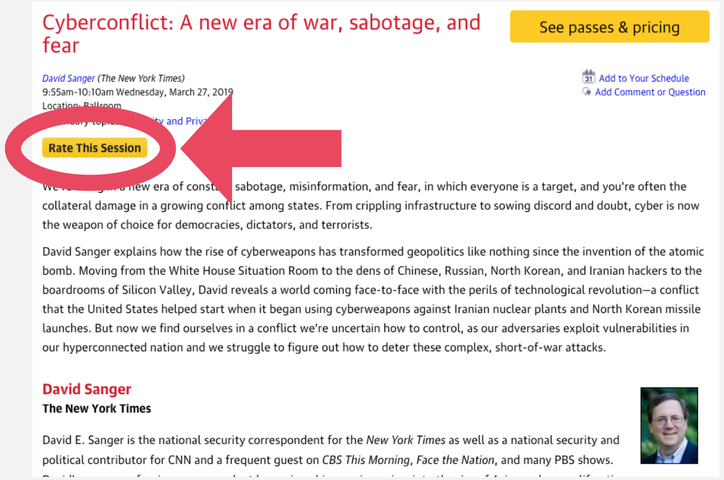
BITS ON DISK

https://www.bitsondisk.com/

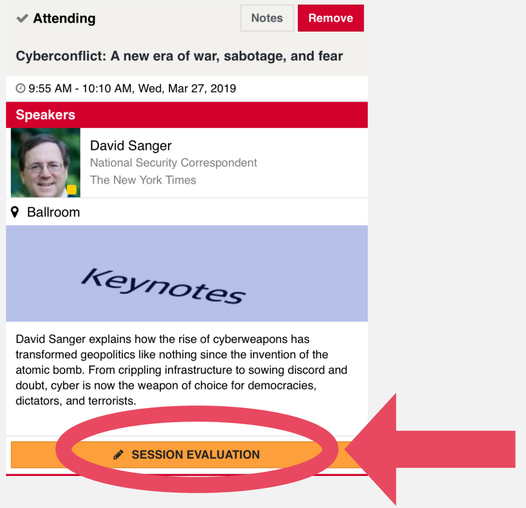
alex@bitsondisk.com

Twitter, GitHub,etc: @alexras

Rate This Talk



Session Page



O'Reilly Events App

References: https://github.com/bitsondisk/velocity-sj-2019