ToroDB internals: open source Java database middleware under the hood

Álvaro Hernández Tortosa

aht@torodb.com

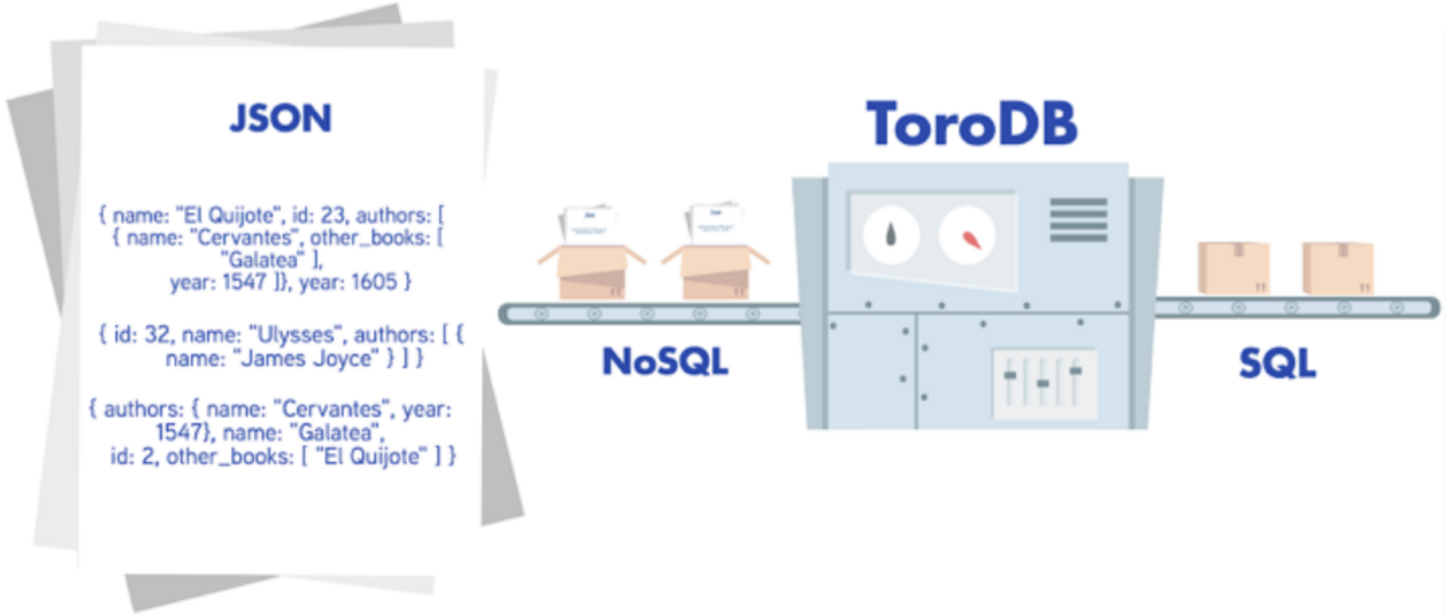
ALVARO HERNANDEZ

CEO

DBA and Java Software Developer
8Kdata and ToroDB Founder
Well-Known member of the PostgreSQL Community
World- Class Database Expert (+30 Talks in last 2 years)

@ahachete

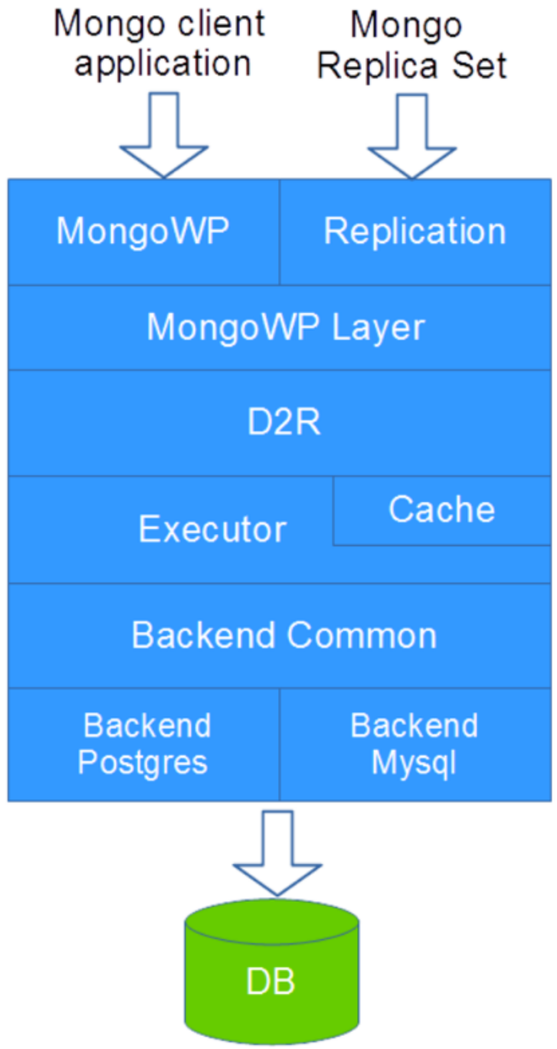
What is ToroDB?



Author	
name	year
Cervantes	1547
James Joyce	

Book			
id	name	author	year
23	El Quijote	Cervantes	1605
2	Galatea	Cervantes	
32	Ulysses	James Joyce	

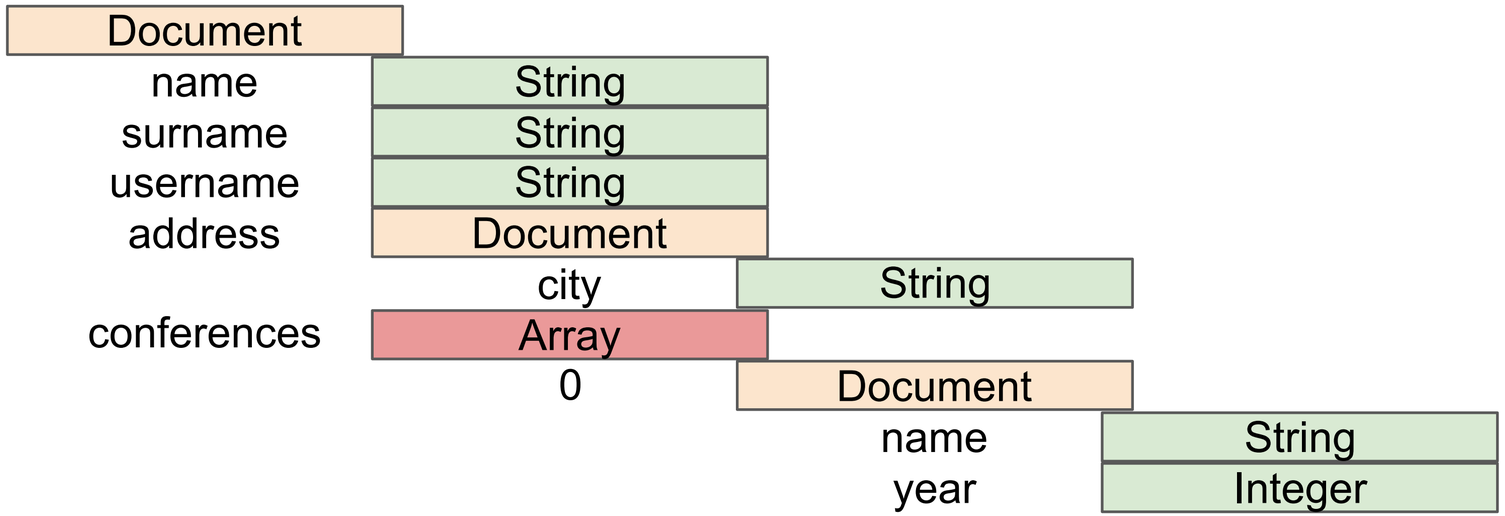
ToroDB Layered Architecture



How D2R (Document-to-Relational) transformation is performed

- JSON document is transformed into our internal document format that allows easy transformation using the Visitor pattern

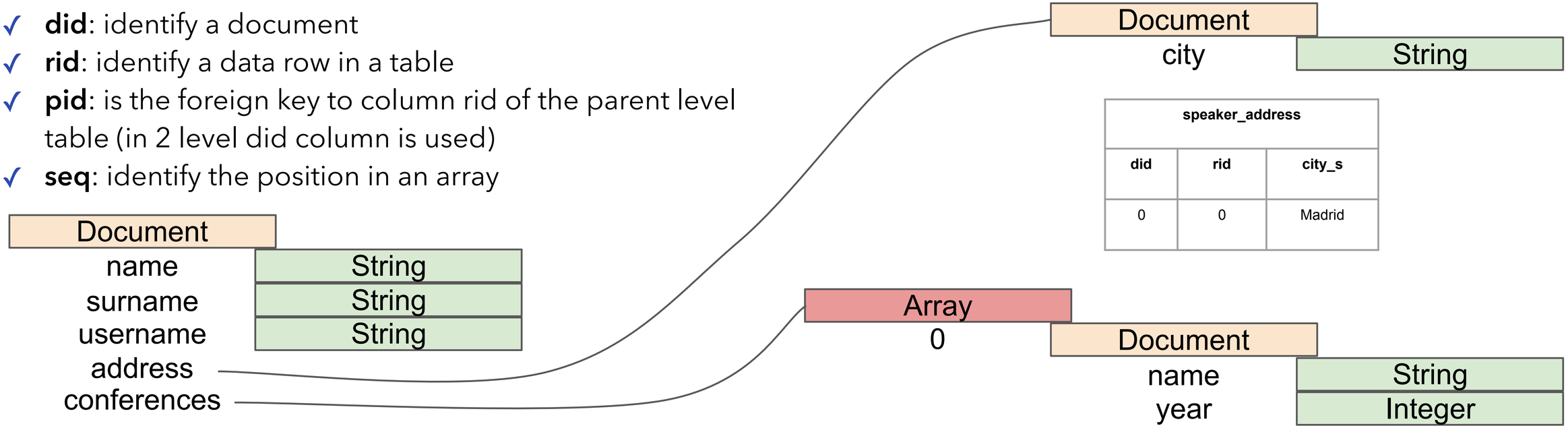
```
{
  “name”: “Álvaro”,
  “surname”: “Hernandez”,
  “username”: “aht”,
  “address”: {
    “city”: “Madrid”
  }
  “conferences”: [
  {
    “name”: “JockerConf”,
    “year”: 2017
  },
  ...
  ]
}
```



How D2R (Document-to-Relational) transformation is performed

- Internal document format is mapped to a table-friendly format that includes some meta columns (and is enabled to perform batch inserts)

- ✓ **did**: identify a document
- ✓ **rid**: identify a data row in a table
- ✓ **pid**: is the foreign key to column rid of the parent level table (in 2 level did column is used)
- ✓ **seq**: identify the position in an array



speaker_address

did	rid	city_s
0	0	Madrid

speaker

did	name_s	surname_s	username_s	address_e	conferences_e
0	Álvaro	Hernandez	aht	f	t

speaker_conferences

did	rid	seq	name_s	year_i
0	0	0	JockerConf	2017

Generated meta-information

- ✓ “doc part” represents a table mapped to the document root or to a subdocument (or subarray)
- ✓ “table ref” is the primary key of a “doc part” and you can think of it as the path inside the document

doc_part

database	collection	table_ref	identifier
world	speaker	[]	speaker
world	speaker	[address]	speaker_address
world	speaker	[conferences]	speaker_conferences

field

database	collection	table_ref	name	type	identifier
world	speaker	[]	name	string	name_s
...	...	...	...	...	...
world	speaker	[conferences]	name	string	name_s

speaker

did	name_s	surname_s	username_s	address_e	conferences_e
0	Álvaro	Hernandez	aht	f	t

speaker_conferences

did	rid	seq	name_s	year_i
0	0	0	JockerConf	2017

speaker_address

did	rid	city_s
0	0	Madrid

Rebuilding the document from the table information

- We retrieve a ResultSet for each table representing a level of the document and start building the original document from the last child levels up to the root level

1

speaker_conferences				
did	**rid**	**seq**	**name_s**	**year_i**
0	0	0	JockerConf	2017

Array
0
Document
name
year
String
Integer

2

speaker_address		
did	**rid**	**city_s**
0	0	Madrid

Document
city
String

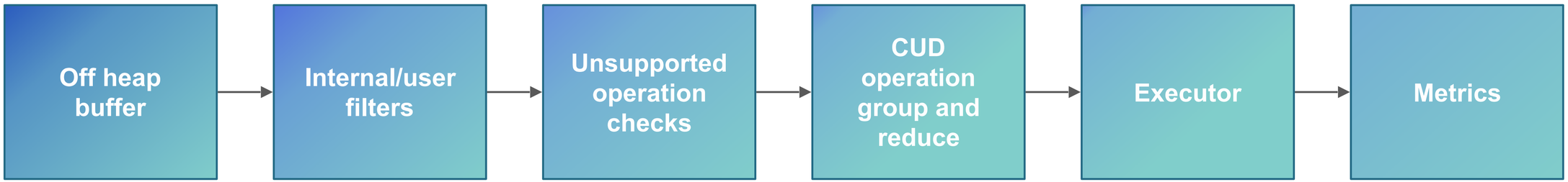
3

speaker					
did	**name_s**	**surname_s**	**username_s**	**address_e**	**conferences_e**
0	Álvaro	Hernandez	aht	f	t

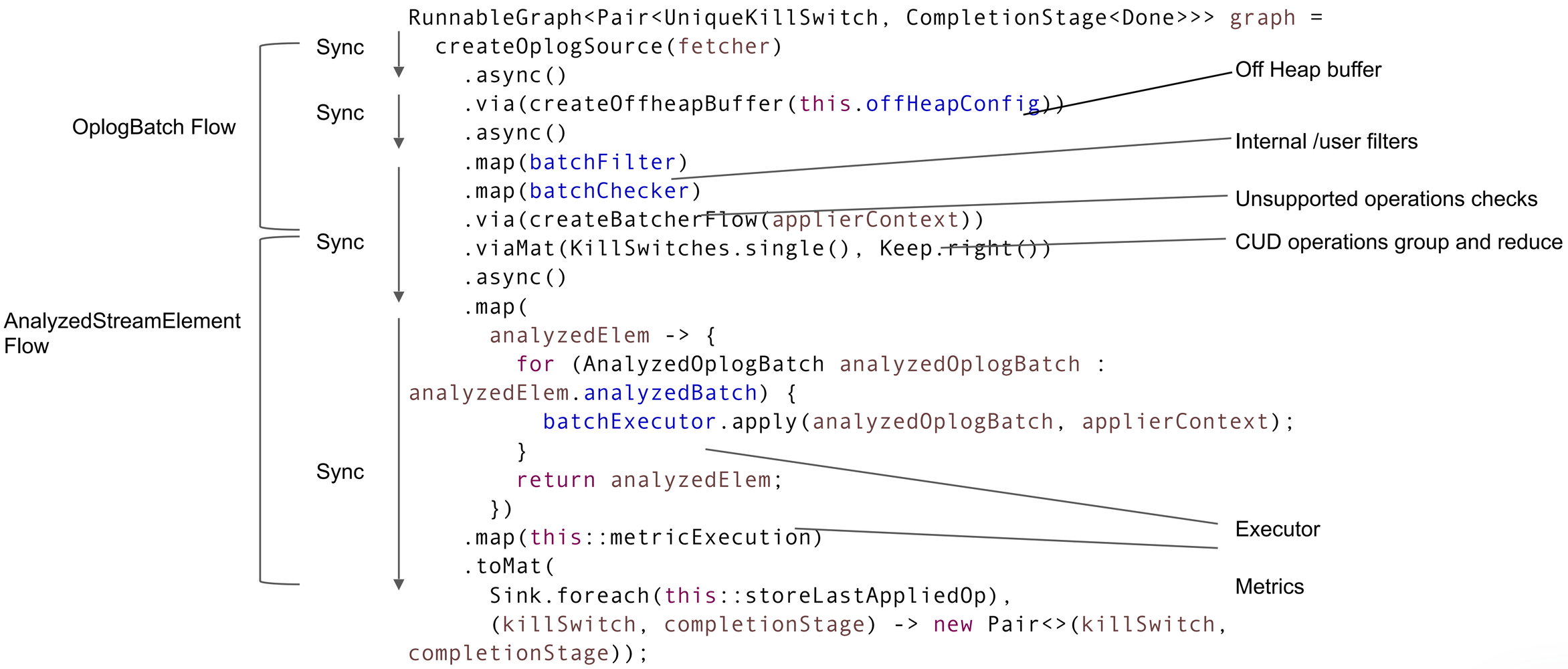
Document
name
surname
username
address
conferences
String
String
String

Replication pipeline

- ToroDB Stampede replicates MongoDB oplog operations using an Akka Streams pipeline.
- Akka has been designed with back pressure in mind that allow to execute a pipeline efficiently and with bounded resource usage.
- The actor model implemented by Akka enable an easy and efficient control over threads usage.
- And make it easy to implement complex pipeline:

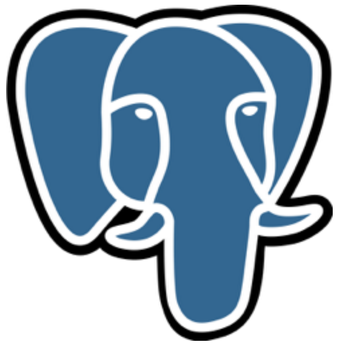


Replication pipeline



DDL changes

- Operations are streamed as part of the replication pipeline, and can mix DML and DDL statements.
- Several operations may happen concurrently.
- This works fine on PostgreSQL (transactional DDL!). But has problems on MySQL/Oracle due to **impicit commit**, violating atomicity.



DDL changes

Fix implicit DDL commit problem: execute DDL on different tx (same or separate thread)

- Backend needs to provide READ COMMITTED DDL (OK on all dbs)
- Deadlock detection needs to be implemented, specially for destructive operations like DROP

TABLE:

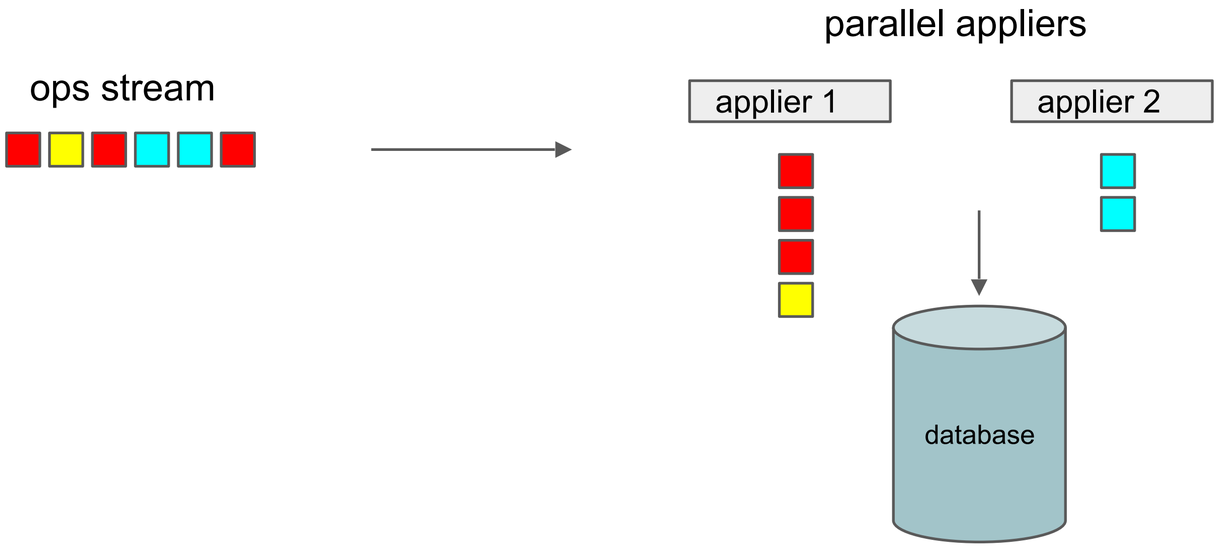
- A reader is reading the table
- It then needs to add a column.
- The DDL tx waits to acquire the exclusive lock
- The reader is blocked waiting for the DDL to proceed

DDL changes

- The solution is to use exclusive transactions for DDL: no other tx will happen in the system (this can later be relaxed).
- No DDL/DML can be mixed: if a tx sees DDL, rollbacks, sends the DDL to the SchemaManager, waits, and retries the operation
- Implementation is non-trivial as different backends require different lockings, but a superset of locks was developed (can be fine-tuned)
- An alternative solution could have been to create a lazy system where C request a drop, M modifies its metadata but not the backend and eventually, when C transaction is closed, M modifies the backend accordingly

CUD Operations

- ToroDB applies the oplog in a very smart way: it fetches a batch of oplog operations and apply them concurrently, grouping them by a hash of the database, collection and _id (primary key).
- NOTE: all updates and deletes on the oplog must query by _id. Inserted documents must always contain the _id.
- Doing this, each group can be executed in any order...



CUD Operations

Single-batch operations can also be partitioned into sub-operations. This can allow some out-of-order execution. Divide operations into sub-operations which can be:

- ✓ insert a document (I)
- ✓ fetch a document (F)
- ✓ remove a document (D)
- ✓ modify a document that is in memory (M)

Oplog operations map to::

- ✓ Create: a I if insertsAsUpserts is false or F, D, M, I if insertsAsUpserts is true
- ✓ Update: a F, D, M, I
- ✓ Delete: a D

(insertAsUpserts == false iif in initial sync state)

CUD Operations

Group sub operations by type and execute them in the following order without breaking the semantics (only if sub operations belong to the same document):

- ✓ First all F sub operations
- ✓ Then all D sub operations and M sub operations (they can be executed on any order)
- ✓ Finally all I sub operations

We can then execute all sub operations of the same type as a single db batch and produce a huge improvement.:

Even more, these batches of sub operations can be executed in parallel!

CUD Operations

Piece of cake with Akka Streams:

```
Flow.of(OplogBatch.class)
  .via(
    new BatchFlow<>(
      batchLimits.maxSize,
      batchLimits.maxPeriod,
      finishBatchPredicate,
      costFunction,
      zeroFun,
      acumFun))
  .filter(rawElem -> rawElem.rawBatch != null && !
rawElem.rawBatch.isEmpty())
  .map(
    rawElem -> {
      List<OplogOperation> rawOps = rawElem.rawBatch.getOps();
      List<AnalyzedOplogBatch> analyzed =
batchAnalyzer.apply(rawOps);
      return new AnalyzedStreamElement(rawElem, analyzed);
    });
```

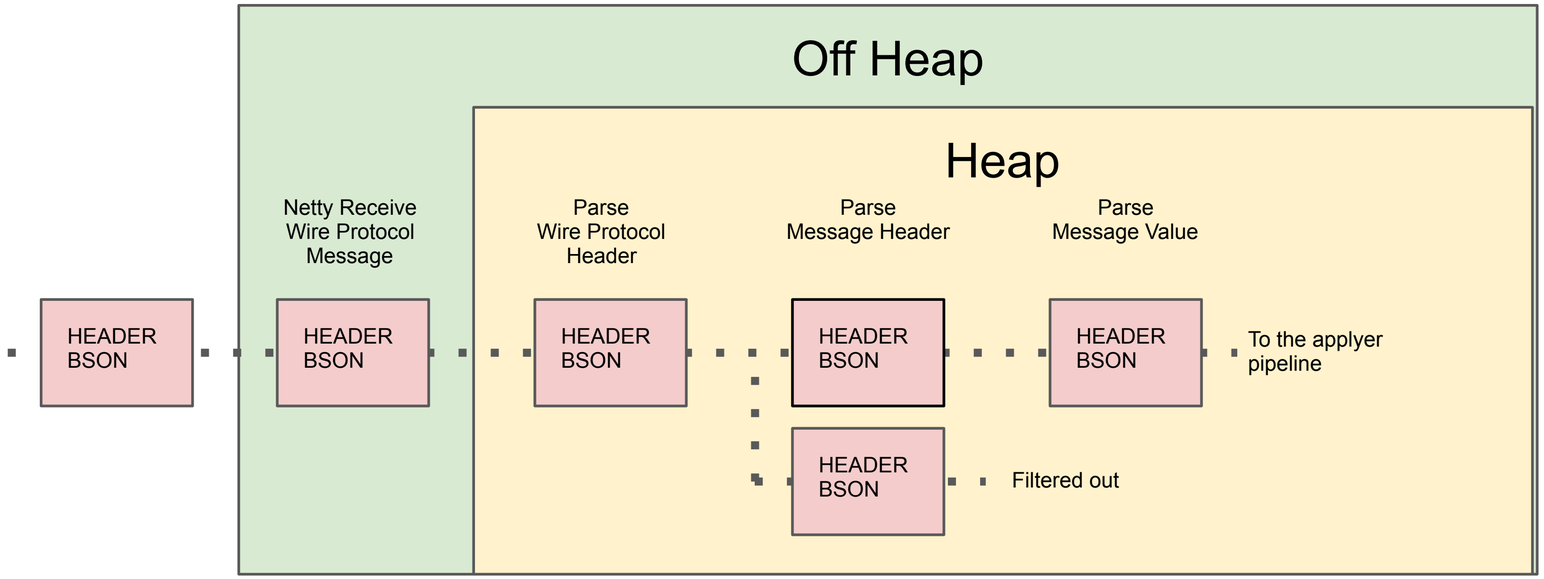
This is a Flow that does not emit until a predicate is true or the buffered elements are evaluated to value higher than a maximum or a finite duration has passed

Filter out empty batches

Extract sub operations, analyze them and divide them into independent batches (here is where the finite state machine can reduce CUD operation into a single composed operation (Insert, Update & Modify, Delete & Create, etc.)

Network I/O with Netty

- Netty is an event based NIO framework that easy the use of non-blocking sockets.
- ToroDB uses Netty heavily and also takes advantage of the off heap support. We lazily deserialize MongoDB messages on demand so that String and Binary values are brought to the Heap only if needed.



Network I/O with Netty

We just skip the reading a String until it is really needed!

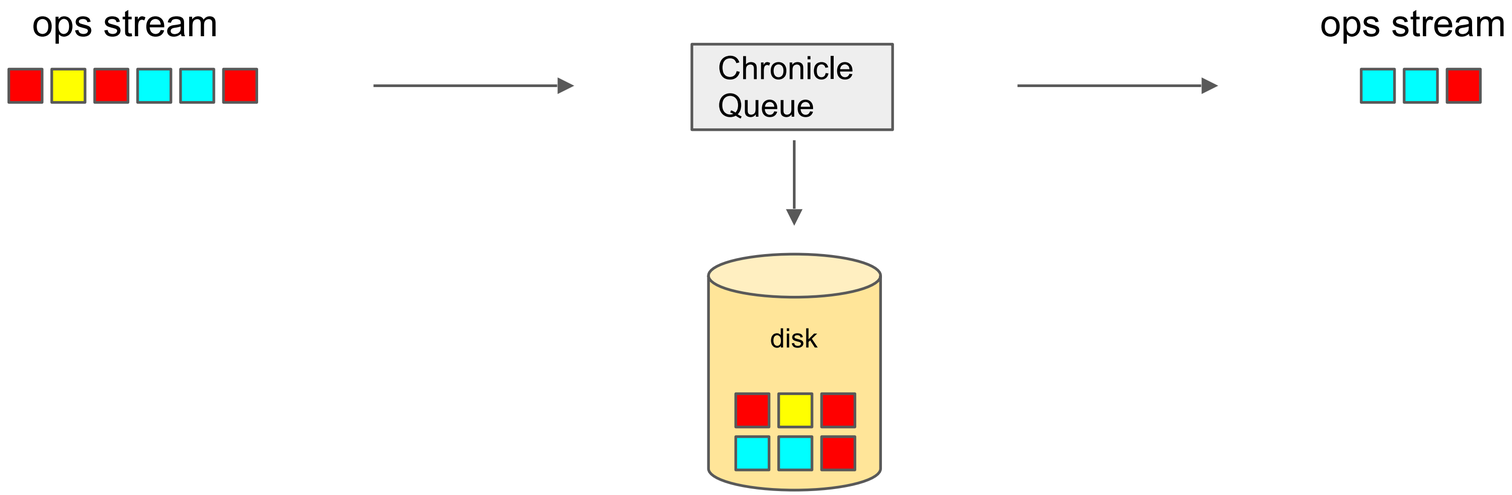
```
public class NettyBsonString extends AbstractBsonString {

  private final ByteBuf byteBuf;

  public NettyBsonString(ByteBuf byteBuf) {
    this.byteBuf = byteBuf;
  }

  @Override
  public String getValue() {
    return getString(byteBuf);
  }

  private String getString(ByteBuf byteBuf) {
    int lenght = getStringLenght(byteBuf);
    byte[] bytes = new byte[lenght];
    byteBuf.getBytes(0, bytes);
    return new String(bytes, Charsets.UTF_8);
  }

  private int getStringLenght(ByteBuf byteBuf) {
    return byteBuf.readableBytes();
  }
}
```

Off-heap buffers

- Chronicle Queue is a distributed unbounded persisted queue. We use it in replication as a persistent buffer of messages that allow ToroDB to cope with oplog burst like an initial import.

- If ToroDB operations can not process operation at the same speed as MongoDB they're stored on disk.

- We can also continue replicating from the last applied operation if for some reasons ToroDB instance crash.



Off-heap buffers

- Really easy to instantiate a queue with Chronicle Queue

```
private static ChronicleQueue
getSingleChronicleQueue(OffHeapBufferConfig offHeapConfig) {

  Path path = createPath(offHeapConfig.getPath());

  StoreFileListener sl =
getStoreFileListener(offHeapConfig.getMaxFiles());

  return SingleChronicleQueueBuilder.binary(path)
    .rollCycle(offHeapConfig.getRollCycle().asCqRollCycle())
    .storeFileListener(sl)
    .build();
}
```

Off-heap buffers

- We created a small project to integrate Chronicle Queue with Akka and put it on our replication pipeline

```
ChronicleQueue scq =
getSingleChronicleQueue(offHeapConfig);

Flow.of(OplogBatch.class)
  .via(
    new ChronicleQueueStreamFactory<>()
  .withQueue(scq)
  .autoManaged()
  .createBuffer(new OplogBatchMarshaller()))
  .map(Excerpt::getElement);
```

Resource consumption

Documents are schema-less...

...a better describing word is “schema-attached”: the schema is shipped with the data in each document!

...this imply a lot of String instances

...that imply a lot of String comparison byte per byte since we use lots of HashMaps

Can we optimize it?

Resource consumption

Yes!

- Retrieve the string from an intern like cache (if string cacheable)

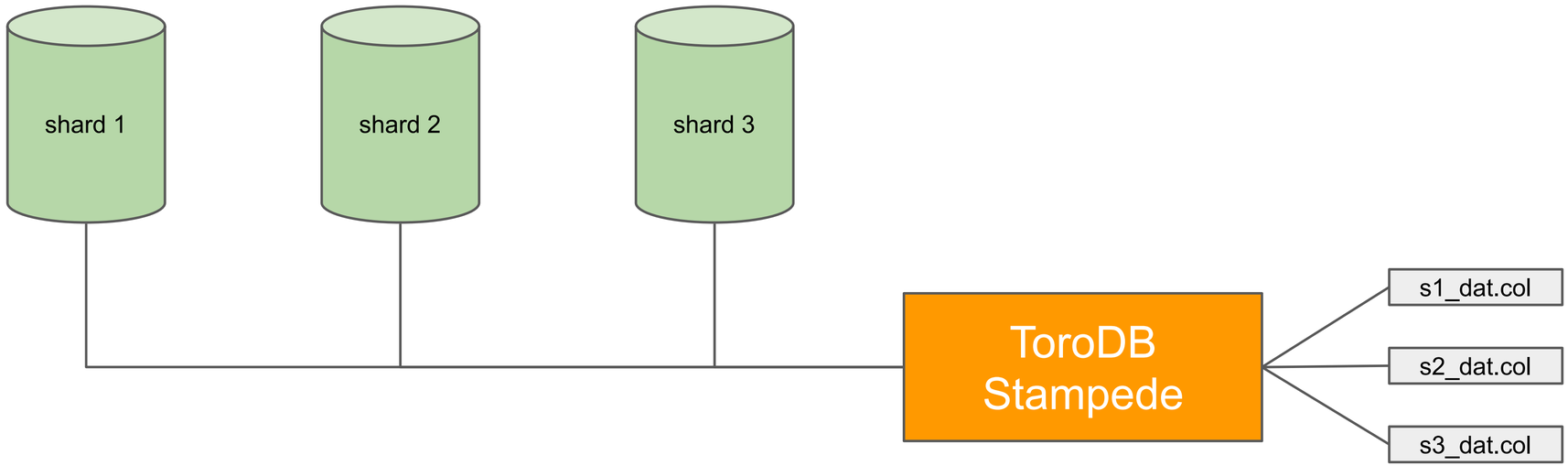
```
String result = stringPool.fromPool(likelyCacheable, buffer.readSlice(pos));
```

- We use the Guava Interner class for efficiency since it let the Garbage Collector do its job on unreaded keys

```
public class WeakMapStringPool extends StringPool {

  private final Interner<String> interner;

  public WeakMapStringPool(StringPoolPolicy heuristic) {
    super(heuristic);
    this.interner = Interners.newWeakInterner();
  }

  @Override
  protected String retrieveFromPool(ByteBuf stringBuf) {
    return interner.intern(getString(stringBuf));
  }

}
```

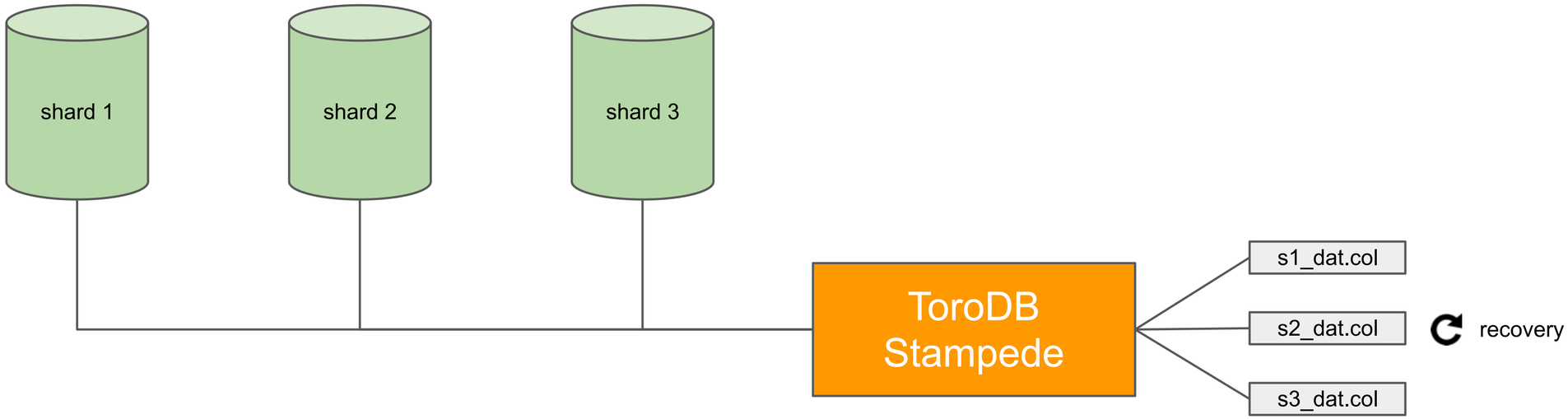
Sharding

- People that requires advanced analytics usually have a lot of data.
- To efficiently manage that amount of data on MongoDB is by using sharding.
- ToroDB Stampede can replicate from several shards!



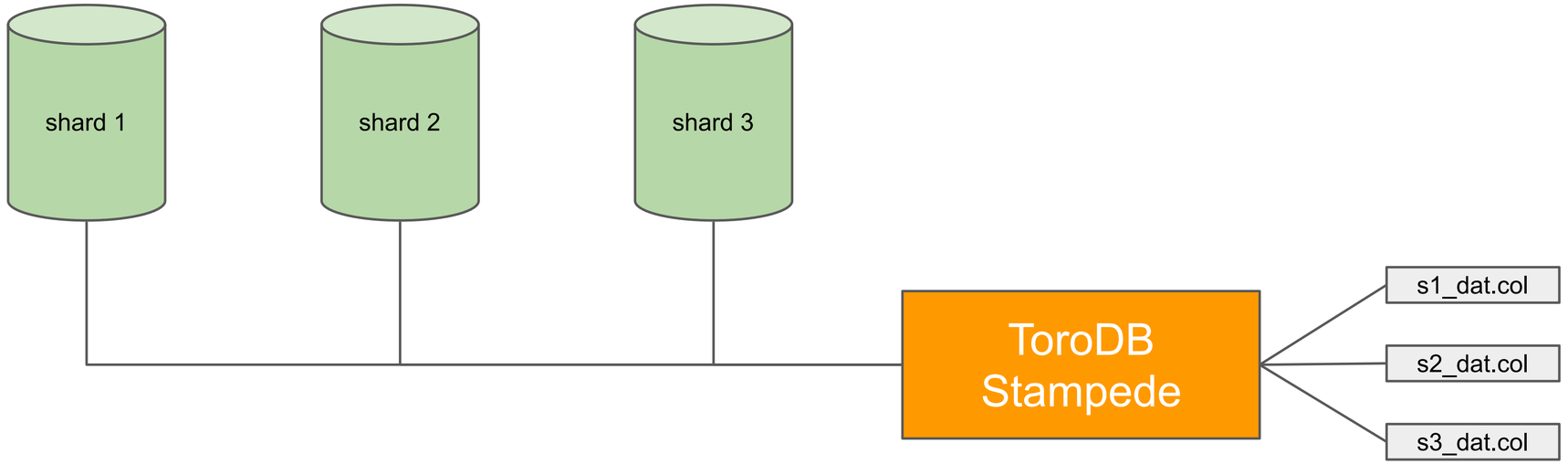
Sharding - issues to deal with

- Different replication state per shard: to replicate from several shards, each one with their own oplog, we can fall to recovery on one shard but still be *online* on the others.



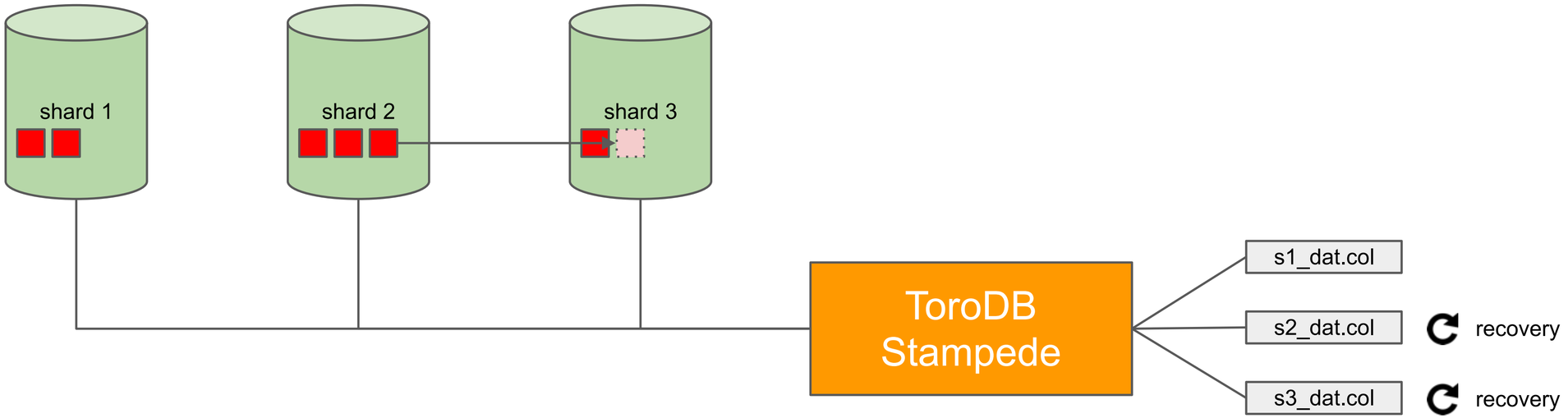
Sharding - issues to deal with

- Drop database on recovery: before implementing sharding, the first thing done by the recovery phase is to drop the whole database. It has sense when replicating from a single shard (or if we share the replication state on all shards) but it is incorrect if we decide to allow different states per shard.



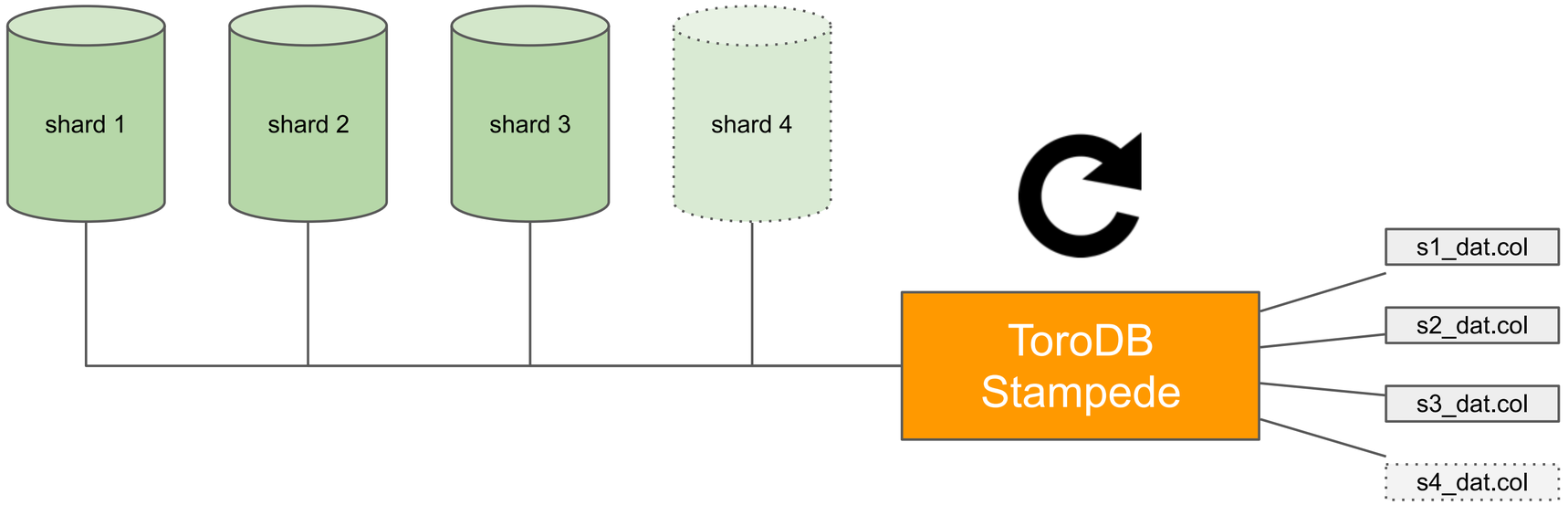
Sharding - issues to deal with

- *Migrate* operations (aka when moving a document to another shard): These operations are stored on at least two different shards and the stored operation is different on each one. For example, when a document is moved from a shard to another, there will be a delete operation on the shard that stored the document and a insert operation on the shard that will be store the document.



Sharding - issues to deal with

- Changes on shards: What happen when a shard is added or removed. It seems difficult to think in advance which implications will this would have.



Sharding - issues to deal with

- Operations of different shards executed out of order: as there is no common clock, the timestamp of two operations on different oplogs cannot be compared.
  - ✓ This is not bad on CUD (Create, Update, Delete) operations that are not migrations, as they affect a single document which must be on a single shard, so different operations on the same document are always comparable by timestamp.

- We need to find synchronization points. Two possible options:
  - ✓ Documents that migrate from one shard to another (identified by _id, uniquely global) happen at the same time.
  - ✓ Some operations like index creations affect all shards

Sharding - bundles

- A single Guice injector is OK to create plain Guava services, but it cannot have several instances of the same module
- Guice injectors only provide singleton instances per injector. There are 2 possible solutions:
  - ✓ Use scopes. It is more complicated and error prone: if there's a bug, you will silently get the singleton and not the other instance.
  - ✓ Use several injectors: easier and probably less performant, but only used at startup time (so it's OK)

Sharding - bundles. Problems encountered

- It is very easy to break the abstraction. Each class of each module can request the injection of any object on the injector, including classes from other modules . Seems fine, and Guice can resolve complex dependency graphs, but it is very difficult to reason about that.
- Example: to improve startup method, backend service is started to check for consistency. Only if it is all OK starts other modules. But backend service relied on other layers, like torod, so it was not easy task.
- Dependencies were implicit. Not only when talking about dependencies between modules, but where a class was used. As it was very easy inject even classes, new dependencies could be added without documentation or reasoning whether they should have rather been be wrapped in some way (like using a delegator or an adaptor).

Sharding - bundles

Guice PrivateModule and Bundles are the key that enabled ToroDB to support sharding:

- PrivateModule force developer to be aware of what to expose as a dependency. So that a we can have a fine grain control over injected instances.

- Bundles are quite simple: They are a Guava service that instantiates something that is returned on the method getExternalInterface. There are some abstract bundles designed to be compliant with DRY principle that implement some common things like to wait until all dependencies are started before starting the bundle.

```
public interface Bundle<ExtIntT> extends
TorodbService,
  SupervisedService, AutoCloseable {

public abstract Collection<Service>
getDependencies();
public abstract ExtIntT getExternalInterface();

public default <O> Bundle<O> map(Function<ExtIntT
O>
    transformationFunction) {
  return new TransformationBundle<>(this,
    transformationFunction);
}

@Override
public default void close() {
  if (isRunning()) {
    this.stopAsync();
    this.awaitTerminated();
  }
}

}
```

Integration Test JUnit5 and Docker

Integration Tests are really important for ToroDB. In particular we have to test that:

- ✓ Backend code is working as expected
- ✓ Replication corner cases are well handled

We need a way to test all of that with real database instances?!

Integration Test JUnit5 and Docker

Docker integration into ***Java*** for fully automated unit and integration testing scenarios, including clusters of servers, failover, etc. ...and JUnit 5 taken to the extreme!

A Postgres docker container is spawned by our open source testing-tools project in each test iteration.

https://github.com/torodb/testing-tools

Contributions are welcome!

```
@RunWith(JUnitPlatform.class)
@RequiresPostgres(version = LATEST)
public class PostgresTest {

  @Test
  public void simpleTest(PostgresService service) {
    assertEquals(Service.State.NEW, service.state(), "The injected
service must start on "
      + Service.State.NEW + " state");

    service.startAsync();
    service.awaitRunning();

    testConnection(service);
  }

  private void testConnection(PostgresService service) {
    assert service.isRunning();

    service.getDslContext().createTable("test")
      .column("aint", SQLDataType.INTEGER)
      .execute();

    for (int i = 0; i < 10; i++) {
      service.getDslContext().insertInto(DSL.table("test"))
        .columns(DSL.field("aint"))
        .values(i)
        .execute();
    }

    Record1<Integer> countRecord =
service.getDslContext().selectCount()
      .from(DSL.table("test"))
      .fetchAny();
    Assertions.assertNotNull(countRecord);
    Assertions.assertEquals(10, countRecord.value1().intValue());
  }
}
```

Integration Test JUnit5 and Docker

```
public class PostgreSqlStructureIT extends
AbstractStructureIntegrationSuite {
  private static PostgresService postgresService;

  @BeforeAll
  public static void beforeAll() {
    postgresService =
PostgresService.defaultService(EnumVersion.LATEST);
    postgresService.startAsync();
    postgresService.awaitRunning();
  }

  @AfterAll
  public static void afterAll() {
    if (postgresService != null &&
postgresService.isRunning()) {
      postgresService.stopAsync();
      postgresService.awaitTerminated();
    }
  }

  @Override
  protected BackendTestContextFactory
getBackendTestContextFactory() {
    return new
PostgreSqlTestContextFactory(postgresService);
  }
}
```

You can also use it without annotation if your test needs require complex configurations

... like you need to spawn a container for each test suite and need a reference before each test

Integration Test JUnit5 and Docker

JUnit 5 also rocks!

Testing replication corner cases made easy with @TestFactory annotation

```
@RunWith(JUnitPlatform.class)
public abstract class AbstractOplogApplierTest {

  protected String getName(OplogApplierTest oplogTest) {
    return oplogTest.getTestName().orElse("unknown");
  }

  protected abstract Supplier<ClosableContext>
contextSupplier();

  @TestFactory
  protected Stream<DynamicTest> createJsonTests() throws
Exception {
    return oplogTestSupplier()
      .map(oplogTest -> OplogApplierTestFactory.oplogTest(
        getName(oplogTest),
        oplogTest,
        contextSupplier())
      );
  }

  protected Stream<OplogApplierTest> oplogTestSupplier() {
    return new DefaultTestsStreamer().get();
  }
}
```

Questions?

www.torodb.com/stampede

info@torodb.com

@nosqlonsql

github.com/torodb