making Datomic

@stuarthalloway

making things

Relevance

Clojure

Datomic

ClojureScript

core.async

Pedestal

Cognitect

Datomic

functional

well-known benefits

- referential transparency, composition, testing
- concurrency friendly

less obvious benefits

- time model
- topology

agility

flexibility

- universal schema
- programming with data
- dynamic

power

- indexes
- logic
- ACID

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

functional, lazy peers

pluggable storage protocol

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

database is a lazily
realized value, available
to all peers equally

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

query databases,
not connections

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

join across databases,
systems, in-memory collections

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");


Database db = conn.db();


Set results = q(..., db);


Set crossDbResults = q(..., db1, db2);


Entity e = db.entity(42);
```

lazy, associative
navigable value

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

information in
generic data structures

ACID, serialized, time aware

contains old db, new db, change

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);  ← time travel

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

one possible future

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history(); ← all history, overlapped

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

monitor all change
from any peer

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

review any
time range

complexities mitigated

lost data

log analysis

ORM

inheritance

structural rigidity

model caching

app caching

managing time

test setup

defensive copying

join tables

relationship direction

logic duplication

imperative code

read transactions

denormalization

eventual consistency

DAOs

DTOs

objects

strings

injection attacks

data duplication

isolation levels

how we did it

simplicity

power

focus

courage

pragmatism

patience

complexity: tables

“People can belong to multiple clubs”

join table

person table

club table

id key in person table

person key in join table

club key in join table

id key in club table

simplicity: datoms

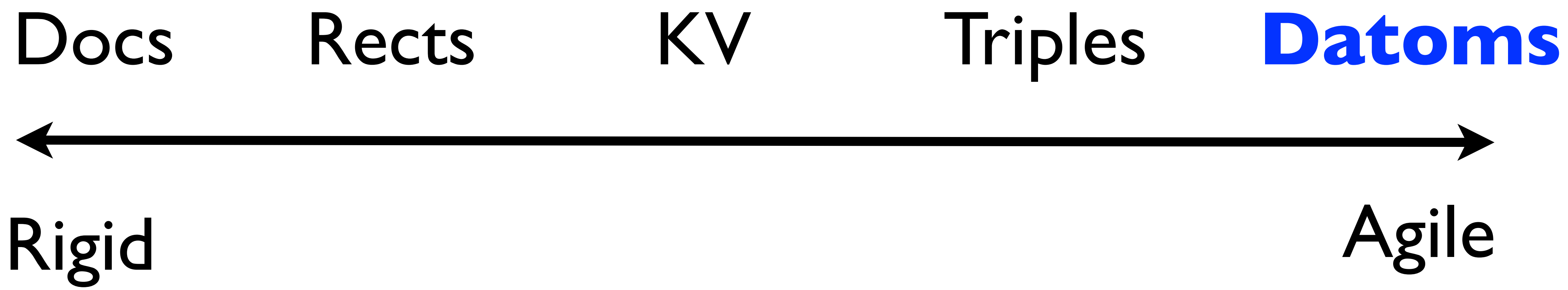
“People can belong to multiple clubs”

```
[?person :club ?club]
```

power

capabilities dictated at storage time	**capabilities chosen at query time**
row store	EAVT, partitions
kv store	AVET, unique keys
column store	AEVT
graph database	VAET
document database	all the above

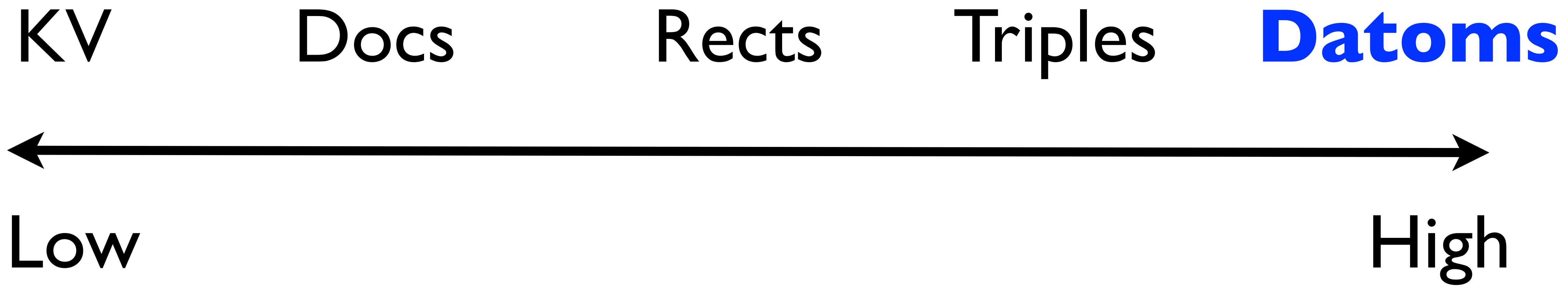
agility



Docs Rects KV Triples **Datoms**

Rigid Agile

leverage



KV Docs Rects Triples **Datoms**

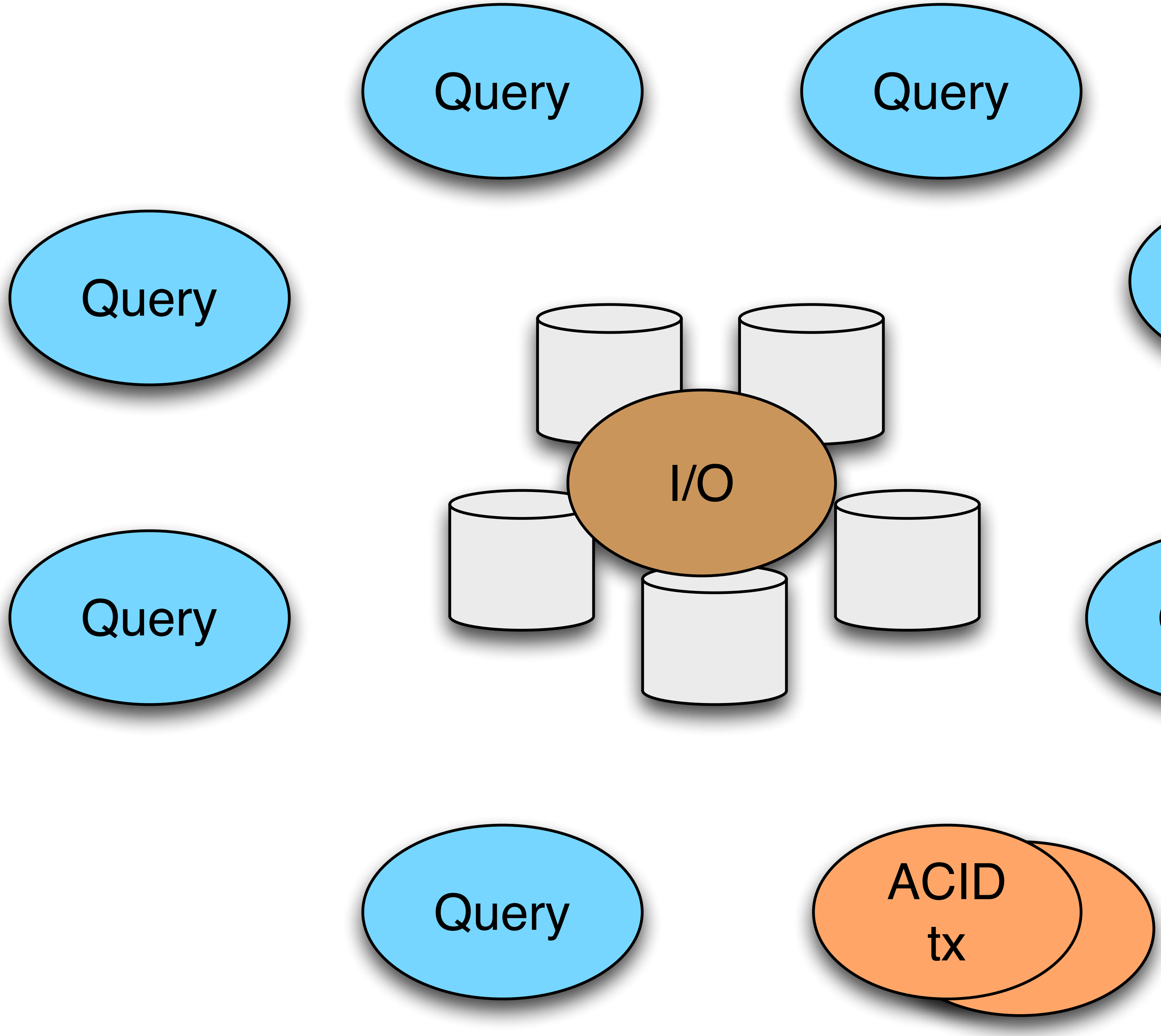
Low High

focus

"Because of the elasticity of Datomic, we were able to reduce our hosting fees by a factor of 10 when we moved off of [a popular NoSQL]."

<redacted>

courage: elastic read, ACID write



pragmatism

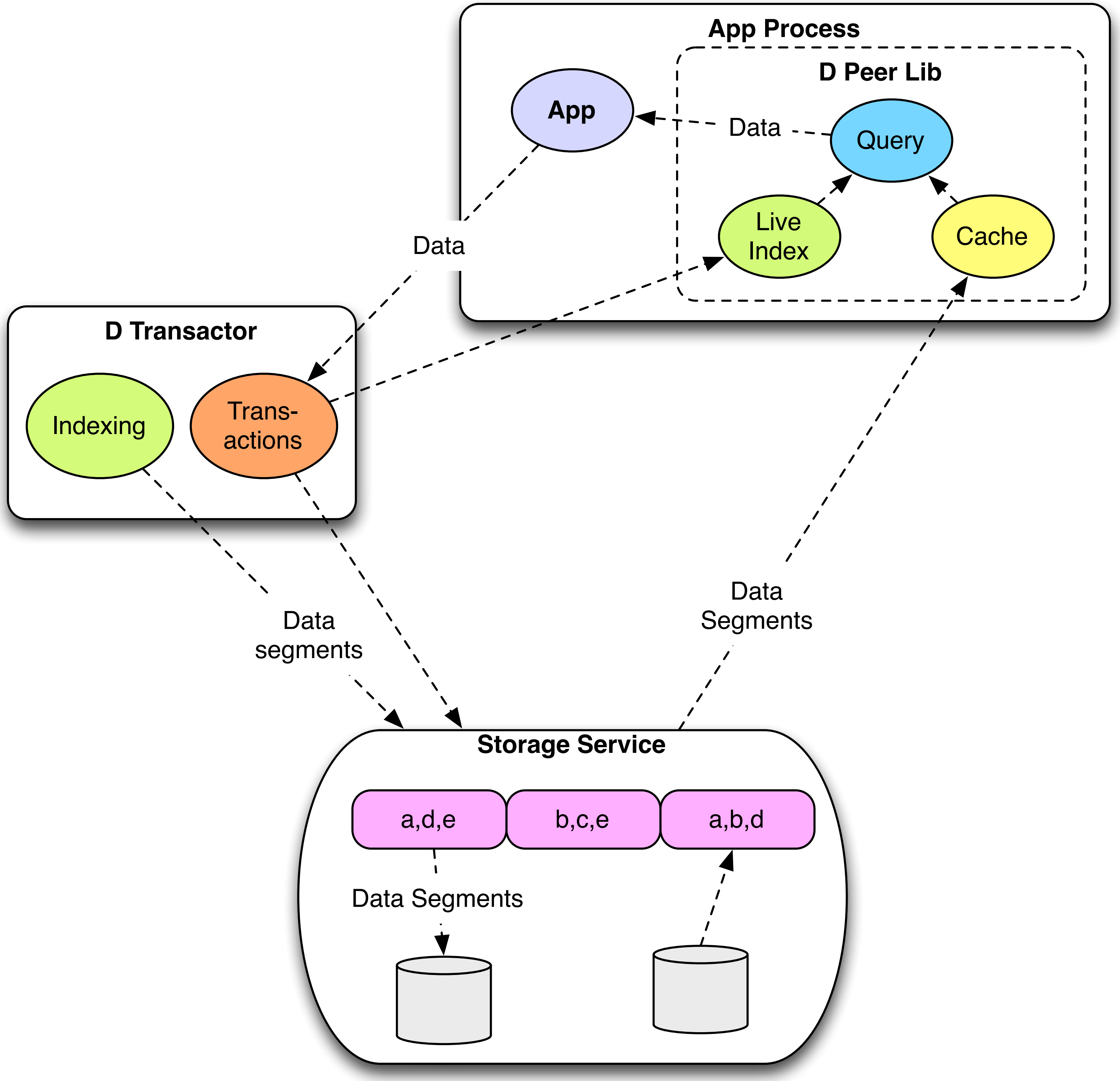
academic (but unfashionable) ideas

- datalog
- persistent data structures
- lisp
- communicating sequential processes

but not these ideas:

- (list omitted for troll protection)

patience


App Process
D Peer Lib
App
Data
Query
Live Index
Cache
Data
D Transactor
Indexing
Trans-actions
Data segments
Data Segments
Storage Service
a,d,e
b,c,e
a,b,d
Data Segments

the problem with example-based testing

example-based-tests (EBT)

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

setup

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

inputs

EBT

execution

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

output

EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

validation

weaknesses of EBT

severely limited coverage

fragility

bespoke

deconstructing EBT

Inputs

Execution

Outputs

Validation

generative testing

Model

Outputs

Execution

Inputs

Validation

loose coupling FTW

decouple	**benefits**
model	improve design generate load
inputs	increase comprehensiveness by running longer
execution	test different layers with same code only part that must change with your app
outputs	expert analysis persist for future study
validation	test generic *properties* run against prod data
all	*functional programming* *feedback loops in test development*

thank you!

cognitect

@stuarthalloway