Datomic for the 96 Percent

cognitect

@stuarthalloway

challenges with SQL

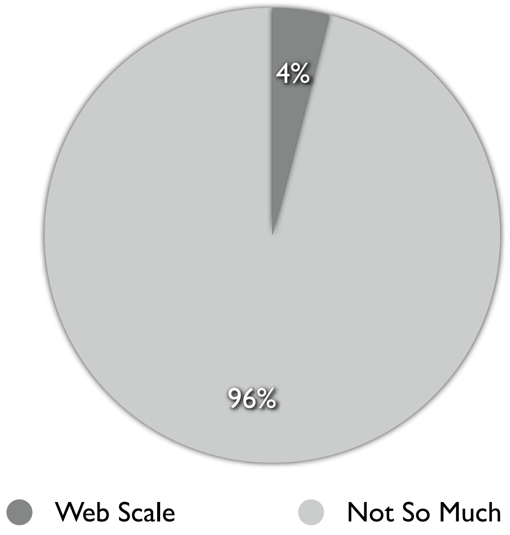
write volumes

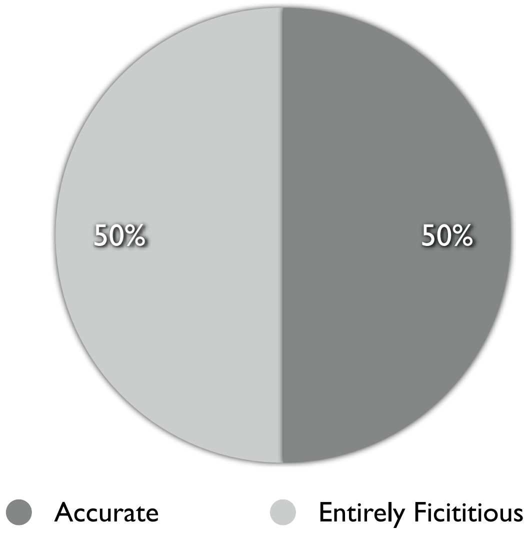
read volumes

deployment rigidity

model rigidity

update-in-place

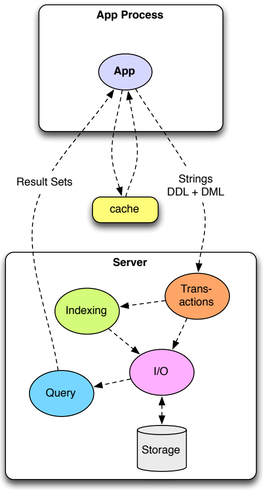




eventual consistency

http://en.wikipedia.org/wiki/Everclear_(alcohol)

rigid deployment



rigid models

“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

the laws

memory is expensive

storage is expensive

machines are precious

resources are dedicated

update in place = transience

characteristic	**transient structure**
sharing	difficult
distribution	difficult
concurrent access	difficult
access pattern	eager
caching	difficult
examples	Java and .NET collections relational databases NoSQL databases

answering the challenge

functional

radical deployment, e.g. local logic query, laziness

isolated writes, serializ**ed** ACID

time-aware

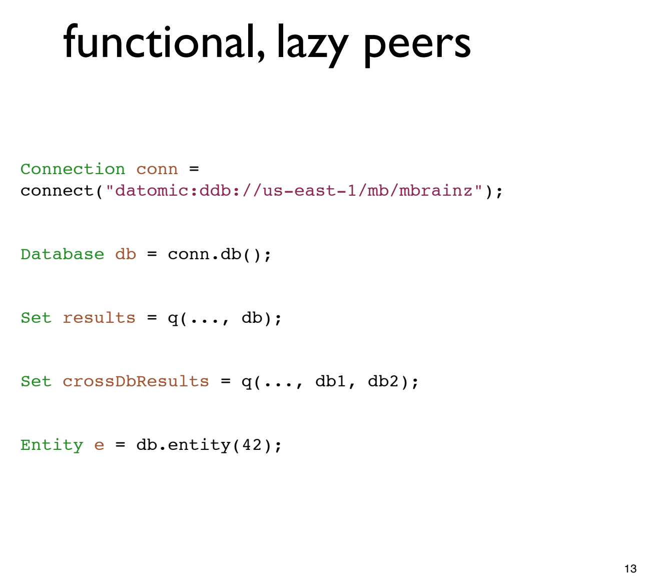
elastic read scaling

flexible, universal attribute schema

programmable

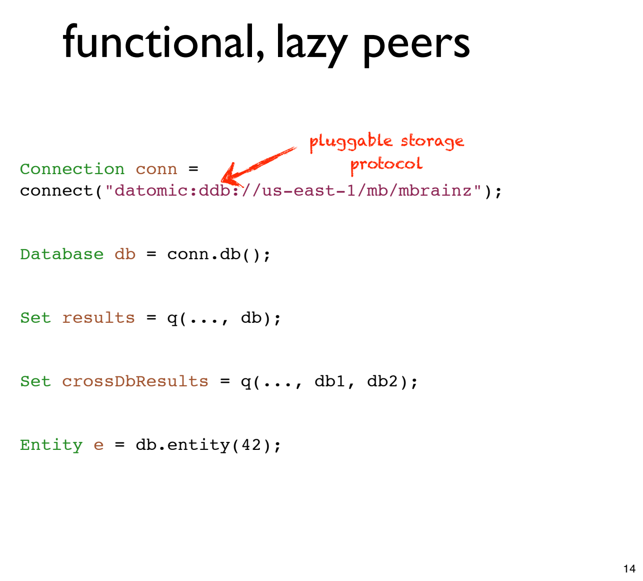
transience vs. persistence

characteristic	**transient**	**persistent**
sharing	difficult	trivial
distribution	difficult	easy
concurrent access	difficult	trivial
access pattern	eager	eager or lazy
caching	difficult	easy
examples	Java, .NET collections relational databases NoSQL databases	Clojure, F# collections Datomic database



functional, lazy peers

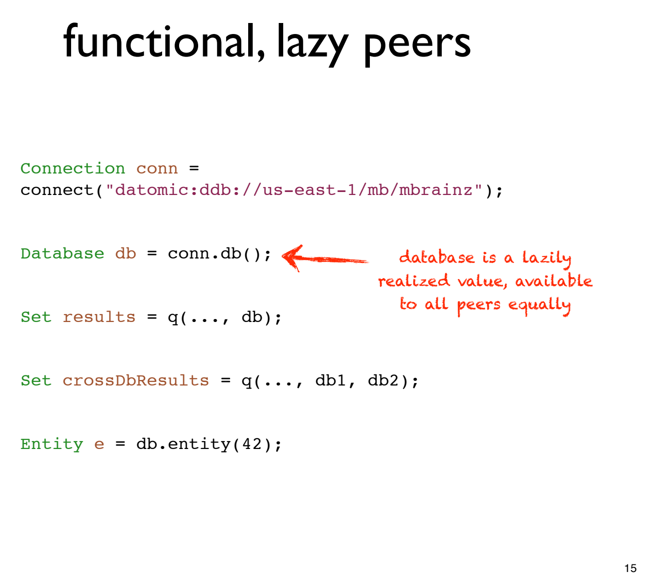
```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

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

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

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



functional, lazy peers

```
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);
```

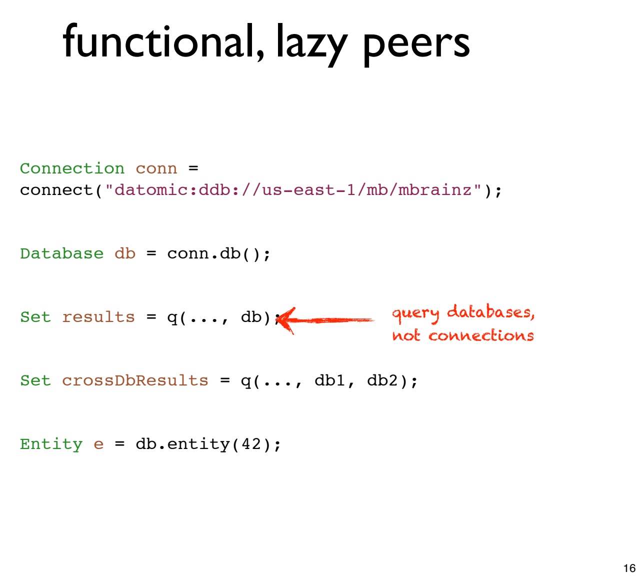
pluggable storage
protocol



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);
```

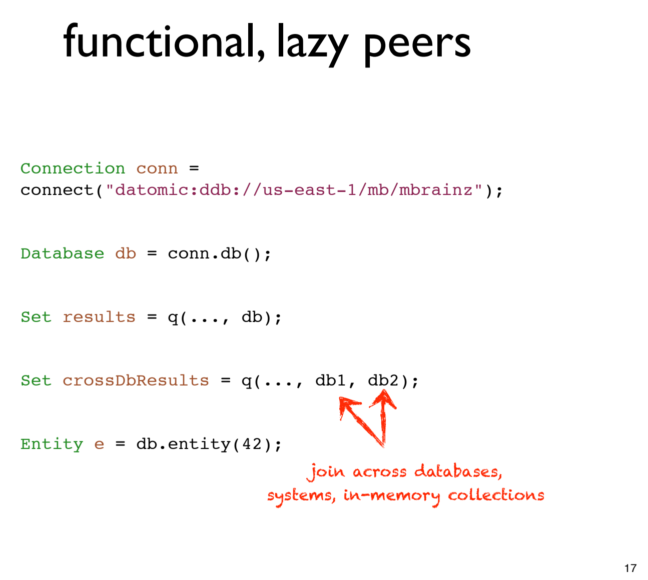
database is a lazily
realized value, available
to all peers equally



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);
```

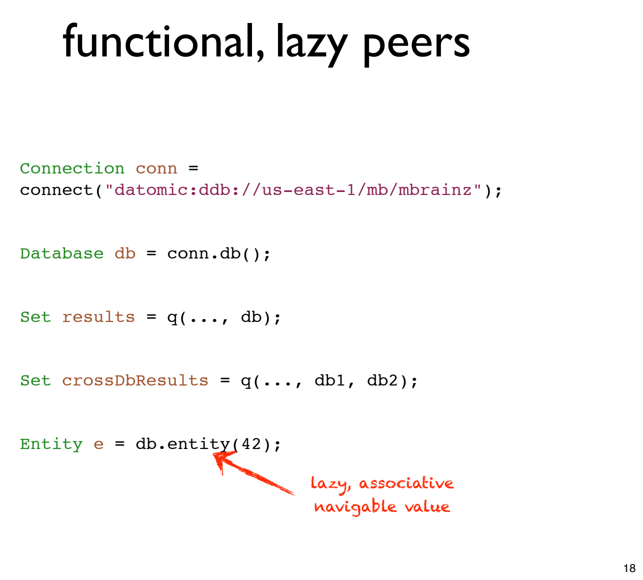
query databases,
not connections



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);

possibleFuture = db.with(...);

allTime = db.history();

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

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

time travel

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();

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

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

all history, overlapped

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);
```

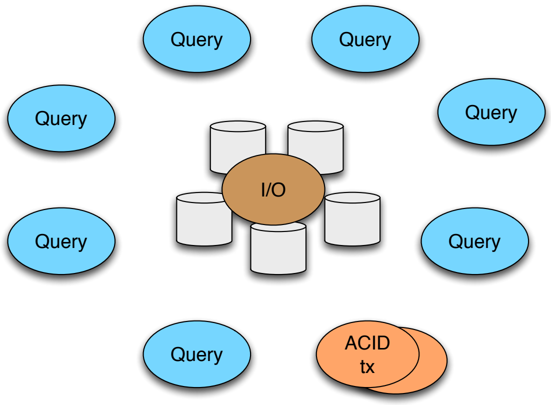
monitor all change
from any peer

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

elastic query scaling



query

why datalog?

Equivalent to Relational Model + Recursion

Better fit than Prolog for query

- No clause order dependency
- Guaranteed termination

Pattern-matching style easy to learn

Example Database

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

Data Pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

entity attribute value

Data Pattern

Constrains the results returned,
binds variables

constant

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

variable variable

```
[?customer :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[?customer :email ?email]
```

Constants Anywhere

"Find a particular customer's email"

```
[42 :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 :email ?email]
```

Variables Anywhere

"What attributes does customer 42 have?

```
[42 ?attribute]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute]
```

Variables Anywhere

"What attributes and values does customer 42 have?

```
[42 ?attribute ?value]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute ?value]
```

Where Clause

data pattern

```
[:find ?customer
 :where [?customer :email]]
```

Find Clause

variable to
return

```
[:find ?customer
 :where [?customer :email]]
```

Implicit Join

“Find all the customers who
have placed orders.”

```
[:find ?customer
 :where [?customer :email]
        [?customer :orders]]
```

API

```
import static datomic.Peer.q;


q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

q

```
import static datomic.Peer.q;


q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

Query

```
import static datomic.Peer.q;


q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

Input(s)

```
import static datomic.Peer.q;


q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

In Clause

Names inputs so you can refer to them elsewhere in the query

```
:in $database ?email
```

Parameterized Query

“Find a customer by email.”

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

First Input

“Find a customer by email.”

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Second Input

“Find a customer by email.”

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Verbose?

“Find a customer by email.”

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Shortest Name Possible

“Find a customer by email.”

```
q([:find ?customer
   :in $ ?email
   :where [$ ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Elide $ in Where

“Find a customer by email.”

```
q([:find ?customer
   :in $ ?email
   :where [ ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

no need to
specify $

Predicates

Functional constraints that can appear in a :where clause

```
[(< 50 ?price)]
```

Adding a Predicate

“Find the expensive items”

```
[:find ?item
 :where [?item :item/price ?price]
        [(< 50 ?price)]]
```

Functions

Take bound variables as inputs and bind variables with output

```
[(shipping ?zip ?weight) ?cost]
```

Function Args

```
[(shipping ?zip ?weight) ?cost]
```

bound inputs

Function Returns

```
[(shipping ?zip ?weight) ?cost]
```

bind return
values

Calling a Function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

navigate from customer to zip

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

get product facts needed *during query*

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

call web service to bind shipCost

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

BYO Functions

Functions can be plain JVM code.

```
public class Shipping {
  public static BigDecimal
  estimate(String zip1, int pounds);
}
```

Calling a Function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

constrain price

Calling a Function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

return customer, product pairs

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

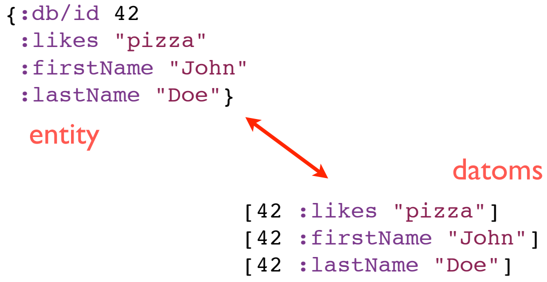
Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

entities

- maplike, point-in-time view of datoms sharing a common **e**

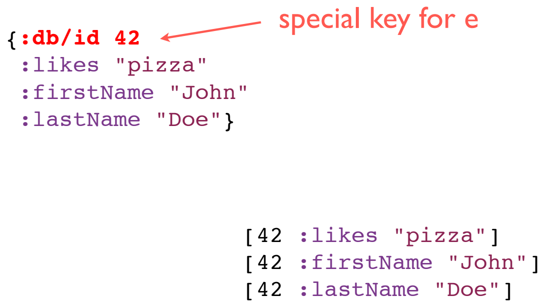


```
{:db/id 42
 :likes "pizza"
 :firstName "John"
 :lastName "Doe"}
```

```
[42 :likes "pizza"]
[42 :firstName "John"]
[42 :lastName "Doe"]
```

entities

- transformation is purely mechanical



```
{:db/id 42
 :likes "pizza"
 :firstName "John"
 :lastName "Doe"}
```

```
[42 :likes "pizza"]
[42 :firstName "John"]
[42 :lastName "Doe"]
```

one database, many indexes

structure	**attribute**
row	EAVT
column	AEVT
document	EAVT, partitions, components
graph	VAET

transactions

ids and partitions

built-in partitions

partition	usage
:db.part/db	schema entities
:db.part/tx	transaction entities
:db.part/user	user entities

create your own partitions

group related entities in a partition

- coarser granularity than e.g. tables

partition is a hint to indexing

- group these things together
- can help locality
- does not affect semantics

creating partitions

```
[{:db.install/_partition :db.part/db,
  :db/id #db/id[:db.part/db],
  :db/ident :inventory}
 {:db.install/_partition :db.part/db,
  :db/id #db/id[:db.part/db],
  :db/ident :customers}]
```

uniqueness

uniqueness

requirement	model with	value types
db-relative opaque id	entity id	opaque (long)
external id	:db.unique/identity attribute	string, uuid, uri
global opaque id	:db.unique/identity squuid	uuid
programmatic name	:db/ident	keyword

squids

semi-sequential UUIDs

do not fragment indexes

```
public class Peer;
    public static UUID squuid();
    public static long squuidTimeMillis(UUID squuid);
    // other methods elided for brevity
}
```

transaction functions

add and retract

```
[[:db/add john :likes pizza]
 [:db/retract john :likes iceCream]]
```

what about update?

```
[[:db/add john :likes pizza]
 [:db/retract john :likes iceCream]
 [:db/add john :balance 110?]]
```

atomic increment

```
[[:db/add john :likes pizza]
 [:db/retract john :likes iceCream]
 [:inc john :account 10]]
```

transaction fns

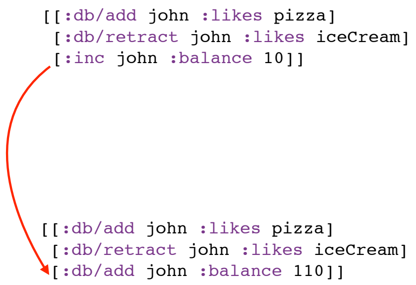
subset of data fns

run inside transactions

have access to in-tx value of database

as first argument

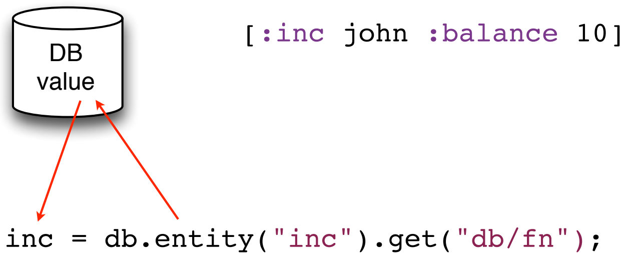
tx function expansion



```
[[:db/add john :likes pizza]
 [:db/retract john :likes iceCream]
 [:inc john :balance 10]]
```

```
[[:db/add john :likes pizza]
 [:db/retract john :likes iceCream]
 [:db/add john :balance 110]]
```

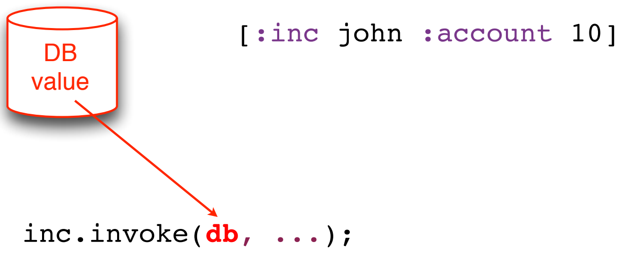
lookup the function



[:inc john :balance 10]

inc = db.entity("inc").get("db/fn");

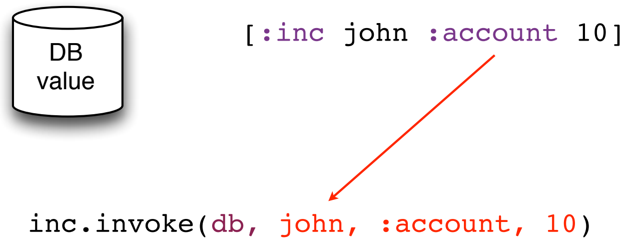
pass in current db



[:inc john :account 10]

inc.invoke(**db**, ...);

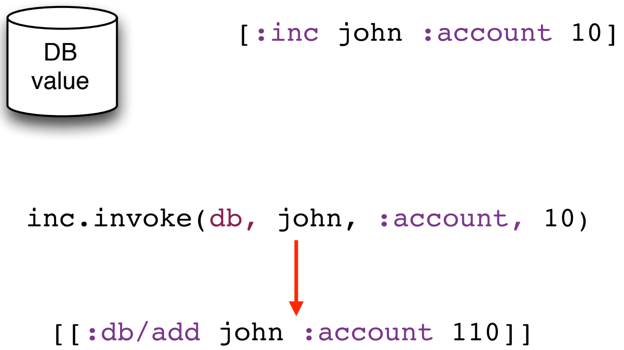
pass in args



[:inc john :account 10]

inc.invoke(db, john, :account, 10)

data out



[:inc john :account 10]

inc.invoke(db, john, :account, 10)

[[:db/add john :account 110]]

inc

```
public static Object inc(Object db, Object e, Object amount)
{
    // lookup entity
    // calculate new balance
    // create assertion
    // return list containing assertion
}
```

inc

```
public static Object inc(Object db, Object e, Object a, Object amount) {
    Entity ent = ((Database)db).entity(e);
    Long balance = (Long) ent.get(a) + (Long) amount;
    List updated = list("db/add", e, a, balance);
    return list(updated);
}
```

modeling

modeling rigiditiy

“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

universal relation

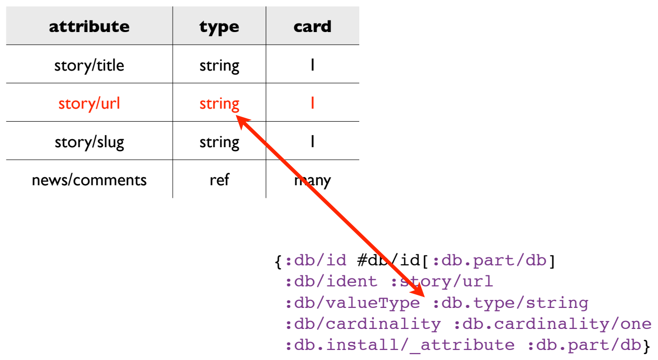
“People can belong to multiple clubs”

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

stories

attribute	type	cardinality
story/title	string	1
story/url	string	1
story/slug	string	1
news/comments	ref	many

schema is plain old data



attribute	type	card
story/title	string	1
story/url	string	1
story/slug	string	1
news/comments	ref	many

```
{:db/id #db/id[:db.part/db]
 :db/ident :story/url
 :db/valueType :db.type/string
 :db/cardinality :db.cardinality/one
 :db.install/_attribute :db.part/db}
```

users

attribute	type	cardinality
user/firstName	string	1
user/lastName	string	1
user/email*	string	1
user/upVotes	ref	many

*unique

cardinality many

```
[:db/add 42 :upvotes 11]
[:db/add 42 :upvotes 12]
```

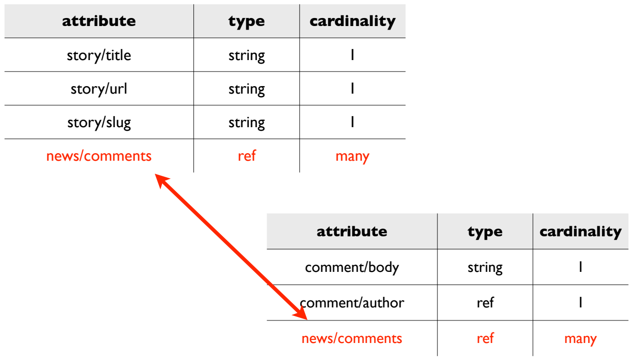
entities

```
john = db.entity(42);
john.get("user/upVotes").size();
```

comments

attribute	type	cardinality
comment/body	string	1
comment/author	ref	1
news/comments	ref	many

types do not dictate attrs



attribute	type	cardinality
story/title	string	1
story/url	string	1
story/slug	string	1
news/comments	ref	many

attribute	type	cardinality
comment/body	string	1
comment/author	ref	1
news/comments	ref	many

relation direction

```
// get child comments
comment.get("news/comments");
```

reversing direction

```
// get parent comment
comment.get("news/_comments");
```

underscore means
“nav backwards”

recursive (graph) query

```
;; base case
[(story-comment ?story ?comment)
 [?story :story/title]
 [?story :new/comments ?comment]]
```

it is a story comment if...

recursive (graph) query

```
;; base case
[(story-comment ?story ?comment)
 [?story :story/title]
 [?story :new/comments ?comment]]
```

... there is a story ...

recursive (graph) query

```
;; base case
[(story-comment ?story ?comment)
 [?story :story/title]
 [?story :new/comments ?comment]]
```

... with a comment

recursive (graph) query

```
;; recursion
[(story-comment ?story ?comment)
 [?parent :news/comments ?comment)
 (story-comment ?story ?parent)]
```

or, it is a story comment if...

recursive (graph) query

```
;; recursion
[(story-comment ?story ?comment)
 [?parent :news/comments ?comment]
 (story-comment ?story ?parent)]
```

... it has a parent comment ...

recursive (graph) query

```
;; recursion
[(story-comment ?story ?comment)
 [?parent :news/comments ?comment)
 (story-comment ?story ?parent)]
```

which is itself a story comment

documents

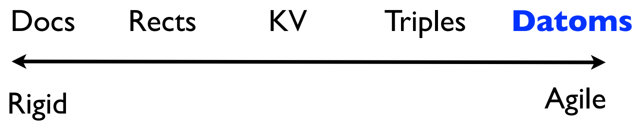
activity “document”

```
// get upvotes
john.get("user/upVotes");
// get title of an upvoted story
anUpvote.get("story/title");
// get John's comments
john.get("comment/_author");
```

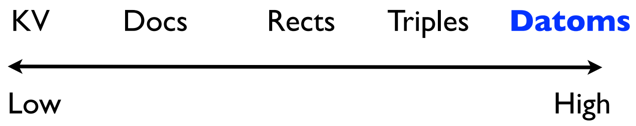
profile “document”

```
// get facts about John
john.get("user/email");
john.get("user/firstName");
```

agility



leverage



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

String Injection

Data Duplication

Isolation Levels

programmability

Make a column name variable?

Make a table name variable?

Treat metadata as first-class data?

first-class attributes

```
[?person ?attr ?value]
```

attribute slot
isn’t special

schema made of ordinary data

```
[?e :db/valueType]
```

find all
attributes

user stories

Room key
Search. Book. Relax.

http://thinkrelevance.com/blog/2013/06/12/kurt-zimmer-of-room-key-podcast-episode-033

https://github.com/candera/strangeloop-2013-datomic/blob/master/slides.org

“We use Datomic as an event-source data store and it works wonderfully!”

cognician

https://groups.google.com/forum/#!topic/datomic/E-M_x-wCjOM

"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>

resources

Datomic

http://www.datomic.com/
http://blog.datomic.com/2013/06/using-datomic-from-groovy-part-1.html
http://blog.datomic.com/2013/05/a-whirlwind-tour-of-datomic-query_16.html
https://github.com/datomic/day-of-datomic

Stuart Halloway

https://github.com/stuarthalloway/presentations/wiki.
http://www.linkedin.com/pub/stu-halloway/0/110/543/
https://twitter.com/stuarthalloway
mailto:stu@cognitect.com