mm-ADT
A Multi-Model Abstract Datatype

Dr. Marko A. Rodriguez
Founder, RReduX Inc.
Project Management Committee, Apache TinkerPop

Collaborators
Daniel Kuppitz and Stephen Mallette

Funded by:

RREDUX DATASTAX

Rodriguez, M.A., “mm-ADT: A Multi-Model Abstract Datatype,” ApacheCon 2019, Las Vegas, NV, September 2019.

WARNING

This Slide Presentation Should Not Be Used as a Reference

As of September 2019, there exists a rough draft of the mm-ADT specification (0.1-alpha) and a Java-based prototype of the mm-ADT-bc compiler. The specifics presented here are likely to change as the project matures.

The goal is to release the following artifacts by early 2020.

1.) A stable 1.0 mm-ADT specification document.
2.) A Java-based mm-ADT-bc compiler and virtual machine.
3.) A basic reference implementation of an mm-ADT database.

Database Datatypes

KEY/VALUE DATABASE


riak
RocksDB

DOCUMENT DATABASE


ArangoDB


mongoDB®

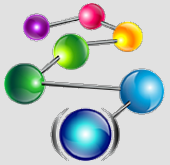
PROPERTY GRAPH DATABASE

DATASTAX


neo4j


OrientDB®

RDF DATABASE


AllegroGraph


STARDOG

RELATIONAL DATABASE

Apache Derby


MySQL™

WIDE-COLUMN DATABASE

APACHE HBASE

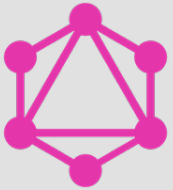

cassandra

Database Components

STORAGE SYSTEM

PROCESSING ENGINE

QUERY LANGUAGE



Universal Database Components

60% solid.

UNIVERSAL DATA STRUCTURE

Presentation is primarily on this topic.

90% solid.

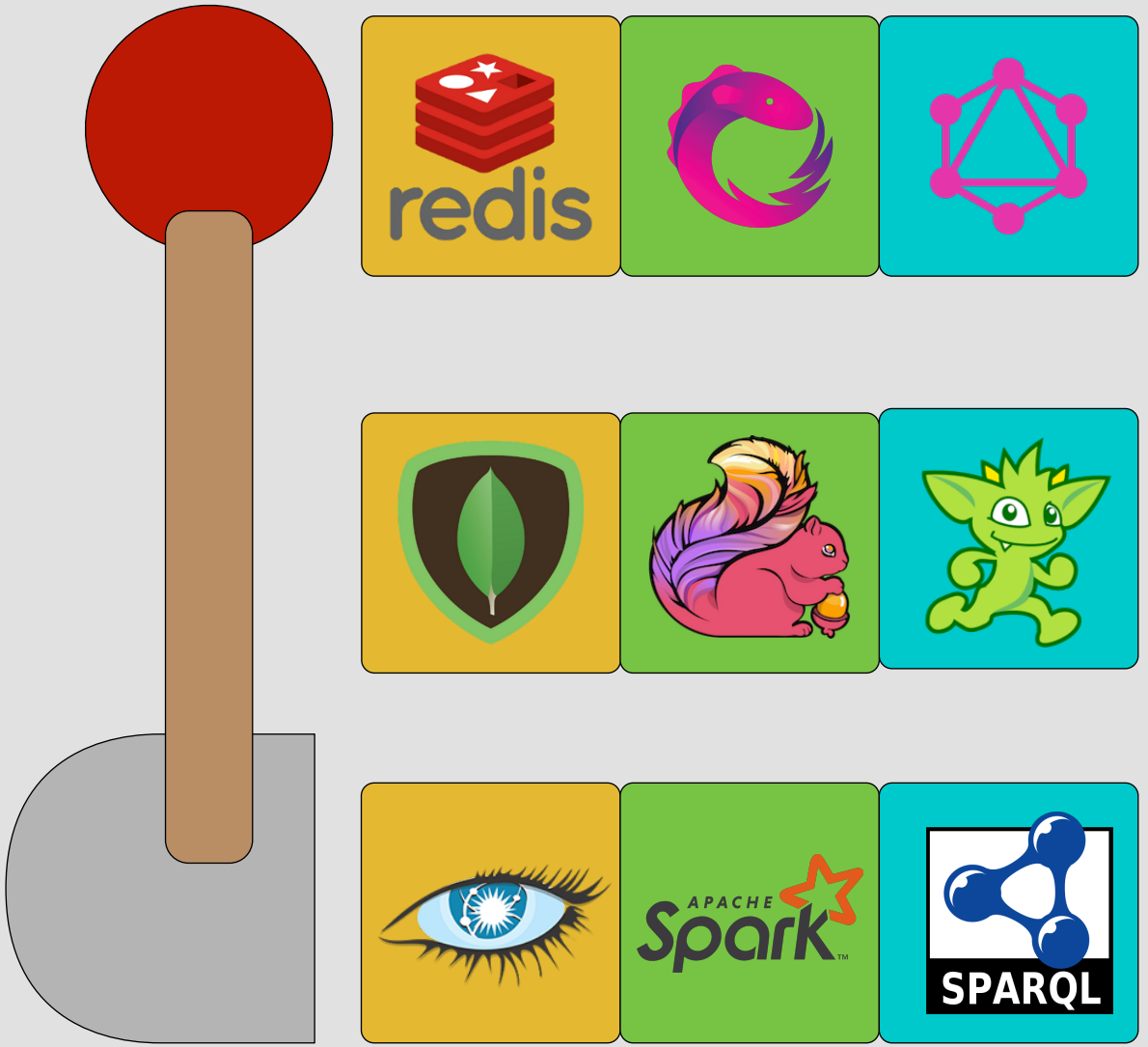
UNIVERSAL PROCESSING MODEL

80% solid.

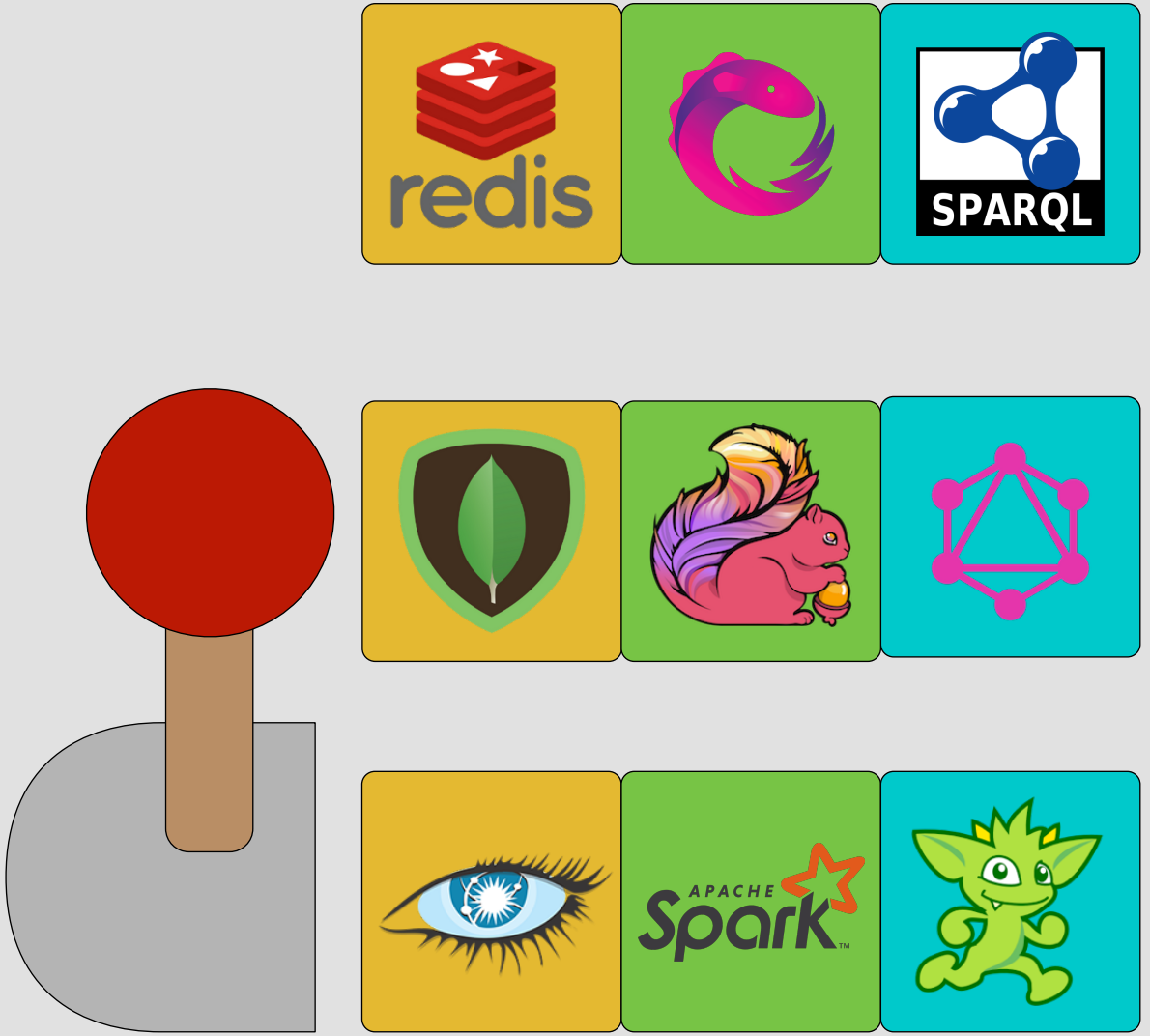
UNIVERSAL INSTRUCTION SET

as of Sept, 2019

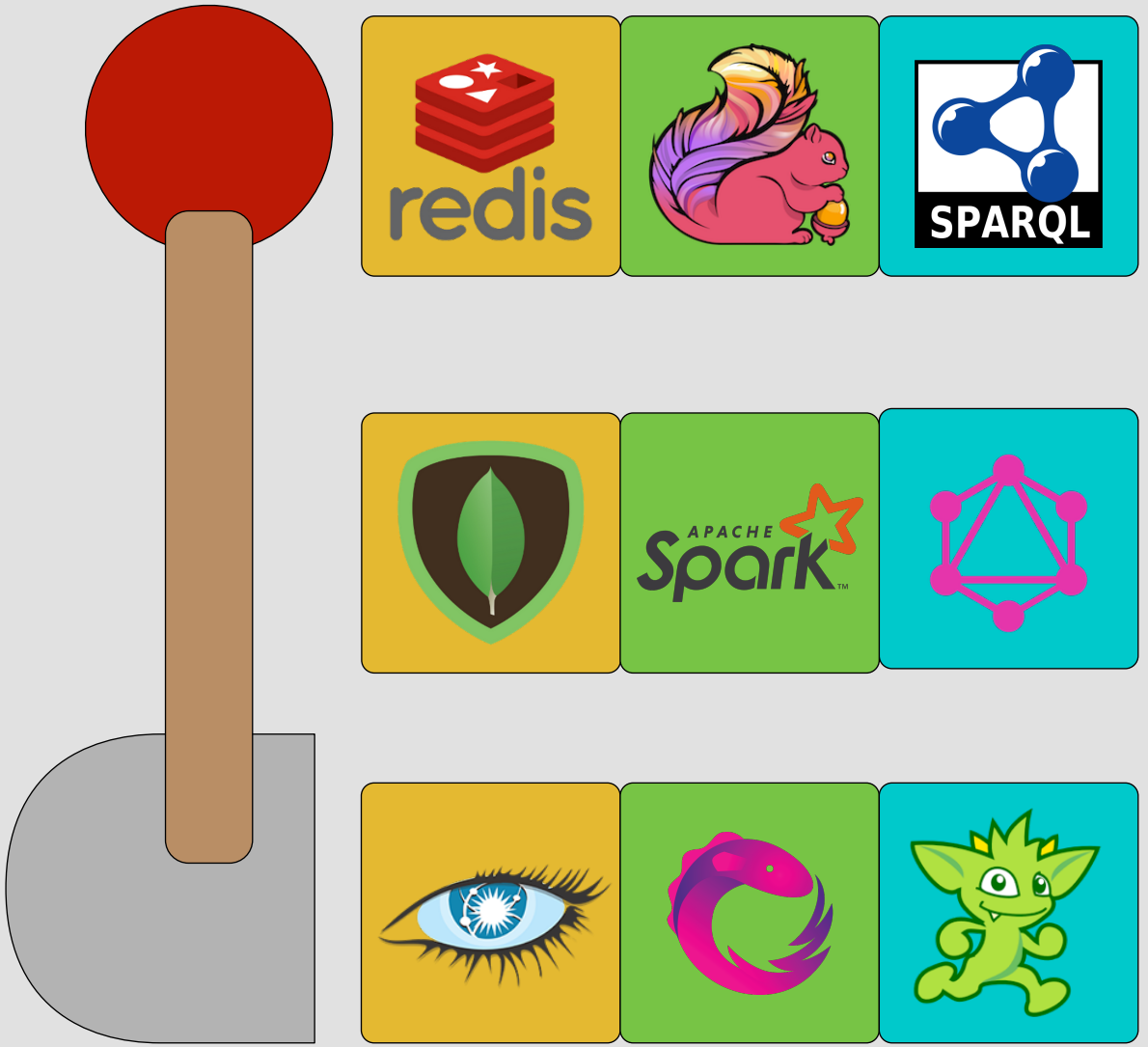
Synthetic Database Systems



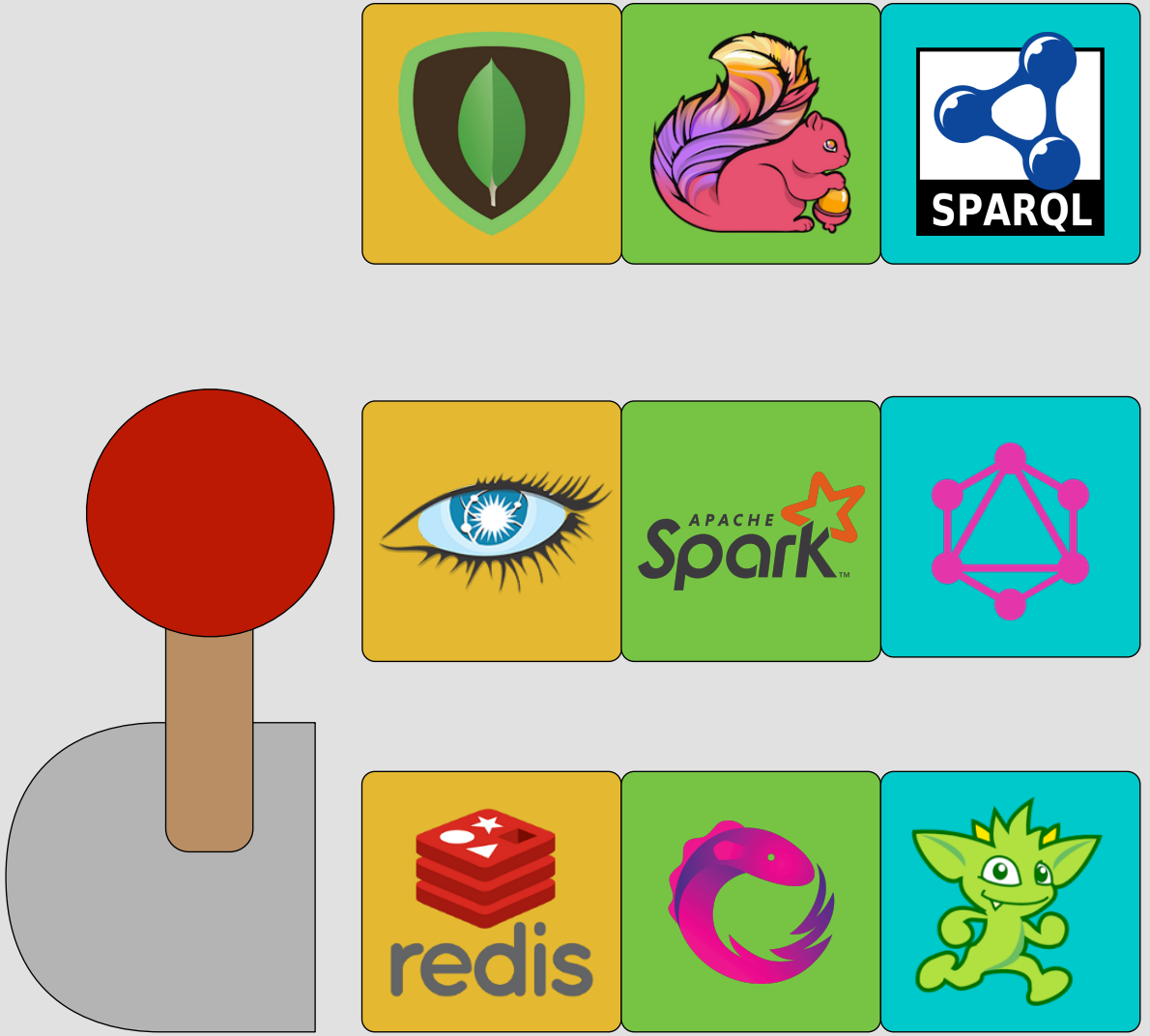
Synthetic Database Systems



Synthetic Database Systems

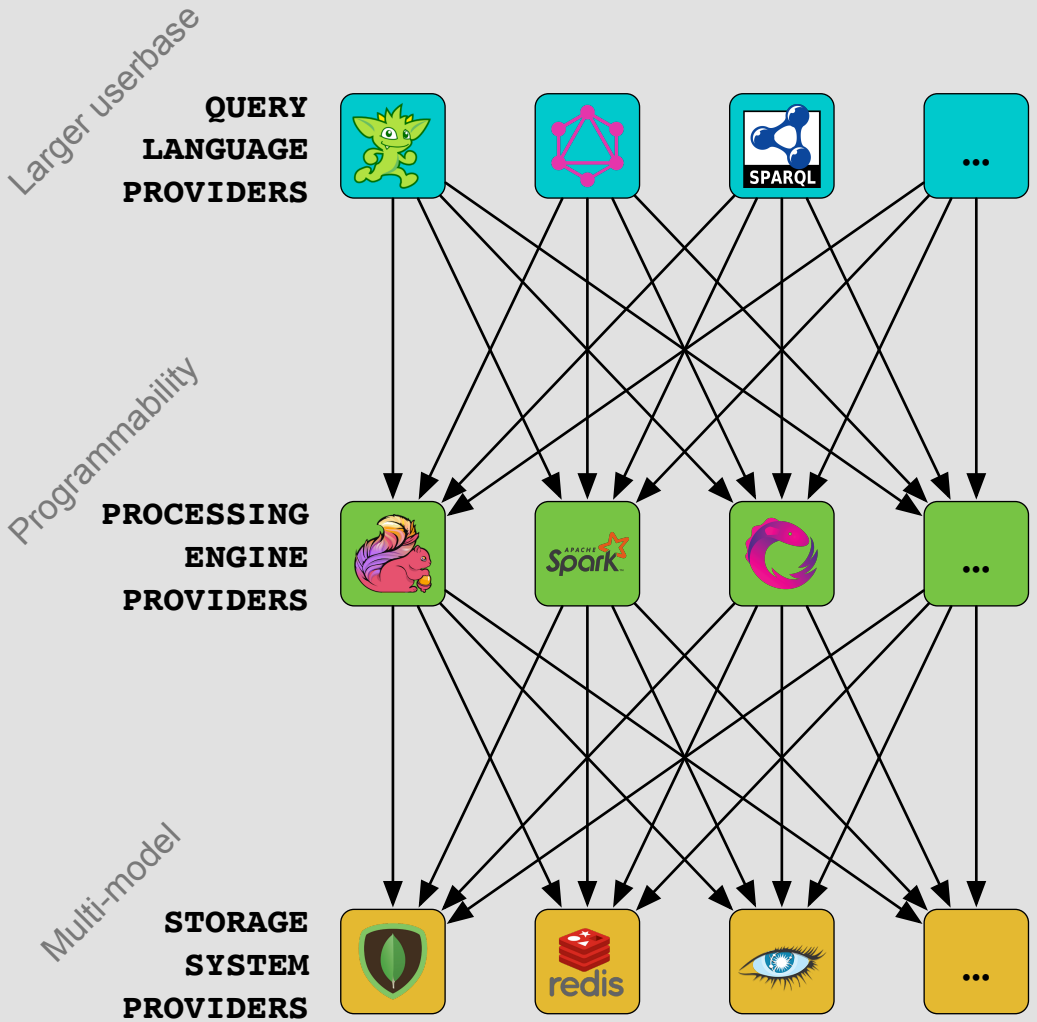


Synthetic Database Systems



mm-ADT Compliant

The Benefit to Component Providers



Your query language's queries can execute real-time, near-time, or batch-time over any database.

Your processing engine can be programmed by any query language and compute over any database.

Your database can be processed by many processors whose queries are defined by many different query languages.

mm-ADT

A Multi-Model Abstract Datatype

Any Data Structure/Language

KEY/VALUE
JSON/XML DOCUMENT
PROPERTY/RDF GRAPH
RELATIONAL
WIDE-COLUMN
QUANTUM
… *yes. any data structure.*
SQL
SPARQL
CYPHER
GREMLIN
GRAPHQL
CQL
QUERY DOCUMENT
QUANTUM UNITIARY MATRICES

yes. any data query language.

Storage/Processing Agnosticism

RECORD LAYOUT
INDEX STRUCTURES
PARTITIONING
DENORMALIZATIONS
SORT ORDERS
…
PULL-BASED ITERATION
PUSH-BASED REACTIVE
MESSAGE PASSING
LINEAR SCAN/FILTER
LAZY/STRICT EVALUTION
QUANTUM WAVE INTERFERENCE

yes. any algorithm.

Universal Operators

GET
PUT
FILTER
SELECT
PROJECT
ORDER
DEDUP
GROUP
COUNT
SUM
MEAN
COALESCE
REPEAT
BRANCH
ARITHMETIC

yes. addition. :)

mm-ADT Requirements

A Cluster-Oriented Bytecode and Virtual Machine

A **type system** for capturing database models and schemas.

Datatypes, composite structures, constraints, foreign keys, cardinalities, dependencies.

A **database perspective** w/ loose coupling for future use cases.

Indices, denormalizations, read/write costs, transactions, offloading computations, complex data access paths, multi-user, distributed data, data locality, real-time/batch executions.

A **bytecode** foundation for use across software and languages.

Compatible with Java, C++, Go, etc.-based databases.

Reasonable compilation strategies to/from:

SQL, SPARQL, Cypher, Gremlin, GraphQL, QueryDocs, …future.

A **universal processing** model able to capture various strategies.

Single-threaded, single-machine queries.

Distributed, cluster-oriented queries.

Lazy and space-efficient for main-memory queries (real-time).

Eager and time-efficient for disk-based queries (batch).

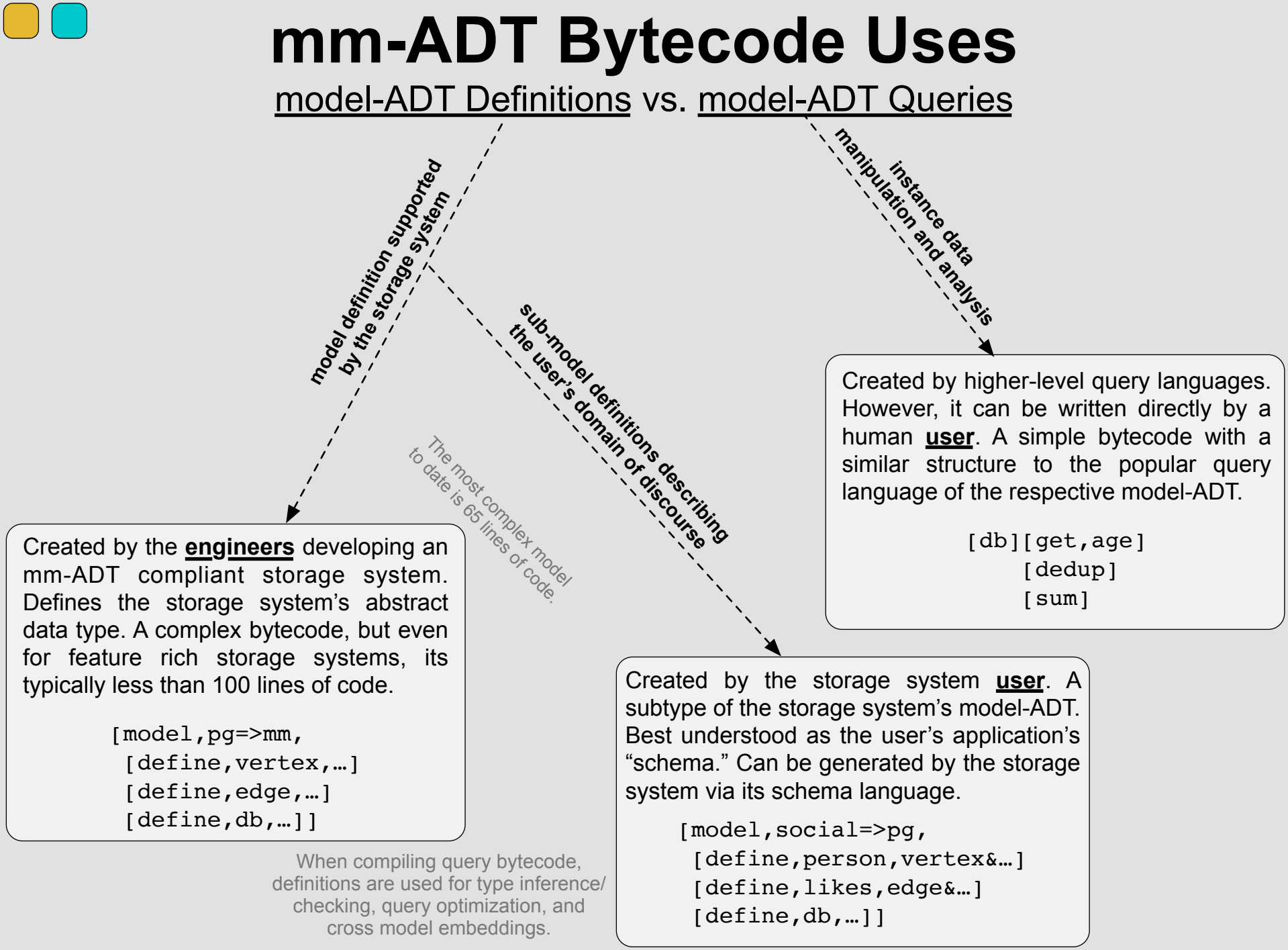
Turing-complete for any known query/algorithm.

Part 1

Universal Data Structure

mm-ADT Bytecode Uses

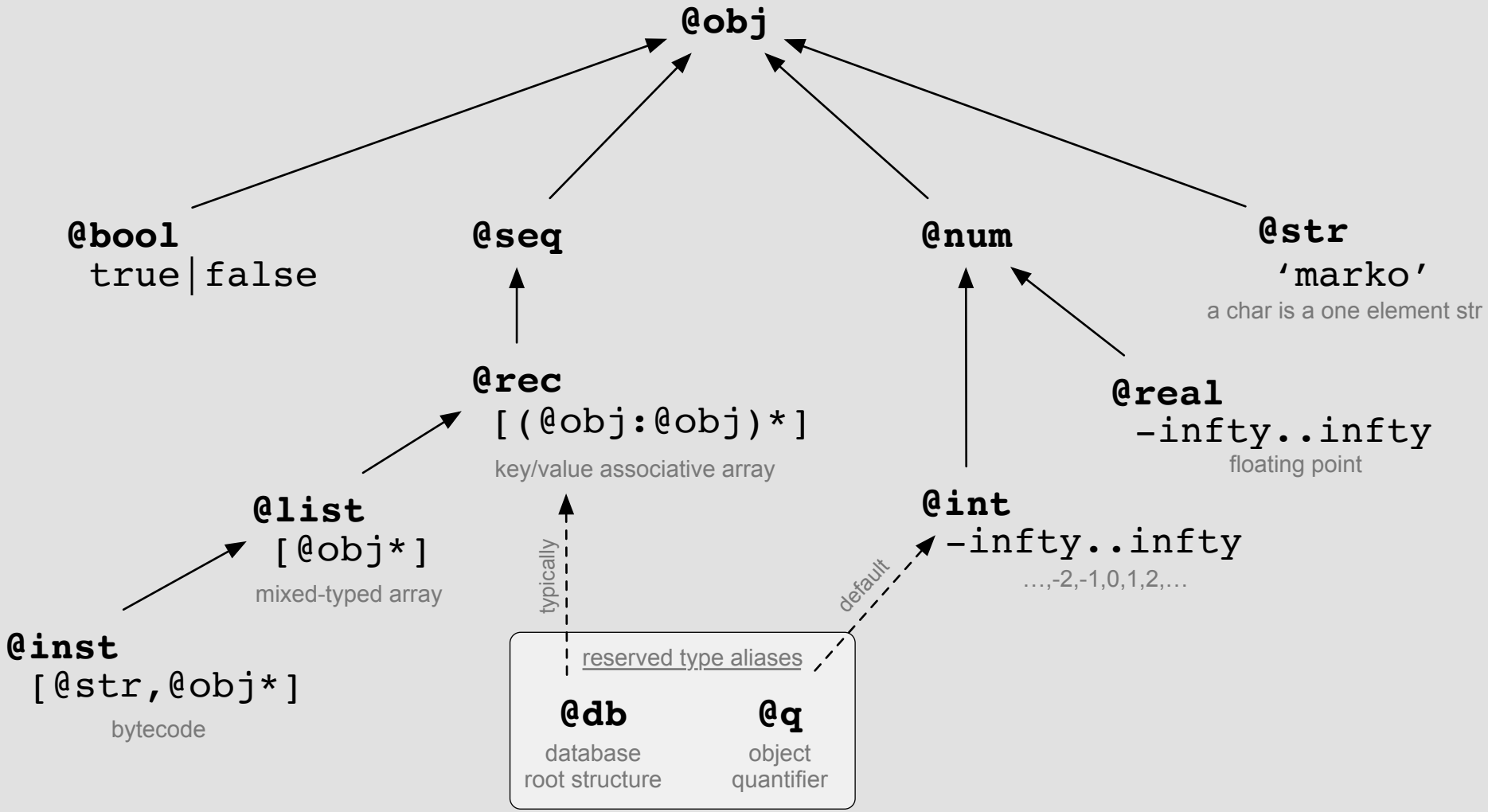
model-ADT Definitions vs. model-ADT Queries



The language used for all examples is the (somewhat) human readable/writable bytecode language called mm-ADT-bc.

Built-In Datatypes

Fundamental Structures Defined Outside of mm-ADT



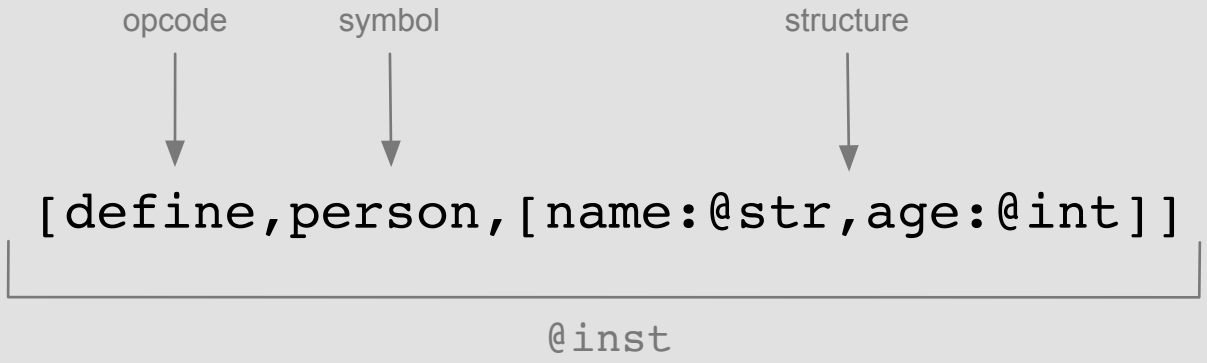
Custom Datatypes

Type Definitions

```
[define,person,[name:@str,age:@int]]
```

Custom Datatypes

Type Definitions



@str

```
[is,[type][eq,≪str≫]]: @obj => @str?
```

@int

```
[is,[type][eq,≪int≫]]: @obj => @int?
```

@person

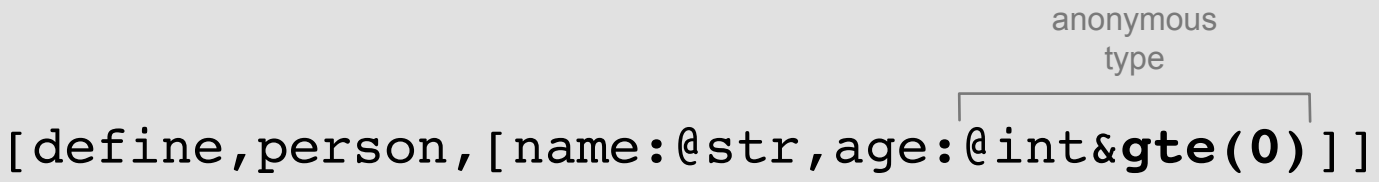
```
[is,[get,name][type][eq,≪str≫]][is,[get,age][type][eq,≪int≫]]: @obj => @person?
```

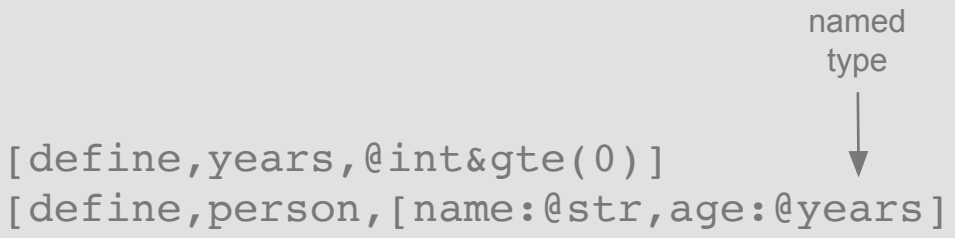
```
[define,person,[name:[is,[type][eq,≪str≫]],age:[is,[type][eq,≪int≫]]]]
```

mm-ADT types are filter bytecode (predicates). If an object is unfiltered by the bytecode, then the object is that type.

Custom Datatypes

Type Refinement






```
[define,probability, @real&gte(0.0)&lte(1.0)]
[define,varchar255,  @str&[is,[len][lte,255]]
[define,short,       @int&gte(-32767)&lte(32767)]
[define,pair,        [@obj,@obj]]
[define,triple,      [@obj,@obj,@obj]]
```

Anonymous types are the norm in mm-ADT.

Custom Datatypes

Type Recursion

```
[define,person,[name:@str,age:@int&gte(0),friend:@person]]
```

Graph model-ADTs are recursive structures

```
[define,vertex,inE:@edge*,outE:@edge*]
[define,edge,outV:@vertex,inV:@vertex]
```

Custom Datatypes

Type Quantifiers

Quantifiers are any algebraic ring with unity.

`[mult][add][sub][zero][one]`

```
[define,person,[name:@str,age:@int&gte(0),friend:@person?]]
```

```
{3,5}: 3 to 5
*    : {0,infty}
+    : {1,infty}
?    : {0,1}
{2}  : {2,2}
{3,} : {3,infty}
```

```
[define,none, @obj{0}   ]
[define,some, @obj{1}   ]  // @obj
[define,maybe,@obj{0,1}]  // @obj?
```

```
@none | @some = @maybe
{0,0} + {1,1} = {0,1}
  {min,max}
```

```
@str?
      @person?
@vertex?
```

Unitary Martrix Quantifiers

Quantifiers can be matrices composed of complex numbers.
Constructive and deconstructive wave interference patterns can be incorporated into a query.
This does not radically alter the bytecode as quantifiers are fundamental to mm-ADT.

```
[define,q,@unitary]
```

```
@obj{[1,0,1,0]}
```

Real Number Quantifiers

Quantifiers can be real numbers (for example, values between 0.0 and 1.0).
Real quantifiers can model energy diffusions, fuzzy set semantics, probabilistic/stochastic behavior.

```
[define,q,@real&gte(0.0)&lte(1.0)]
```

```
@obj{0.92}
```

All standard query languages are bound to the natural numbers with standard multiplication and addition definitions.

Custom Datatypes

Subtypes

Note that there are also axioms for the composition of a type's instruction rewrites. Not discussed in this presentation.

{0,1}

```
[define,person,[name:@str,age:@int&gte(0),friend:@person?]]
[define,loner,@person&[friend:@person{0}]]
```

super type
loner <: person

{0,0}

composition

```
[name:@str,age:@int&gte(0),friend:@person?]&[friend:@person{0}]
                          =>
[name:@str,age:@int&gte(0),friend:@person{0}]
```

{0,1}&{0,0}=>{0,0}=>{0}

type quantifier composition

```
{x1,x2}&{y1,y2} => {x1*y1,x2*y2}
{x1,x2}|{y1,y2} => {min(x1,y1),max(x2,y2)}
```

Composition through & and | are defined for each type with default definitions from the underlying fundamental type.

Custom Datatypes

Type Dependents

```
[define,person,[name:@str,age:@int&gte(0),friend:@person?]]
[define,older,@person[age   :@int&gte(0)~x,
                      friend:@person&[age:@int&gte(0)&lt(~x)]?]]
```

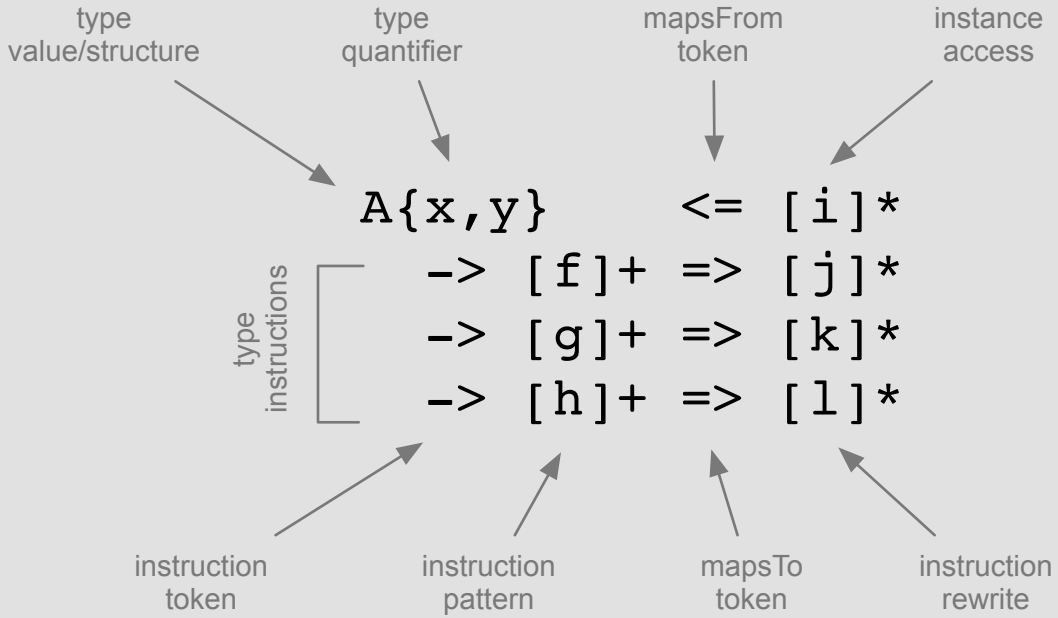
anonymous subtype

```
[define,complex,[@real~x1,@real~x2]
  -> [add,@complex&[@real~y1,@real~y2]]  => [map,@complex&[~x1+~y1,~x2+~y2]]
  -> [mult,@complex&[@real~y1,@real~y2]] => [map,@complex&[(~x1*~y1)-(~x2*~y2),
                                                          (~x1*~y2)-(~x2*~y1)]]
```

Infix arithmetic notation is still being designed. Primary issue is * and + parser ambiguities due to quantifier shorthands.

mm-ADT Datatypes

The General Structure of a Type



$f : A \to B$ $\quad f(a) = b$

domain spec range def

```
A <= [i]
  -> [f] => [B <= [j]]
  -> [g] => [C <= [k]]
```

Inspired by OOP where methods/instructions grouped by domain class/type.

$f : A \to B$
$g : A \to C$

```
A -> [f] => B
  -> [g] => C
```

\+ Types with instructions denote function branches.

$f \cdot h : A \to D$

```
A -> [f] => B -> [h] => D
```

* Paths from type to type via instructions denote function composition.

The type is represented in mm-ADT-bc syntax. All mm-ADT bytecode generating languages leverage these components.

mm-ADT Datatypes

Instance Data Access Path

```
[define,person,
  [name:@str~x,age:@int] <= [db][get,people][is,[get,name][eq,~x]]]
```

instance access
(canonical representation)

```
A <= […]
```

instance
```
@person&[name:marko,age:29]
```
all constant values

type
```
@person&[age:29]
```
name unbound
"people 29 years of age"

reference
```
@person&[name:marko]
```
accessible via <=

mm-ADT is a cluster-oriented programming language where data access has different costs depending on locality.

mm-ADT Datatypes

Instruction Rewrites

```
[define,person,[name:@str,age:@int]]
[define,db,person*
 -> [order,[gt,[get,age]]]      =>
 -> [dedup,[get,name]]          =>
 -> [count]                     => [ref,@int     <= [db][count]]
 -> [is,[get,name][eq,@str~x]] => [ref,@person? <= [db][is,[get,name][eq,~x]]]
```

no-op

aggregate

index lookup

submitted bytecode

rewritten bytecode

```
[db][is,[get,name][eq,marko]] => [ref,@person? <= [db][is,[get,name][eq,marko]]]
```

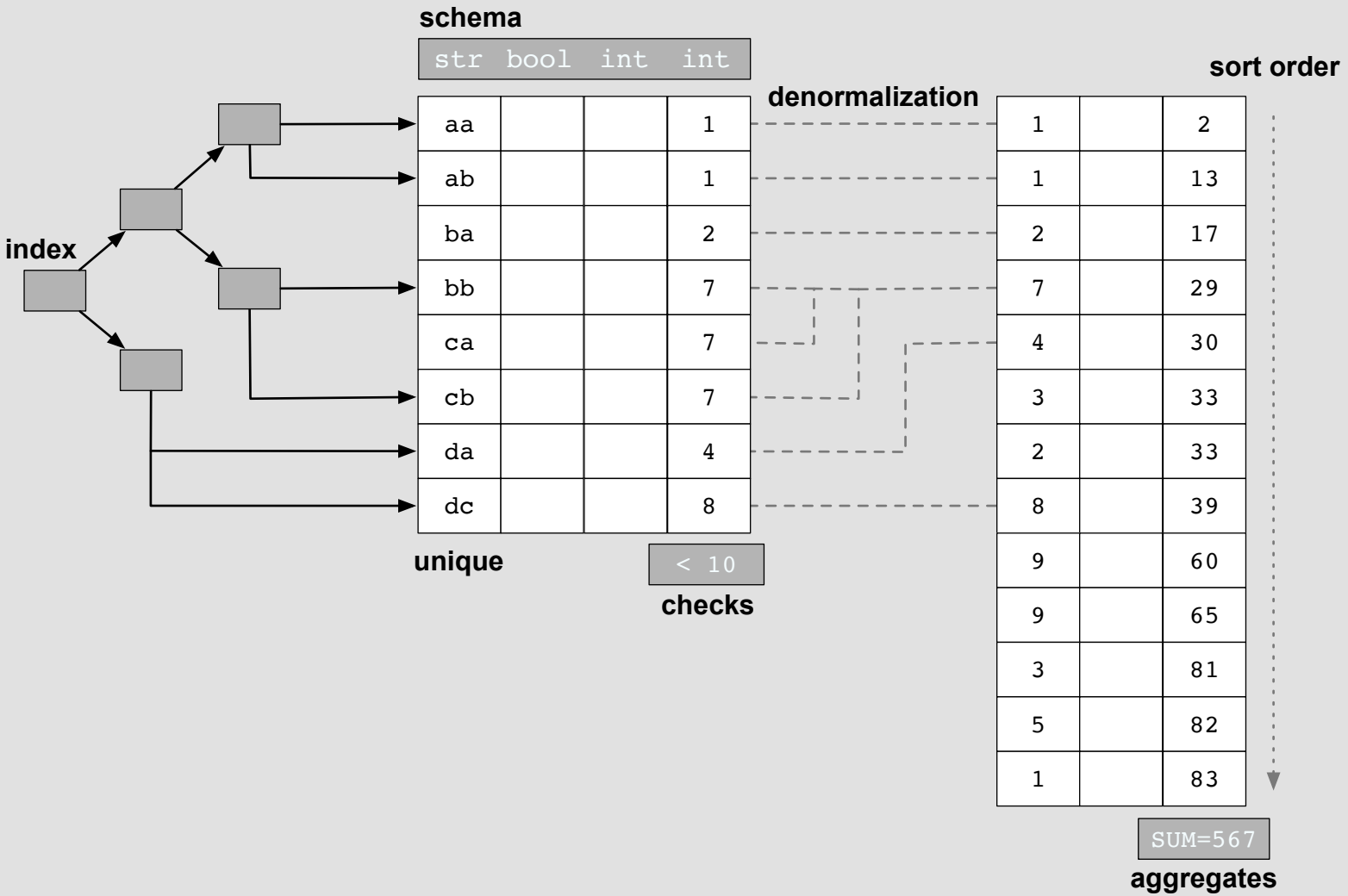
`O(n)` linear scan

`O(log n)` index query

All [ref] instructions are resolved by the storage system or processing engine and are used to access secondary structures.

Relational model-ADT

Primary and Secondary Structures



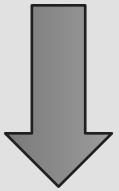
☐ **Primary Structure**: The structure associated with the database's conceptual model.

☐ **Secondary Structures**: The auxiliary structures used to improve query performance and ensure data integrity.

Relational model-ADT

RELATIONAL DATABASE

```
[model,rdb=>mm,
 [define,value,@bool|@num|@str]
 [define,row,[(@str:@value)*]]
 [define,table,@row*]
 [define,db,[(@str:@table)*]]]
```



DOMAIN SPECIFIC RELATIONAL SCHEMA

```
[model,social=>rdb,
 [define,person,[name:@str?,age:@int?]]
 [define,persons,@person*]
 [define,db,[people:@persons]]]
```

Relational model-ADT

Primary Structure (Subtype)

```
[define,person,[name:@str?,age:@int?]]
[define,persons,@person*]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int
)
```

Relational model-ADT

Secondary Structures (Primary Key)

```
[define,person,[name:@str,age:@int?]]
[define,persons,@person*
  -> [dedup,[get,name]] => ]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int,
  PRIMARY KEY(name)
)
```

Relational model-ADT

Secondary Structures (Sort Orders)

```
[define,person,[name:@str,age:@int?]]
[define,persons,@person*
  -> [dedup,[get,name]]      =>
  -> [order,[gt,[get,name]]] => ]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int,
  PRIMARY KEY(name,ASC)
)
```

Relational model-ADT

Secondary Structures (Null Values)

```
[define,person,[name:@str,age:@int]]
[define,persons,@person*
  -> [dedup,[get,name]]      =>
  -> [order,[gt,[get,name]]] => ]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  PRIMARY KEY(name,ASC)
)
```

Relational model-ADT

Secondary Structures (Check Values)

```
[define,person,[name:@str,age:@int&gte(0)]]
[define,persons,@person*
  -> [dedup,[get,name]]      =>
  -> [order,[gt,[get,name]]] => ]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0)
)
```

Relational model-ADT

Secondary Structures (Foreign Key)

```
[define,person,[name:@str~x,
                age:@int&gte(0),
                friend:@person&[name:@str]] <= [db][get,people]
                                                  [is,[get,name]
                                                      [eq,~x]]]
[define,persons,@person*
  -> [dedup,[get,name]]      =>
  -> [order,[gt,[get,name]]] => ]
[define,db,[people:@persons]]
```

instance access

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  friend varchar(255),
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0),
  FOREIGN KEY (friend) REFERENCES people(name)
)
```

Relational model-ADT

Secondary Structures (Single Key Index)

```
[define,person,[name:@str~x,
                age:@int&gte(0),
                friend:@person&[name:@str]] <= [db][get,people]
                                                  [is,[get,name]
                                                      [eq,~x]]
[define,persons,@person*
  -> [dedup,[get,name]]           =>
  -> [order,[gt,[get,name]]]      =>
  -> [is,[get,name][eq,@str~x]]  => [ref,@person&[name:~x]?]]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  friend varchar(255),
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0),
  FOREIGN KEY (friend) REFERENCES people(name),
  CREATE UNIQUE INDEX name_idx ON people(name)
)
```

Relational model-ADT

Secondary Structures (Multi-Key Index)

```
[define,person,[name:@str~x,
                age:@int&gte(0),
                friend:@person&[name:@str]] <= [db][get,people]
                                                  [is,[get,name]
                                                      [eq,~x]]
[define,persons,@person*
  -> [dedup,[get,name]]              =>
  -> [order,[gt,[get,name]]]         =>
  -> [is,[get,name][eq,@str~x]]      => [ref,@person&[name:~x]?
    -> [is,[get,age][eq,@int~y]]     => [ref,@person&[name:~x,age:~y]?]]
  -> [is,[get,age][eq,@int~y]]       => [ref,@person&[age:~y]*
    -> [is,[get,name][eq,@str~x]]    => [ref,@person&[name:~x,age:~y]?]]]
[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  friend varchar(255),
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0),
  FOREIGN KEY (friend) REFERENCES people(name),
  CREATE UNIQUE INDEX name_idx ON people(name),
  CREATE INDEX name_age_idx ON people(name,age)
)
```

Relational model-ADT

Secondary Structures (Aggregates)

```
[define,person,[name:@str~x,
                age:@int&gte(0),
                friend:@person&[name:@str]] <= [db][get,people]
                                                  [is,[get,name]
                                                      [eq,~x]]

[define,persons,@person*
  -> [dedup,[get,name]]              =>
  -> [order,[gt,[get,name]]]         =>
  -> [is,[get,name][eq,@str~x]]      => [ref,@person&[name:~x]?
    -> [is,[get,age][eq,@int~y]]     => [ref,@person&[name:~x,age:~y]?]]
  -> [is,[get,age][eq,@int~y]]       => [ref,@person&[age:~y]*
    -> [is,[get,name][eq,@str~x]]    => [ref,@person&[name:~x,age:~y]?]]
  -> [count]                         => [ref,@int <= [db][get,people]
                                                        [count]]]

[define,db,[people:@persons]]
```

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  friend varchar(255),
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0),
  FOREIGN KEY (friend) REFERENCES people(name),
  CREATE UNIQUE INDEX name_idx ON people(name),
  CREATE INDEX name_age_idx ON people(name,age)
)
```

Relational model-ADT

Secondary Structures (Domain Logic)

conceptual secondary structures

```
[define,person,[name:@str~x,
                age:@int&gte(0),
                friend:@person&[name:@str]] <= [db][get,people]
                                                  [is,[get,name]
                                                      [eq,~x]
  -> [get,friend][get,friend]     => ]
[define,persons,@person*
  -> [dedup,[get,name]]           =>
  -> [order,[gt,[get,name]]]      =>
  -> [is,[get,name][eq,@str~x]]   => [ref,@person&[name:~x]?
    -> [is,[get,age][eq,@int~y]]  => [ref,@person&[name:~x,age:~y]?]]
  -> [is,[get,age][eq,@int~y]]    => [ref,@person&[age:~y]*
    -> [is,[get,name][eq,@str~x]] => [ref,@person&[name:~x,age:~y]?]]
  -> [count]                      => [ref,@int <= [db][get,people]
                                                     [count]]]

[define,db,[people:@persons]]
```

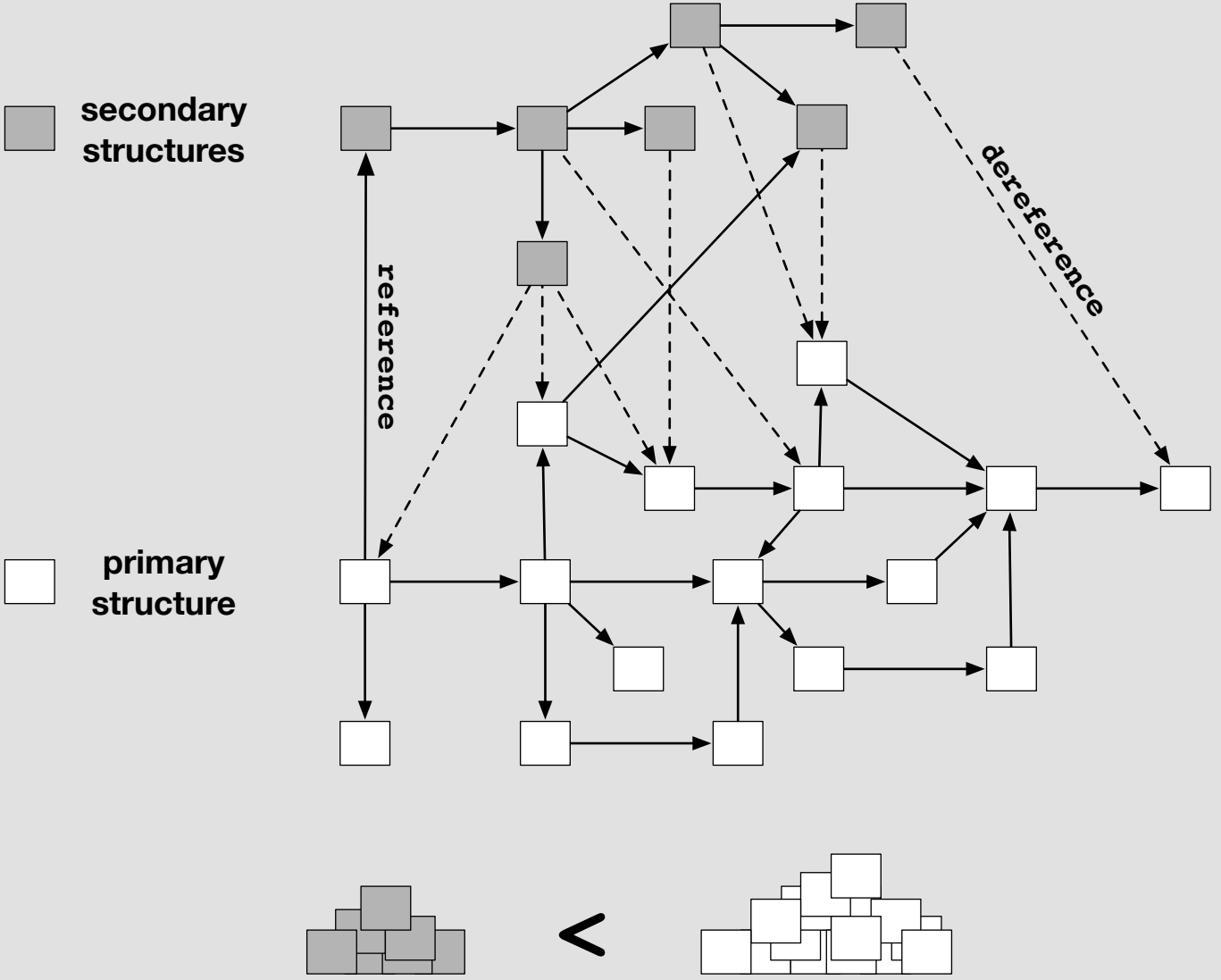
no-op

Friends are pair bonded.
Its a weird example,
but we got stuck with such a simple model.

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  friend varchar(255),
  PRIMARY KEY(name,ASC),
  CHECK (age >= 0),
  FOREIGN KEY (friend) REFERENCES people(name),
  CREATE UNIQUE INDEX name_idx ON people(name),
  CREATE INDEX name_age_idx ON people(name,age)
)
```

model-ADT Data Access Graph

Primary and Secondary Structures Specify Data Access Paths



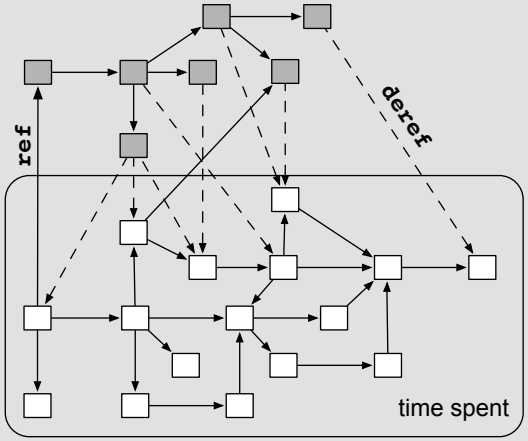
Secondary structures “teleport” the processor from one location in the primary structure to another.

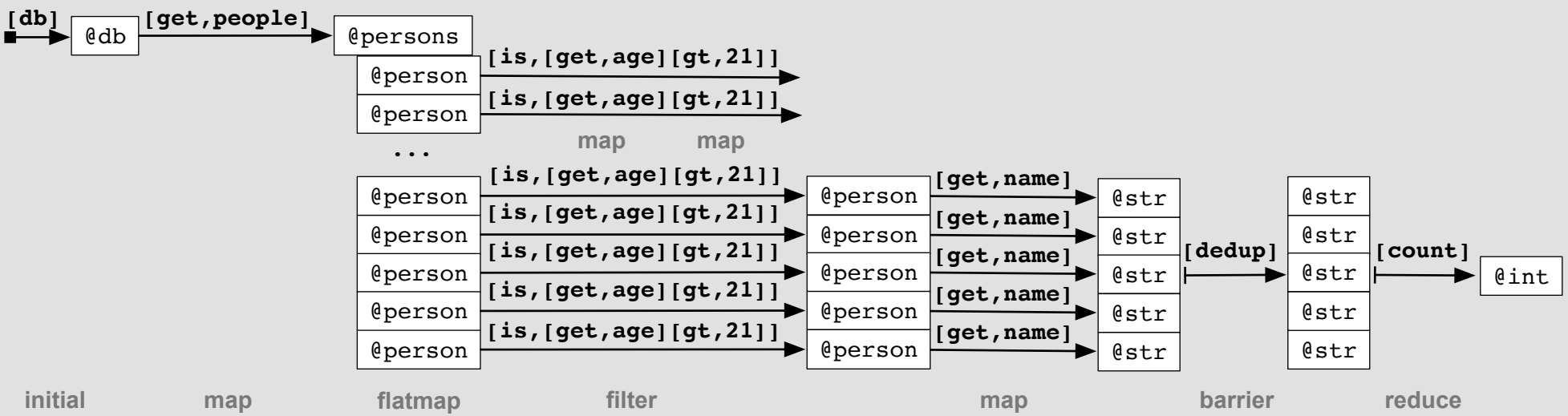
Data Access Path

Primary Structure Only

How many unique names are there for people over 21 years of age?

```
[db][get,people]
    [is,[get,age][gt,21]]
    [get,name]
    [dedup]
    [count]
```



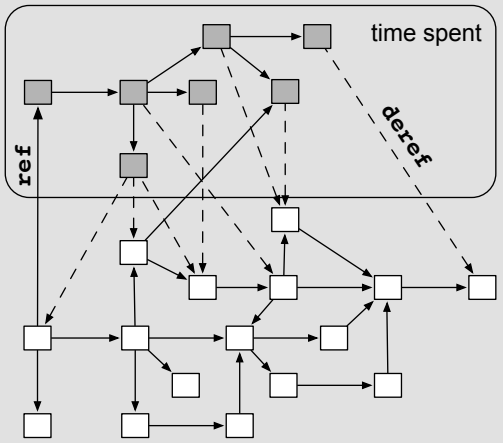



```
[define,person,[name:@str,age:@int]]
[define,persons,@person*]
[define,db,[people:@persons]]
```

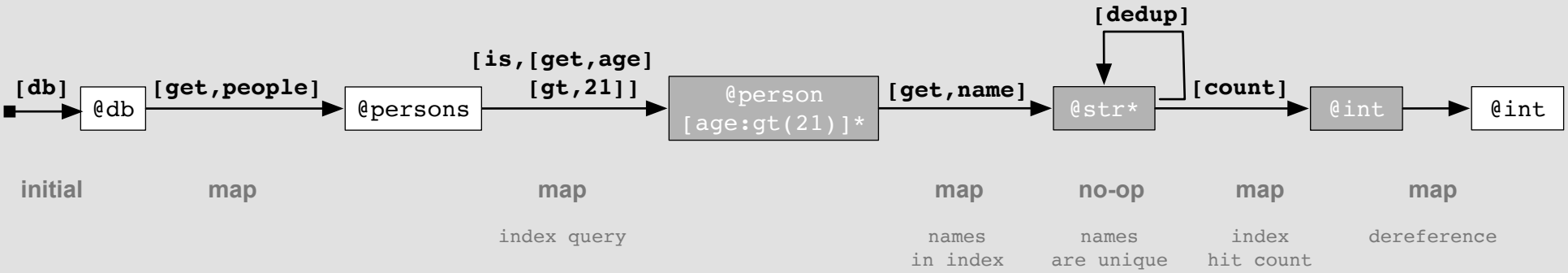
```
CREATE TABLE people (
  name varchar(255) NOT NULL,
  age  int NOT NULL
)
```

Data Access Path

Primary and Secondary Structures




```
[db][get,people]
    [is,[get,age][gt,21]]
    [get,name]
    [dedup]
    [count]
```




```
[define,person,[name:@str,age:@int]]
[define,persons,@person*
 -> [dedup,[get,name]]                =>
 -> [is,[get,age][@rel~x,@int~y]] => [ref,@person[age:~x(~y)]*
  -> [get,name]                       => [ref,@str* <= […]
   -> [dedup]                         =>
   -> [count]                         => [ref,@int  <= […]]]]]
[define,db,[people:@persons]]
```

runtime instruction rewrites

```
CREATE TABLE people (
  name varchar(255),
  age  int NOT NULL,
  PRIMARY KEY(name),
  CREATE INDEX idx ON people(age)
)
```

Common model-ADTs

Primary Structure

[model]
domain of discourse

KEY-VALUE STORE

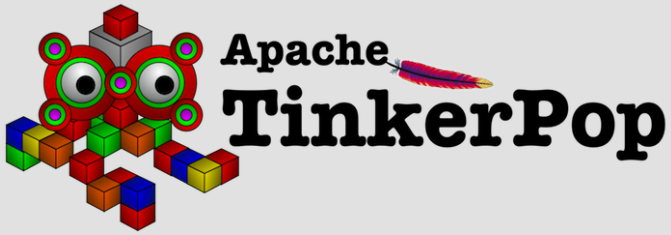
```
[model,kv=>mm,
 [define,k,@num|@str]
 [define,v,@obj]
 [define,kv,[@k,@v]]
 [define,db,@kv*]]
```

DOCUMENT DATABASE

```
[model,doc=>mm,
 [define,dval,@bool|@num|@str|@dobj|@dlist]
 [define,dlist,[(@dval)*]]
 [define,dobj,[(@str:@dval)*]]
 [define,doc,@dobj&[_id:@str]]
 [define,collection,@doc*]
 [define,db,[(@str:@collection)*]]]
```

PROPERTY GRAPH DATABASE

```
[model,pg=>mm,
 [define,properties,[(@str:@str|@num|@bool)*]]
 [define,element,@properties&[id:@obj,label:@str]]
 [define,edge,@element&[outV:@vertex,inV:@vertex]]
 [define,vertex,@element&[inE:@edge*,outE:@edge*]]
 [define,db,@vertex*]]
```



RELATIONAL DATABASE

```
[model,rdb=>mm,
 [define,value,@bool|@num|@str]
 [define,row,[(@str:@value)*]]
 [define,table,@row*]
 [define,db,[(@str:@table)*]]]
```

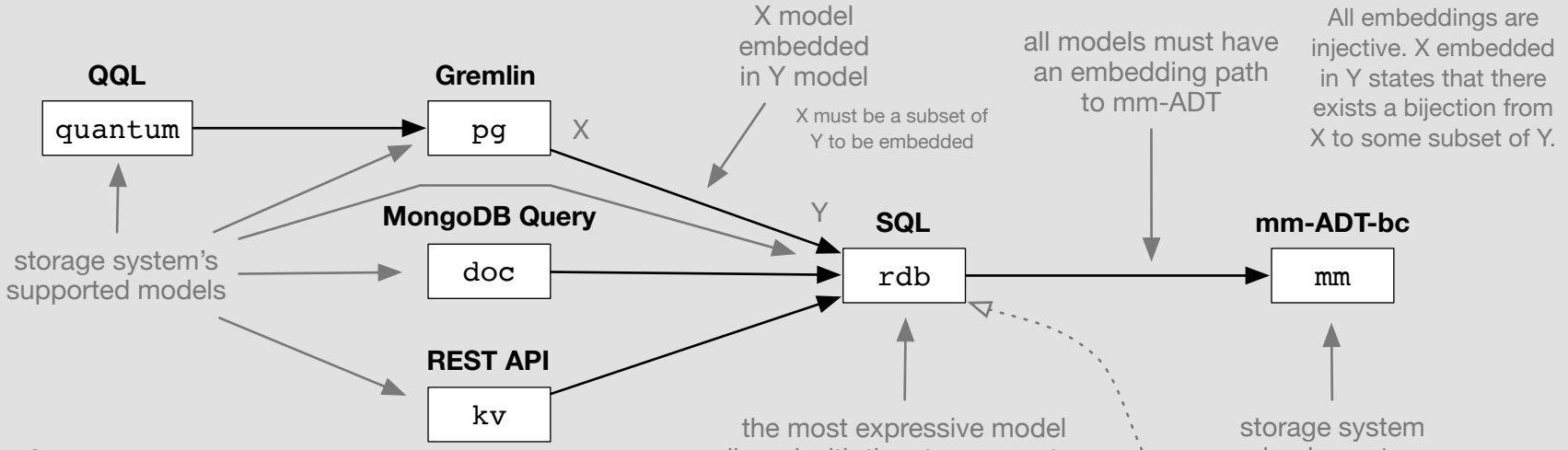
A model-ADT is an abstract datatype modeled/*embedded* in mm-ADT.

Multiple model-ADTs

Models, Embeddings, and Queries

model-ADT definition bytecode

```
[model,    rdb=>mm, [define,…][define,…][define,…]]
[model,     pg=>rdb,[define,…][define,…][define,…]]
[model,    doc=>rdb,[define,…][define,…][define,…]]
[model,     kv=>rdb,[define,…][define,…][define,…]]
[model,quantum=>pg, [define,…][define,…][define,…]]
```



model-ADT query bytecode

property graph query (Gremlin=>pg)

```
[db][is,[get,label][eq,person]]
    [get,outE]
    [is,[get,label][eq,phones]]
    [get,inV]
    [get,home]
```

key/value query (REST=>kv)

```
[db][get,people]
    [get,phones]
    [get,home]
```

document query (MongoDB Query=>doc)

```
[db][get,people]
    [get,phones]
    [get,home]
```

relational query (SQL=>rdb)

```
[db][get,people]
    [at,[db][get,phones],
      [get,person_id][is,[eq,[get,id]]]]
    [get,home]
```

A query language compiles to the most aligned model-ADT. The generated bytecode is translated to mm-ADT via rewrites.

Part 2

Universal Processing Model

mm-ADT Bytecode

Query Instructions

```
[define,person,[name:@str,age:@int]]
[define,db,[people:@person*]]
```

What are the ages of the people named marko?

```
[db]
[get,people]
[is,
    [get,name]
    [eq,marko]]
[get,age]
```

```
:@obj{0} => @db
:@db     => @person*
:@person => @person?
 :@person => @str
 :@str    => @bool
:@person => @int
```

mm-ADT Bytecode

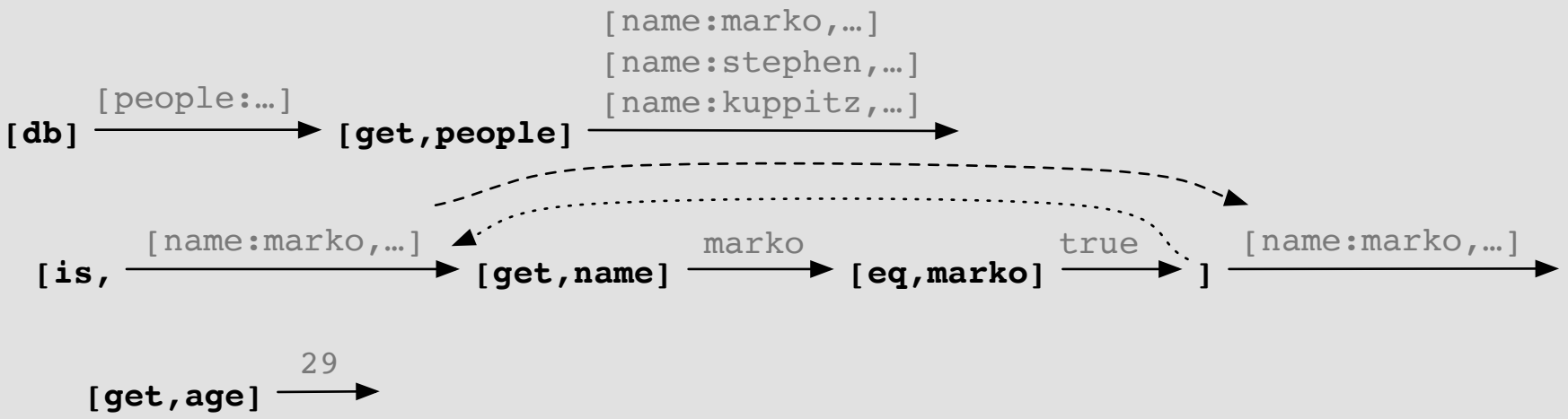
Query Execution

What are the ages of the people named marko?

```
[db]
[get,people]
[is,
    [get,name]
    [eq,marko]]
[get,age]
```

Universal Computing via Streams

```
:@obj{0} => @db
:@db     => @person*
:@person => @person?
 :@person => @str
 :@str    => @bool
:@person => @int
```



Stream Ring Theory

An Algebraic Ring for Stream Computing

Turing Complete
https://zenodo.org/record/2565243

`f*g`	“Multiplication” is instruction composition
`f+g`	“Addition” is stream branching (clone/split)
`-f`	“Negative” inverts the object quantifier
`0`	`[_]{0}: @obj* => @obj{0}`
`1`	`[_]{1}: @obj* => @obj*`

not XOR

Axioms

`f+(g+h) = (f+g)+h`	addition associativity
`f*(g*h) = (f*g)*h`	multiplication associativity
`f*(g+h) = (f*g)+(f*h)`	left distributive
`(f+h)*g = (f*g)+(h*g)`	right distributive
`f*0 = 0`	multiplicative zero
`f*1 = f`	multiplicative one
`f+0 = f`	additive zero
`f+1 = f+1`	additive one
`f-f = f+(-f) = 0`	negative

The stream ring is the product of the quantifier ring and the instruction ring.

Stream Ring Theory

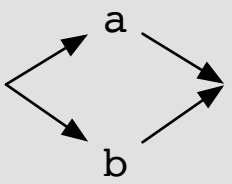
Addition, Multiplication, and Distribution

```
a * b * c
```

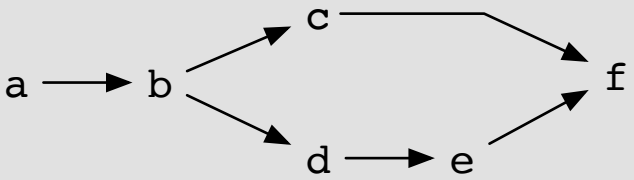
```
@inst~a * @inst~b = ~a~b
```


```
a + b
```

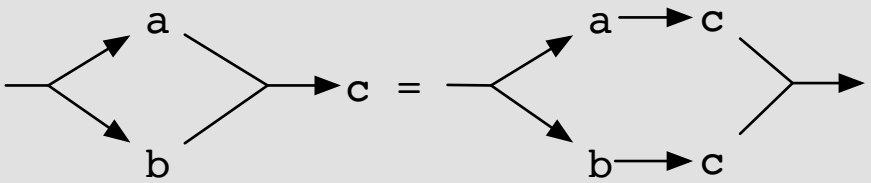
```
@inst~a + @inst~b = [branch,~a,~b]
```



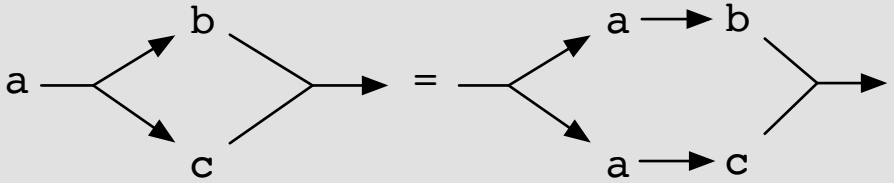

```
a * b * (c + (d*e)) * f
```




```
(a+b)*c = (a*c)+(b*c)
```




```
a*(b+c) = (a*b)+(a*c)
```




```
@inst~a{x,y} * @inst~b{w,z} = ~a~b{x*w,y*z}
@inst~a{x,y} + @inst~b{w,z} = [branch,~a,~b]{x+w,y+z}
```

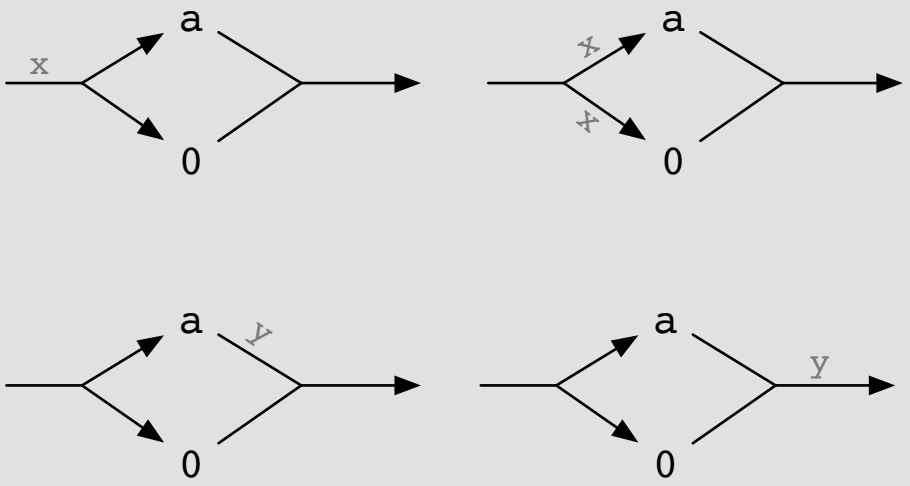
Instruction and object quantifiers must be from the same algebraic ring as instruction quantifiers modulate object quantifiers.

Stream Ring Theory

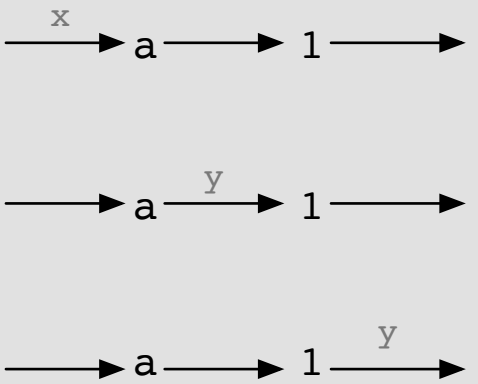
Additive and Multiplicative Identities

```
a(x) = y        filter
   0 = [_]{0}
   1 = [_]{1}
identity
```

$a + 0 = a$



$a * 1 = a$




```
a: x => x?
b: x => x?
```

multi-set union $a \uplus b$

```
  a + b    = x{0,2}
x(a + b)   =
 xa + xb   =

-------------------
  0 + 0    = 0
  0 + x    = x
  x + 0    = x
  x + x    = x{2}
     x{0,2}
```

set union $a \cup b$

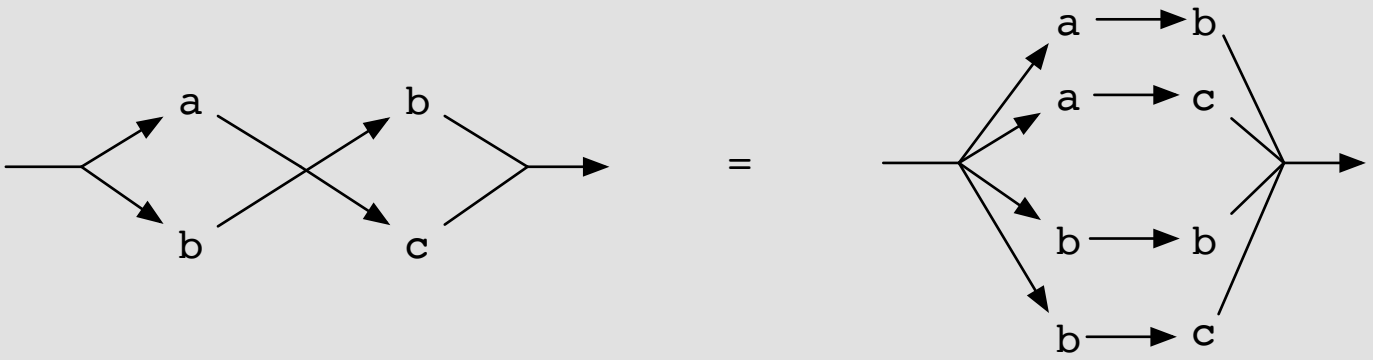
```
  a + b  -    ab  = x?
x(a + b  -    ab) =
 xa + xb - x(ab) =
 xa + xb + -x(ab) =

-------------------------
  0 + 0  +     0  = 0
  x + 0  +     0  = x
  0 + x  +     0  = x
  x + x  +    -x  = x
     x{0,1} => x?
```

Stream Ring Theory

Binomial and Multinomial Expansions

FOIL method

$$(a+b)*(b+c) = ab + ac + b^2 + bc$$

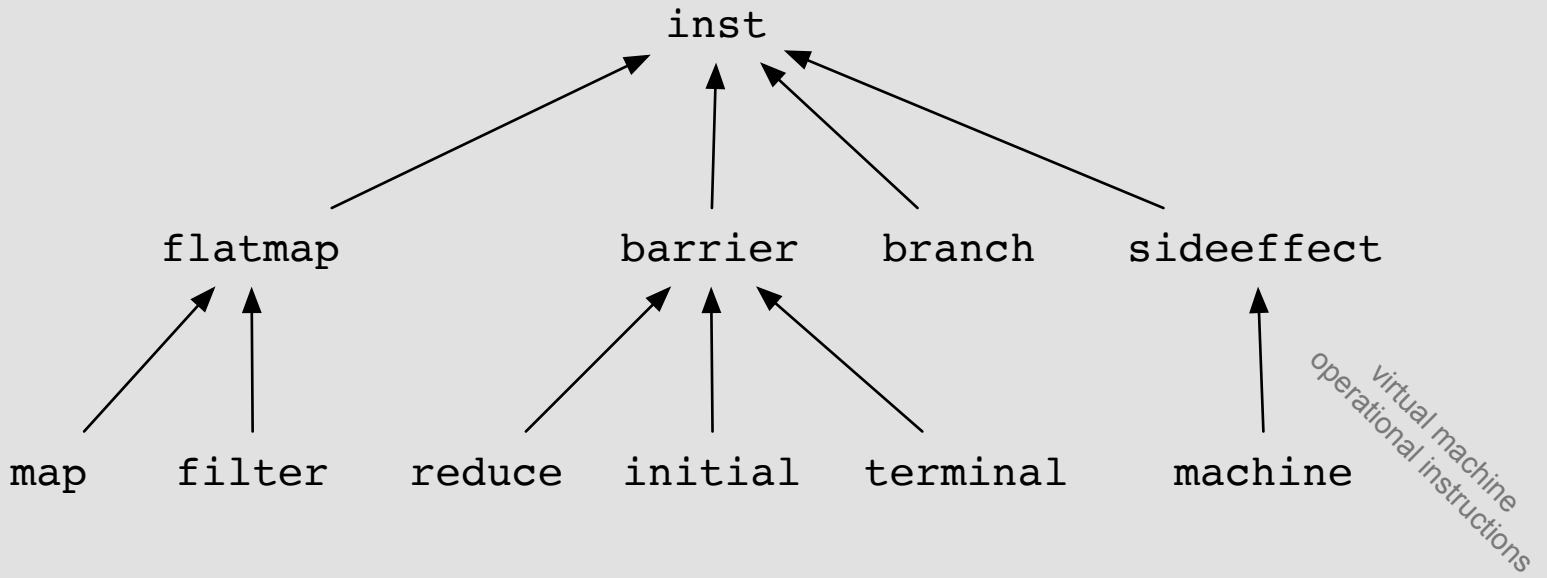


Part 3

Universal Instruction Set

mm-ADT Instruction Set

Instruction Types




```
flatmap : @obj     =>  @obj*
map     : @obj     =>  @obj
filter  : @obj     =>  @obj?
barrier : @obj*    =>  @obj*
reduce  : @obj*    =>  @obj
initial : @obj{0}  =>  @obj*
terminal: @obj*    =>  @obj{0}
```

ring
ring
commutative ring
ring
near-ring

Type checking includes both object type and quantifier.

mm-ADT Instruction Set

Instruction Types

MAP

```
[get,@obj]: @rec => @obj
[path]    : @obj => @list
[type]    : @obj => @obj
…
```

FILTER

```
[_]            : @obj* => @obj*
[and,@obj{2,}]: @obj  => @obj?
[or,@obj{2,}] : @obj  => @obj?
[is,@bool]    : @obj  => @obj?
…
```

REDUCE

```
[count]: @obj* => @int
[max]  : @num* => @num
[sum]  : @num* => @num
…
```

SIDE-EFFECT

```
[put,@obj,@obj] : @seq  => @seq
[drop,@obj]     : @seq  => @seq
[fit,@int?,@obj]: @list => @list
…
```

BARRIER

```
[dedup,@obj*]          : @obj* => @obj*
[order,[lt|gt,@obj]*]: @obj* => @obj*
[range,@int,@int]     : @obj* => @obj*
…
```

INITIAL

TERMINAL

MACHINE

```
[ref,@obj*]: @obj => @obj*
```

BRANCH

```
[repeat,@inst{1,3}]     : @obj => @obj*
[ifelse,@bool,@inst{2}]: @obj => @obj*
[choose,[@bool,@inst]+]: @obj => @obj*
[coalesce,@inst{2,}]    : @obj => @obj*
…
```

Query Language Compilation

Generating model-ADT Bytecode

```
SELECT name FROM people WHERE age < 29
```

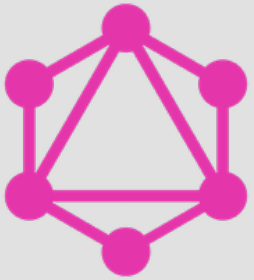
```
[db][get,people]
    [is,[get,age][lt,29]]
    [get,name]
```


```
g.V(1).out('created')
      .in('created')
      .hasId(neq(1))
      .groupCount().by('name')
```

```
[db][get,V]
    [is,[get,id][eq,1]]
    [get,outE]
    [is,[get,label][eq,created]]
    [get,inV]
    [get,inE]
    [is,[get,label][eq,created]]
    [get,outV]
    [is,[get,id][neq,1]]
    [group,[get,name],[_],[count]]
```

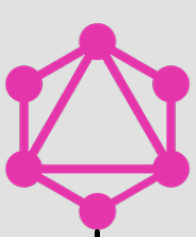
```
person(name:marko){friends{name}}
```

```
[db][get,people]
    [is,[get,name][eq,marko]]
    [get,friends]
    [get,name]
```



Query Language Compilation

Query Language and Storage System Decoupling



mm-ADT objects
or
language translation

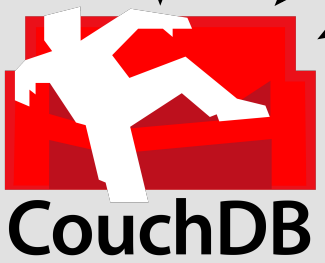
Query languages compile to their respective model-ADT bytecode specification.

`[model,doc=>mm]` `[model,rdb=>mm]` `[model,pg=>mm]`

doc=>pg
doc=>rdb
rdb=>doc
rdb=>pg
pg=>doc
pg=>rdb

model-ADT *embeddings* use rewrite rules to encode one model-ADT within another.

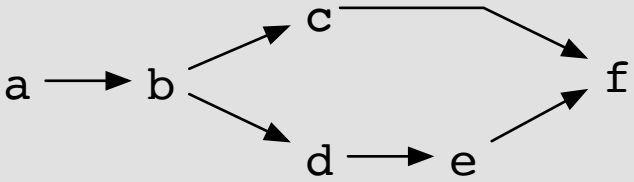
This enables query bytecode intended for one model-ADT to execute against another model-ADT.



mm-ADT Components

Storage System, Processing Engine, and Query Language


```
[model,wc=>mm,…]
[model,kv=>mm,…]
[model,pg=>mm,…]
```



```
[db][get,outE]
    [get,inV]
    [get,name]
```

An mm-ADT **storage system** publishes the model-ADTs it supports (and is optimized for). Within a particular model-ADT context, the database will produce respective mm-ADT objects for the processor to consume.

For all unsupported models, a model-ADT *embedding* can be used to translate to a supported model-ADT. However, while semantically correct, the storage system might lack appropriate secondary structures.

An mm-ADT **processing engine** is able to accept arbitrary mm-ADT bytecode and generate an execution pipeline that can read/write mm-ADT objects to/from the underlying mm-ADT compliant storage system.

mm-ADT processing engines are agnostic to the model-ADT bytecode encoding. They must faithfully implement every instruction in the mm-ADT instruction set.

An mm-ADT **query language** has a compiler to a specific model-ADT bytecode specification.

The processing engine translates the language compiler's model-ADT bytecode into an execution pipeline. At runtime, the processor and storage system communicate via mm-ADT objects to ultimately yield the answer to the query.

mm-ADT Compliant Components

Storage System, Processing Engine, and Query Language

```
compilation: @str                                  => @inst*
             g.V().out().values('name')            => [db][get,outE][get,inV][get,name]

rewrite    : @inst*                                => @inst*
             [db][get,outE][get,inV][get,name]     => [db][get,name]

evaluation : @inst*                                => @obj*
             [db][get,name]                        => marko,kuppitz,stephen
```

Common model-ADTs

Key/Value Store

mm-ADT
kv=>mm Bytecode

GET/PUT
semantics

```
[db][count]                                  // 0
[db][put,[a,[name:marko,age:29]]]
    [put,[b,[name:vadas,age:27]]]
    [put,[c,[name:josh,age:32]]]
    [put,[d,[name:peter,age:10]]]
[db][put,[d,[name:peter,age:35]]]            // updated d value
[db][count]                                  // 4
[db][is,[get,0][eq,a]]
    [get,1]                                  // [name:marko,age:29]
[db][group,[get,1]
           [get,age]
           [ifelse,[is,[gt,30]],
                   [map,old],
                   [map,young]],
            [map,1],
            [sum]]                           // [old:2,young:2]
```

Key/Value Store

Primary and Secondary Structures

```
[model,kv=>mm,
 [define,k,@num|@str]
 [define,v,@bool|@num|@str|@list|@rec]
 [define,kv,[@k,@v]
  -> [put,0,@k]              => [error]
  -> [drop,0|1]              => [error]]
 [define,db,kv*
  -> [put,@kv[@k~k,@v~v]]  => [coalesce,
                                 [is,[get,0][eq,~k]][put,1,~v],
                                 [put,[~k,~v]]]
  -> [dedup,[get,0]]         =>
  -> [count]                 => [ref,@int <= [db][count]]
  -> [is,[get,0][eq,@k~k]] => [ref,@kv? <= [db][is,[get,0]
                                                [eq,~k]]
  -> [group,@inst*{3}~mr]  => [ref,@rec <= [db][group,~mr]]]]
```

sorted keys

key counter

key index

map-reduce framework

MongoDB
JSON QueryDoc

```
db.people.insertOne({name:marko,
                     age:29,
                     projects:[{name:lop,lang:java}],
                     knows:[{name:josh},{name:vadas}]})
db.people.find({"knows.name":{$regex:/^j.*/}},{name:1,age:1})
```

Document Database

mm-ADT
doc=>mm Bytecode

```
[db][get,people]
    [put,[name:marko,
          age:29,
          projects:[[name:lop,lang:java]],
          knows:[[name:josh],[name:vadas]]]]
[db][get,people]
    [is,[get,knows][get,name][regex,'^j.*']]
    [ref,[name:@string,age:@int]]
```

Document Database

Primary and Secondary Structures

```
[model,doc=>mm,
 [define,dval,@bool|@num|@str|@dobj|@dlist
  -> [is,[get,@str~x][eq,@dval~y]] => [ref,@dval? <= […]]]
 [define,dlist,[(@dval)*]]
 [define,dobj,[(@str:@dval)*]]
 [define,doc,@dobj&[_id:@str]]
 [define,collection,@doc*
  -> [dedup,[get,_id]]              =>
  -> [is,[get,_id][eq,@str~x]]      => [ref,@doc? <= […][is,[get,_id][eq,~x]]]
  -> [count]                        => [ref,@int  <= […][count]]
  -> [group,@inst*{3}~mr]           => [ref,@rec  <= […][group,~mr]]]
 [define,db,[(@str:@collection)*]]]
```

recursive pattern matching

index by _id

doc counter

map-reduce framework

Property Graph Database

Property Graph Database

Primary and Secondary Structures

```
[model,pg=>mm,
 [define,tokens,id|label|inE|outE|outV|inV]
 [define,properties,[(@str&not(tokens):@str|@num|@bool)*]]
 [define,element,@properties&[id:@obj~x,label:@str]
  -> [drop,id|label]                  => [error]
  -> [put,id|label,@obj]              => [error]]
 [define,vertex,@element&[inE:@edge*,outE:@edge*]    <= [db][is,[get,id][eq,~x]]
  -> [get,outE]                       => [ref,@edge*    <= […][get,outE]
   -> [put,@edge~x]                   => [branch,[put,~x],[get,inV][get,inE][put,~x]]
   -> [is,[get,@str~y][eq,@obj~z]] => [ref,@edge*    <= […][is,[get,~y][eq,~z]]
    -> [count]                        => [ref,@int      <= […][count]]]
   -> [is,[get,label][eq,@str~y]]  => [ref,@edge*    <= […][is,[get,label][eq,~y]]
    -> [count]                        => [ref,@int      <= […][count]]
    -> [get,inV]                      => [ref,@vertex*  <= […][get,inV]
      -> [get,(label|id)~y]           => [ref,@str|@obj <= […][get,~y]]
      -> [count]                      => [ref,@int      <= […][count]]]]]]
 [define,edge,@element&[outV:@vertex,inV:@vertex]
  -> [put,outV|inV,@vertex]           => [error]
  -> [drop,outV|inV]                  => [error]]
 [define,db,@vertex*                                   <= [db]
  -> [dedup,[get,id]]                 =>
  -> [get,outE]                       => [ref,@edge*    <= […][get,outE]
   -> [dedup,[get,id]]                => ]
  -> [is,[get,id][eq,@obj~x]]     => [ref,@vertex?  <= […][is,[get,id][eq,~x]]]
  -> [count]                          => [ref,@int      <= […][count]]
  -> [get,(inE|outE)~x]               => [ref,@edge*    <= […][get,~x]
   -> [count]                         => [ref,@int      <= […][count]]
   -> [is,[get,label][eq,@str~y]]  => [ref,@edge*    <= […][is,[get,label][eq,~y]]
    -> [count]                        => [ref,@int      <= […][count]]]]
```

vertex-centric edge property indices

vertex-centric edge label indices

adjacent vertex id and label denormalization

vertex count

edge count

edge count by label

RDF Triple Store

RDF Triple Store

Primary and Secondary Structures

```
[model,rdf=>mm,
 [define,uri,@str&regex('//^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?')]
 [define,lit,@str|@num|@bool]
 [define,bnode,@str&regex('_:.*')]
 [define,s,@uri|@bnode]
 [define,p,@uri]
 [define,o,@uri|@bnode|@lit]
 [define,stmt,[s:@s,p:@p,o:@o]
  -> [put,s|p|o,@object] => [error]
  -> [drop,s|p|o]        => [error]]
 [define,db,stmt*
  -> [dedup]                  =>
  -> [count]                  => [ref,@int   <= […][count]
  -> [is,[get,s][eq,@s~x]]    => [ref,@stmt? <= […][is,[get,s][eq,~x]]
   -> [is,[get,p][eq,@p~y]]   => [ref,@stmt? <= […][is,[get,s][eq,~x]][is,[get,p][eq,~y]]]
   -> [at,[db],[is,…]         => [ref,@stmt? <= […]
   -> [is,[get,o][eq,@p~y]]   => [ref,@stmt? <= […][is,[get,s][eq,~x]][is,[get,o][eq,~y]]]]
  -> [is,[get,p][eq,@p~x]]    => [ref,@stmt? <= […][is,[get,p][eq,~x]]
   -> [dedup]
    -> [count]                => [ref,@int   <= […][count]]]
  -> [is,[get,o][eq,@o~x]]    => [ref,@stmt? <= […][is,[get,o][eq,~x]]]]
```

statement count

statements indexed by subject

statements indexed by subject/predicate

unique predicate count

denormalized length-2 paths

Next Generation Models

mm-ADT is for cluster-oriented, general purpose computing.

Credits

Marko A. Rodriguez
Daniel Kuppitz
Stephen Mallette

Patronage from RReduX and DataStax