Graph Operations With MongoDB

Charles Sarrazin

Senior Consulting Engineer, MongoDB

Graph Operations With MongoDB

Charles Sarrazin

Senior Consulting Engineer, MongoDB

Agenda

01 MongoDB Introduction

02 Graph Use & Concepts

03 New Lookup Operators

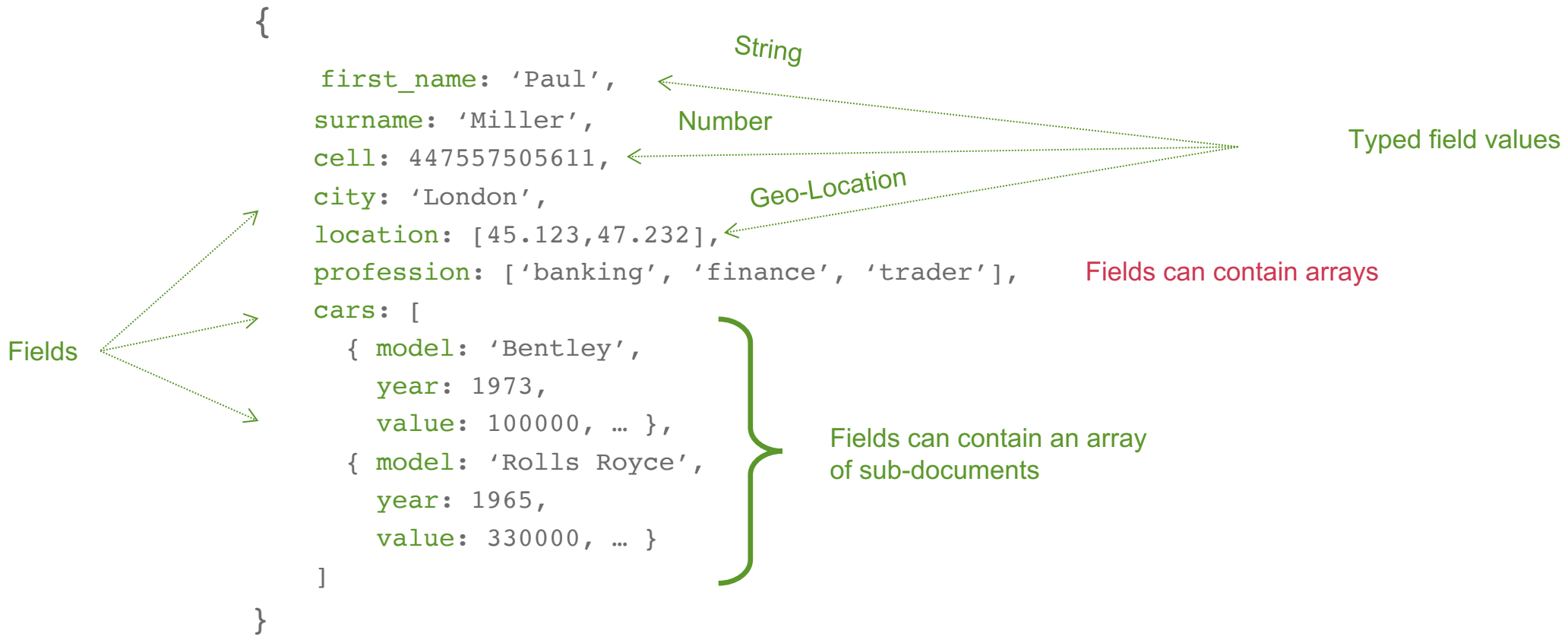
04 Example Scenarios

05 Design & Performance Considerations

06 Wrap-up

MongoDB Introduction

Documents


{
first_name: 'Paul',
surname: 'Miller',
cell: 447557505611,
city: 'London',
location: [45.123,47.232],
profession: ['banking', 'finance', 'trader'],
cars: [
{ model: 'Bentley',
year: 1973,
value: 100000, … },
{ model: 'Rolls Royce',
year: 1965,
value: 330000, … }
]
}
String
Number
Geo-Location
Typed field values
Fields
Fields can contain arrays
Fields can contain an array
of sub-documents

Query Language

```
db.collection.find({'city':'London'})
db.collection.find({'profession':{'$in':['banking','trader']}},{'surname':1,'profession':1})
db.collection.find({'cars.year':{'$lte':1968}}).sort({'surname':1}).limit(10)
db.collection.find({'cars.model':'Bentley','cars.year':{'$lt':1966}})
db.collection.find({'cars':{'$elemMatch':{'model':'Bentley','year':{'$lt':1966}}}})
db.collection.find({'location':{'$geoWithin': { '$geometry': {
            'type': 'Polygon',
            coordinates: [ <array-of-coordinates> ]
}}}})
```

Secondary Indexes

compound, geospatial, text, multikey, hashed,
unique, sparse, partial, TTL

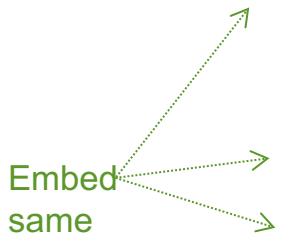
Query Language

```
db.collection.aggregate ( [
 {$match:{'profession':{'$in':['banking','trader']}}},
 {$addFields:{'surnameLower':{$toLower:"$surname"},'prof':{$ifNull:["$prof","Unknown"]}},
 {$group: { ... } },
 {$sort: { ... } },
 {$limit: { ... } },
 {$match: { ... } },
 ...
] )
```

Aggregation pipeline

Schema Design

```
{
    first_name: ‘Paul’,
    surname: ‘Miller’,
    cell: 447557505611,
    city: ‘London’,
    location: [45.123,47.232],
    profession: [‘banking’, ‘finance’, ‘trader’],
    cars: [
      { model: ‘Bentley’,
        year: 1973,
        value: 100000, … },
      { model: ‘Rolls Royce’,
        year: 1965,
        value: 330000, … }
    ]
}
```



Schema Design

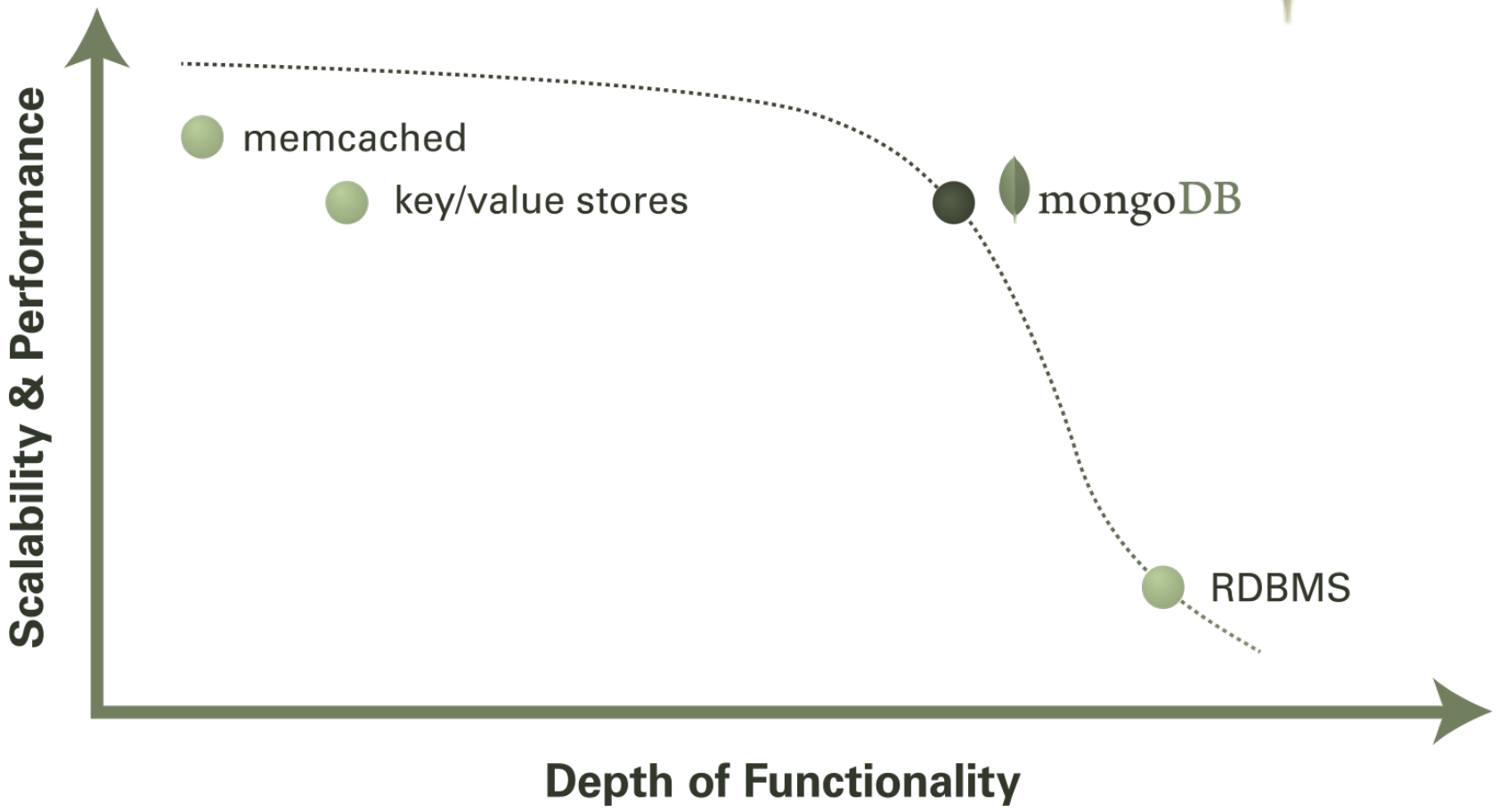
```
{
    first_name: ‘Paul’,
    surname: ‘Miller’,
    cell: 447557505611,
    city: ‘London’,
    location: [45.123,47.232],
    profession: [‘banking’, ‘finance’, ‘trader’],
    cars: [
      { model: ‘Bentley’,
        year: 1973,
        value: 100000, … },
      { model: ‘Rolls Royce’,
        year: 1965,
        value: 330000, … }
    ]
}
```

Embed same document

```
{
    first_name: ‘Paul’,
    surname: ‘Miller’,
    cell: 447557505611,
    city: ‘London’,
    location: [45.123,47.232],
    profession: [‘banking’, ‘finance’, ‘trader’]
}
cars:
   { owner_id: 146
     model: ‘Bentley’,
     year: 1973,
     value: 100000, … },
   { owner_id: 146
     model: ‘Rolls Royce’,
     year: 1965,
     value: 330000, … }
```

Separate Collection

with reference


Scalability & Performance
memcached
key/value stores
mongoDB
RDBMS
Depth of Functionality

Functionality Timeline

2.0 – 2.2

- Geospatial Polygon support
- Aggregation Framework

2.4 – 2.6

- New 2dsphere index
- Aggregation Framework efficiency optimisations
- Full text search

3.0 – 3.2

- Join functionality
- Increased geo accuracy
- New Aggregation operators
- Improved case insensitivity

3.4

- Recursive graph traversal
- Faceted search
- Multiple collations

MongoDB 3.4 - Multi-Model Database

Document
Rich JSON Data Structures
Flexible Schema
Global Scale

Graph
Graph & Hierarchical
Recursive Lookups

Relational
Left-Outer Join
Views
Schema Validation

Search
Text Search
Multiple Languages
Faceted Search

Key/Value
Horizontal Scale
In-Memory

GeoSpatial
GeoJSON
2D & 2DSphere

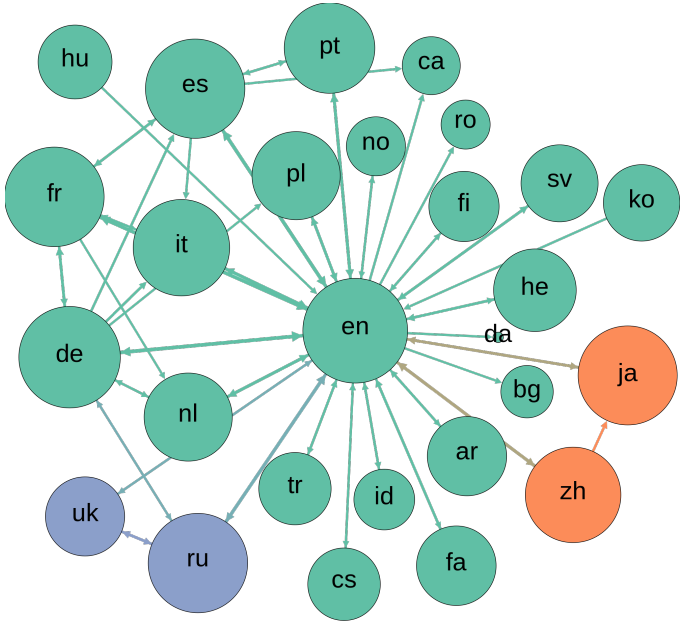
Binaries
Files & Metadata
Encrypted

Graph Use & Concepts

Common Use Cases

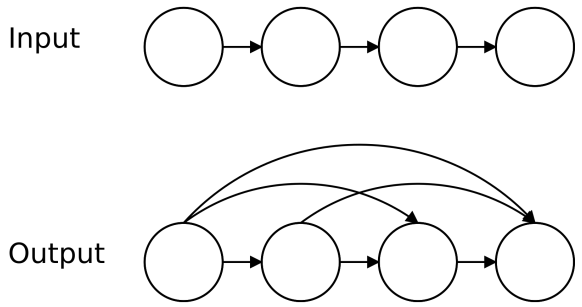
- Networks
  - Social – circle of friends/colleagues
  - Computer network – physical/virtual/application layer
- Mapping / Routes
  - Shortest route A to B
- Cybersecurity & Fraud Detection
  - Real-time fraud/scam recognition
- Personalisation/Recommendation Engine
  - Product, social, service, professional etc.

Graph Key Concepts



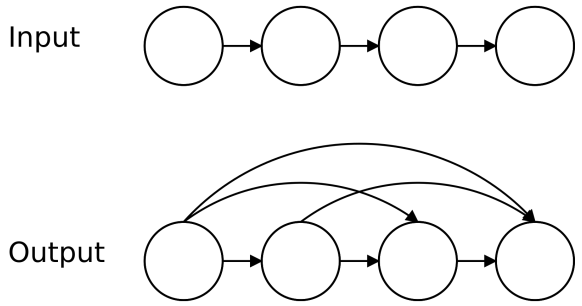
- Vertices (nodes)
- Edges (relationships)
- Nodes have properties
- Relationships have name & direction

Relational DBs Lack Relationships



- “Relationships” are actually JOINs
- Raw business or storage logic and constraints – not semantic
- JOIN tables, sparse columns, null-checks
- More JOINS = degraded performance and flexibility

Relational DBs Lack Relationships

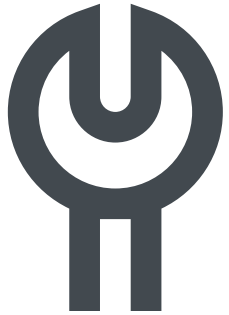


- How expensive/complex is:
  - Find my friends?
  - Find friends of my friends?
  - Find mutual friends?
  - Find friends of my friends of my friends?
  - And so on…

Native Graph Database Strengths

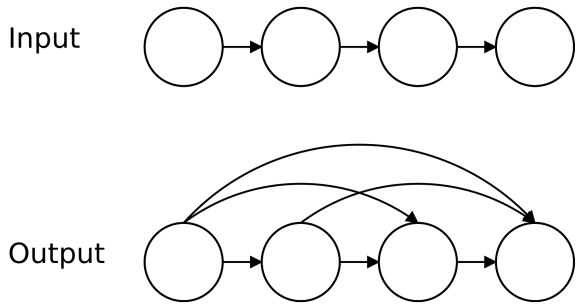
- Relationships are first class citizens of the database
- Index-free adjacency
- Nodes “point” directly to other nodes
- Efficient relationship traversal

Native Graph Database Challenges



- Complex query languages
- Poorly optimized for non-traversal queries
  - Difficult to express
  - May be memory intensive
- Less often used as System Of Record
  - Synchronisation with SOR required
  - Increased operational complexity
  - Consistency concerns

NoSQL DBs Lack Relationships

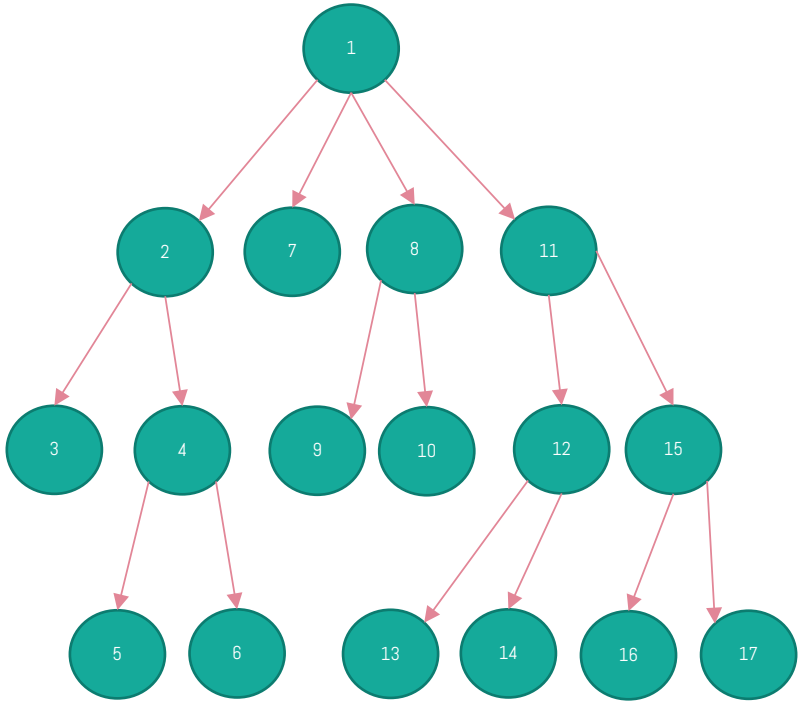


- “Flat” disconnected documents or key/value pairs
- “Foreign keys” inferred at application layer
- Data integrity/quality onus is on the application
- Suggestions re difficulty of modeling ANY relationships efficiently with aggregate stores.
- However…

Friends Network – Document Style

```
{
        _id: 0,
        name: "Bob Smith",
        friends: ["Anna Jones", "Chris Green"]
},
{
        _id: 1,
        name: "Anna Jones",
        friends: ["Bob Smith", "Chris Green", "Joe Lee"]
},
{
        _id: 2,
        name: "Chris Green",
        friends: ["Anna Jones", "Bob Smith"]
}
```

Schema Design – before $graphLookup



- Options
  - Store an array of direct children in each node
  - Store parent in each node
  - Store parent and array of ancestors
- Trade-offs
  - Simple queries…
  - …vs simple updates

Why MongoDB For Graph?

Lookup Operators

$lookup

Syntax

```
$lookup: {
        from: <target lookup collection>,
        localField: <field from the input document>,
        foreignField: <field from the target collection to connect to>,
        as: <field name for resulting array>
}
```

$graphLookup

Syntax

```
$graphLookup: {
        from: <target lookup collection>,
        startWith: <expression for value to start from>,
        connectToField: <field name in target collection to connect to>,
        connectFromField: <field name in target collection to connect from – recurse from here>,
        as: <field name for resulting array>,
        maxDepth: <max number of iterations to perform>,
        depthField: <field name for number of recursive iterations required to reach this node>,
        restrictSearchWithMatch: <match condition to apply to lookup>
}
```

Things To Note

- **startWith** value is an expression
  - Referencing **value of** a field requires the '$' prefix
  - Can do things like `{$toLower: "$name" }`
  - Handles array fields automatically

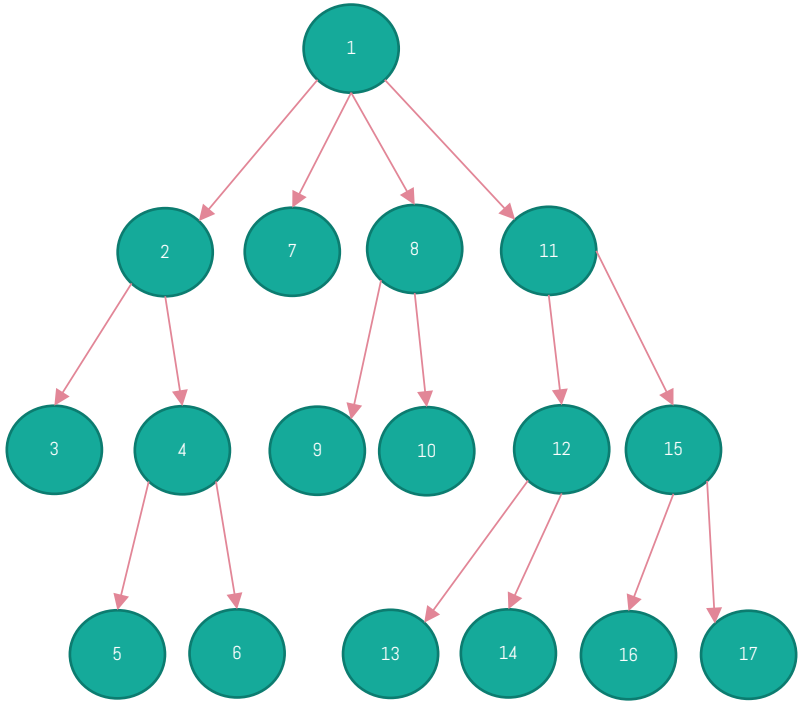
- **connectToField** and **connectFromField** take field names

- **restrictSearchWithMatch** takes a standard query expressions

Things To Note

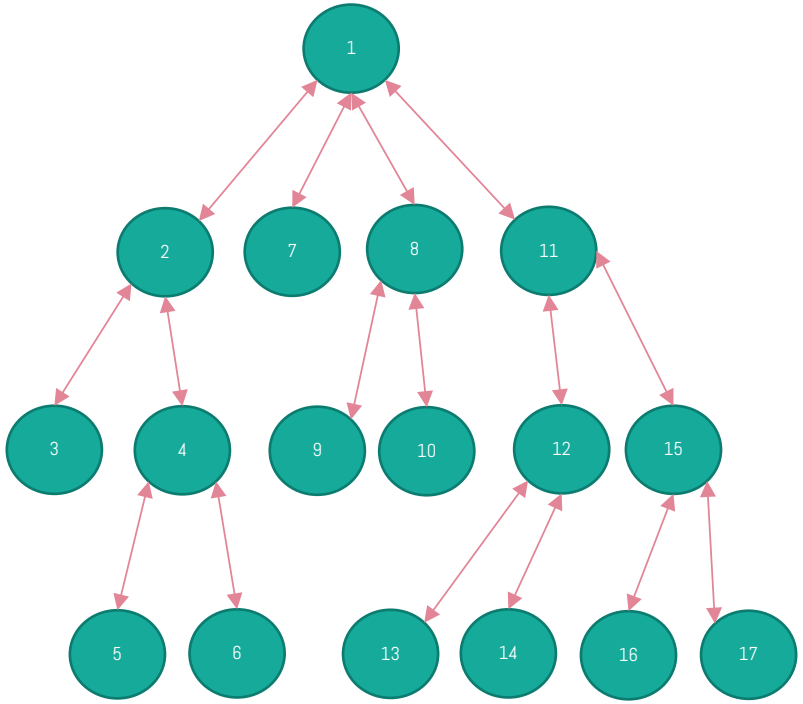
- Cycles are automatically detected
- Can be used with 3.4 views:
  - Define a view
  - Recurse across existing view ('base' or 'from')
- Can be used multiple times per Aggregation pipeline

Schema Design – before $graphLookup



- Options
  - Store an array of direct children in each node
  - Store parent in each node
  - Store parent and array of ancestors
- Trade-offs
  - Simple queries…
  - …vs simple updates

Schema Design – with $graphLookup



- Options
  - Store immediate parent in each node
  - Store immediate children in each node
- Traverse in multiple directions
  - Recurse in same collection
  - Join/recurse into another collection

“So just how suitable is MongoDB for the many varied graph use cases I have then?"

75%

of use cases*

*based on beta test user feedback

Example Scenarios

Scenario: Calculate Friend Network

```
{
        _id: 0,
        name: "Bob Smith",
        friends: ["Anna Jones", "Chris Green"]
},
{
        _id: 1,
        name: "Anna Jones",
        friends: ["Bob Smith", "Chris Green", "Joe Lee"]
},
{
        _id: 2,
        name: "Chris Green",
        friends: ["Anna Jones", "Bob Smith"]
}
```

Scenario: Calculate Friend Network

```
[
    {
        $match: { "name": "Bob Smith" }
    },
    {
        $graphLookup: {
                from: "contacts",
                startWith: "$friends",
                connectToField: "name",
                connectFromField: "friends",
                as: "socialNetwork"
         }
    },
    {
        $project: { name: 1, friends:1, socialNetwork: "$socialNetwork.name"}
    }
]
```

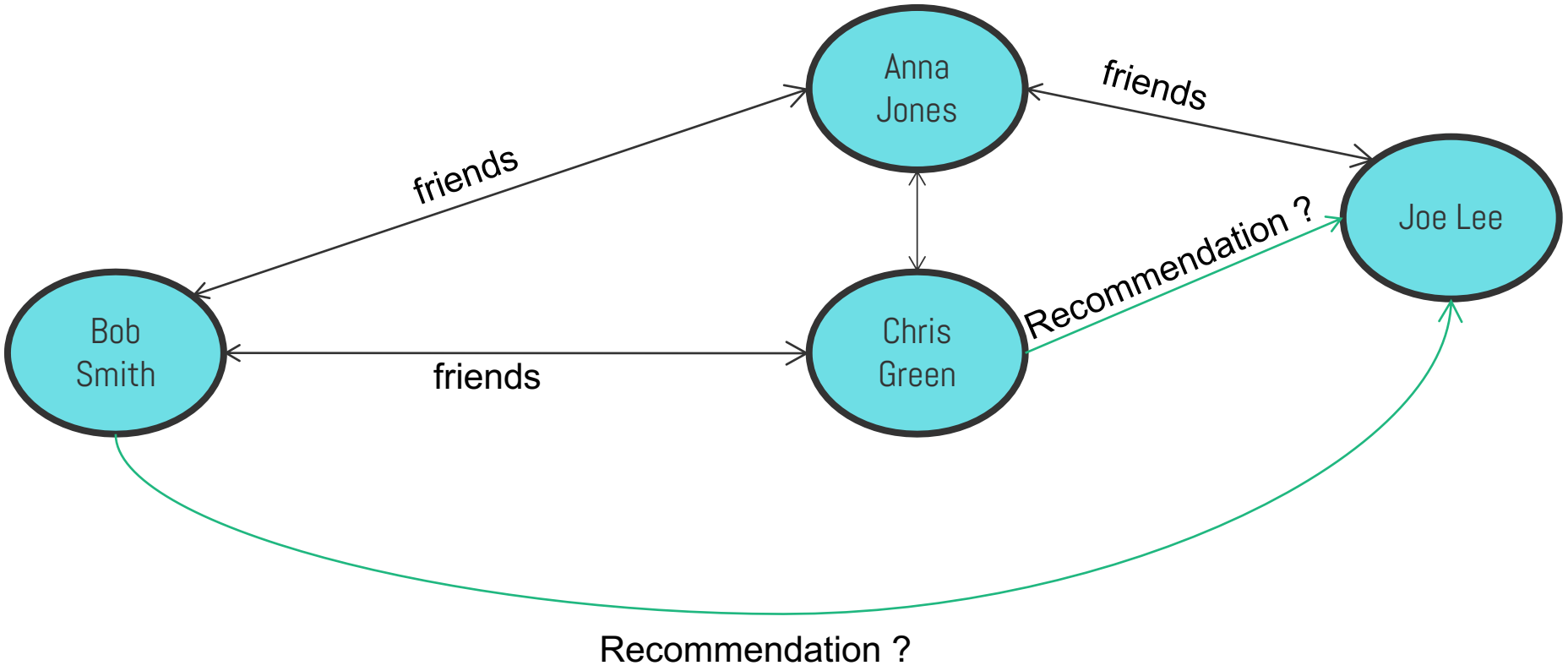
This field is an array

No maxDepth set

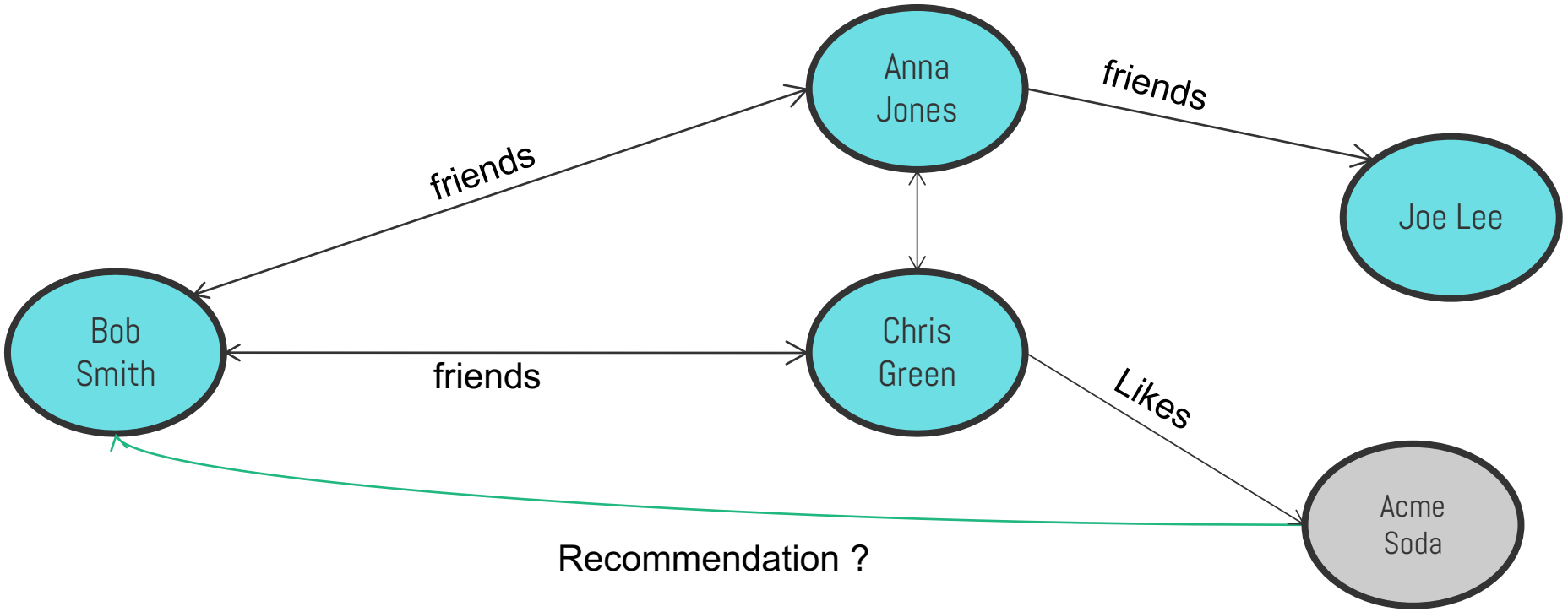
Scenario: Calculate Friend Network

```
{
        "_id" : 0,
        "name" : "Bob Smith",
        "friends" : [
                "Anna Jones",
                "Chris Green"
        ],
        "socialNetwork" : [
                "Joe Lee",
                "Fred Brown",
                "Bob Smith",
                "Chris Green",
                "Anna Jones"
        ]
}
```

Friends Network - Social

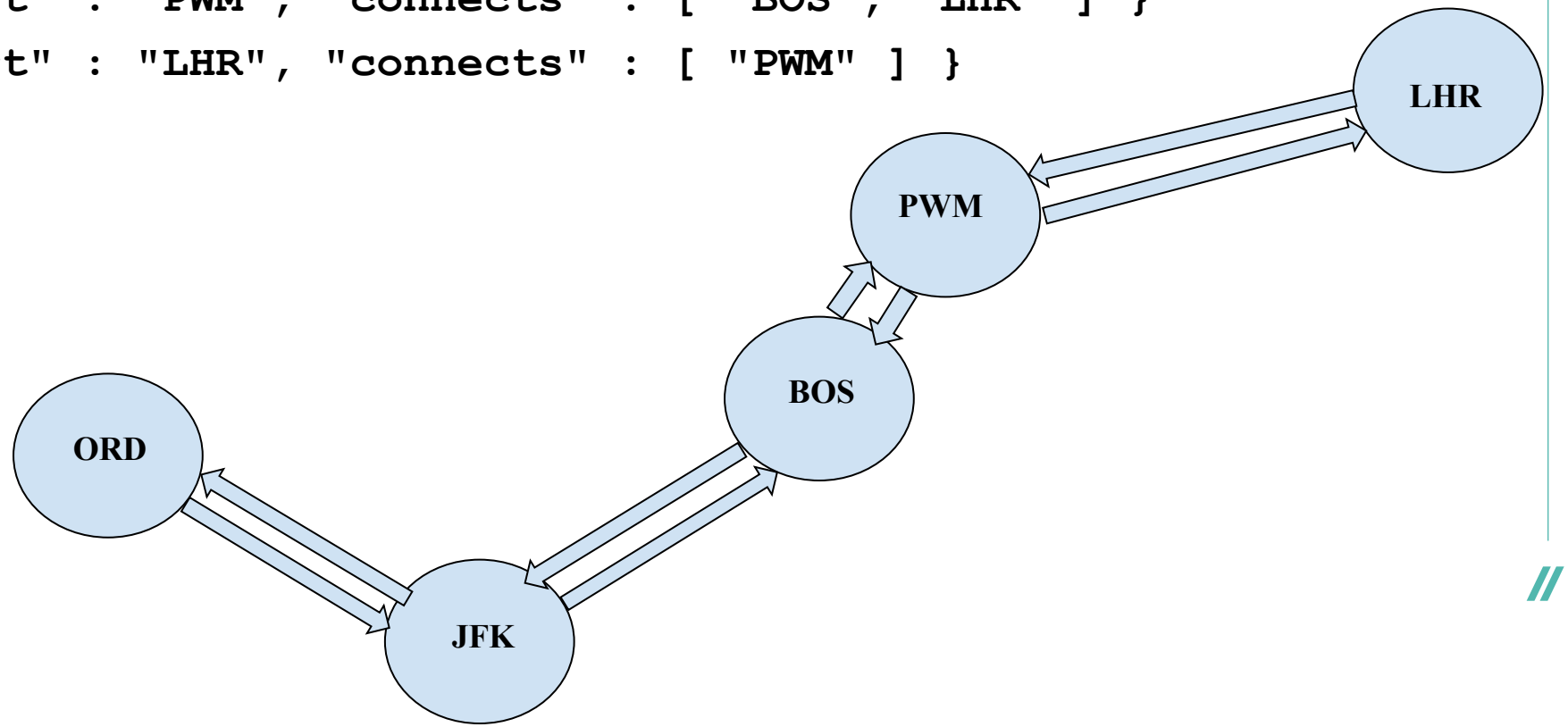


Friends Network - Social



Scenario: Determine Air Travel Options

```
{ "_id" : 0, "airport" : "JFK", "connects" : [ "BOS", "ORD" ] }
{ "_id" : 1, "airport" : "BOS", "connects" : [ "JFK", "PWM" ] }
{ "_id" : 2, "airport" : "ORD", "connects" : [ "JFK" ] }
{ "_id" : 3, "airport" : "PWM", "connects" : [ "BOS", "LHR" ] }
{ "_id" : 4, "airport" : "LHR", "connects" : [ "PWM" ] }
```



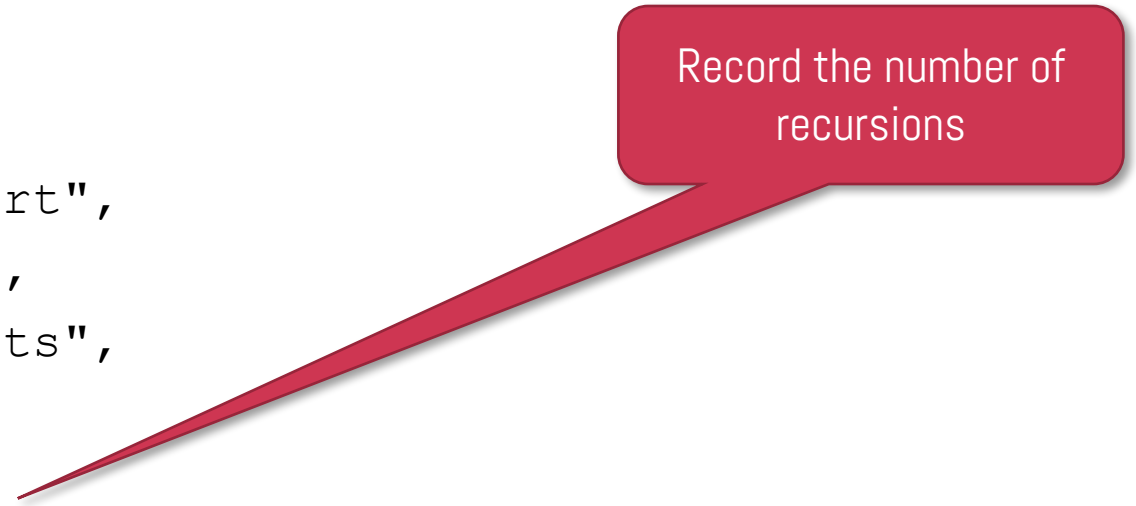
Scenario: Determine Air Travel Options

Meet Lucy

```
{ "_id" : 0, "name" : "Lucy", "nearestAirport" : "JFK" }
```

Scenario: Determine Air Travel Options

```
[
    {
       "$match": {"name":"Lucy"}
    },
    {
       "$graphLookup": {
              from: "airports",
              startWith: "$nearestAirport",
              connectToField: "airport",
              connectFromField: "connects",
              maxDepth: 2,
              depthField: "numFlights",
              as: "destinations”
        }
    }
]
```

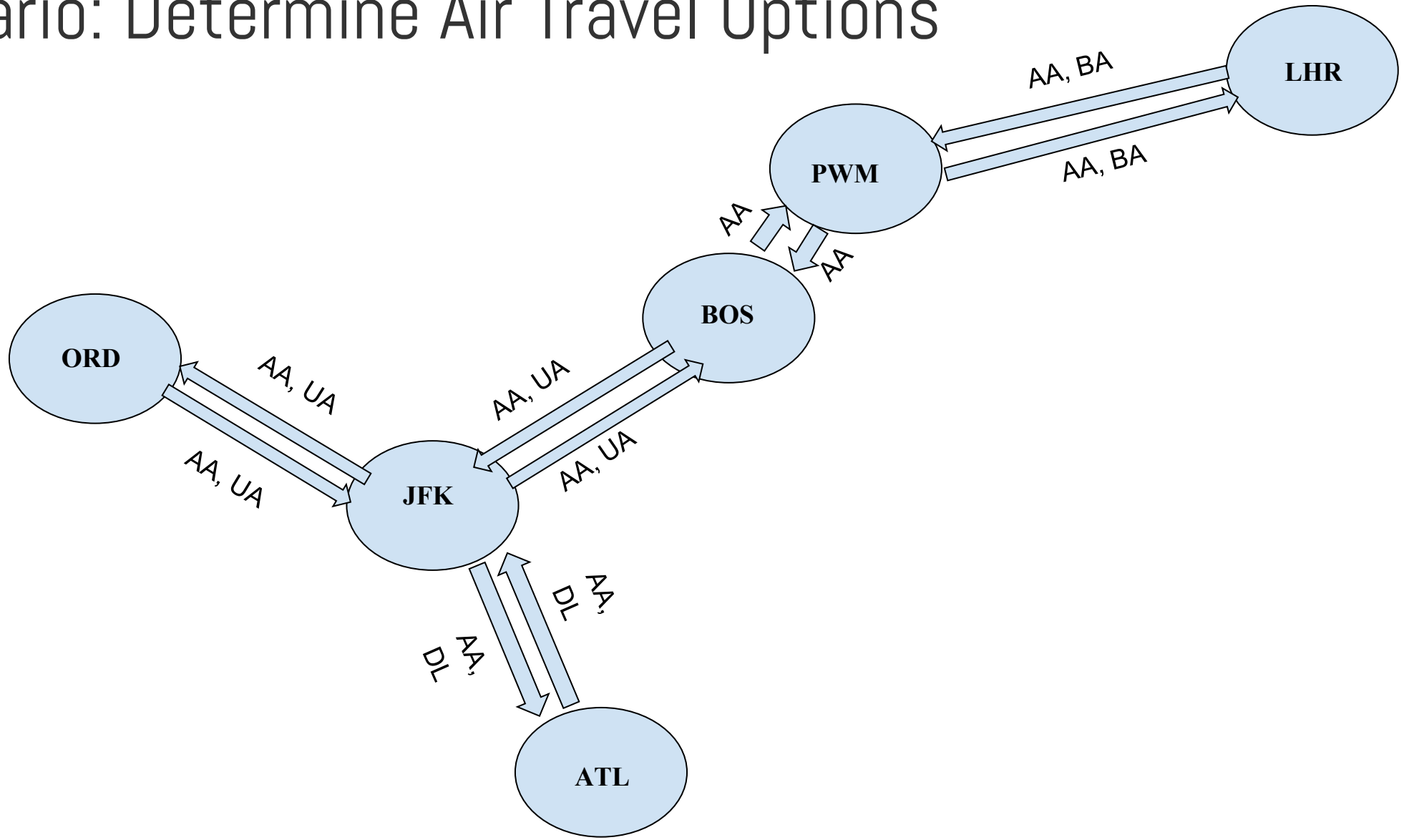


Scenario: Determine Air Travel Options

```
{
  name: "Lucy”,
  nearestAirport: "JFK",
  destinations: [
    { _id: 0, airport: "JFK", connects: ["BOS", "ORD"], numFlights: 0 },
    { _id: 1, airport: "BOS", connects: ["JFK", "PWM"], numFlights: 1 },
    { _id: 2, airport: "ORD", connects: ["JFK"], numFlights: 1 },
    { _id: 3, airport: "PWM", connects: ["BOS", "LHR"], numFlights: 2 }
  ]
}
```

How many flights this would take

Scenario: Determine Air Travel Options



Scenario: Determine Air Travel Options

```
{ "_id" : 0, "airport" : "JFK", "connects" : [
                    { "to" : "BOS", "airlines" : [ "UA", "AA" ] },
                    { "to" : "ORD", "airlines" : [ "UA", "AA" ] },
                    { "to" : "ATL", "airlines" : [ "AA", "DL" ] }] }
{ "_id" : 1, "airport" : "BOS", "connects" : [
                    { "to" : "JFK", "airlines" : [ "UA", "AA" ] },
                    { "to" : "PWM", "airlines" : [ "AA" ] } ]] }
{ "_id" : 2, "airport" : "ORD", "connects" : [
                    { "to" : "JFK", "airlines" : [ "UA","AA" ] }] }
{ "_id" : 3, "airport" : "PWM", "connects" : [
                    { "to" : "BOS", "airlines" : [ "AA" ] }] }
```

Scenario: Determine Air Travel Options

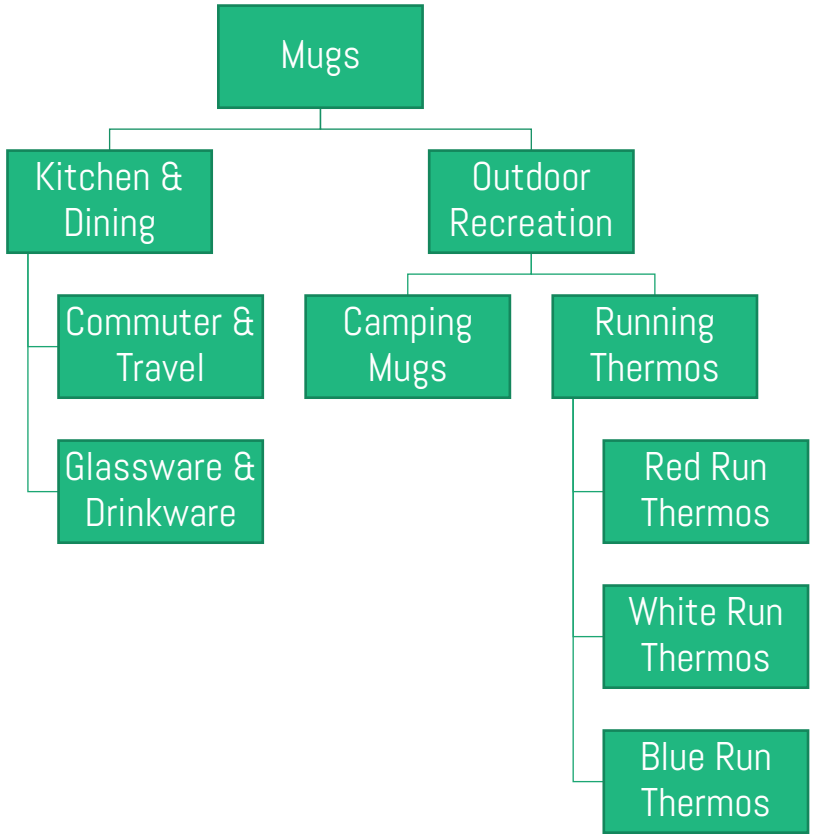
```
[
    {
       "$match":{"name":"Lucy"}
    },
    {
       "$graphLookup": {
              from: "airports",
              startWith: "$nearestAirport",
              connectToField: "airport",
              connectFromField: "connects.to”,
              maxDepth: 2,
              depthField: "numFlights”,
              restrictSearchWithMatch: {"connects.airlines":"UA"},
              as: ”UAdestinations"
       }
    }
]
```

We've added a filter

Scenario: Determine Air Travel Options

```
{
        "name" : "Lucy",
        "from" : "JFK",
        "UAdestinations" : [
            { "_id" : 2, "airport" : "ORD", "numFlights" : NumberLong(1) },
            { "_id" : 1, "airport" : "BOS", "numFlights" : NumberLong(1) }
        ]
}
```

Scenario: Product Categories




```
{ "_id" : 1, "name" : "Mugs" }
{ "_id" : 2, "name" : "Kitchen & Dining", "parentId" : 1 }
{ "_id" : 3, "name" : "Commuter & Travel", "parentId" : 2 }
{ "_id" : 4, "name" : "Glassware & Drinkware", "parentId" : 2 }
{ "_id" : 5, "name" : "Outdoor Recreation", "parentId" : 1 }
{ "_id" : 6, "name" : "Camping Mugs", "parentId" : 5 }
{ "_id" : 7, "name" : "Running Thermos", "parentId" : 5 }
{ "_id" : 8, "name" : "Red Run Thermos", "parentId" : 7 }
{ "_id" : 9, "name" : "White Run Thermos", "parentId" : 7 }
{ "_id" : 10, "name" : "Blue Run Thermos", "parentId" : 7 }
```

Scenario: Product Categories

```
db.categoryGraph.aggregate([
  { $match:       { _id: 1 }},
  { $graphLookup: {
        from:             "categoryGraph",
        startWith:        "$_id",
        connectFromField: "_id",
        connectToField:   "parentId",
        depthField:       "depth",
        maxDepth:         1,
        as:               "Children" }}
]);
```

Get all children 2 levels deep – flat result


```
{
    "_id": 1,
    "name": "Mugs",
    "Children": [
        { "_id": 7, "name": "Running Thermos",       "parentId": 5, "depth": 1 },
        { "_id": 6, "name": "Camping Mugs",          "parentId": 5, "depth": 1 },
        { "_id": 4, "name": "Glassware & Drinkware", "parentId": 2, "depth": 1 },
        { "_id": 3, "name": "Commuter & Travel",     "parentId": 2, "depth": 1 },
        { "_id": 5, "name": "Outdoor Recreation",    "parentId": 1, "depth": 0 },
        { "_id": 2, "name": "Kitchen & Dining",      "parentId": 1, "depth": 0 }
    ]
}
```

Scenario: Product Categories

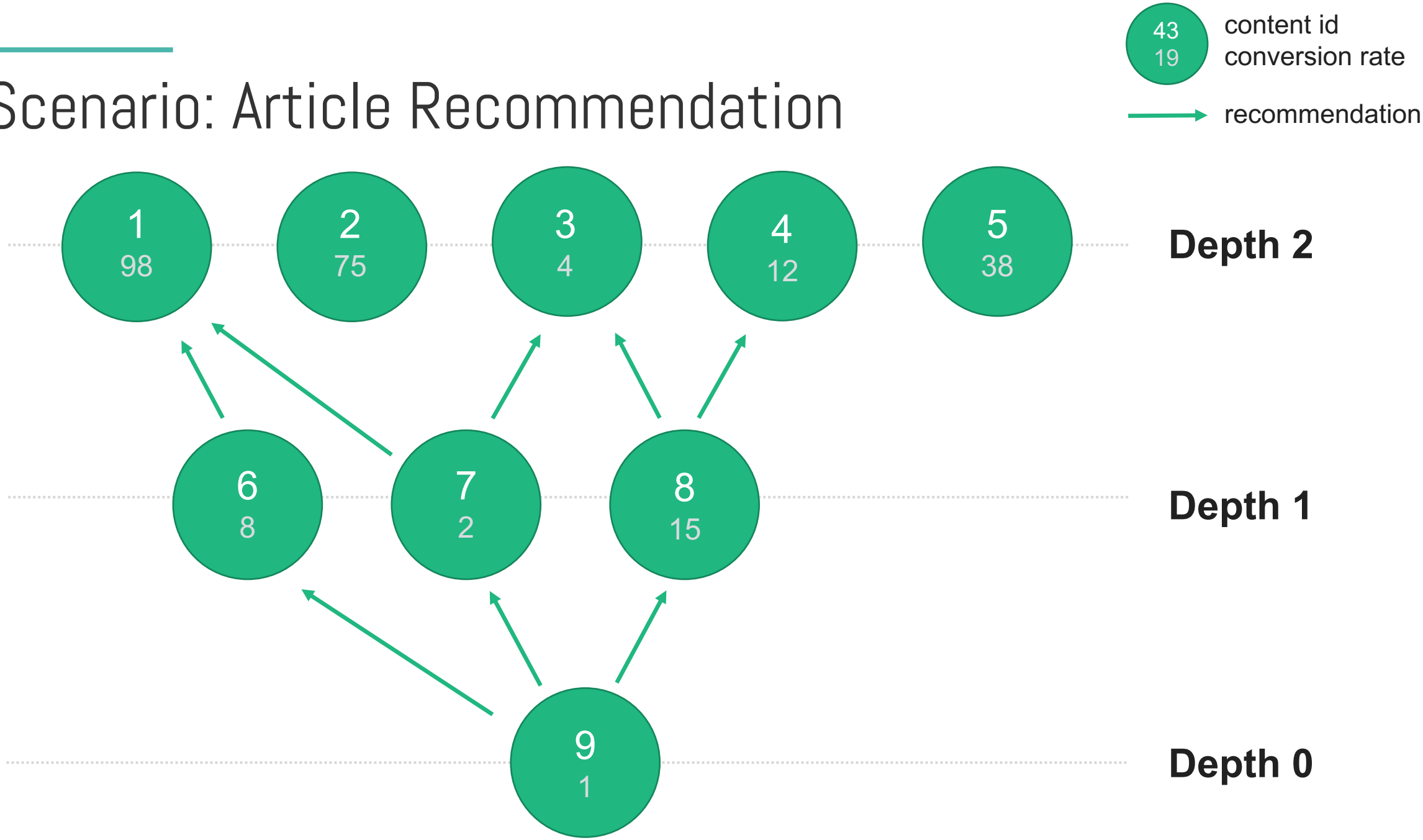
```
db.createView("vwCategoryDepth1", "categoryGraph", [
    { $graphLookup: {
         from:             "categoryGraph",
         startWith:        "$_id",
         connectFromField: "_id",
         connectToField:   "parentId",
         maxDepth:         0,
         as:               "Children" }}
]);

db.categoryGraph.aggregate([
  { $match:       { _id: 1 }},
  { $graphLookup: {
         from:             "vwCategoryDepth1",
         startWith:        "$_id",
         connectFromField: "_id",
         connectToField:   "parentId",
         maxDepth:         0,
         as:               "Children" }}
]);
```

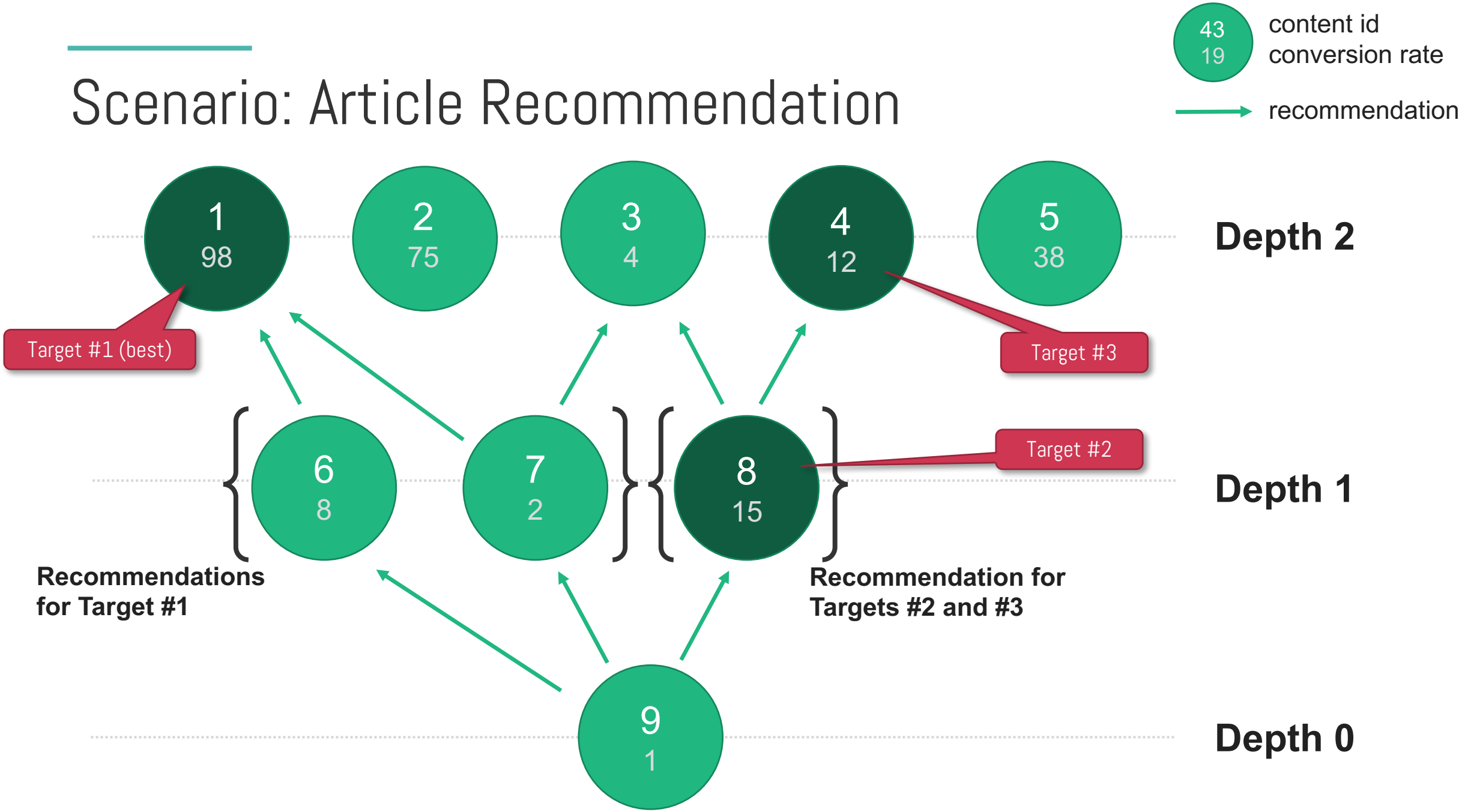
Get all children 2 levels deep – nested result

```
{
    "_id" : 1,
    "name" : "Mugs",
    "Children" : [
        {   "_id" : 5,
            "name" : "Outdoor Recreation",
            "parentId" : 1,
            "Children" : [
                { "_id" : 7, "name" : "Running Thermos", "parentId" : 5 },
                { "_id" : 6, "name" : "Camping Mugs",    "parentId" : 5 }
            ]},
        {   "_id" : 2,
            "name" : "Kitchen & Dining",
            "parentId" : 1,
            "Children" : [
                { "_id" : 4, "name" : "Glassware & Drinkware", "parentId" : 2 },
                { "_id" : 3, "name" : "Commuter & Travel",     "parentId" : 2 }
            ]}
    ]
}
```

Scenario: Article Recommendation



Scenario: Article Recommendation



Syntax

```
db.articles.aggregate([
        { "$match"  : { cid : 9 }},
        { "$unwind" : "$furtherReading" },
        { "$graphLookup" : {
          from : "articles",
          startWith : "$furtherReading",
          connectFromField : "furtherReading",
          connectToField : "cid",
          maxDepth : 1, /* stop after recursing once */
          depthField : "depth",
          as : "target" }},
        { "$unwind" : '$target' },
        { "$sort" : { 'target.conversion' : -1 }},
        { "$group" : {
          _id : "$target.cid",
          recommend : { $addToSet: "$furtherReading"},
          target : { $first : "$target" }}},
        { "$sort" : { 'target.conversion' : -1 }},
        { "$project" : {
          'target._id' : 0, '_id' : 0,
          'target.furtherReading' : 0, 'target.title' : 0 }},
        { "$limit" : 3 }
]).pretty()
```

Syntax

```
{
    "recommend" : [
        7,
        6
    ],
    "target" : {
        "cid" : 1,
        "conversion" : 98,
        "depth" : NumberLong(1)
    }
},
{
    "recommend" : [
        8
    ],
    "target" : {
        "cid" : 8,
        "conversion" : 15,
        "depth" : NumberLong(0)
    }
},
{
    "recommend" : [
        8
    ],
    "target" : {
        "cid" : 4,
        "conversion" : 12,
        "depth" : NumberLong(1)
    }
}
```

Design & Performance Considerations

The Tale of Two Biebers

VS

Follower Churn

- Everyone worries about scaling content
- But follow requests can be >> message send rates
  - Twitter enforces per day follow limits

Edge Metadata

- Models – friends/followers
- Requirements typically start simple
- Add Groups, Favorites, Relationships

Firstpost

Twitter Testers Gain Access To A New **Timeline** Called "Fave People ...

TechCrunch - by Sarah Perez - Mar 17, 2014

Twitter is experimenting with a new feature called "Fave People," which allows you to keep track of your favorite **Twitter** users and accounts in a ...

Twitter testing 'Fave People' **timeline**, lets you keep your favorite ...

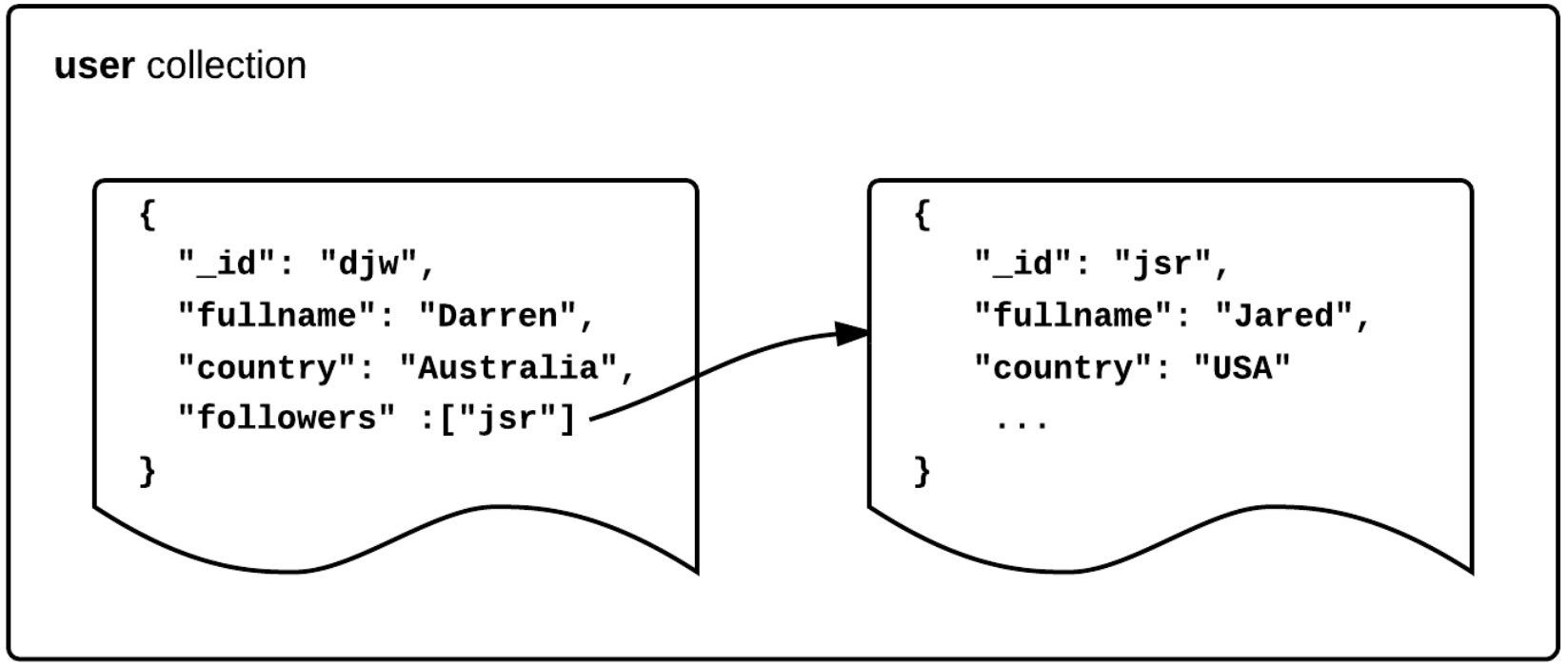
Digital Trends - Mar 18, 2014

Twitter Is Reportedly Testing a "Fave People" **Timeline**...For Your Favs

Gizmodo - Mar 17, 2014

Options for Storing Graphs in MongoDB

Option One – Embedding Edges



Embedded Edge Arrays

```
{
        "_id" : "djw",
        "fullname" : "Darren Wood",
        "country" : "Australia",
        "followers" : [ "jsr", "ian"],
        "following" : [ "jsr", "pete"]
}
```

- Storing connections with user (popular choice)
  - ✓Most compact form
  - ✓Efficient for reads
- However….
  - User documents grow
  - Upper limit on degree (document size)
  - Difficult to annotate (and index) edge

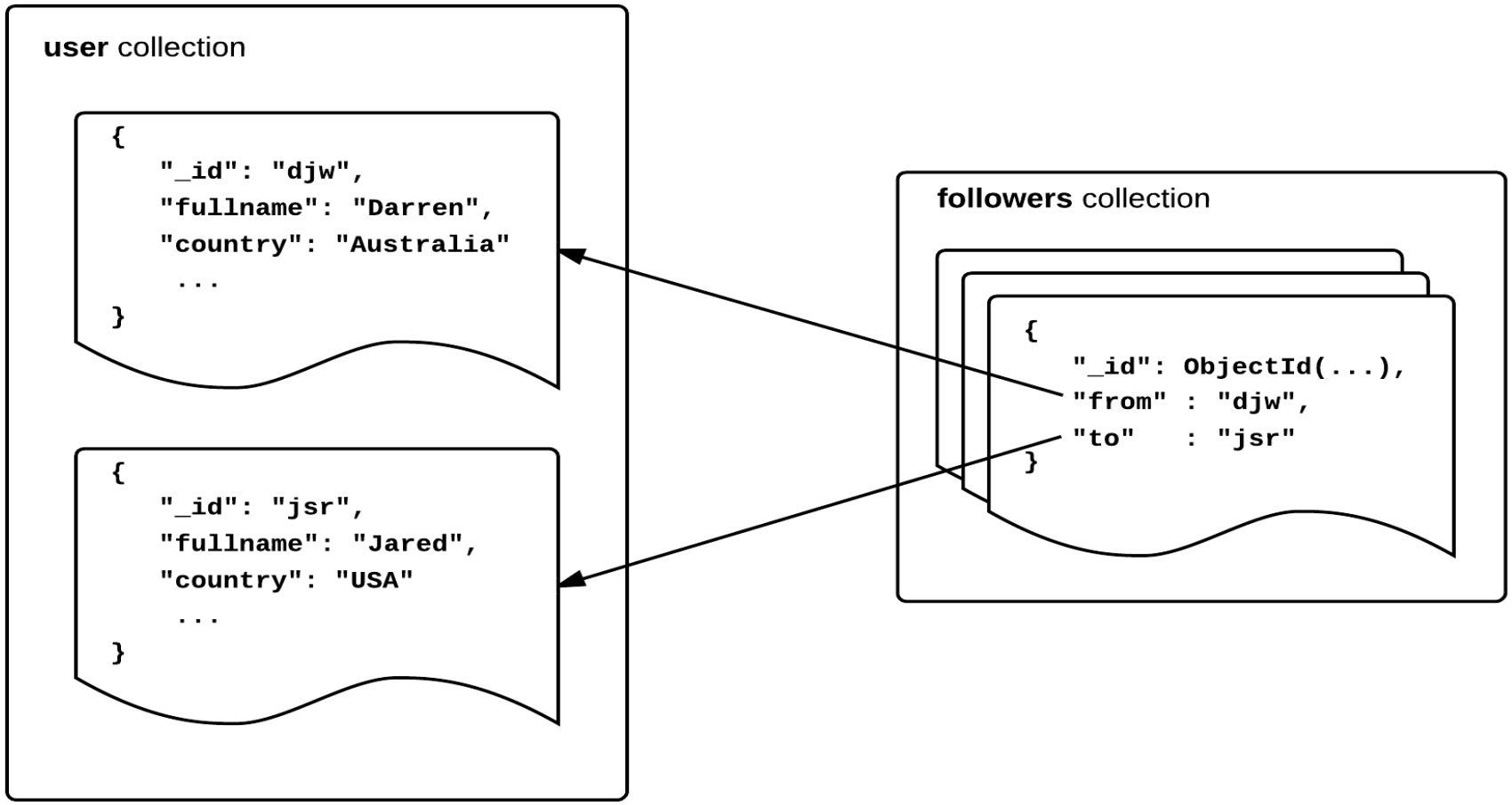
Embedded Edge Arrays

- Creating Rich Graph Information
  - Can become cumbersome

```
{
        "_id" : "djw",
        "fullname" : "Darren Wood",
        "country" : "Australia",
        "friends" : [
                {"uid" : "jsr", "grp" : "school"},
                {"uid" : "ian", "grp" : "work"} ]
}

{
        "_id" : "djw",
        "fullname" : "Darren Wood",
        "country" : "Australia",
        "friends" : [ "jsr", "ian"],
        "group" : [ ”school", ”work"]
}
```

Option Two – Edge Collection



Edge Collections

- Document per edge

```
> db.followers.findOne()
{
        "_id"  : ObjectId(…),
        "from" : "djw",
        "to"   : "jsr"
}
```

- Very flexible for adding edge data

```
> db.friends.findOne()
{
        "_id"   : ObjectId(…),
        "from"  : "djw",
        "to"    : "jsr",
        "grp"   : "work",
        "ts"    : Date("2013-07-10")
}
```

Edge Collection Indexing Strategies

Finding Followers

Find followers in single edge collection :

```
> db.followers.find({from : "djw"}, {_id:0, to:1})
{
        "to" : "jsr"
}
```

Using index :

```
{
        "v" : 1,
        "key" : { "from" : 1, "to" : 1 },
        "unique" : true,
        "ns" : "socialite.followers",
        "name" : "from_1_to_1"
}
```

Covered index when searching on "from" for all followers

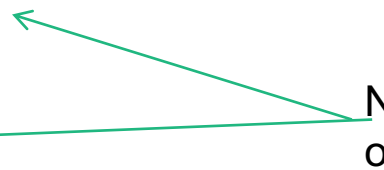
Specify only if multiple edges cannot exist

Finding Following

Could use a reverse covered index :

What about who a user is following?

```
{
        "v" : 1,
        "key" : { "from" : 1, "to" : 1 },
        "unique" : true,
        "ns" : "socialite.followers",
        "name" : "from_1_to_1"
}
{
        "v" : 1,
        "key" : { "to" : 1, "from" : 1 },
        "unique" : true,
        "ns" : "socialite.followers",
        "name" : "to_1_from_1"
}
```



Notice the flipped field order here

Wait ! There may be an issue with the reverse index…..

Finding Following

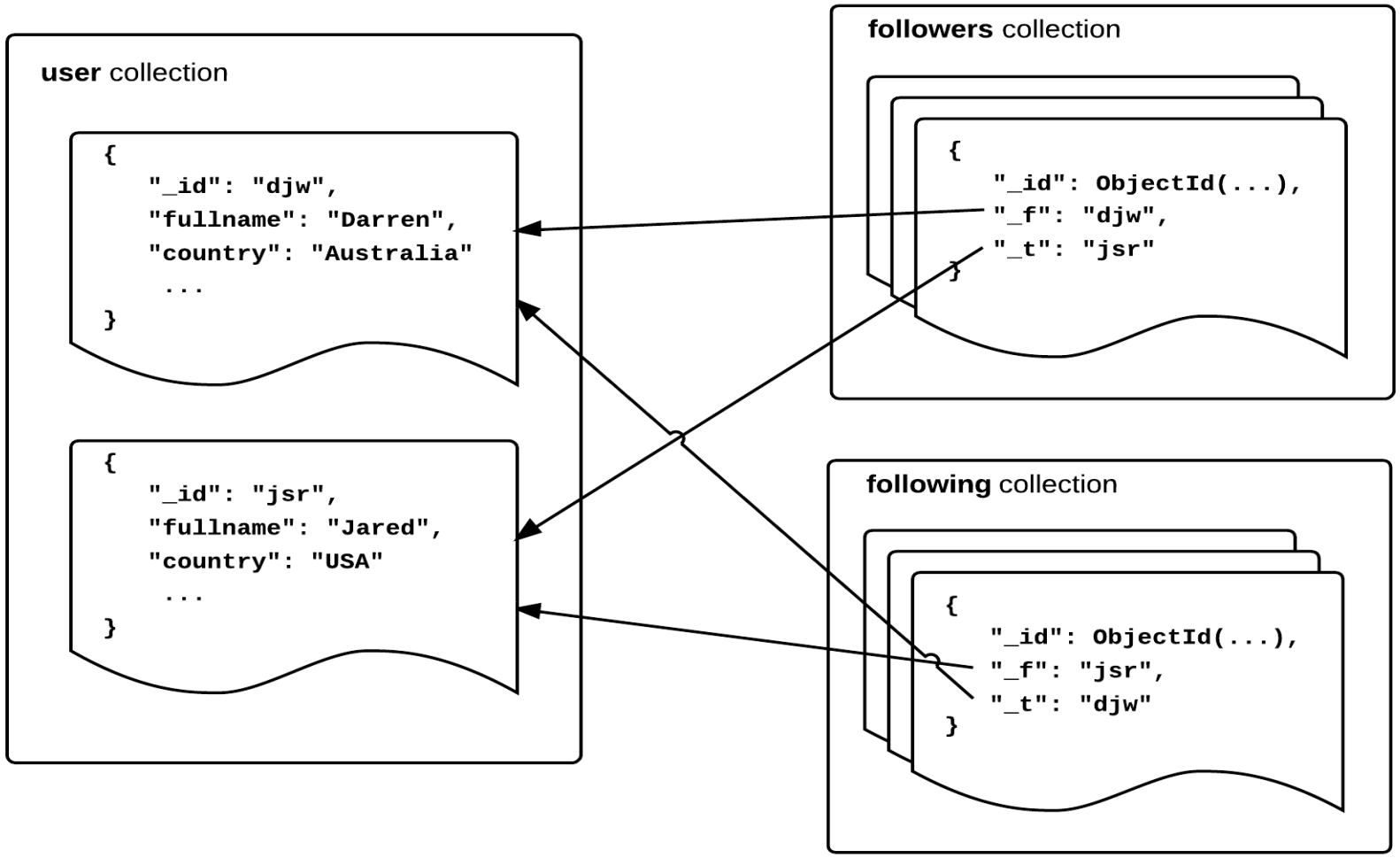
SHARDING!

```
{
        "v" : 1,
        "key" : { "from" : 1, "to" : 1 },
        "unique" : true,
        "ns" : "socialite.followers",
        "name" : "from_1_to_1"
}
{
        "v" : 1,
        "key" : { "to" : 1, "from" : 1 },
        "unique" : true,
        "ns" : "socialite.followers",
        "name" : "to_1_from_1"
}
```

If we shard this collection by "from", looking up followers for a specific user is "targeted" to a shard

To find who the user is following however, it must scatter-gather the query to all shards

Dual Edge Collections



Dual Edge Collections

- When "following" queries are common
  - Not always the case
  - Consider overhead carefully

- Can use dual collections storing
  - One for each direction
  - Edges are duplicated reversed
  - Can be sharded independently

Wrap-up

MongoDB $graphLookup

- Efficient, index-based recursive queries
- Familiar, MongoDB query language
- Use a single System Of Record
  - Cater for all query types
  - No added operational overhead
  - No synchronization requirements
  - Reduced technology surface area

Graph Operations With MongoDB

Charles Sarrazin

Senior Consulting Engineer, MongoDB