Building a Graphy Time Machine

Michael Hackstein
ArangoDB Inc.

About Me

1. Michael Hackstein
2. ArangoDB Core Team
    a. Leading Graph Development Team
        i. Graph Features
        ii. Smart Graphs
        iii. UI
3. Masters Degree
    a. Spec. in Databases and Information Systems
4. Twitter/Github: @mchacki

RWTH AACHEN UNIVERSITY

Motivation

1. Data evolves over time:
    a. Documents are added or removed
    b. Attributes are changed
2. We often need a full history or some kind of version control
3. This includes graph data

Requirement: Property Graphs

1. Allow to store attributes on vertices
2. Allow to store attributes on edges

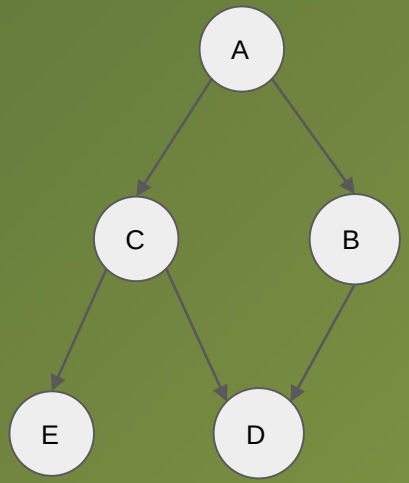
Time-Slicing

Use-Case

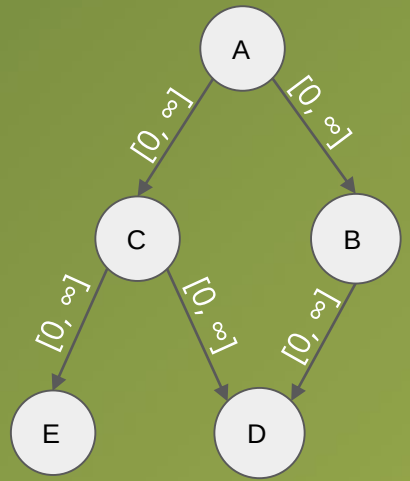
1. We have a graph-data model
2. The data is changing over time
3. We have a requirement that we can see how the data looked like at a certain point in time
4. We are able to compress, archive or delete data older than a certain point.
    a. If we cannot clean up we need to be prepared for "infinite" production of data.

Example Step 1: Start

Original Graph data, the starting point

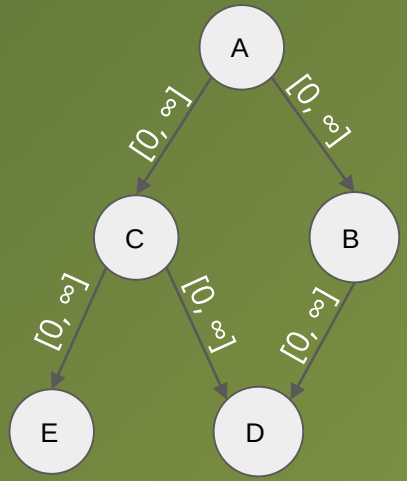


Add {created: ts1, expires: ts2} to every edge



Example Step 2: Modify B -> B'

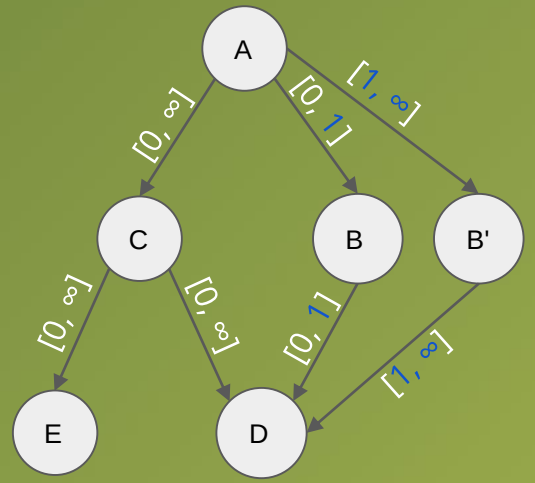
Full Graph, all nodes are active



Copy on

Write

A->B->D is obsolete in 1. A->B'->D is active at 1



Querying for Time-Slicing

Query the neighbors of a vertex (A) at a certain time

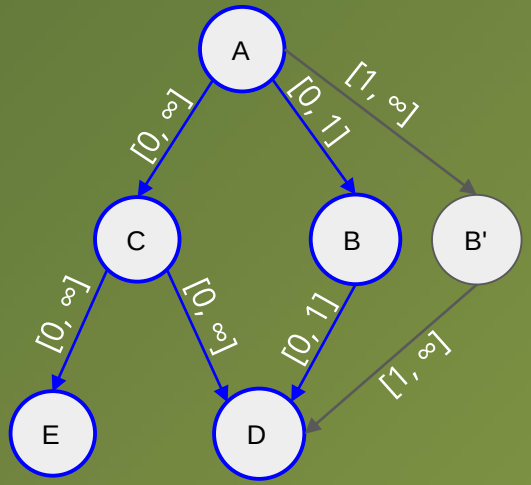
```
FOR n, e IN 1 OUTBOUND 'vertex/A' GRAPH 'timemachine'
  FILTER e.created >= @timestamp
  FILTER e.expires < @timestamp
  RETURN n
```

Query a (sub-) graph starting at vertex (A) at a certain time

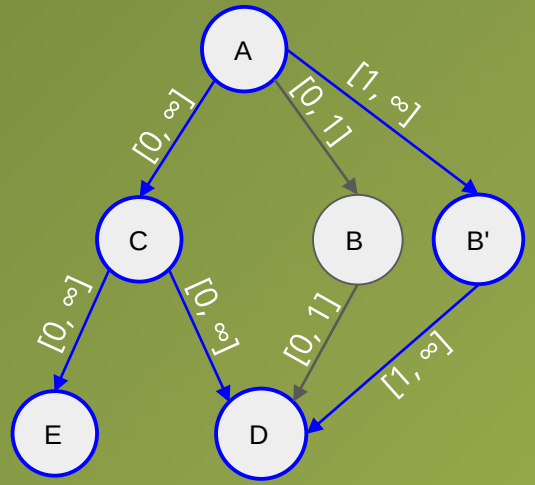
```
FOR n, e, p IN 1..@depth OUTBOUND 'vertex/A' GRAPH 'timemachine'
  FILTER p.edges[*].created ALL >= @timestamp
  FILTER e.edges[*].expires < @timestamp
  RETURN n
```

Example Step 3: Querying

Timestamp == 0



Timestamp == 5



DEMO TIME

Problem solved?

1. Solved:
    a. Store complete history
    b. Query for any point in time
2. New / Open:
    a. We would need infinite space
    b. What about complexity / cost of inserts?
    c. What about complexity of reads?

Infinite space

1. Scale to more machines
    a. Delays the issue
2. In most use-cases:
    a. Most recent data needs full versions
    b. Old data can be compacted and archived
    c. => Use-case specific "garbage" collection

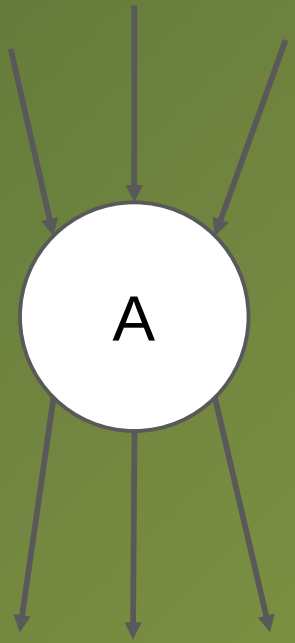
Overhead of a modification

1. "Mental Model":
    a. Update one Attribute (1 operation)
2. "Real Data":
    a. Copy the Vertex and overwrite this attribute (1 operation)
    b. Copy all ***n*** connected edges with new created/expires (***n*** operations)
    c. Modify expire of all ***n*** connected edges (***n*** operations)

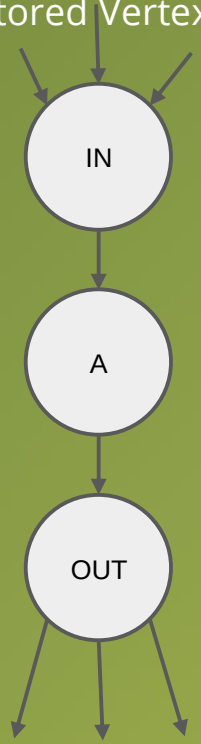
Vertex-Proxies

Vertex Proxies to the rescue

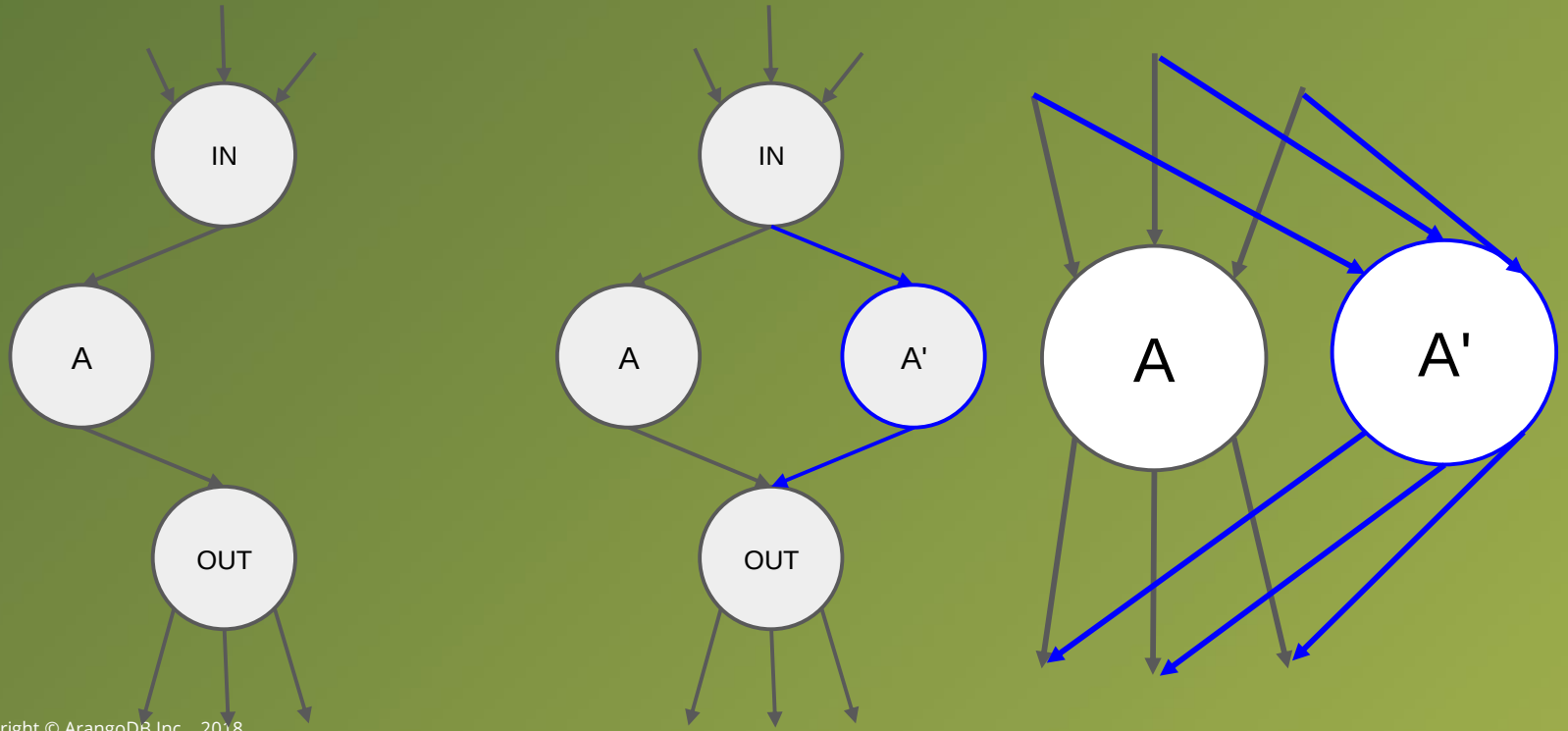
Logical Vertex



Stored Vertex



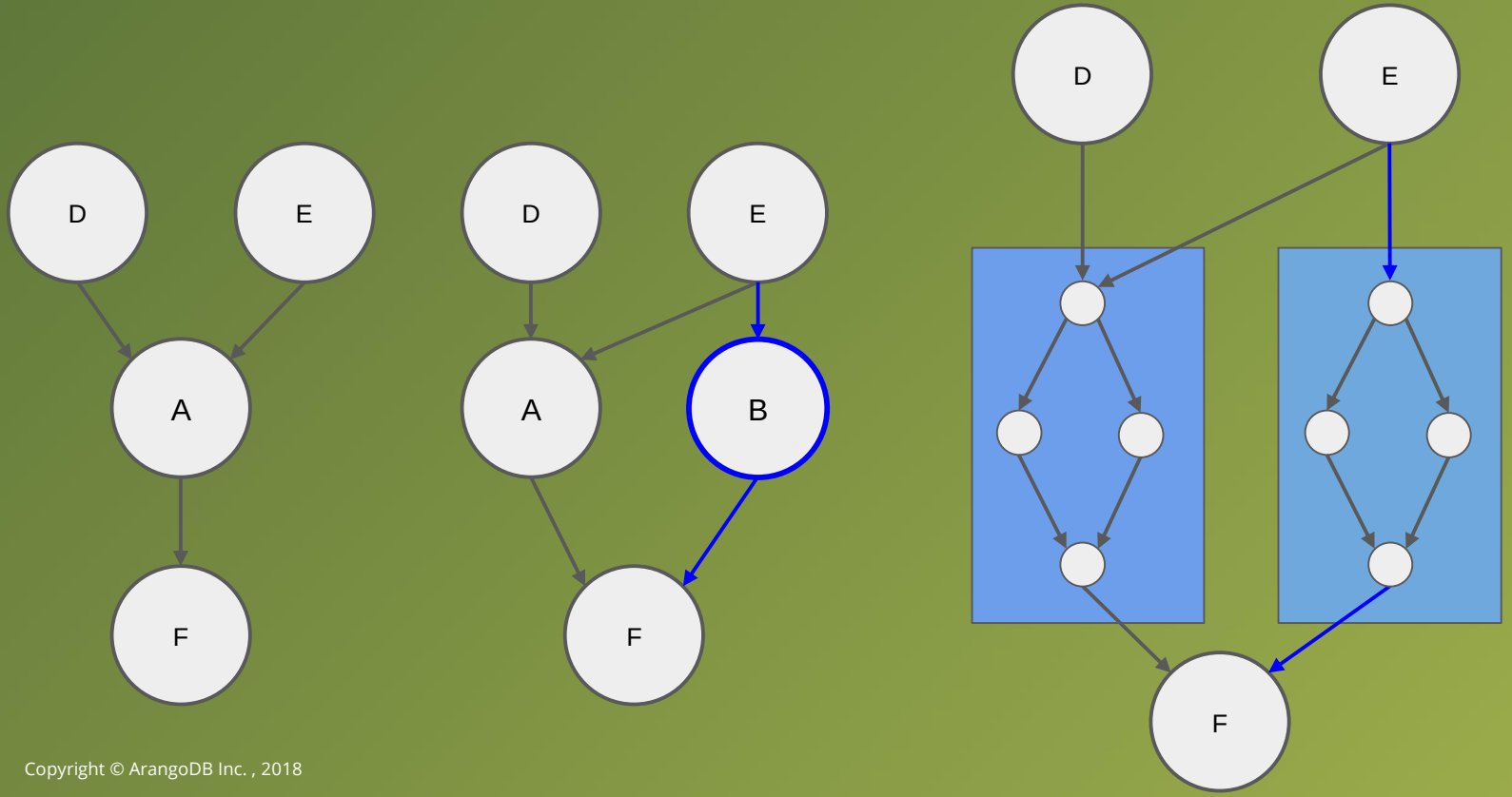
Modify A



Overhead of a modification (Proxy)

1. "Mental Model":
    a. Update one Attribute (1 operation)
2. "Real Data":
    a. Copy the Vertex and overwrite this attribute (1 operation)
    b. Create edges **IN -> Vertex' -> OUT** (2 operations)
    c. Modify expire **IN -> Vertex -> OUT** (2 operations)

Replace A with new B



Overhead of a replace (Proxy)

1. "Mental Model":
    a. Replace one Vertex (1 operation)
    b. Update **k** connected edges (**k** operations)
2. "Real Data":
    a. Create new Vertex (1 operation)
    b. Create In and Out Proxy (2 operations)
    c. Create **k** new connected edges (**k** operations)
    d. Expire **k** old connected edges (**k** operations)

DEMO TIME

Number of Edges created

1. It sounds like the number of edges can explode?
    a. Total Number of edges is high
    b. #Edges (Proxy) < #Edges (noProxy) if #Edges/Vertex > 3
2. Number of edges per Vertex:
    a. Assume (A) has **k** inbound edges, and **n** outbound edges and **L** versions
    b. IN-Proxy: **k** inbound, **L** outbound
    c. Out-Proxy: **L** inbound, **n** outbound
    d. A (each): **1** inbound, **1** outbound

Influence on Query costs

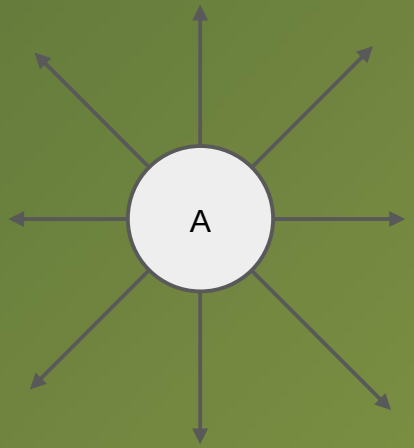
1. For simplicity assume we only search with direction on edges
    a. For reverse-direction same complexity applies
2. And assume we **only** search for **one** specific timestamp
    a. Timestamp- Range-search is possible, just makes the formula more complicated ;)
3. "Mental-Model":
    a. Every vertex A has **n** outbound edges => (O(n)) for depth 1, ($O(n^{\mathbf{d}})$) for depth **d**
4. "Proxy-Model":
    a. Specific version of A has 1 outbound edge to OUT => (O(1))
    b. OUT has n outbound edges to IN => (O(n)) for depth 1, ($O(n^{\mathbf{d}})$) for depth **d**
    c. IN has **L** outbound edges, only **1** is relevant => (O(L)), linear search
5. "No-Proxy-Model":
    a. Specific version of A has **n * L** outbound edges, only **n** are relevant => (O(n*L))

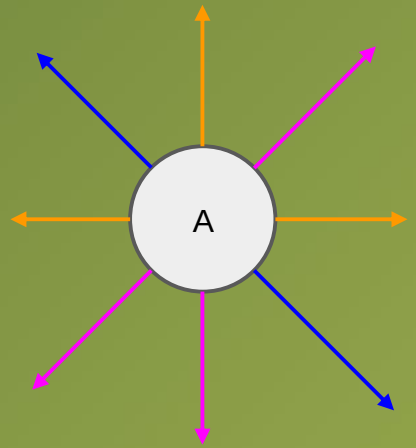
Vertex-Centric index

Linear Scan does not scale

1. This step is optional, depends on use-case
    a. Small number of versions **L** => linear might be okay
    b. High number of versions **L** => you may need this optimization
2. The number of neighbors **n** does not matter in the proxy solution.
    a. We need all neighbors **n** in every query.
    b. Improving the time to find one specific neighbor is of no help.

Vertex-Centric Index





No index => scan all connected edges

index (_from, color) => scan all edges of color

Vertex-Centric Index applied

1. We need a Sorted index
    a. First sort by vertex **(_from)**
    b. Second sort by **created**
    c. In ArangoDB: **Skiplist(["_from", "created"])**
2. Find the relevant version in **O(log L)** instead of **O(L)**
3. Drawbacks:
    a. Uses additional space
    b. Increase cost of write
4. Only relevant on OUT -> Vertex edges
    a. In ArangoDB use three edge collections, e.g.: **outToV**, **vToIn**, **inToOut**

DEMO TIME

Problem solved?

1. Solved:
    a. Store complete history
    b. Query for any point in time
2. New / Open:
    a. We would need infinite space
    b. What about complexity / cost of inserts?
    c. What about complexity of reads?

Problem solved?

1. Solved:
    a. Store complete history
    b. Query for any point in time
    c. We would need infinite space (garbage collection)
    d. What about complexity / cost of inserts? (Vertex Proxies)
    e. What about complexity of reads? (Minor Penalty here, Vertex Centric Indexes)
2. New:
    a. What about new versions?

Next Use-case

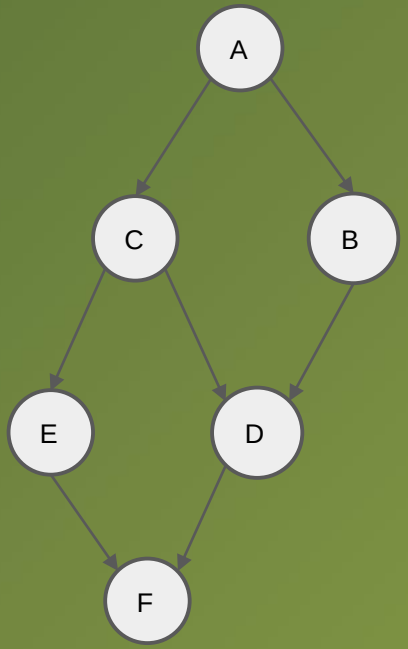
1. We need a new version of the graph e.g.:
    a. Two projects using the same graph as start
    b. We have version 1, and want to develop version 2 and apply fixes to 1
2. We could copy the complete graph
    a. Expensive
    b. Maybe the only option

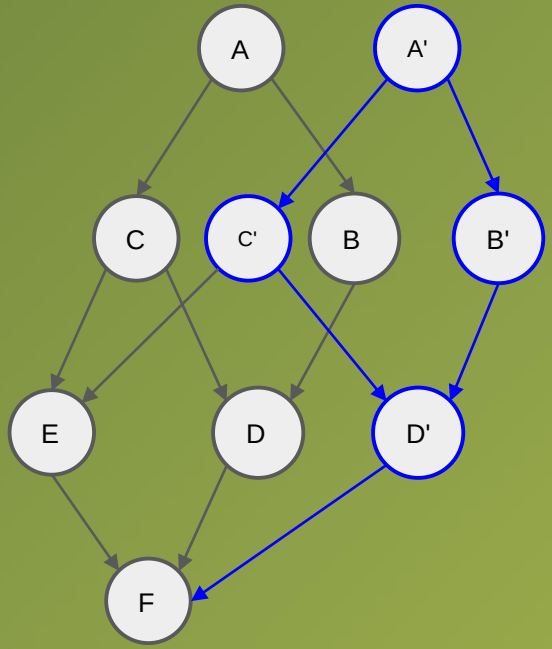
Branches

We are lazy, so is the copy

1. Instead of a full-copy on create we do the following:
    a. We create a new "root" determining the new branch
    b. By application logic we need to remember on which branch we are now
2. If we now modify something in the new branch:
    a. Apply lazy path copying
    b. From this point on, we actually have a "partial-copy" of the graph
    c. Eventually we have copied the entire graph
        i. At least the part that is necessary to copy

Path-Copying by Example





Path-Copying Explained

1. We need to create a new version of a vertex at timestamp **t**
2. We traverse all the way back up to the root(s)
3. We can stop at every vertex already on this new version
4. For every vertex:
    a. We create a new copy in this new version
    b. We copy all non-expired edges from the old version downwards, with [<old>,t]
5. We copy all edges between these vertices with [t, ∞]

Complexity

1. A path copy is potentially expensive
    a. Especially on long paths
2. But it is still "under control"
3. It is typically less expensive than copying the complete graph
    a. In deep and narrow graphs you copy large portions at a time.
    b. In shallow and broad graphs path-copying is rather cheap

Graph Requirements

1. All parts up to path-copying:
    a. Can be applied to any graph
    b. We used a directed graph here
    c. Graph could be non-directed
        i. Graph DBS typically only offer directed anyways
    d. Graph can have cycles
2. Path-copying
    a. Requires directed, acyclic graphs
        i. We need a direction to go "upwards"
        ii. We cannot have cycles because:

DEMO TIME

Just one more thing...

1. All this information needs to be correct in any case.
2. Actually we only want to expose the "mental" model of the graph
    a. It is error prone and tedious to handle proxies and time-slices in every application
    b. If one application does it wrong, others may suffer

Foxx

Micro-Service Framework

1. Add REST-API to ArangoDB
2. Written in JavaScript
    a. Many node modules available (not all)
3. Generate interactive Documentation (Swagger)

Micro-Service Framework

1. Designed to work as a data-intensive micro-service
2. Perfect fit to transform "mental-model" <-> "data-model"
3. Simply expose the API you need
    a. Hide the proxy logic and just return logical vertices

Existing App: ArangoDB Timetravel

1. Created by a community member @KevinExtremo
2. Code here: https://github.com/Bonsaya/arangodb-timetravel/

DEMO TIME

Thank you very much.
Any questions?

Please star:
https://github.com/arangodb/arangodb
Twitter: @arangodb

Rate My Session