

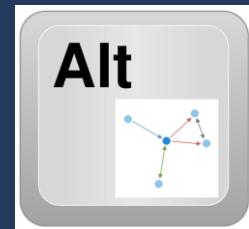
# Azure Cosmos DB – AltGraph

Chris Joakim, Microsoft, Cosmos DB Global Black Belt (GBB)

<https://www.linkedin.com/in/chris-joakim-4859b89>

<https://github.com/cjoakim/azure-cosmosdb-altgraph>

[https://www.youtube.com/watch?v=SGih\\_Kj\\_1yk](https://www.youtube.com/watch?v=SGih_Kj_1yk)



# What is AltGraph?

**AltGraph is a set of Alternative Graph Implementations built on:**

A Set of Designs:

- Using the Azure **Cosmos DB NoSQL API**
- **Fast In-memory processing** vs DB and Disk Traversal
- RDF-like “**Triples**” (**v1**) or the **JGraphT library (v2)**
- Azure Redis Cache or Cosmos DB Integrated **Cache**

Two Reference Implementations – v1/NPM and v2/IMDb, using:

- The **Java** programming language
- **Spring Boot** and **Spring Data** frameworks
- <https://github.com/cjoakim/azure-cosmosdb-altgraph>

# Presentation Outline

- **Influences**
  - Real-world Use Cases
  - Previous Cosmos DB Live TV Sessions
  - LinkedIn / Liquid,
- **Perception:** How you See the Problem often determines your solution
  - Sample Database Diagrams
  - Types of Databases
- **Think Differently:** Why another Graph Implementation?
- **Design**
- **Demonstration** of the Reference Application

# Influences

## Real-World Customer Use-Cases

- Manufacturing **Bill-of-Material** (BOM)
- **Social Network Systems** - People, Messages, Posts, Tags, etc.
- Knowledge Graphs
- Java and Spring and Spring Data

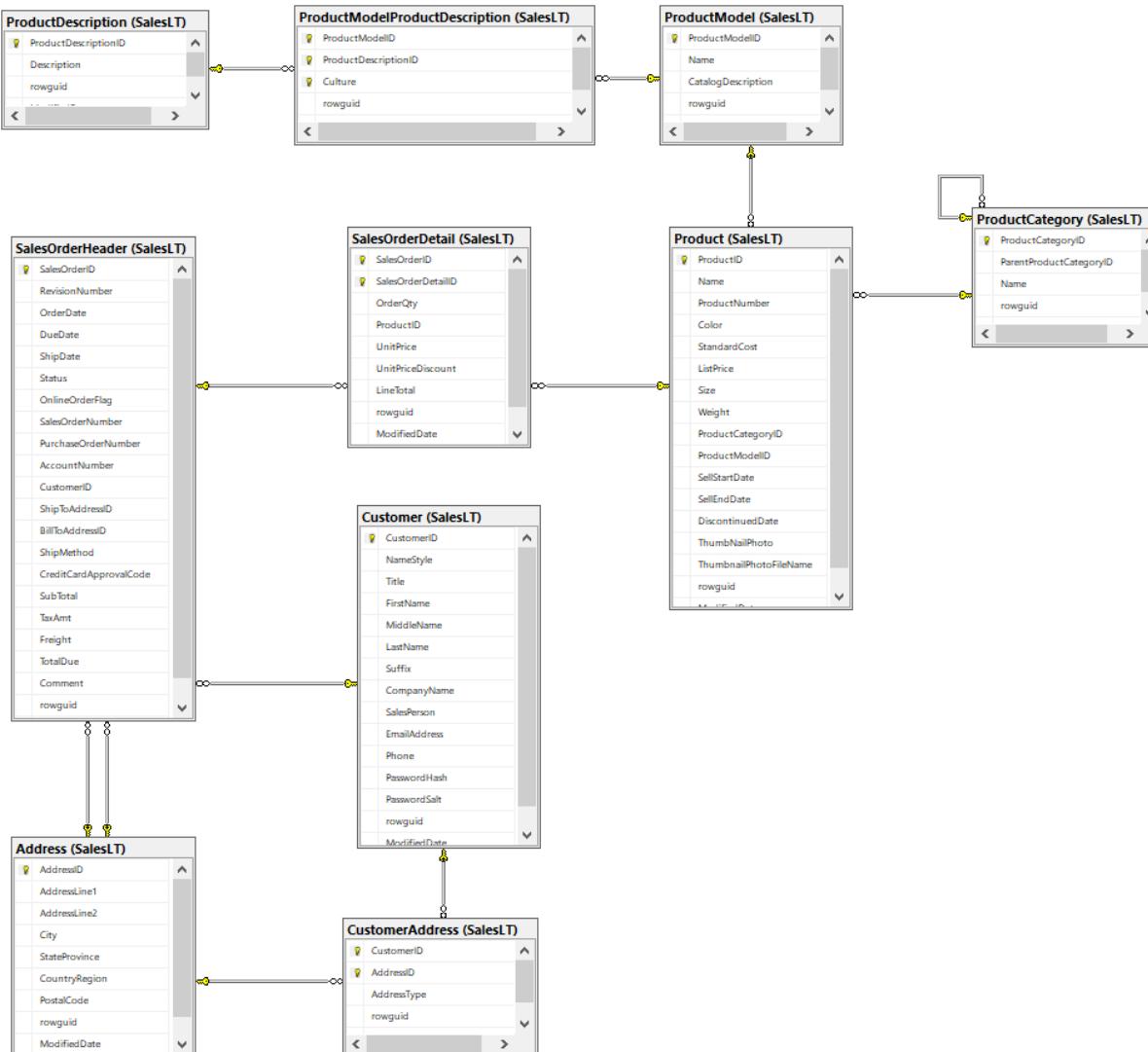
## Previous Cosmos DB Live TV Sessions

- Kushagra Thapar, Spring Data, 2022/02/03
- Mark Heckler, Spring Boot, 2022/06/23  
    Spring Boot: Up and Running – O'Reilly Media Book
- List of Episodes  
<https://www.youtube.com/playlist?list=PLmamF3YkHLoKMzT3gP4oqHiJbjMaiiLEh>

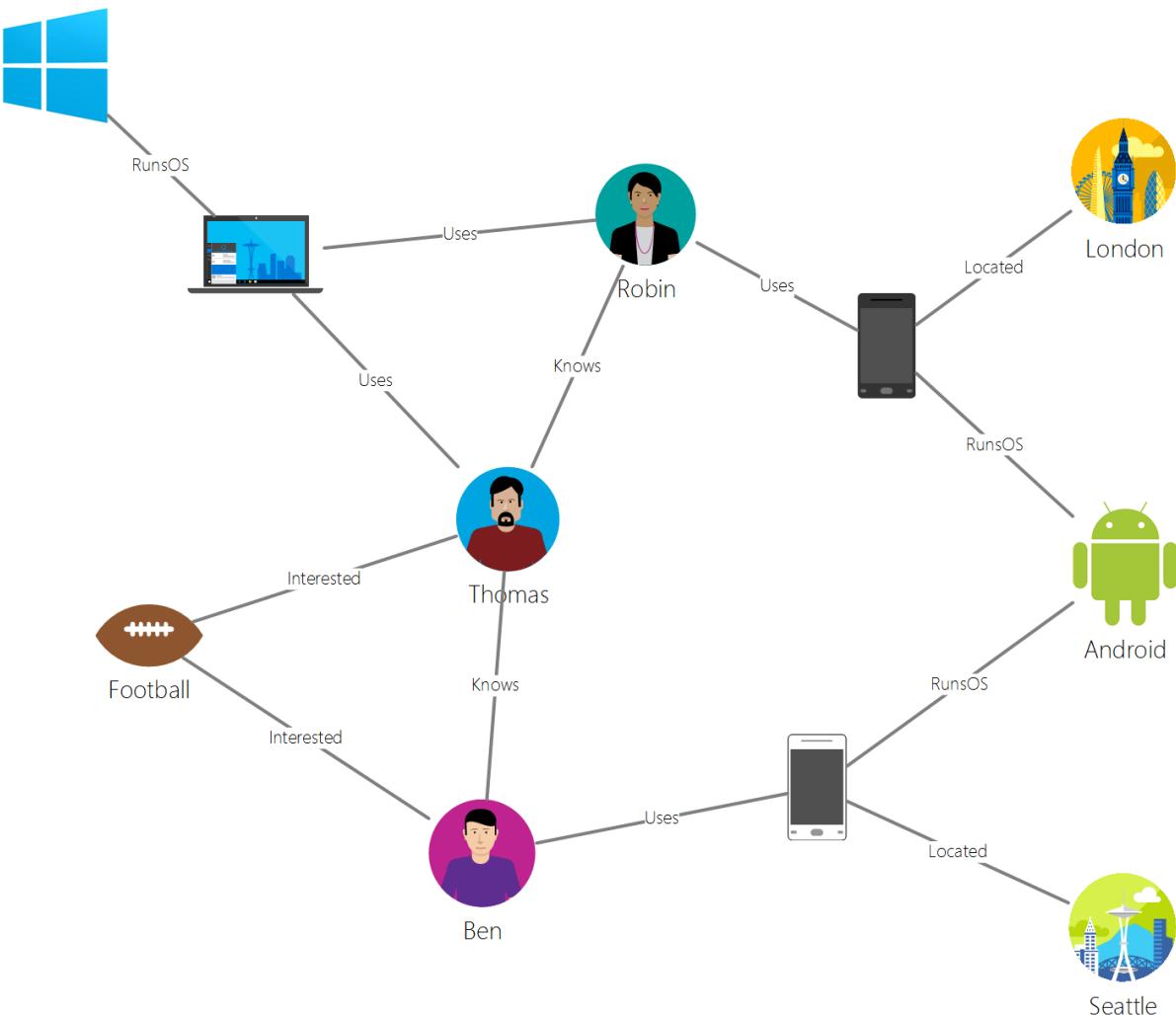
## LinkedIn / Liquid

- In-memory graph
- <https://engineering.linkedin.com/blog/2020/liquid-the-soul-of-a-new-graph-database-part-1>

# Perception: What solution would you use if the problem was drawn like this?



## Perception: Or if the problem was drawn like this?

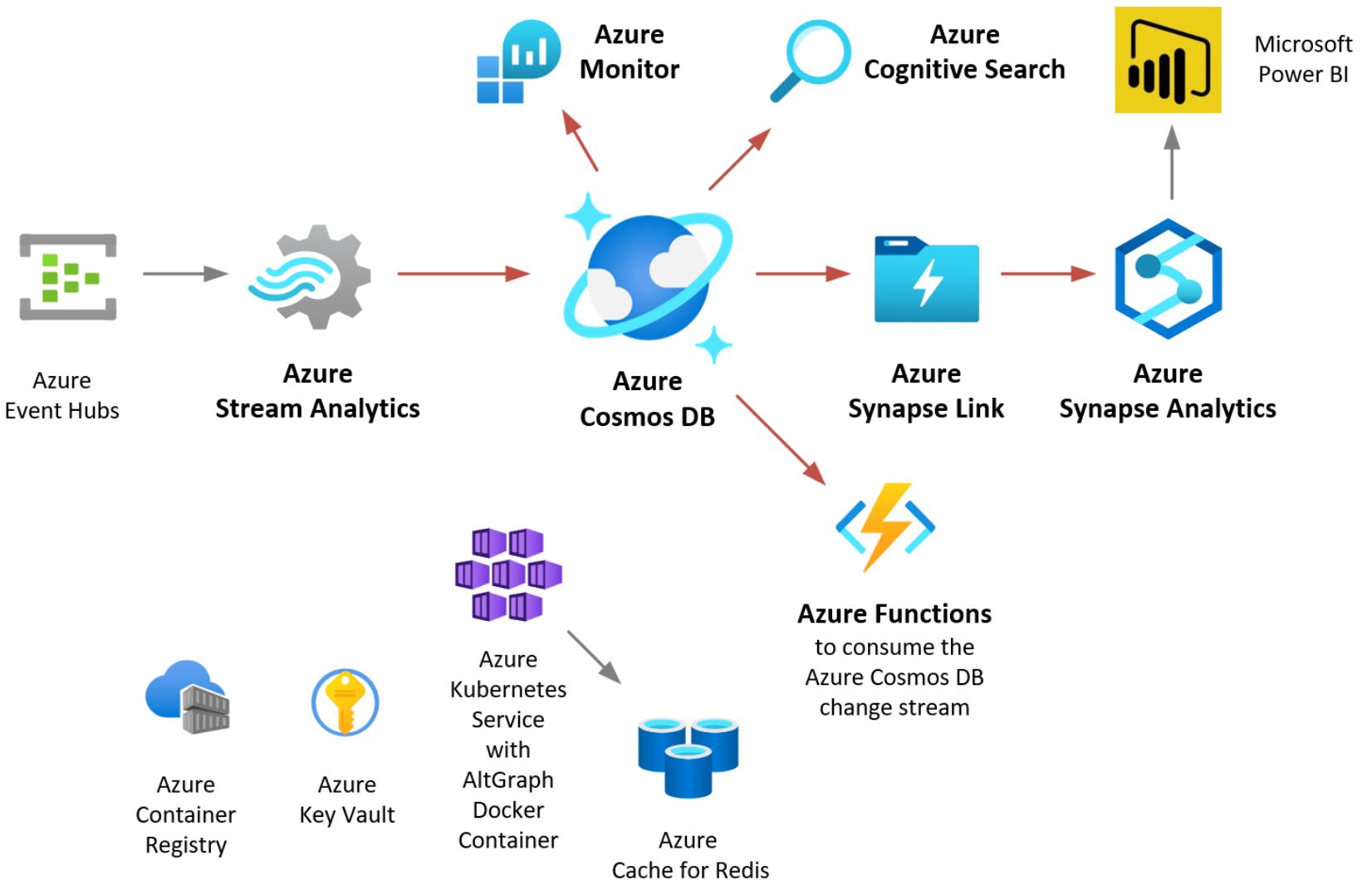


We see this a lot in the field.

A Total Solution involves  
**more than just the**  
Database.

**Database Integrations**  
are important. The **red**  
lines show native  
integrations.

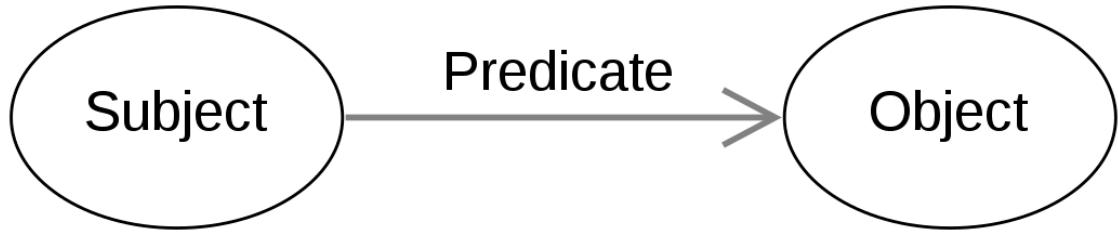
The **Cosmos DB NoSQL**  
**API** offers excellent  
integration with other  
Azure Paas Services.



# Database Solutions

- **Types of Databases**
  - **Relational:** Transactional use-cases
  - **Graph:** Graph-specific use-cases. **RDF (triplestores)** and **LPG (vertices, edges)**
  - **NoSQL:** including the Cosmos DB NoSQL API: **General Purpose**
- **Think Differently; Why another Graph Solution?**
  - **Fast execution speed**, and **lower Cosmos DB RU costs**
  - Lower barrier to entry for new apps: **simplicity, based on SQL**
    - Structured Query Language. Zero-to-low learning curve vs Graph DB syntaxes
  - **Reusable design**
  - **Faster time-to-market.** Zero to POC in days. A Reference Implementation
  - Enables **better integration** with the rest of Azure

# Design Foundations: The concept of RDF Triples and Triplestores



## Examples:

Microsoft	is_a	Technology Company
Java	is_a	Programming Language
C	is_a	Programming Language
Cosmos DB	is_a	Database System
Cosmos DB	is_a	NoSQL Database System
Cosmos DB	has_a_sdk_for	Java
Cosmos DB	has_a_sdk_for	C#
Chris	works_at	Microsoft
Chris	has_role	GBB

The triples are quite granular, typical solution has many many of these

# Design Foundations: The concept of an Index (as in Book)

Indexes enable you to quickly find what you're looking for.

It's quite small relative to the size of the Book it indexes.

248

INDEX

starters (*continued*)  
  spring-boot-starter-groovy-templates 190  
  spring-boot-starter-hateoas 190  
  spring-boot-starter-honeyq 190  
  spring-boot-starter-integration 190  
  spring-boot-starter-jdbc 190  
  spring-boot-starter-jersey 191  
  spring-boot-starter-jetty 191  
  spring-boot-starter-jooq 191  
  spring-boot-starter-jta-atomikos 191  
  spring-boot-starter-jta-bitronix 191  
  spring-boot-starter-log4j 191  
  spring-boot-starter-log4j2 192  
  spring-boot-starter-logging 192  
  spring-boot-starter-mail 192  
  spring-boot-starter-mobile 192  
  spring-boot-starter-mustache 192  
  spring-boot-starter-parent 192  
  spring-boot-starter-redis 192  
  spring-boot-starter-remote-shell 192  
  spring-boot-starter-security 193  
  spring-boot-starter-social-facebook 193  
  spring-boot-starter-social-linkedin 193  
  spring-boot-starter-social-twitter 193  
  spring-boot-starter-test 193  
  spring-boot-starter-thymeleaf 193  
  spring-boot-starter-tomcat 193  
  spring-boot-starter-undertow 194  
  spring-boot-starter-validation 194  
  spring-boot-starter-velocity 194  
  spring-boot-starter-web 194

spring-boot-starter-websocket 194  
spring-boot-starter-ws 194  
symbolic links 8

T

test-on-borrow property 166  
test-on-return property 166  
test-while-idle property 166  
testing  
  integration testing auto-configuration 77–79  
  running applications  
    overview 86–87  
    starting server on random port 87–88  
  testing pages with Selenium 88–90  
web applications  
  mocking Spring MVC 80–83  
  overview 79–80  
  security testing 83–85

tests

  class created by Spring Initializr 28–29  
  running for CLI-based applications 102–105  
testService() method 78  
Thymeleaf  
  configuration properties 229–230  
  template caching for 58  
time-between-eviction-runs-millis property 166  
Tomcat configuration 205–206  
trace endpoint 125, 136  
TraceRepository interface 153  
transitive dependencies,  
  overriding 35–37  
trigger-file property 182  
Twitter support 193, 229

U

Undertow configuration 206–207  
uploads, multi-part 195  
url property 166

use command 10  
UserDetails interface 55  
UserDetailsService  
  interface 112, 157  
username property 166

V

validation-query property 166  
VCAP\_SERVICES environment variable 176  
Velocity configuration 230–231  
views, using Grails 120–123

W

WAR files 162–164  
web applications, testing  
  mocking Spring MVC 80–83  
  overview 79–80  
  security testing 83–85  
@WebAppConfiguration annotation 80–81  
webAppContextSetup()  
  method 80  
@WebIntegrationTest annotation 86–87, 89  
WebSecurityConfigurerAdapter class 51–52  
Windows, command-line completion and 12  
withDetail() method 156  
@WithMockUser annotation 84–85  
@WithUserDetails annotation 84–85

X

 -x parameter 21  
XSS (cross-site scripting) 200

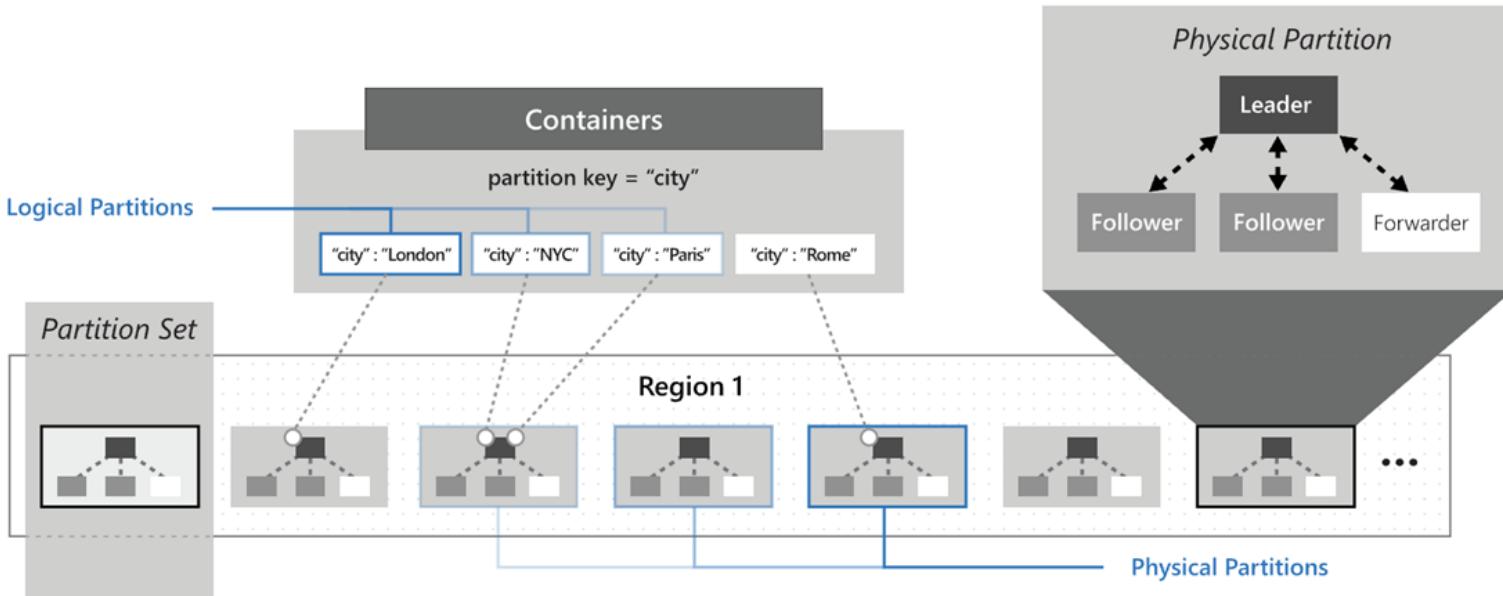
Y

YAML files 70–71

Z

ZooKeeper 210

# Design Foundations: Cosmos DB Partitioning



Reads within the same logical and physical partition are faster.  
The **Triples** and **Seed Data** (see following pages) can reside in the same logical partition.

# Design Foundations: Performance Optimizations

- **Cosmos DB Indexing and Composite Indexes**  
Index individual attributes, and as well as sets of attributes (i.e. – composite indexes) to match your queries
- **Cosmos DB "Point Reads"**  
Read by Document ID and Partition Key for fastest speed and lowest cost
- **In-Memory Processing is much faster than DB Processing**  
Traversing an in-memory data structure is 1000s of times faster than reading a DB or disk
- **Caching**
  - Eliminate costly and redundant reads to the database
  - **Azure Redis Cache**
    - <https://azure.microsoft.com/en-us/services/cache/>
  - **Cosmos DB Integrated Cache** (currently in preview mode)
    - <https://docs.microsoft.com/en-us/azure/cosmos-db/integrated-cache>

# Design Foundations: Spring Boot, Spring Data, Project Lombok

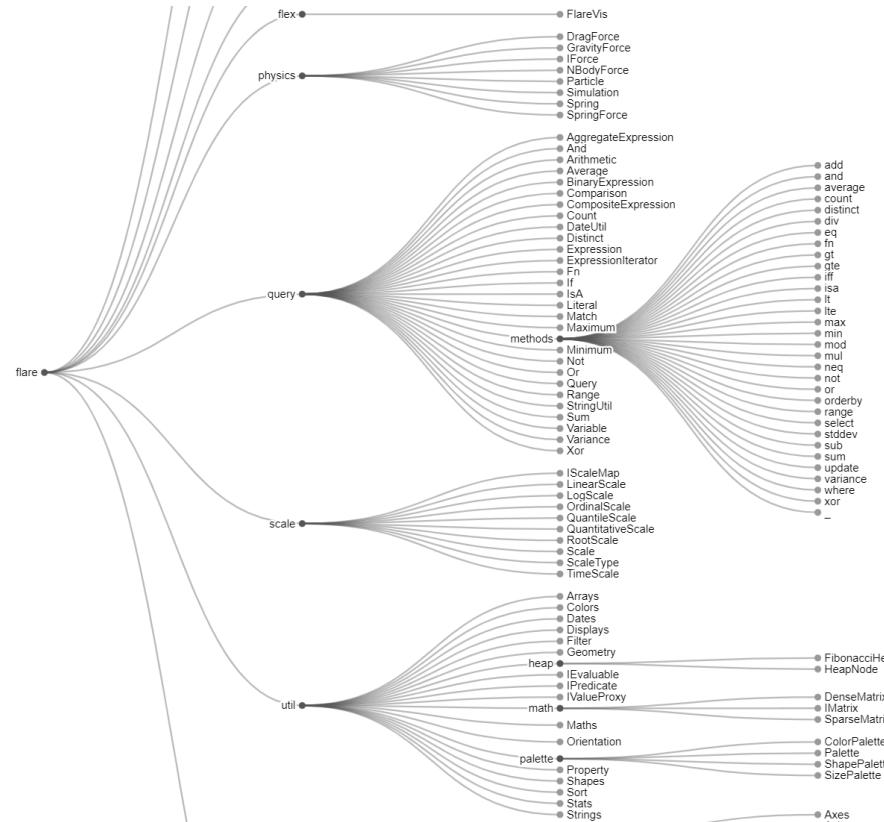
- **Spring Boot**
  - Dependency Injection, “Convention over Configuration”
  - Similar to Ruby on Rails – lots of magick happens if you follow the conventions
  - Thus, high Developer productivity
  - <https://spring.io/projects/spring-boot>
- **Spring Data**
  - Nice abstraction and simplification for database access. Repositories, Templates
  - <https://spring.io/projects/spring-data>
  - **Spring Data for Cosmos DB SDK**
    - <https://docs.microsoft.com/en-us/azure/developer/java/spring-framework/how-to-guides-spring-data-cosmosdb>
- **Project Lombok**
  - Eliminates verbose and low-value boilerplate code. Getters, setters, constructors, etc.
  - Generates bytecode at compile time. Nice IDE support, too
  - <https://projectlombok.org>

# Design Foundations – v2: Java Graph Libraries, intern() Strings

- **JGraphT**
  - A mature widely-used in-memory graph Java library
  - Implements many graph algorithms so you don't have to
    - Page Rank
    - Katz Centrality
    - Dijkstra Shortest Path
    - Others
  - <https://jgraph.org/>
- **Java String intern()**
  - Use a single value in the JVM for a frequently occurring value
  - For example, use just one instance of "nm0000102" (Kevin Bacon) instead of dozens/hundreds. This conserves JVM memory, thus enabling large in-memory graphs.
  - Similar to a Ruby programming language Symbol. 'kevin\_bacon' vs :kevin\_bacon

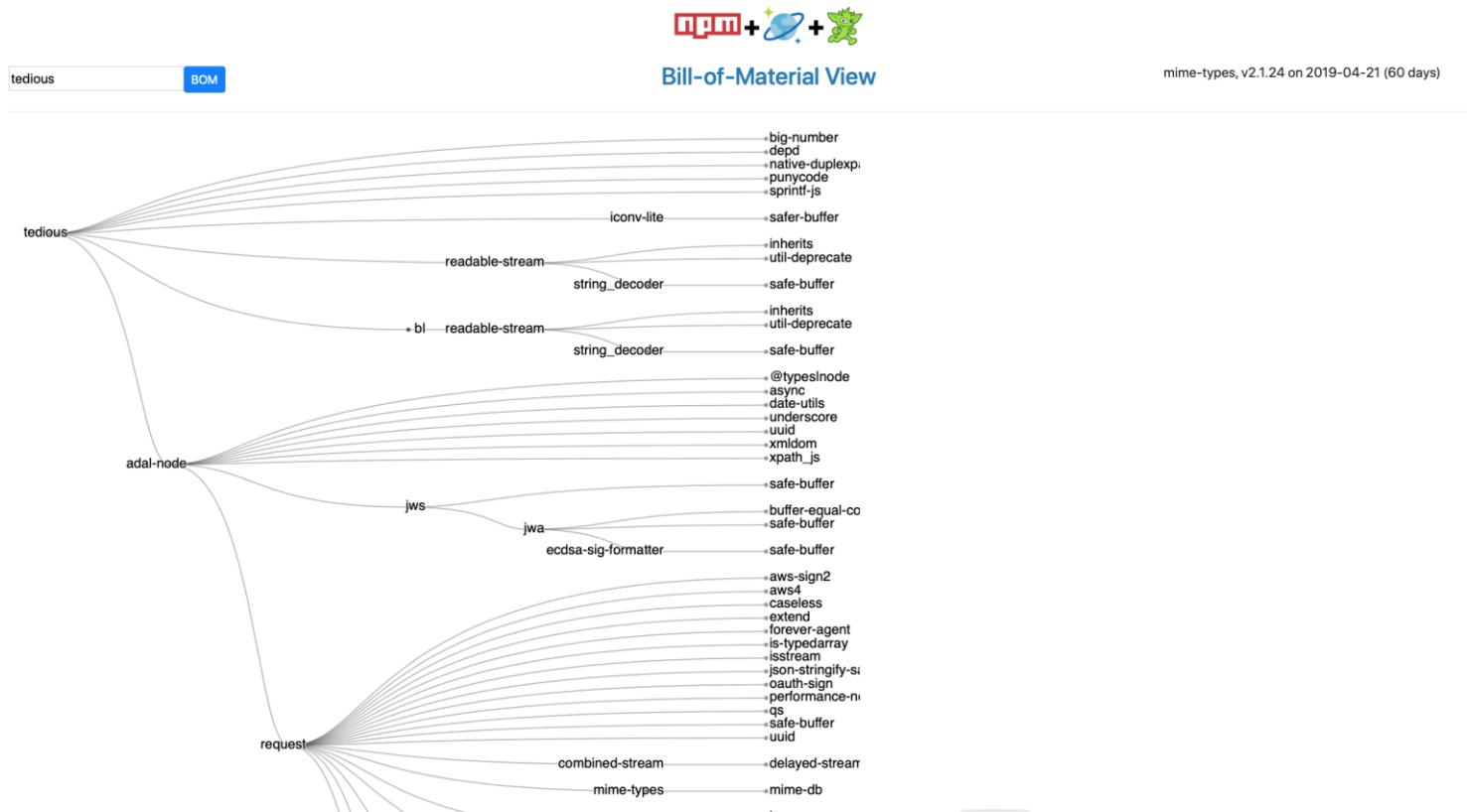
# Design Foundations: D3.js

- **D3.js JavaScript library for in-browser data visualizations**
  - Implements many out-of-the-box visualizations. Open-source. <https://d3js.org>
  - Alternatively, Bring-Your-Own-Visualization-Library (BYOViz)
  - <https://learn.microsoft.com/en-us/azure/cosmos-db/graph/graph-visualization-partners>



# Design Foundations: Previous Implementation

- **Cosmos DB Gremlin API Implementation of a Node.js NPM “Bill-of-Material” Graph**
  - AltGraph v1 uses the **same NPM data** as this previous implementation
  - <https://github.com/Azure-Samples/azure-cosmos-db-graph-npm-bom-sample>



# Implementation – v1/NPM: Cosmos DB NoSQL API

- **Use a Single Container: npm\_graph**
  - Partition key is **/pk**
  - Each document has a **doctype** attribute to distinguish the various entities
  - Reference implementation has a **tenant Id** attribute for multi-tenant use-cases
  - Reference implementation has a **lob** attribute for multiple lines-of-business in a tenant
  - Document types for this NPM graph are: **triple, library, author, maintainer**
- Enabling **Synapse Link** is optional, depending on your requirements
  - This is one of the excellent integrations that Cosmos DB offers
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/synapse-link>
  - <https://github.com/cjoakim/azure-cosmosdb-synapse-link>
- **Heirarchical Partition Keys** (currently in preview mode) may also be used
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/hierarchical-partition-keys>
- Provision the Request Units (RU) as necessary – Serverless, Manual, or Autoscale
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/set-throughput>
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/serverless>

# Implementation – v2/IMDb: Cosmos DB NoSQL API

- **Use Two Containers:** `imdb_graph` and `imdb_seed`
  - Partition key is `/pk` for each container
  - Each document has a **doctype** attribute to distinguish the various entities
    - `imdb_graph` container contains the **people** and **movie** documents
    - `imdb_seed` container **contains the “seed data” to reload the graph into memory**
      - The seed data is used instead of the “RDF Triples” in the v1 implementation
      - The movie data, for example, all resides in the same logical partition
      - Reference data contains over 1-million vertices and 3.9 million edges
      - Load time is approximately 50-seconds from home network
      - Larger in-memory graphs are possible
      - The in-memory graph is mutable for realtime use-cases
  - Navigation of the graph is mostly done with the built-in functionality in **JGraphT**
  - Load the data into memory once, then apply surgical changes to the mutable graph
  - **Graph Traversal is fast and consumes zero Cosmos DB RUs**

# Implementation – v1/NPM: Sample Library Document

This is a JSON document which describes a Node.js NPM Library. Libraries are the “raw material” for the graph.

The **dependencies** object (at line 14) is the data that we'll use to build a graph. This sample document is intentionally small. This library has only one dependency: **xml2js**

Note the **author** and **maintainers** attributes, as well. The graph will include these.

```
1  {
2    "doctype" : "library",
3    "label" : "tcx-js",
4    "id" : "f0b734d9-3240-44c5-9868-cb25597f1e3b",
5    "pk" : "tcx-js",
6    "_etag" : "\"9c00f125-0000-0100-0000-62d9c5440000\"",
7    "tenant" : "123",
8    "lob" : "npm",
9    "cacheKey" : "library|tcx-js",
10   "graphKey" : "library^tcx-js^f0b734d9-3240-44c5-9868-cb25597f1e3b^tcx-js",
11   "name" : "tcx-js",
12   "desc" : "A Node.js library for parsing TCX/XML files, such as from a Garmin GPS device.",
13   "keywords" : [ "tcx", "garmin", "forerunner", "gps" ],
14   "dependencies" : {
15     "xml2js" : "^0.4.19"
16   },
17   "devDependencies" : {
18     "mocha-multi-reporters" : "^1.1.7",
19     "chai" : "^4.2.0",
20     ... others omitted ...
21     "typescript" : "^3.5.2"
22   },
23   "author" : "Chris Joakim",
24   "maintainers" : [ "cjoakim <christopher.joakim@gmail.com>" ],
25   "version" : "1.0.1",
26   "versions" : [ "0.0.1", "0.1.0", "0.1.1", "0.1.2", "1.0.0", "1.0.1" ],
27   "homepage" : "https://github.com/cjoakim/tcx-js",
28   "library_age_days" : 1755,
29   "version_age_days" : 32
30 }
```

# Implementation – v1/NPM: Sample Array of Triples

**Triple** documents have a Subject, Predicate, and Object just like RDF triples. Up to **20 million** of these 1K docs can reside in the same **logical partition** (20GB limit).

This graph contains 6382 triples. They are small in size (1kb) and many can be read into the JVM for **in-memory processing** and traversal. Pagination-based processing is also possible.

They point to the adjacent "Vertices" via the Id/Pk attributes for **point-reads**.

The **tags** enable optimized searching of important Vertex attributes.

```
22949 }, {  
22950   "id" : "0e2cc67f-b566-4b22-aba3-b9a9a7cb6b81",  
22951   "pk" : "triple|123",  
22952   "_etag" : "\"0f0082b6-0000-0100-0000-62d9c5840000\"",  
22953   "tenant" : "123",  
22954   "lob" : "npm",  
22955   "doctype" : "triple",  
22956   "subjectType" : "library",  
22957   "subjectLabel" : "tedious",  
22958   "subjectId" : "4cc0e552-e501-47d4-ada1-2e0cfdafc388",  
22959   "subjectPk" : "tedious",  
22960   "subjectKey" : "library^tedious^4cc0e552-e501-47d4-ada1-2e0cfdafc388^tedious",  
22961   "subjectTags" : [ "author|Mike D Pilsbury <mike.pilsbury@gmail.com>", "maintaine  
22962   "predicate" : "used_in_lib",  
22963   "objectType" : "library",  
22964   "objectLabel" : "mssql",  
22965   "objectId" : "2aa4fc9e-7cd5-41a7-a521-b303ff184303",  
22966   "objectPk" : "mssql",  
22967   "objectKey" : "library^mssql^2aa4fc9e-7cd5-41a7-a521-b303ff184303^mssql",  
22968   "objectTags" : [ "author|Patrik Simek (https://patriksimek.cz)", "maintainer|artl  
22969 }, {
```

## Implementation – v1/NPM: Primary Java Classes

**Cache.java** - implements caching logic, to local disk or Azure Redis Cache

D3CsvBuilder.java - Creates node and edge CSV files for D3.js

Graph.java - An in-memory graph created from a TripleQueryStruct

GraphBuilder.java - Builds a graph by iterating an in-memory TripleQueryStruct

**TripleQueryStruct.java** - Represents **an Array of the Triples** for your graph. It is the “Index”.

**Library.java** - An NPM library document

**Triple.java** - One Triple document

**LibraryRepository.java** - **Spring Data Repository** for Libraries

TripleRepository.java - Spring Data Repository for Libraries

TripleRepositoryExtensions.java - Extensions of the Repository for more complex SQL

TripleRepositoryExtensionsImpl.java

**GraphController.java** - The primary Controller, handles interaction with the UI

# Implementation – v1/NPM: The Spring Data TripleRepository

```
17  @Component
18  @Repository
19  public interface TripleRepository extends CosmosRepository<Triple, String>, TripleRepositoryExtensions {
20      Iterable<Triple> findBySubjectType(String subjectType);
21      Iterable<Triple> findBySubjectLabel(String subjectLabel);
22      1 usage
23      Iterable<Triple> findByTenantAndSubjectLabel(String tenant, String subjectLabel);
24      1 usage
25      @Query("select value count(1) from c")
26      long countAllTriples();
27      1 usage
28      @Query("select value count(1) from c where c.subjectLabel = @subjectLabel")
29      long getNumberOfDocsWithSubjectLabel(@Param("subjectLabel") String subjectLabel);
30      1 usage
31      @Query("select * from c where c.pk = @pk and c.blob = @blob and c.subjectType = @subjectType and c.objectType = @objectType")
32      List<Triple> getByPkBlobAndSubjects(
33          @Param("pk") String pk,          // "pk": "triple|123"
34          @Param("blob") String blob,
35          @Param("subjectType") String subjectType,
36          @Param("objectType") String objectType);
```

Method **getByPkBlobAndSubjects** is used to query the Triples and load them into memory as a **TripleQueryStruct** that can then be **cached**. It is the “Index” to your graph.

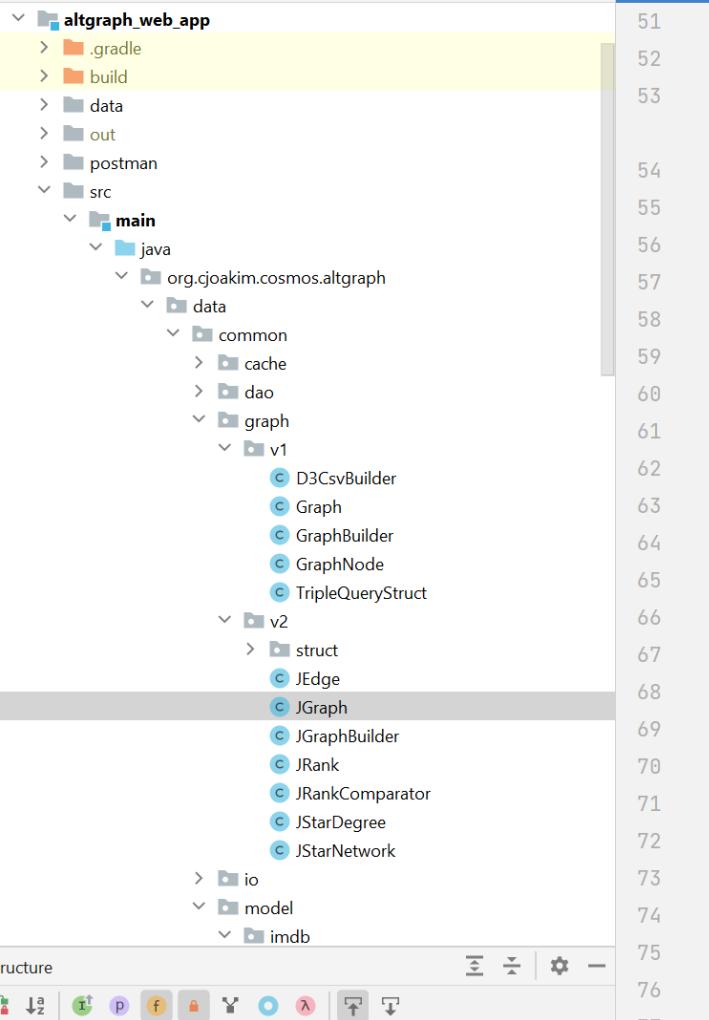
# Implementation – v1/NPM: Building the Graph and Creating D3.js CSV

- Ok, great, we have a **TripleQueryStruct** in memory, now what?
- Optionally Cache it for the next Web Request
  - Class **Cache**
- Build The Graph in Memory
  - Class **GraphBuilder**
  - Iterates, in memory, the many **Triples** in the TripleQueryStruct to build the Graph object
  - Alternatively, for huge graphs, paginate the Triples and build the graph with each page
- Build the two CSV files for D3.js UI visualizations
  - Class **D3CsvBuilder**
- Can we please see the demo now?

# Implementation – v2/IMDb: Primary Java Classes

See the classes in package org.cjoakim.cosmos.altgraph.data.common.graph.v2.

Example shows the use of the JGraphT **DijkstraShortestPath.findPathBetween** method on line 64

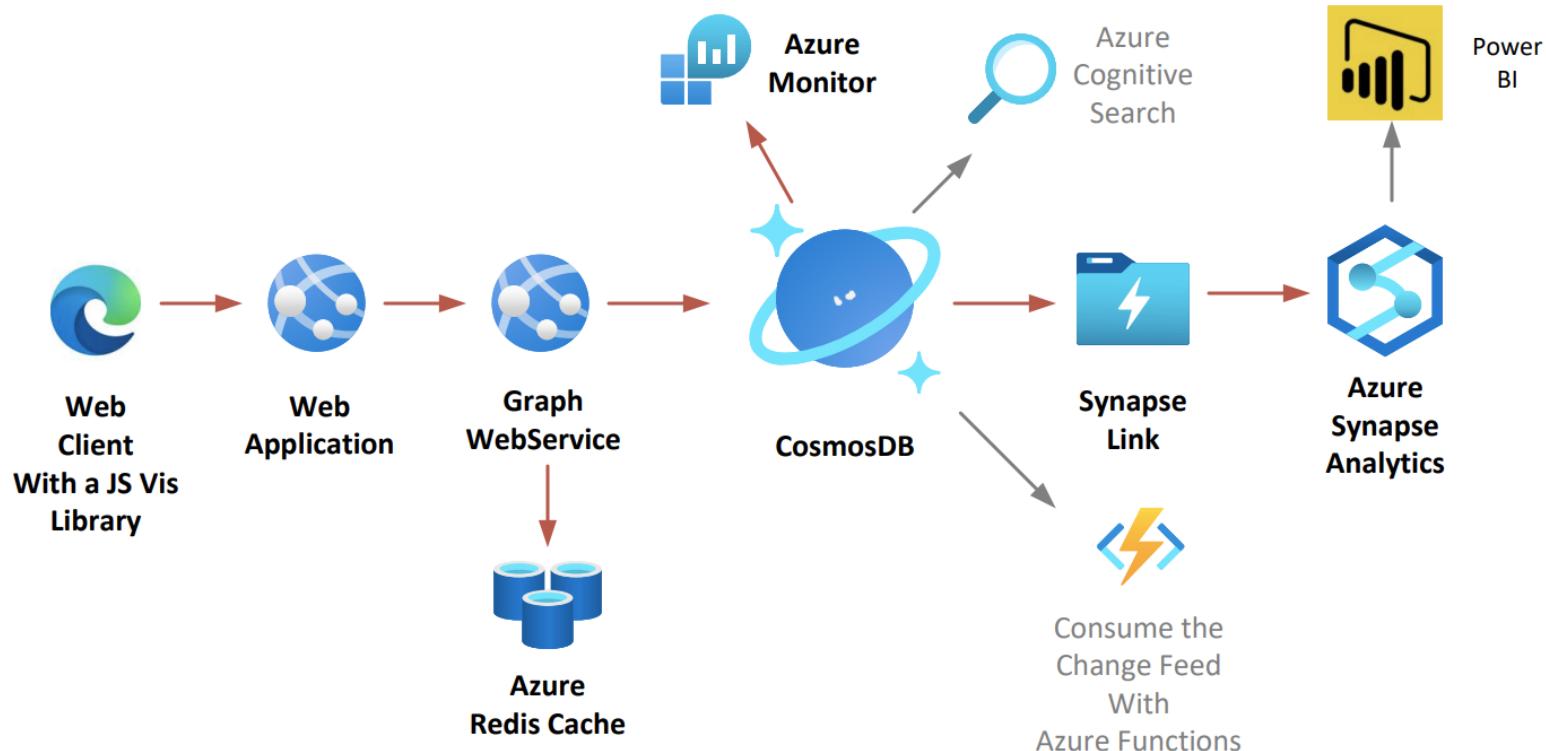


The screenshot shows the file structure of the `altgraph_web_app` project in an IDE. The `src/main/java/org.cjoakim.cosmos.altgraph` package contains several sub-packages: `common`, `data`, `graph`, `v1`, `v2`, `io`, and `model`. The `v2` package is currently selected. Inside `v2`, the `JGraph` class is highlighted. The code editor displays the implementation of the `getShortestPath` method:

```
51  /**
52  * Find the shortest path with the DijkstraShortestPath class in JGraphT.
53  */
54 1 usage
55  public GraphPath<String, DefaultEdge> getShortestPath(String v1, String v2) {
56      log.warn("getShortestPath, v1: " + v1 + " to v2: " + v2);
57      long start = System.currentTimeMillis();
58      if (!isVertexPresent(v1)) {
59          return null;
60      }
61      if (!isVertexPresent(v2)) {
62          return null;
63      }
64      GraphPath<String, DefaultEdge> path =
65          DijkstraShortestPath.findPathBetween(graph, v1, v2);
66      long elapsed = System.currentTimeMillis() - start;
67
68      if (path == null) {
69          log.warn("path is null");
70      } else {
71          log.warn("elapsed milliseconds: " + elapsed);
72          log.warn("path getLength: " + path.getLength());
73          log.warn("path getStartVertex: " + path.getStartVertex());
74          log.warn("path getEndVertex: " + path.getEndVertex());
75      }
76      return path;
77 }
```

# AltGraph Architecture

Recommended solution in **Bold** and with **Red Lines**

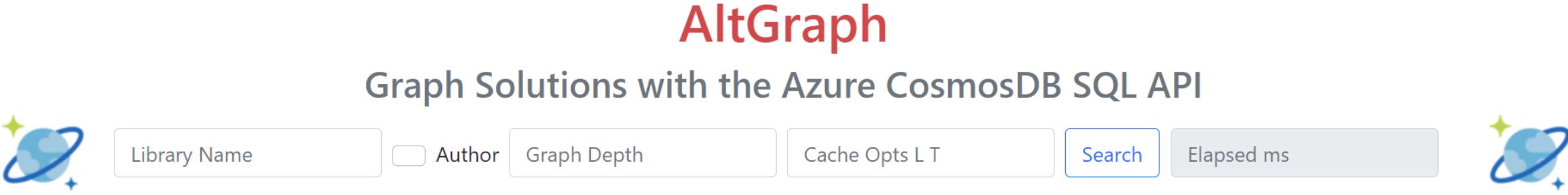


For demonstration purposes, the Web Application and Graph Web Service logic can run in just one **Azure Container Instance**.  
Optionally use the **Cosmos DB Integrated Cache**.

# Demonstration – v1/NPM: The Search Form

## AltGraph

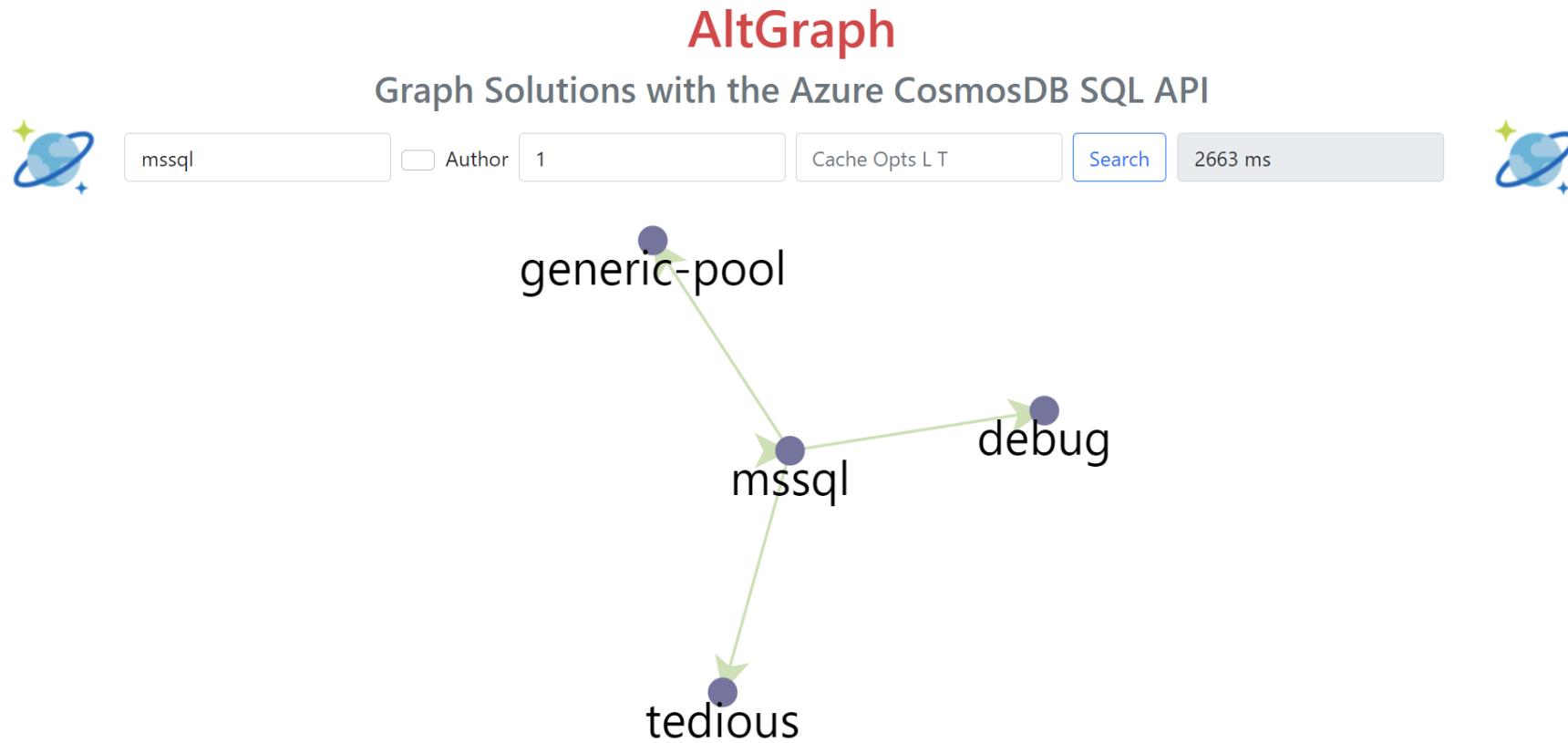
### Graph Solutions with the Azure CosmosDB SQL API



The screenshot shows a search form titled "AltGraph" with the subtitle "Graph Solutions with the Azure CosmosDB SQL API". The form includes input fields for "Library Name" (with placeholder "Enter Library Name"), "Author" (with a checked checkbox), "Graph Depth" (with placeholder "Enter depth"), "Cache Opts L T" (with placeholder "Enter Cache Options"), and an "Elapsed ms" field. A "Search" button is highlighted in blue. Two decorative icons of planets with rings are positioned on either side of the input fields.

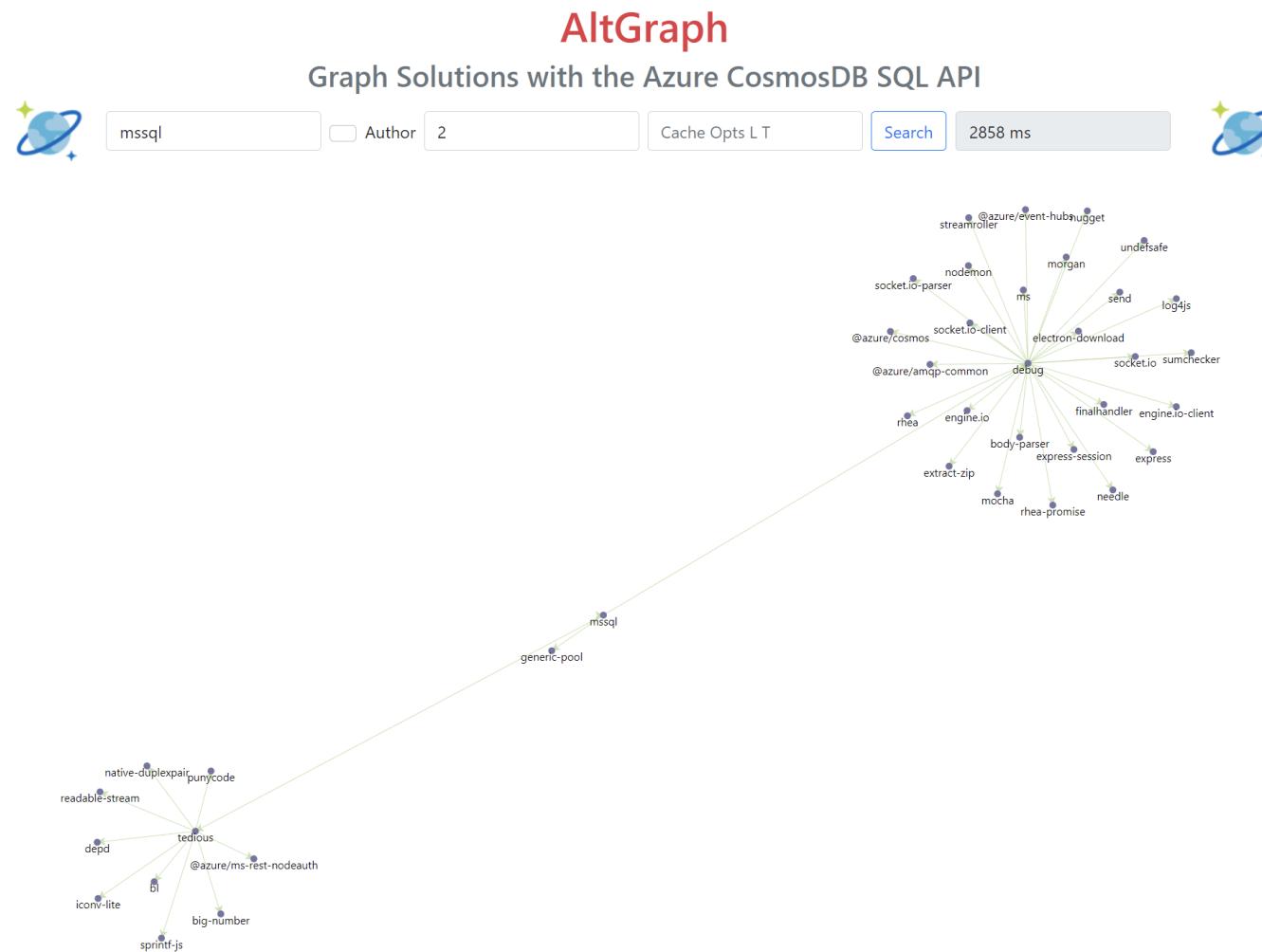
Enter a Library name, and integer graph “depth”.  
Optional **Cache Opts** - “L” for Library caching, “T” for Triple caching.  
The Elapsed ms field will be populated when the graph is displayed.  
The Author checkbox will toggle between a Library and Author graph.

# Demonstration – v1/NPM: Graph with a Depth of 1



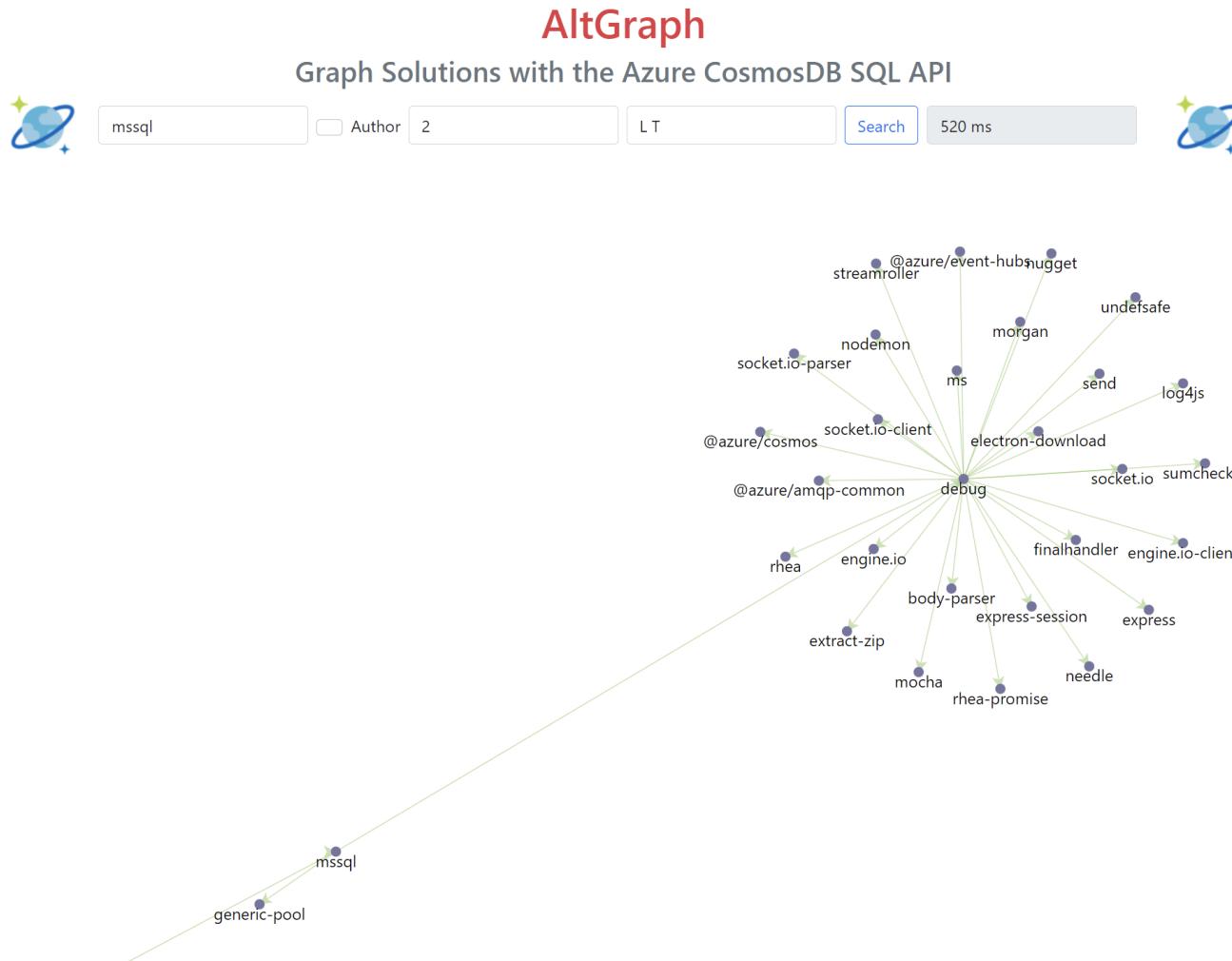
Graph with a depth of 1 and no caching.  
Single click a node for Library info. Double-click to show the graph for that node.

# Demonstration – v1/NPM: Graph with a Depth of 2



Graph with a depth of 2 and no caching. D3.js positions the nodes.

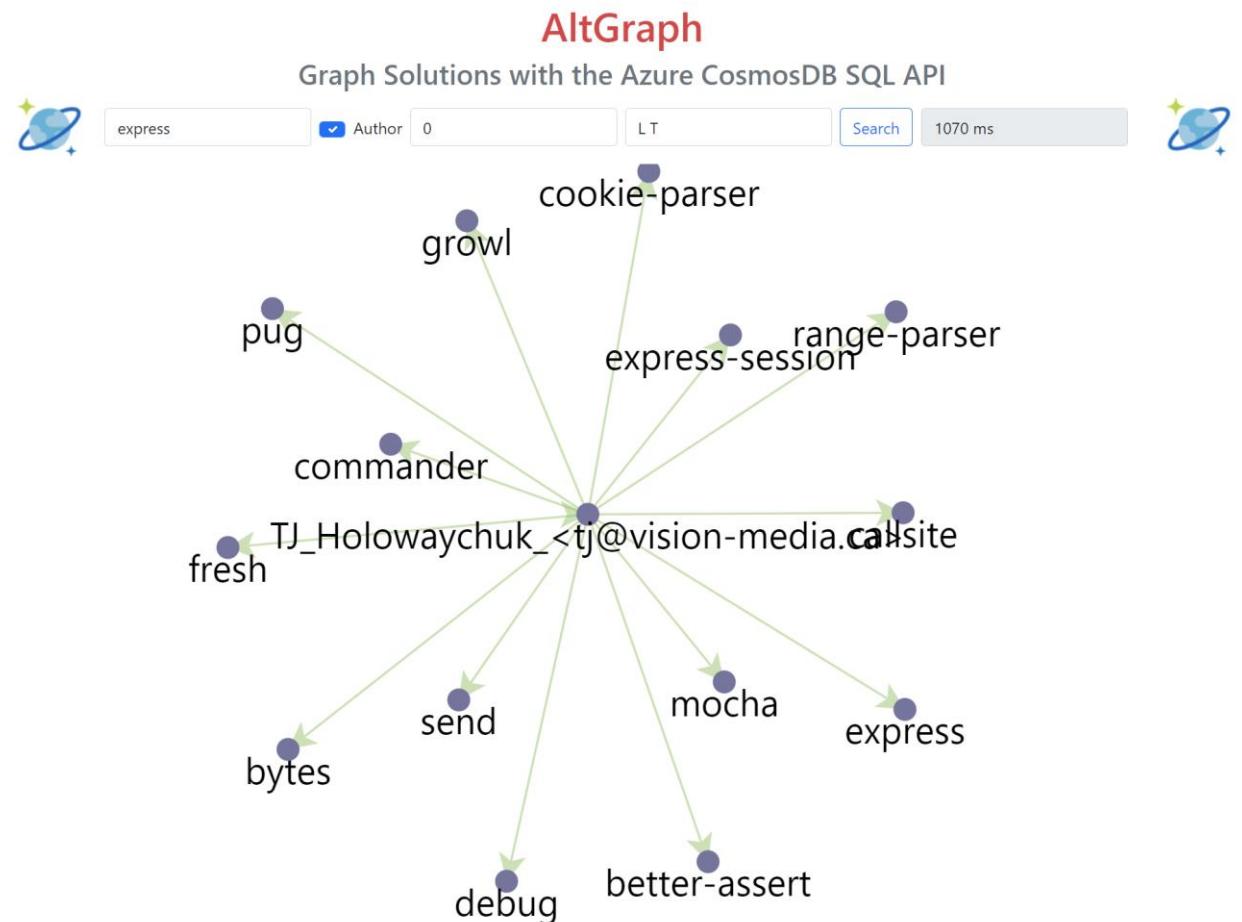
# Demonstration – v1/NPM: Graph with a Depth of 2, with Caching



Graph with a depth of 2 and **caching**. Notice the **speed improvement**.  
This example used Azure Redis Cache from my (slow) home WiFi network.

# Demonstration – v1/NPM: Author-to-Library Graph using the Triple tag values

```
}, {  
  "id" : "7da48a99-23d3-44bf-a878-c3d41c833f32",  
  "pk" : "triple|123",  
  "_etag" : "\"0f001bb8-0000-0100-0000-62d9c5920000\"",  
  "tenant" : "123",  
  "lob" : "npm",  
  "doctype" : "triple",  
  "subjectType" : "library",  
  "subjectLabel" : "pug",  
  "subjectId" : "0b92a30f-8341-4739-8225-a3287afdb54d",  
  "subjectPk" : "pug",  
  "subjectKey" : "library^pug^0b92a30f-8341-4739-8225-a3287afdb54d^pug",  
  "subjectTags" : [ "author|TJ Holowaychuk <tj@vision-media.ca>", "maintainer|forbeslindesay" ],  
  "predicate" : "uses_lib",  
  "objectType" : "library",  
  "objectLabel" : "pug-linker",  
  "objectId" : "3649661e-f7ba-4a57-9b40-4ba3034cdf3b",  
  "objectPk" : "pug-linker",  
  "objectKey" : "library^pug-linker^3649661e-f7ba-4a57-9b40-4ba3034cdf3b",  
  "objectTags" : [ "author|Forbes Lindesay", "maintainer|forbeslindesay" ]  
}, {
```



# Demonstration – v2/IMDB: Graph and JVM Statistics

**AltGraph**  
Graph Solutions with the Azure CosmosDB SQL API



Graph Stats ▾   Search 0

Metric	Value
Date	Wed Dec 07 16:41:29 GMT 2022
In-Memory Vertex Count	1,035,729
In-Memory Edge Count	3,914,288
JVM Total MB	2,048
JVM Free MB	971.491
JVM Pct Free	0.4743608683347702
JVM Max MB	2,048
Elapsed Ms	0
Refresh Date	2022-11-22T21:30:43.227+00:00
Refresh Source	cosmos
Refresh Ms	53750

Query the size and JVM memory-consumption of the graph. This query typically takes 0 to 1 ms.

Note that this JVM graph was loaded on 11/22 and was still in-memory on 12/7.

# Demonstration – v2/IMDB: Graph and JVM Statistics

**AltGraph**  
Graph Solutions with the Azure CosmosDB SQL API

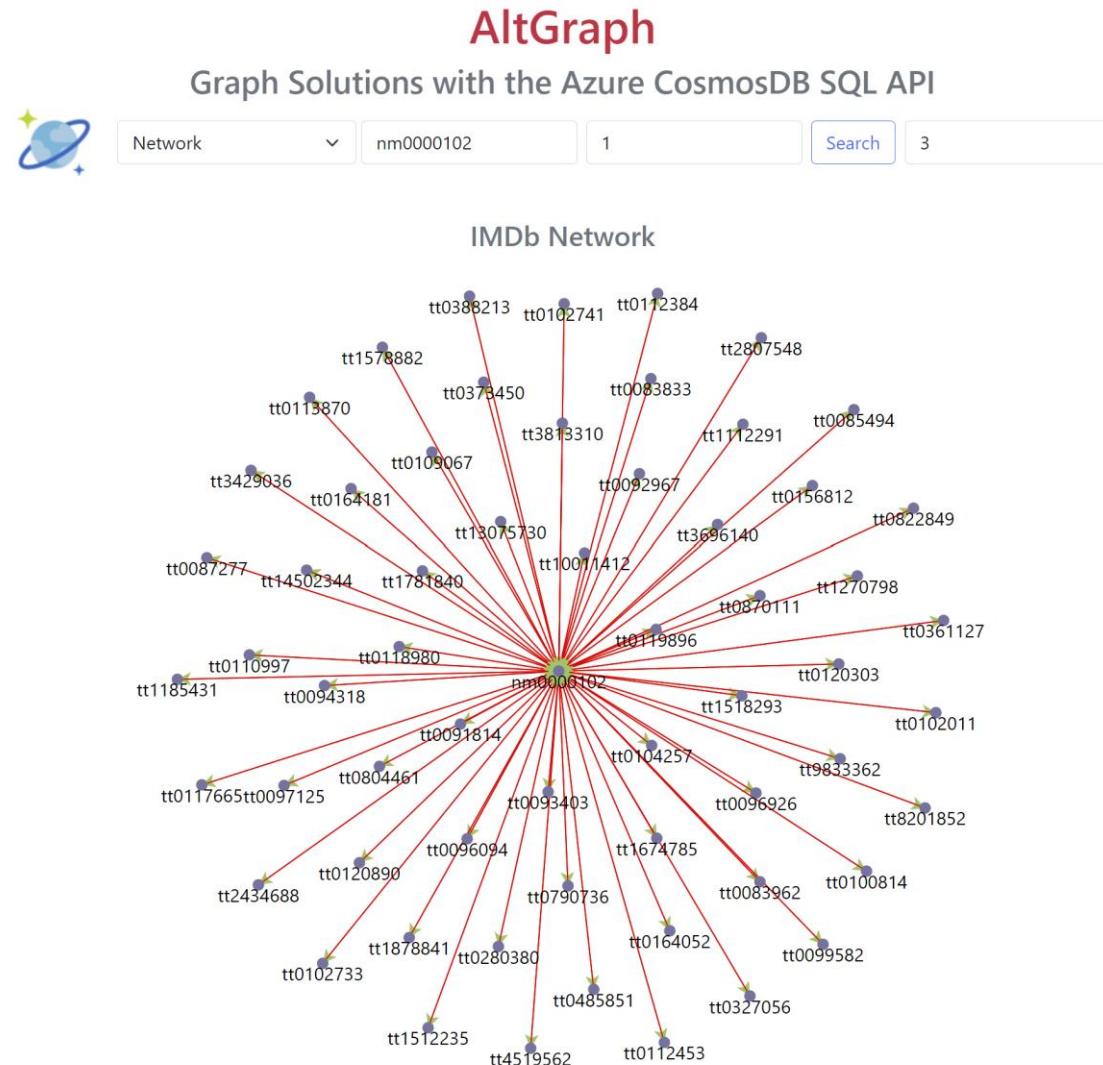


The screenshot shows the AltGraph application interface. At the top left is a blue and white logo. To its right is a dropdown menu labeled "Graph Stats". Next is a "reload" button with a circular arrow icon. To the right of the reload button is a search bar containing the number "51528". Below the header is a table titled "Graph and JVM Statistics" with two columns: "Metric" and "Value". The table lists various metrics such as Date, In-Memory Vertex Count, In-Memory Edge Count, JVM Total MB, JVM Free MB, JVM Pct Free, JVM Max MB, Elapsed Ms, Refresh Date, Refresh Source, and Refresh Ms.

Metric	Value
Date	Wed Dec 07 16:43:36 GMT 2022
In-Memory Vertex Count	1,035,729
In-Memory Edge Count	3,914,288
JVM Total MB	2,048
JVM Free MB	964.3
JVM Pct Free	0.47084957733750343
JVM Max MB	2,048
Elapsed Ms	51,528
Refresh Date	2022-12-07T16:44:27.676+00:00
Refresh Source	cosmos
Refresh Ms	51526

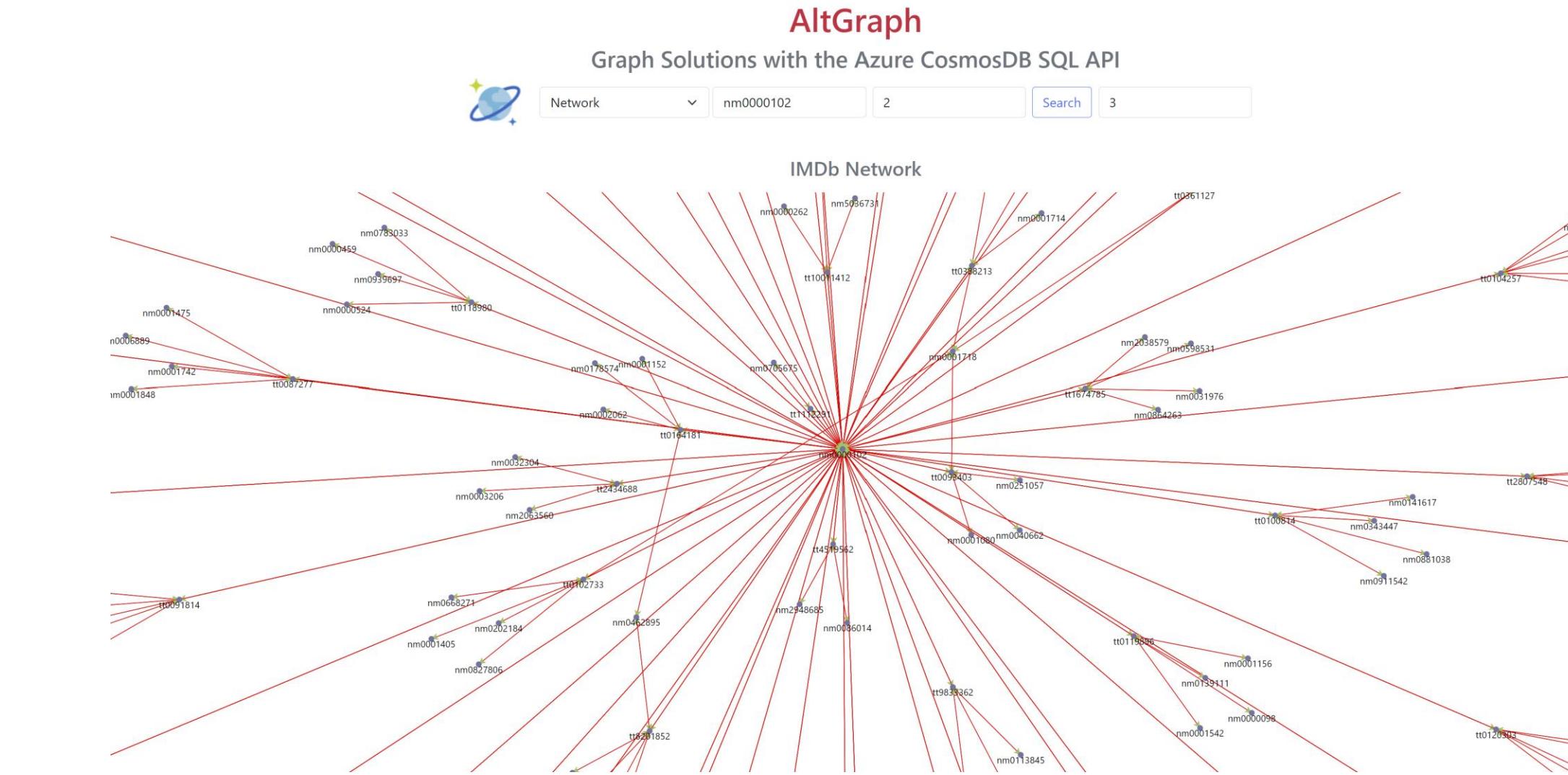
Trigger a reload of the in-memory graph from the Cosmos DB imdb\_seed container.  
The JVM can be running for days/weeks or longer. Note **the 1m Vertices and 3.9m edges**.

# Demonstration – v2/IMDB: Network Graph, “One Degree of Kevin Bacon”

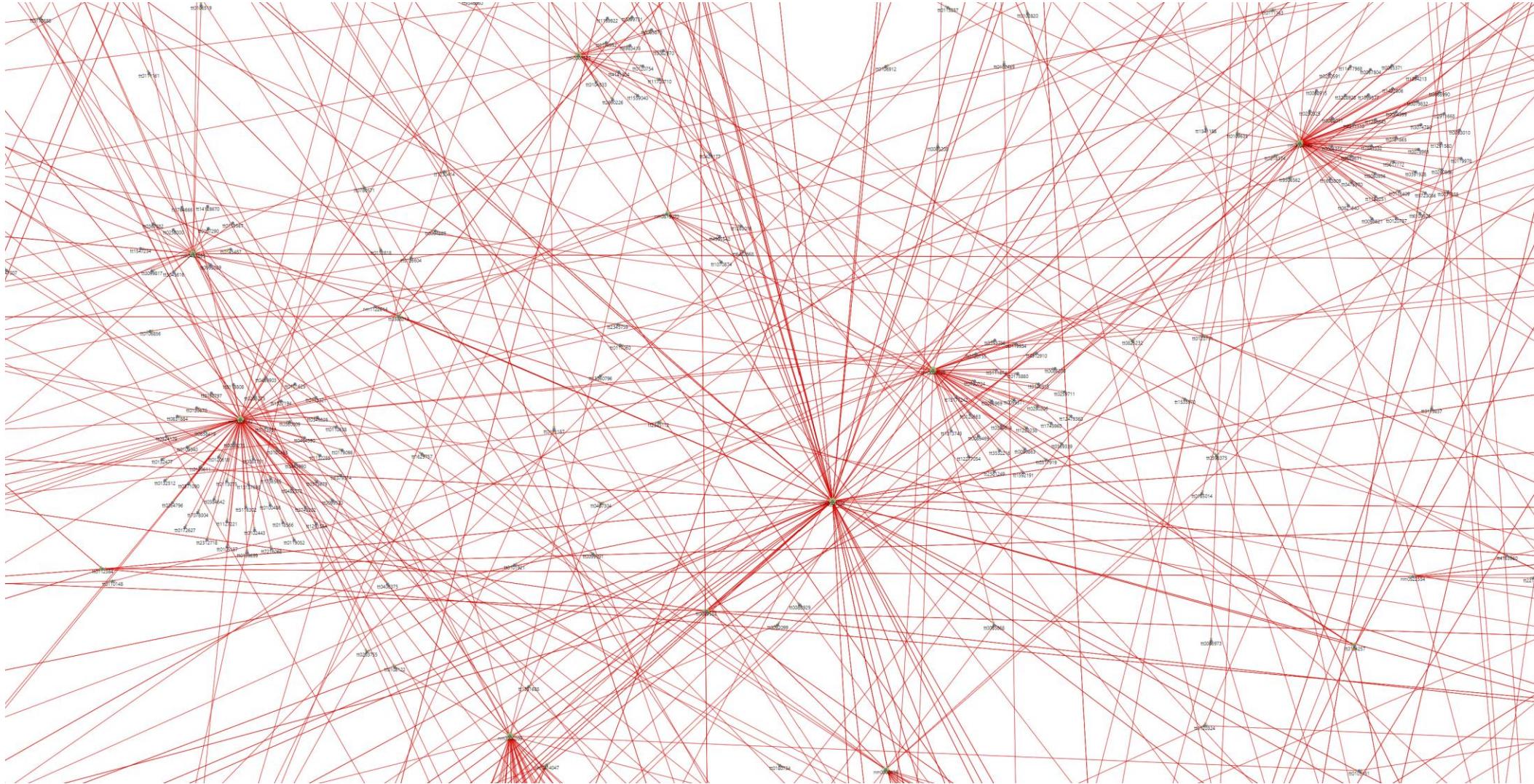


Traverses the adjacent vertices in 3 milliseconds, consumes zero Cosmos DB RUs.

# Demonstration – v2/IMDB: Network Graph, “Two Degrees of Kevin Bacon”



## Demonstration – v2/IMDB: Network Graph, “Three Degrees of Kevin Bacon”



# Demonstration – v2/IMDB: Shortest Path – Kevin Bacon to Charlotte Rampling

**AltGraph**  
Graph Solutions with the Azure CosmosDB SQL API

Shortest Path    nm0000102    nm0001648    Search    87

Shortest Path between Two Vertices

Vertex 1		Vertex 2
<a href="#">nm0000102</a>	-->	<a href="#">tt0097125</a>
<a href="#">nm0949744</a>	-->	<a href="#">tt0097125</a>
<a href="#">nm0949744</a>	-->	<a href="#">tt0381690</a>
<a href="#">nm0001648</a>	-->	<a href="#">tt0381690</a>

```
graph TD; nm0000102 --> tt0097125; tt0097125 --> nm0949744; nm0949744 --> tt0381690; tt0381690 --> nm0001648
```

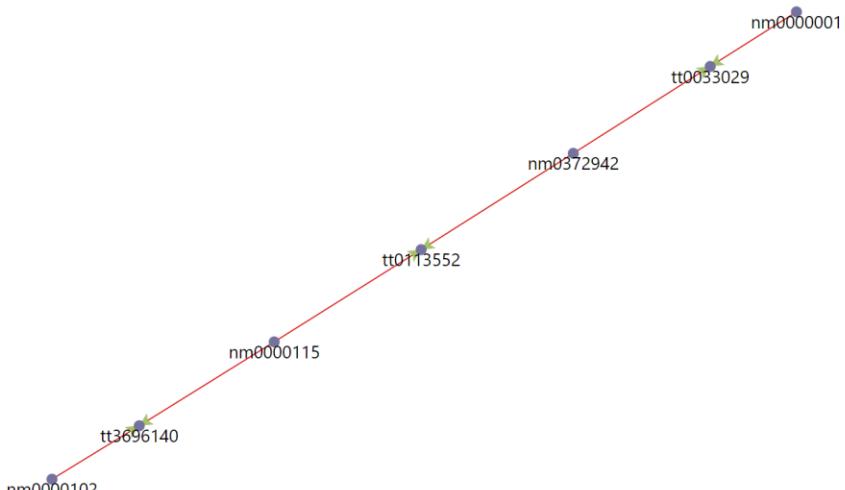
# Demonstration – v2/IMDB: Shortest Path – Kevin Bacon to Fred Aistaire

**AltGraph**  
Graph Solutions with the Azure CosmosDB SQL API

 Shortest Path  nm0000102   1048

Shortest Path between Two Vertices

Vertex 1	-->	Vertex 2
<a href="#">nm0000102</a>	-->	<a href="#">tt3696140</a>
<a href="#">nm0000115</a>	-->	<a href="#">tt3696140</a>
<a href="#">nm0000115</a>	-->	<a href="#">tt0113552</a>
<a href="#">nm0372942</a>	-->	<a href="#">tt0113552</a>
<a href="#">nm0372942</a>	-->	<a href="#">tt0033029</a>
<a href="#">nm0000001</a>	-->	<a href="#">tt0033029</a>



# Demonstration – v2/IMDB: Page Rank Algorithm

**AltGraph**  
Graph Solutions with the Azure CosmosDB SQL API



PageRank  100   3743

Vertex PageRanks

Rank	Vertex	Value	Comment
1	<a href="#">nm0183659</a>	0.00014375625797007131	
2	<a href="#">nm0756966</a>	0.00010948154781558972	
3	<a href="#">nm0644554</a>	0.00006151494573422499	
4	<a href="#">nm0462051</a>	0.00005744026497552381	
5	<a href="#">nm0457554</a>	0.000056200687467845174	
6	<a href="#">nm0619107</a>	0.00005216369278385738	
7	<a href="#">nm0997109</a>	0.00004890868058151003	
8	<a href="#">nm0297876</a>	0.000042653495383222324	
9	<a href="#">nm0103977</a>	0.000042239964918056494	
10	<a href="#">nm3266654</a>	0.00003957139305846257	
11	<a href="#">nm0793813</a>	0.000039275210592778356	



# Thank you!

# Questions?