# Azure CosmosDB – AltGraph

Chris Joakim, Microsoft, CosmosDB 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

# What is AltGraph?

**AltGraph is an _Alternative Graph Implementation_ built on:**

A Design:
- Azure **CosmosDB SQL API**
- RDF-like **"Triples"**
- Azure Redis Cache or CosmosDB Integrated **Cache**
- **Fast In-memory processing** vs DB and Disk Traversal

A Reference Implementation:
- Java programming language
- **Spring Boot** and **Spring Data** frameworks
- https://github.com/cjoakim/azure-cosmosdb-altgraph

# Presentation Outline

- **Influences**
  - Previous CosmosDB Live TV Sessions
  - Real-world Use Cases

- **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

**Previous CosmosDB 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

**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

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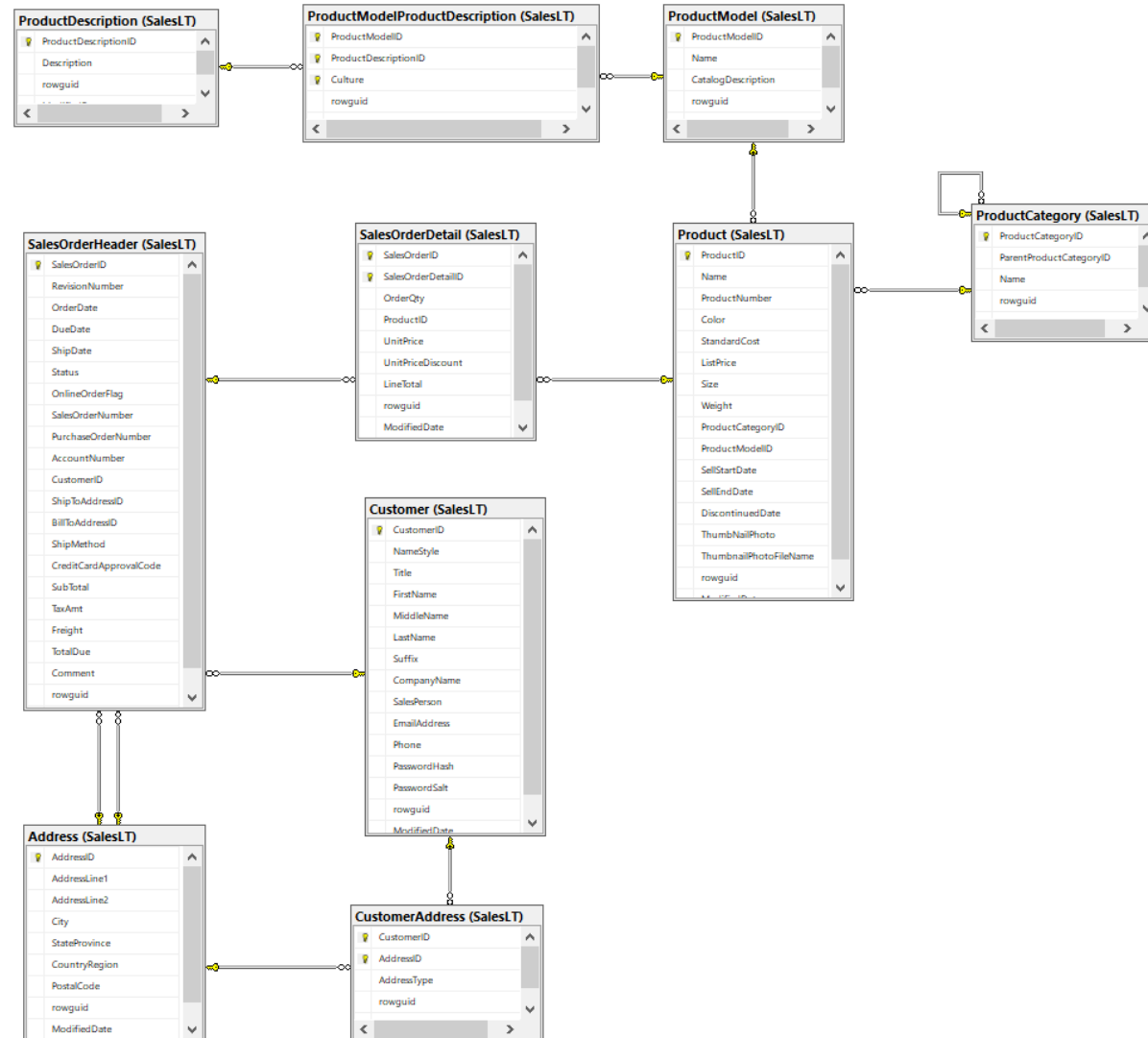
# Perception: Or if the problem was drawn like this?
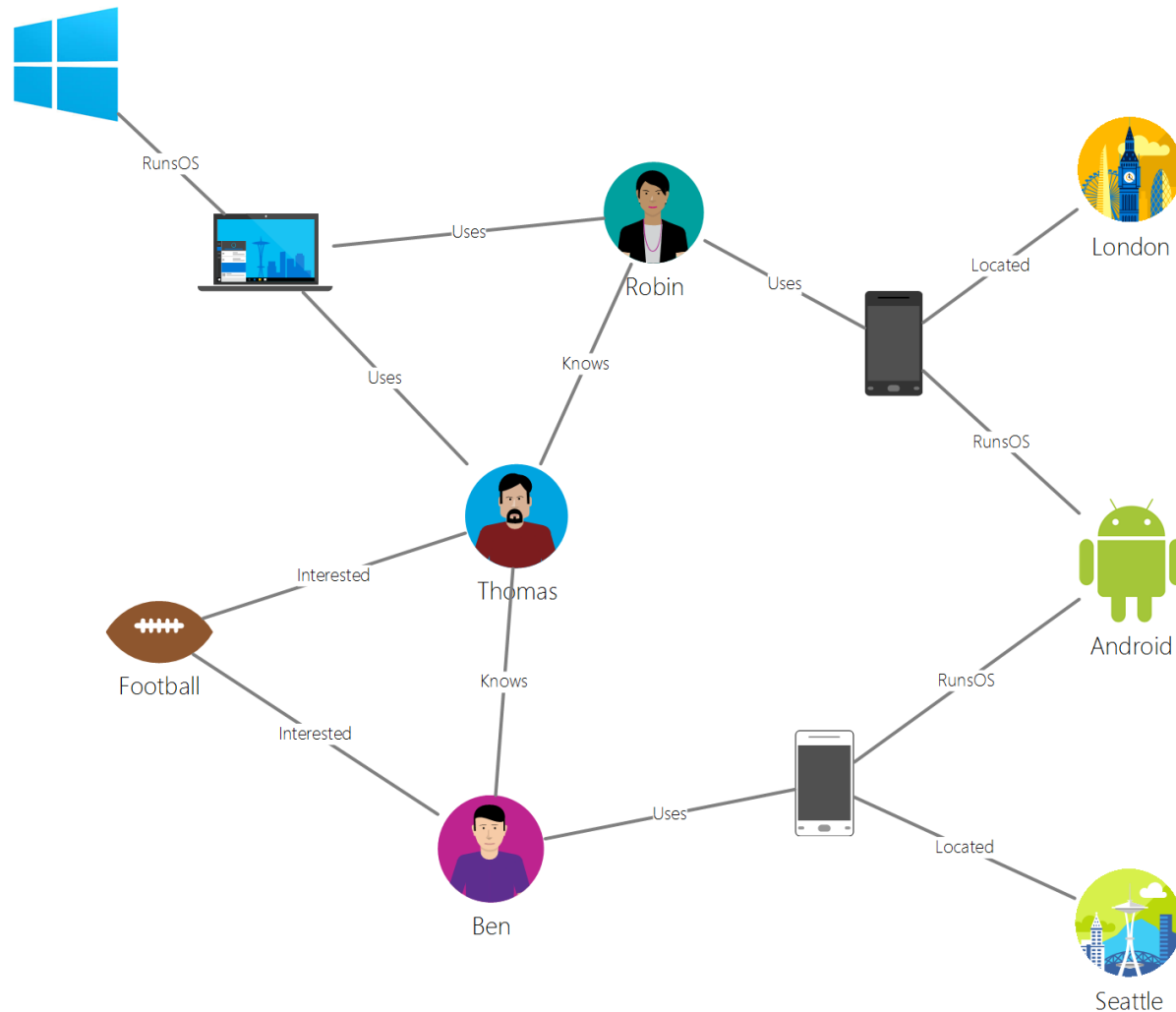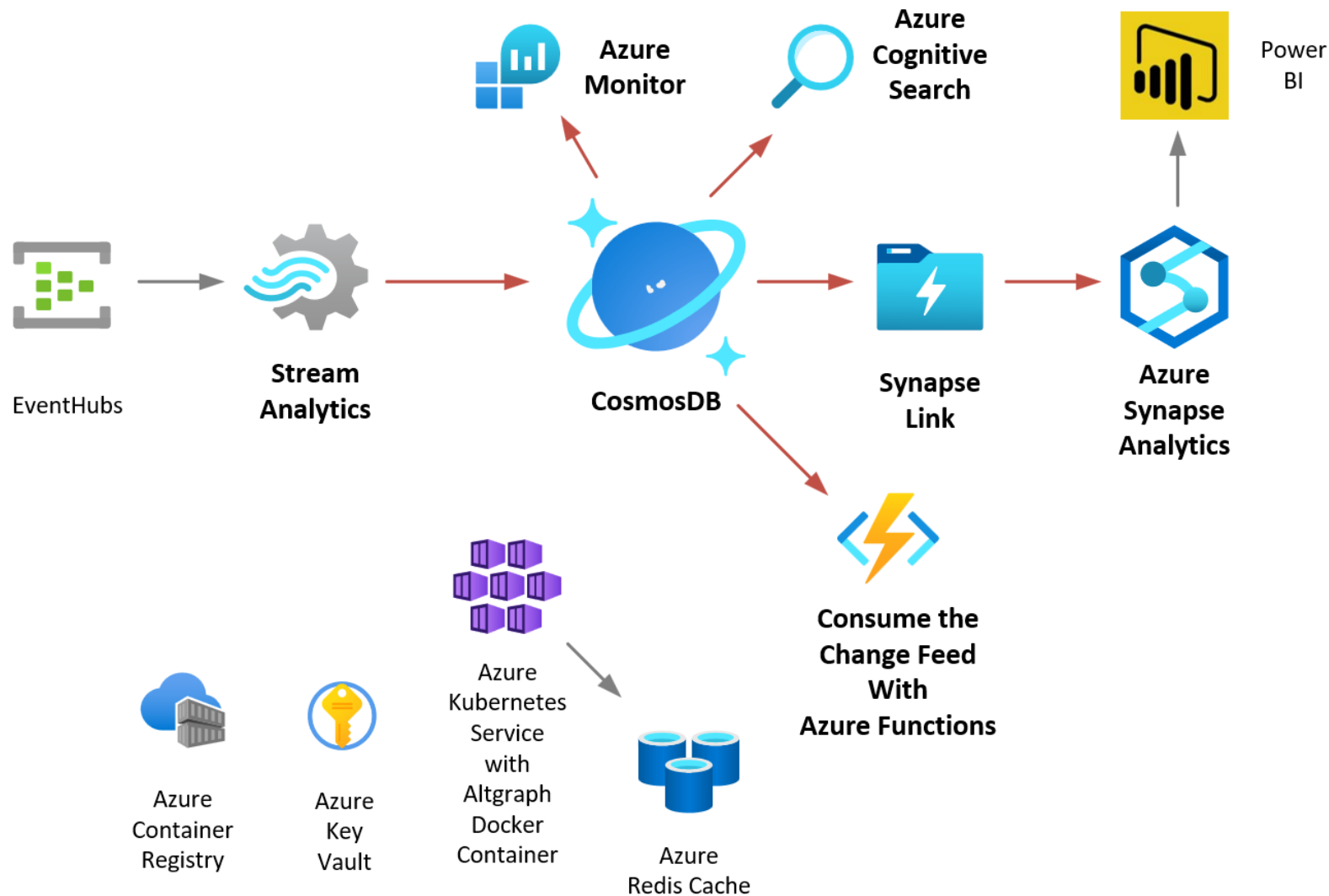


I see this a lot in the field.

# AltGraph Architecture and CosmosDB SQL API Integrations

Direct Integrations in **Bold** and with **Red Lines**

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

**Database Integrations** are important.

The **CosmosDB SQL API** offers excellent integration with other Azure Paas Services.

EventHubs

**Stream Analytics**

Azure Monitor

Azure Cognitive Search

Power BI

CosmosDB

**Synapse Link**

Azure Synapse Analytics

Consume the Change Feed With Azure Functions

Azure Container Registry

Azure Key Vault

Azure Kubernetes Service with Altgraph Docker Container

Azure Redis Cache

©Microsoft Corporation
Azure

# Database Solutions

- **Types of Databases**

  - **Relational**: Transactional use-cases
  - **Graph**: Graph-specific use-cases.  **RDF** (**triplestores**) and **LPG** (**vertices, edges**)
  - **NoSQL:** including the CosmosDB SQL API: **General Purpose**

- **Think Differently; Why another Graph Solution?**

  - **Fast execution speed**, and **lower CosmosDB RU costs**
  - Lower barrier to entry for new apps: **conceptual simplicity, based on SQL**
  - **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
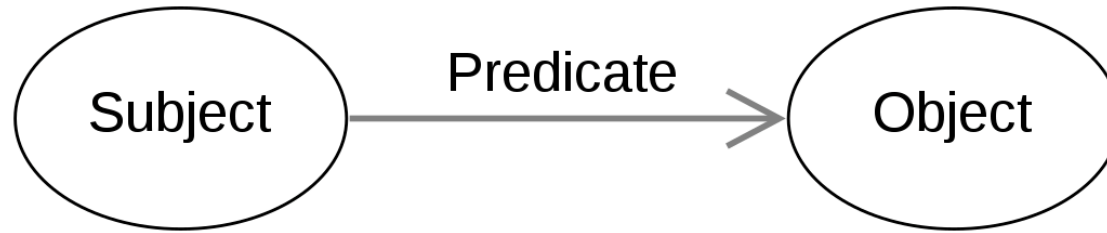


**Examples:**

```
Microsoft      is_a                 Technology Company
Java           is_a                 Programming Language
C              is_a                 Programming Language
CosmosDB       is_a                 Database System
CosmosDB       is_a                 NoSQL Database System
CosmosDB       has_a_sdk_for        Java
CosmosDB       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.

# Design Foundations: CosmosDB Partitioning
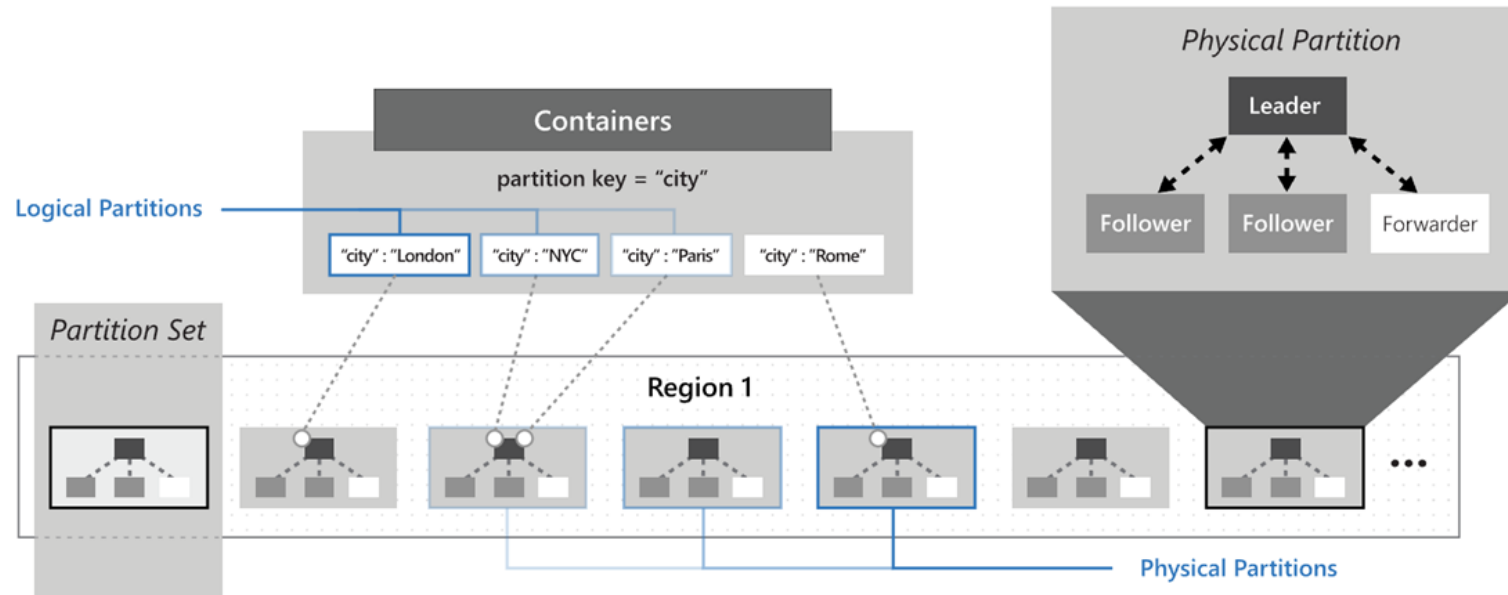


Reads within the same logical and physical partition are faster.
The Triples (see following pages) can reside in the same logical partition.

# Design Foundations:  Performance Optimizations

- **CosmosDB Indexing and Composite Indexes**
  Index individual attributes, and as well as sets of attributes (i.e. – composite indexes)
  to match your queries

- **CosmosDB "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/
  - **CosmosDB 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 CosmosDB 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:  D3.js

- **D3.js JavaScript library for in-browser data visualizations**
  - Many out-of-the-box visualizations.  Open-source.  https://d3js.org
  - Or, Bring-Your-Own-UI-Library
  - https://learn.microsoft.com/en-us/azure/cosmos-db/graph/graph-visualization-partners
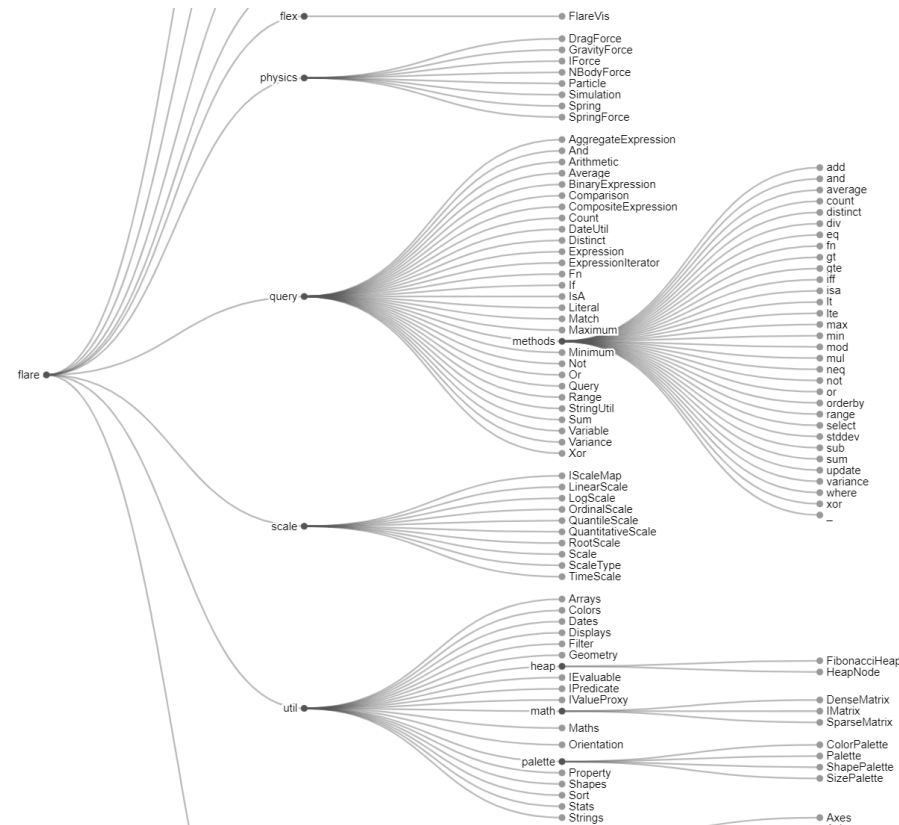
# Design Foundations: My Previous Implementation

- **CosmosDB Gremlin API Implementation of a Node.js NPM "Bill-of-Material" Graph**
  - I wanted to use the **same data** in AltGraph, but re-implement the solution w/AltGraph
  - https://github.com/Azure-Samples/azure-cosmos-db-graph-npm-bom-sample
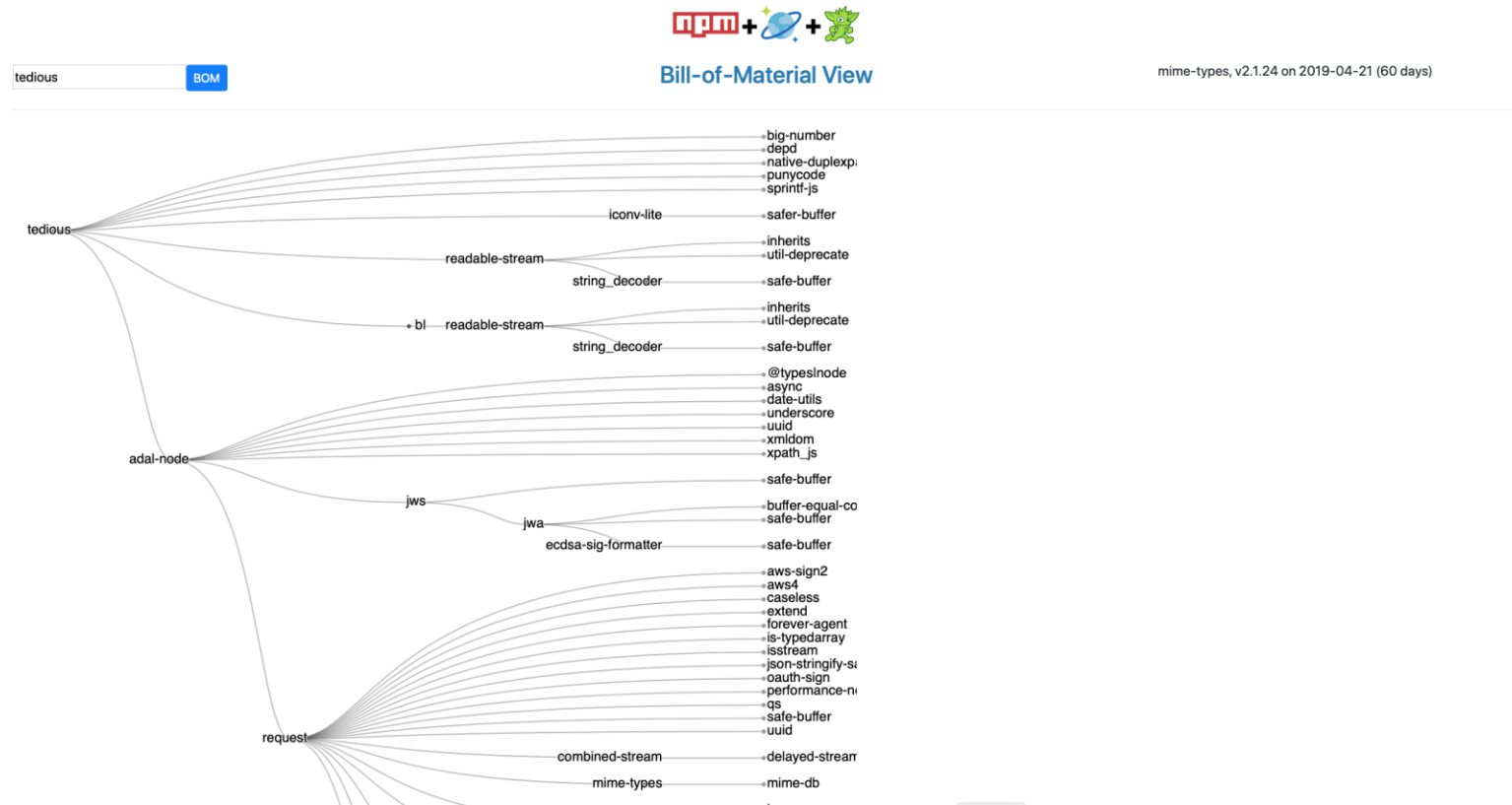
# Implementation:  CosmosDB SQL API

- **Use a Single Container:  altgraph**

  - 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 CosmosDB 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: 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.

```json
{
    "doctype" : "library",
    "label" : "tcx-js",
    "id" : "f0b734d9-3240-44c5-9868-cb25597f1e3b",
    "pk" : "tcx-js",
    "_etag" : "\"9c00f125-0000-0100-0000-62d9c5440000\"",
    "tenant" : "123",
    "lob" : "npm",
    "cacheKey" : "library|tcx-js",
    "graphKey" : "library^tcx-js^f0b734d9-3240-44c5-9868-cb25597f1e3b^tcx-js",
    "name" : "tcx-js",
    "desc" : "A Node.js library for parsing TCX/XML files, such as from a Garmin GPS device.",
    "keywords" : [ "tcx", "garmin", "forerunner", "gps" ],
    "dependencies" : {
        "xml2js" : "^0.4.19"
    },
    "devDependencies" : {
        "mocha-multi-reporters" : "^1.1.7",
        "chai" : "^4.2.0",
        ... others omitted ...
        "typescript" : "^3.5.2"
    },
    "author" : "Chris Joakim",
    "maintainers" : [ "cjoakim <christopher.joakim@gmail.com>" ],
    "version" : "1.0.1",
    "versions" : [ "0.0.1", "0.1.0", "0.1.1", "0.1.2", "1.0.0", "1.0.1" ],
    "homepage" : "https://github.com/cjoakim/tcx-js",
    "library_age_days" : 1755,
    "version_age_days" : 32
}
```

# Implementation:  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 **logicical 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>", "maintainer
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|art
22969        }, {
```

# Implementation:  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: The Spring Data TripleRepository

```java
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);
            1 usage
22          Iterable<Triple> findByTenantAndSubjectLabel(String tenant, String subjectLabel);
            1 usage
23          @Query("select value count(1) from c")
24          long countAllTriples();
            1 usage
25          @Query("select value count(1) from c where c.subjectLabel = @subjectLabel")
26          long getNumberOfDocsWithSubjectLabel(@Param("subjectLabel") String subjectLabel);
            1 usage
27          @Query("select * from c where c.pk = @pk and c.lob = @lob and c.subjectType = @subjectType and c.objectType = @objectType")
28          List<Triple> getByPkLobAndSubjects(
29                  @Param("pk") String pk,       // "pk": "triple|123"
30                  @Param("lob") String lob,
31                  @Param("subjectType") String subjectType,
32                  @Param("objectType") String objectType);
33
```

Method **getByPkLobAndSubjects** 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: 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?**

# Demonstration:  The Search Form

## AltGraph

### Graph Solutions with the Azure CosmosDB SQL API

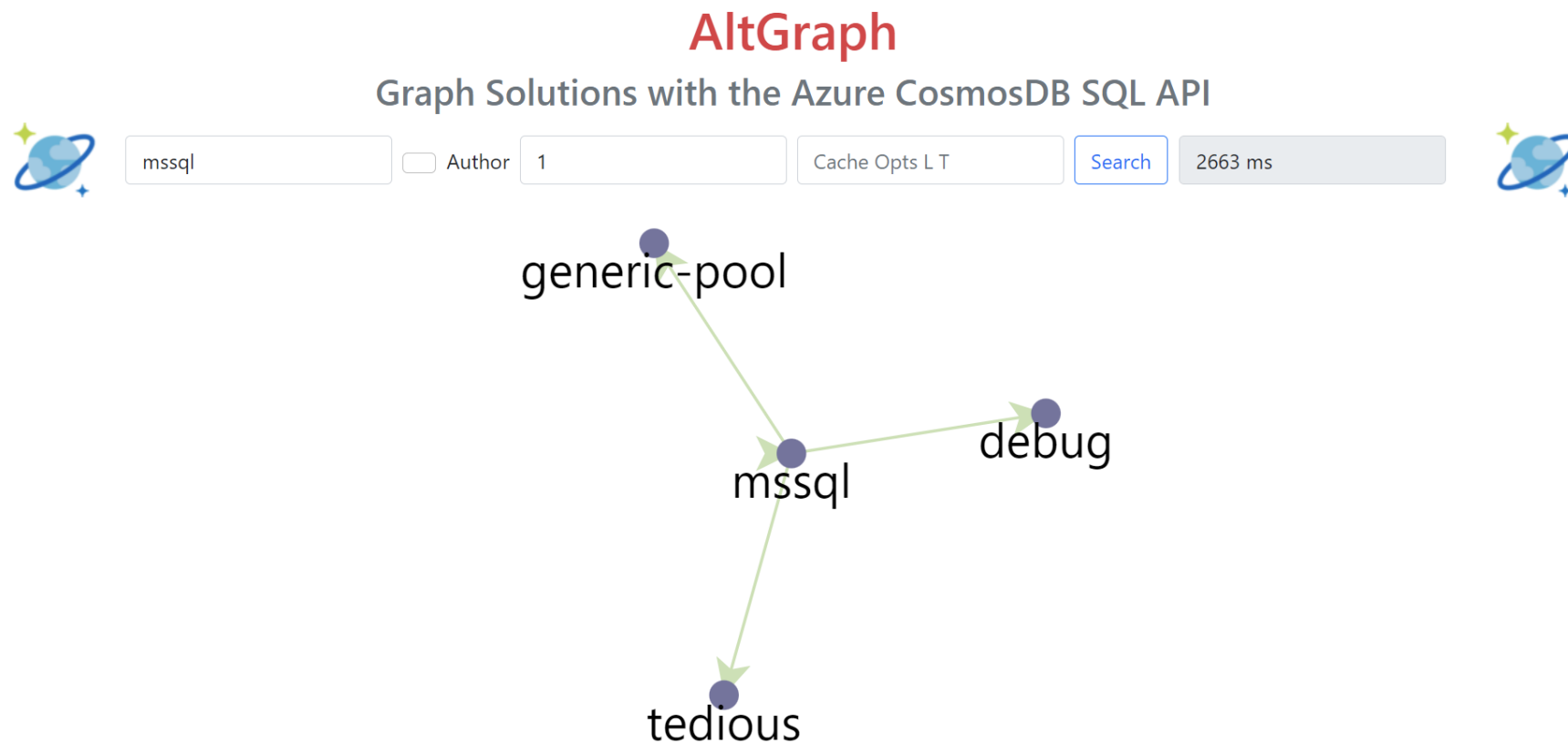| Library Name | ☐ Author | Graph Depth | Cache Opts L T | Search | Elapsed ms |

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.

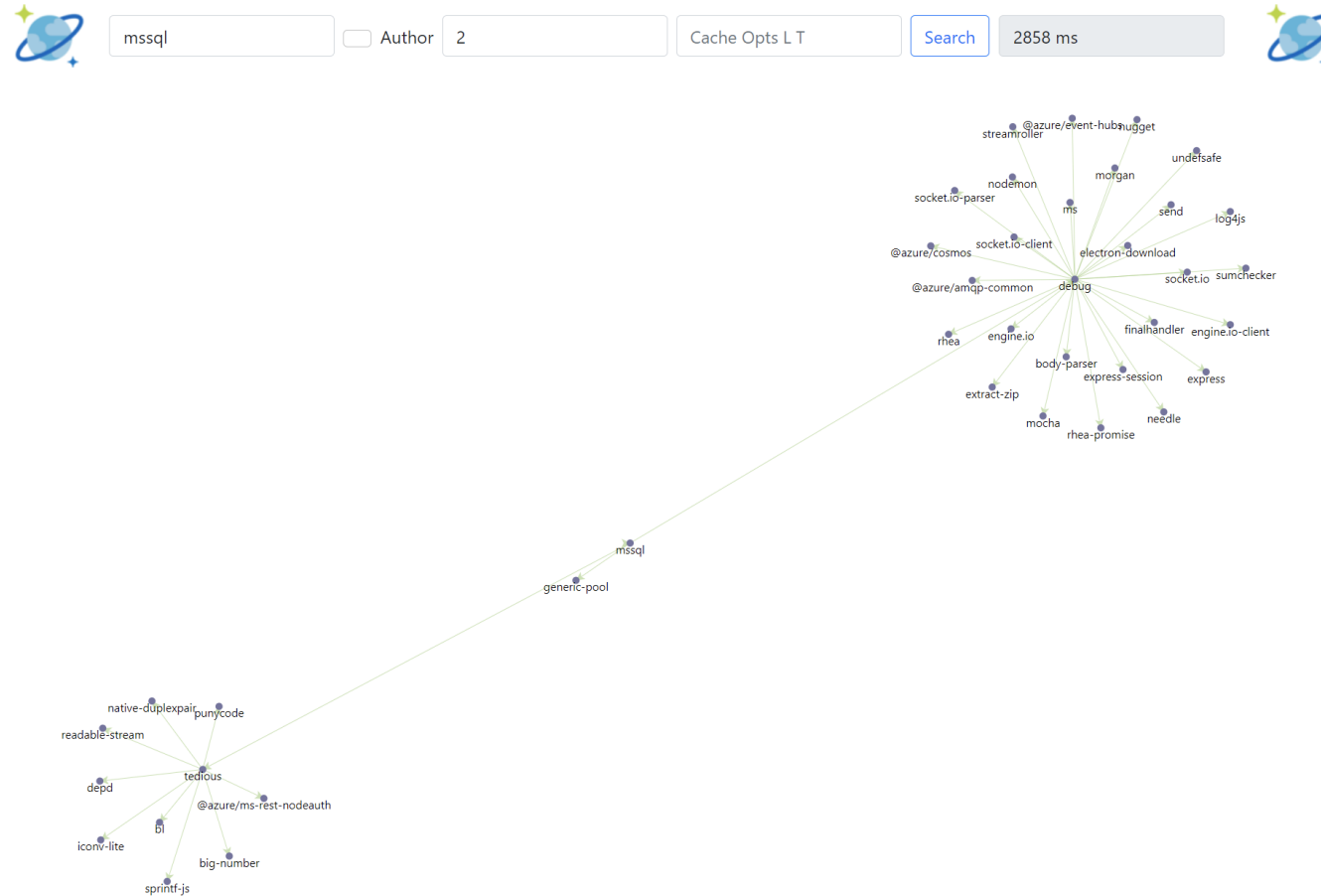# Demonstration:  Graph with a Depth of 1



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:  Graph with a Depth of 2
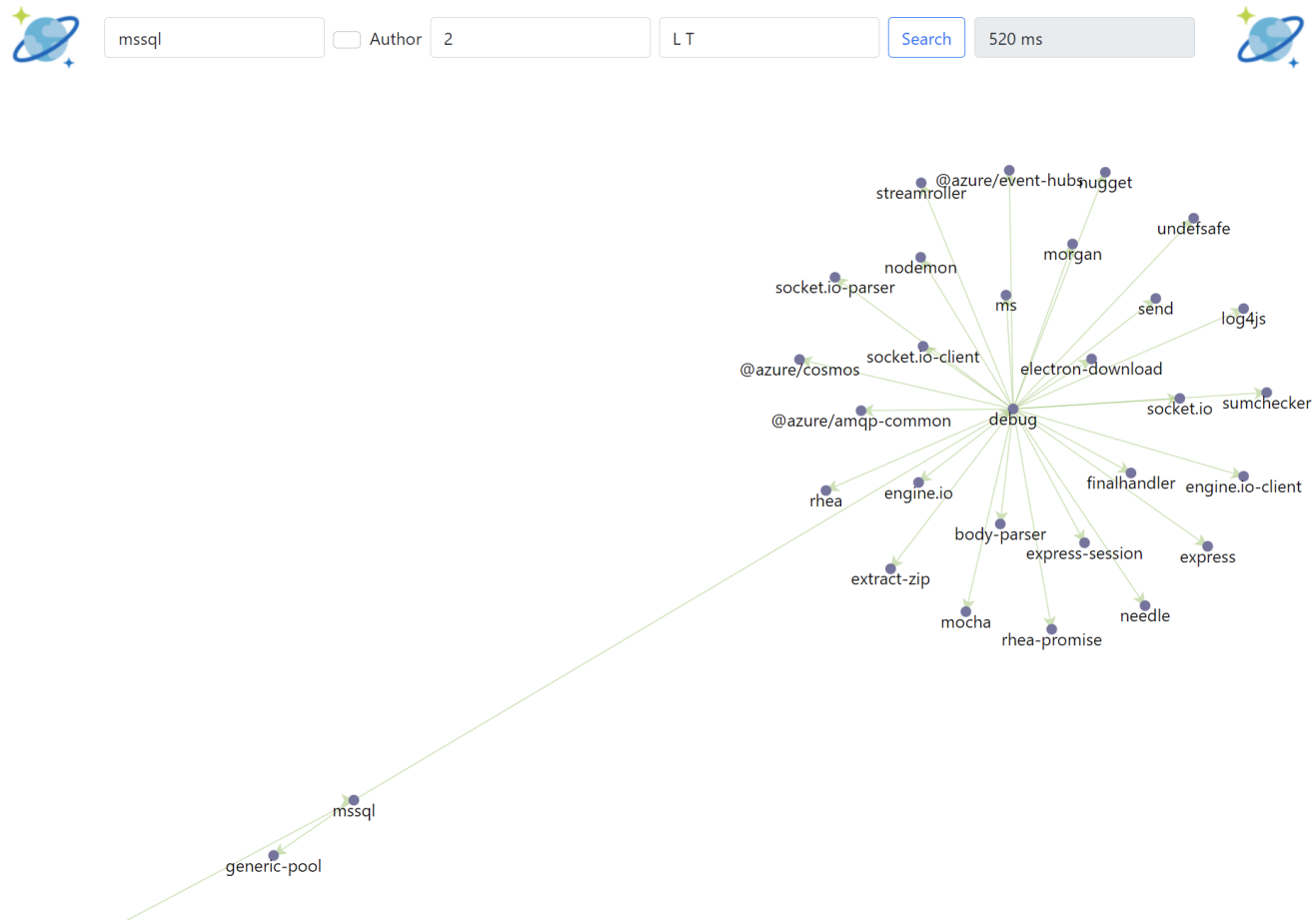


Graph with a depth of 2 and no caching.  D3.js positions the nodes.

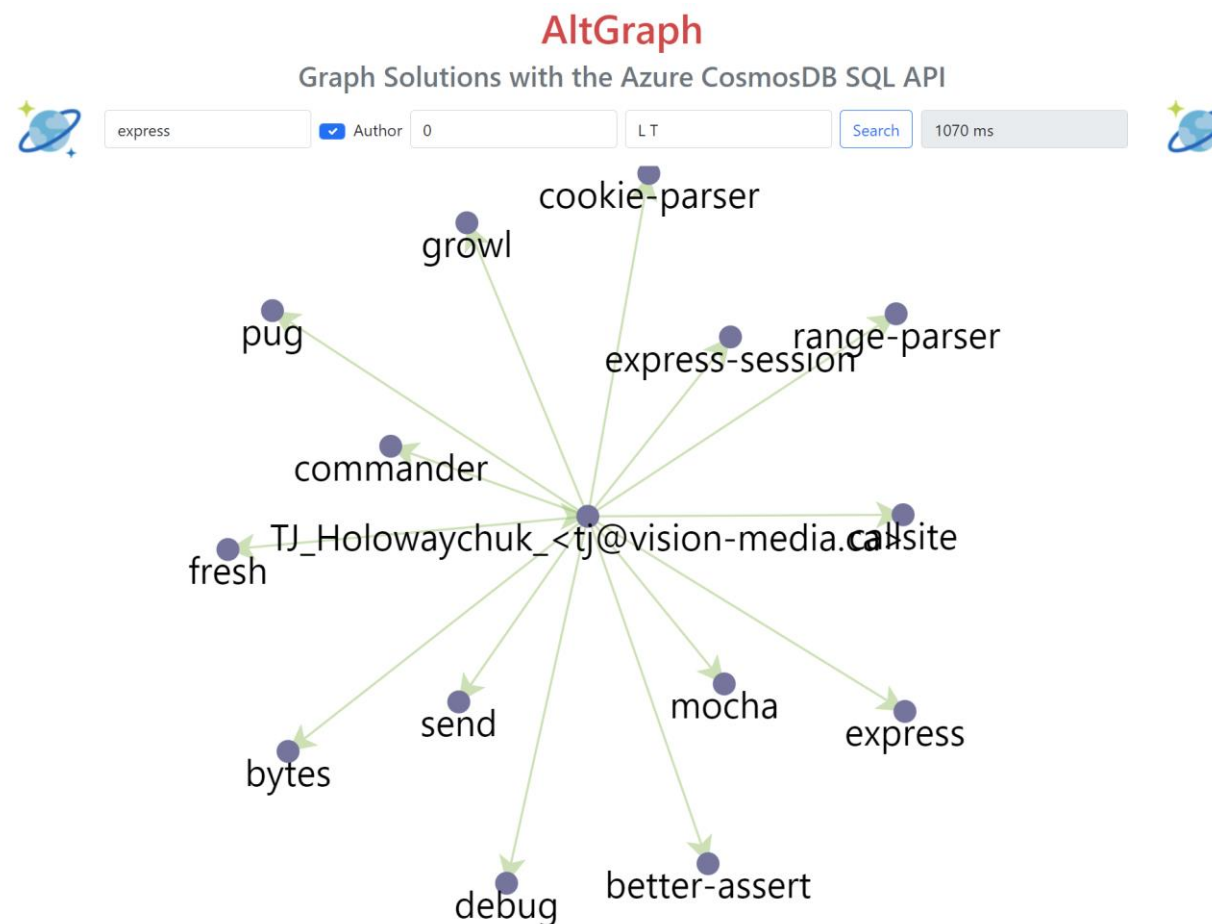# Demonstration: Graph with a Depth of 2, with Caching



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: 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>", "mainta
  "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 '
}, {
```

AltGraph
Graph Solutions with the Azure CosmosDB SQL API

Graph, from tags, showing the Libraries authored by TJ Holowaychuk

**Microsoft**

# Thank you!

# Questions?