Document DB (On Azure)

Dan Matthews

Overview

- Schema-free
- Storage
- Indexing Policies
- Sharding

Azure Storage Options

- Azure SQL
- DocumentDB
- Azure Tables

Meet Azure DocumentDB

- Azure DocumentDB is a NoSQL document database service designed from the ground up to natively support JSON and JavaScript directly inside the database engine.
- It is great for applications that run in the cloud, when predictable throughput, low latency, and flexible queries.

Schema-free JSON data

- A common problem for developers is that application schemas constantly evolve.
- Schema: A database schema is a structure in a formal language supported by the database management system that refers to the organization of data as a blue print of how a database is constructed.
  - Divided into database tables in the case of relational databases
- DocumentDB automatically indexes all JSON documents added to the database, then lets you use familiar SQL syntax to query them without specifying schema or secondary indices up front.

Benefits of schema-free

- One benefit is that you can often represent in a single entity a construct that would require several tables to properly represent in a relational db.
- No schema migrations: Since DocumentDB is schema free, your code will define your schema
- Also, there is a clear path to horizontal scalability which can improve the speed of queries on large data sets
- Even Hadoop jobs over data stored in DocumentDB are supported using the connector

Storage

- DocumentDB stores its data as a native JSON Object
  - This allows attachments to documents, while azure tables (Azures other NoSQL storage option) stores data in XML.
  - The advantage of storing native JSON objects is that the data maps directly into JSON objects in the client application code
- DocumentDB can store petabytes of information.
  - One petabyte is approximately 2^50 bytes, or 2,ooo terabytes

DocumentDB Indexing Policies

- DocumentDB is a true schema-free database. It does not assume or require any schema for the JSON documents it indexes.
  - This allows you to quickly define and iterate on application data models
- As you add documents to a collection, DocumentDB automatically indexes all document properties so they are available for you to query.
  - Automatic indexing also allows you to store heterogeneous types of documents.
  - DocumentDB supports a sustained volume of fast writes while still serving consistent queries.

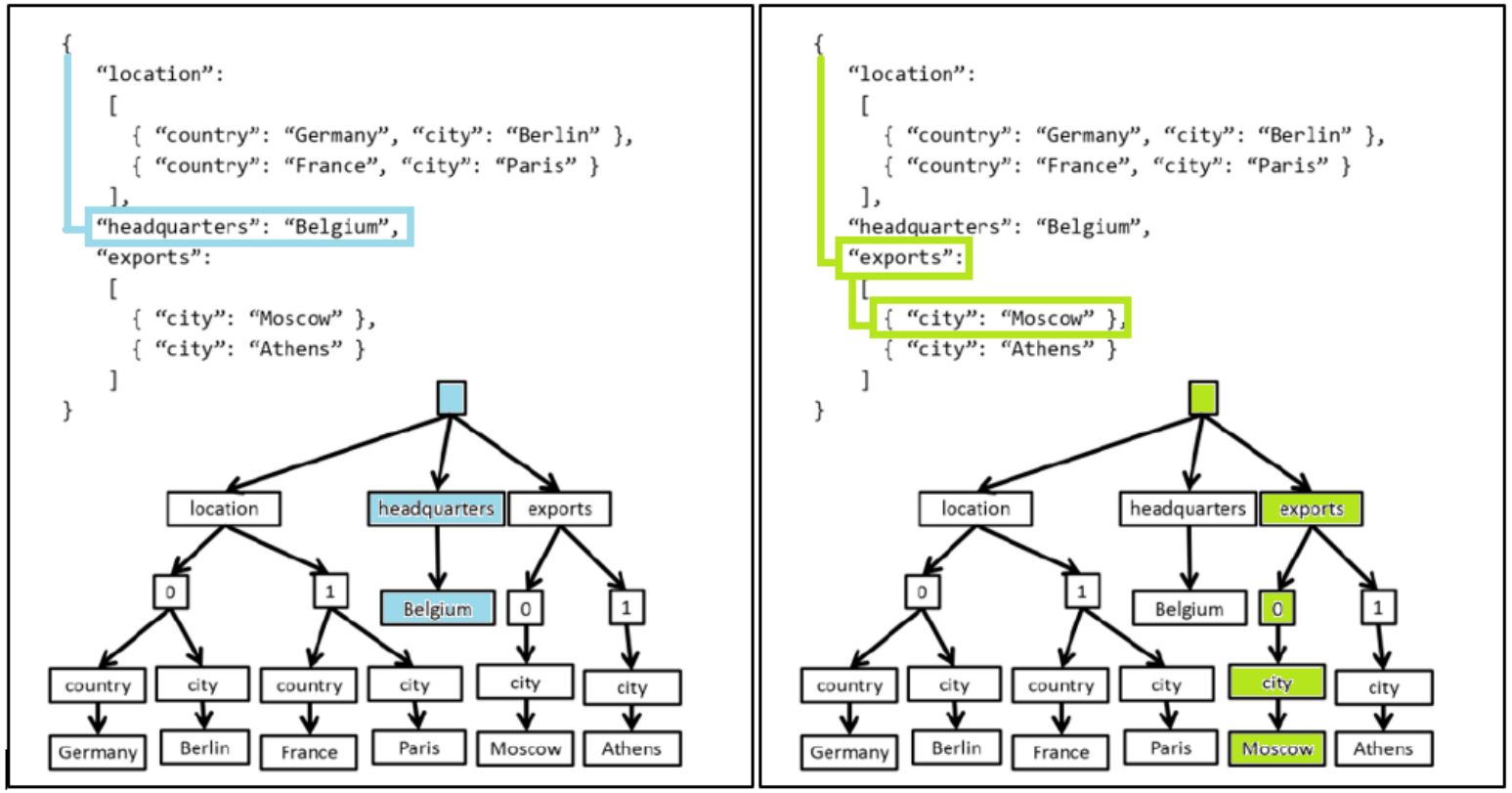
DocumentDB Indexing Policies(1)

- The DocumentDB indexing subsystem is designed to support:
  - Efficient, rich hierarchical and relational queries without any schema or index definitions.
  - Consistent query results while handling a sustained volume of writes. For high write throughput workloads with consistent queries, the index is updated incrementally, efficiently, and online while handling a sustained volume of writes.
  - Storage efficiency. For cost effectiveness, the on-disk storage overhead of the index is bounded and predictable.
  - Storage efficiency. For cost effectiveness, the on-disk storage overhead of the index is bounded and predictable.

DocumentDB Indexing Poilicies(2)

- How DocumentDB Indexing works
  - The indexing in DocumentDB takes advantage of the fact that JSON grammar allows documents to be represented as trees.
  - For a JSON document to be represented as a tree, a dummy root node needs to be created which parents the rest of the actual nodes in the document underneath
  - Each label including the array indices in a JSON document becomes a node of the tree.

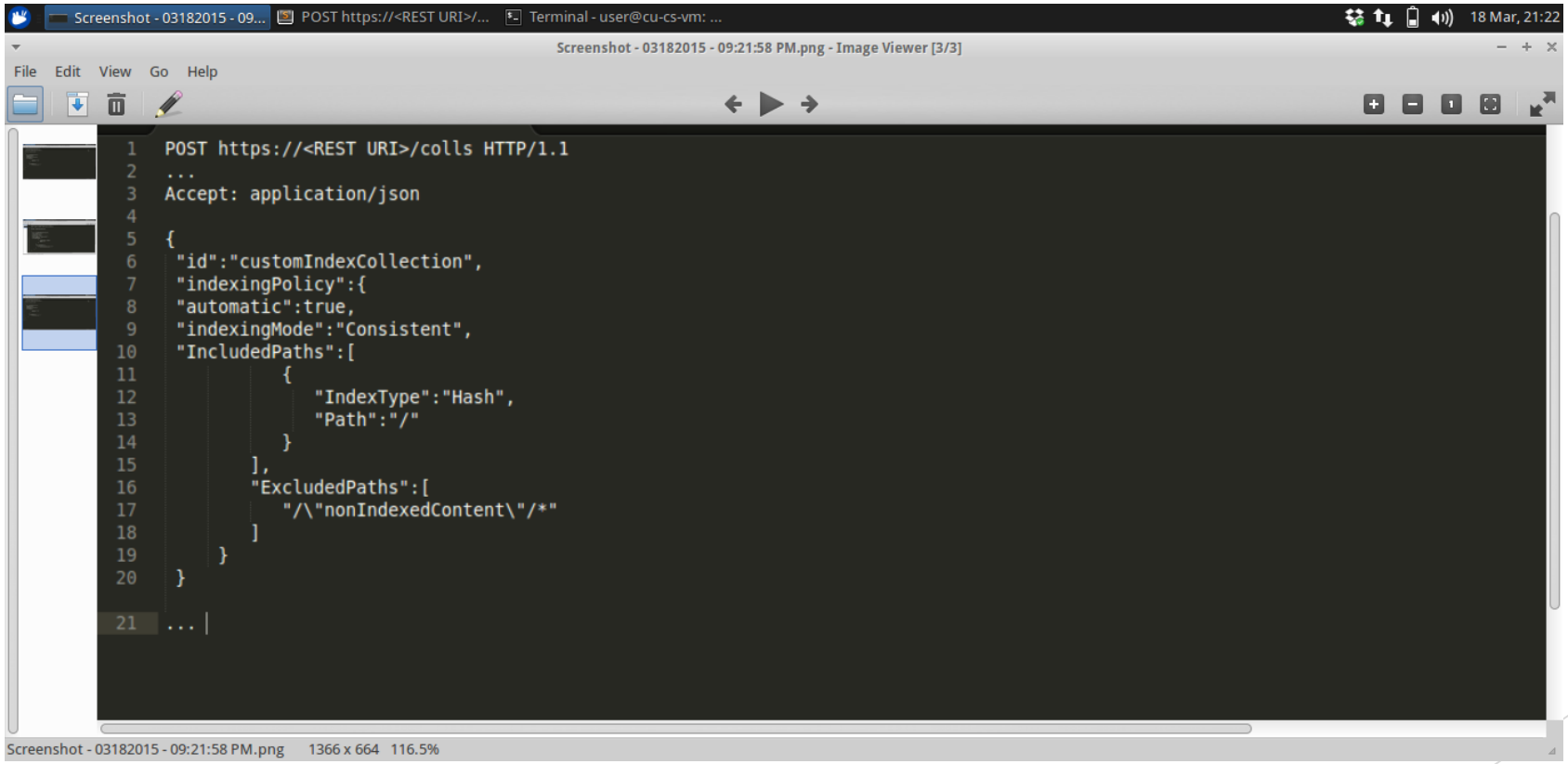
DocumentDB Indexing Policies(3)



DocumentDB Indexing Policies(4)

- In the previous picture: the JSON property {"headquarters": "Belgium"} property in the above example corresponds to the path /"headquarters"/"Belgium“
- The JSON array {"exports": [{"city": “Moscow"}, {"city": Athens"}]} correspond to the paths /"exports"/0/"city"/"Moscow" and /"exports"/1/"city"/"Athens“
  - **Note** The path representation blurs the boundary between the structure/schema and the instance values in documents, allowing DocumentDB to be truly schema-free.

Configuring the Indexing Policy of a Collection



Configuring the Indexing Policy of a Collection(2)

- The Previous sample shows how to set a custom indexing policy during the creation of a collection, by using the DocumentDB REST API. The sample shows the indexing policy expressed in terms of paths, index types, and precisions.
  - The indexing policy is set to: Automatic
  - The Indexing mode is set to: Consisten
  - The indexing type is set to: Hash

- **Note:** The indexing policy of a collection must be specified at the time of creation. Modifying the indexing policy after collection creation is not allowed, but will be supported in a future release of DocumentDB.
- **Note:** By default, DocumentDB indexes all paths within documents consistently with a hash index. The internal Timestamp (_ts) path is stored with a range index.

Automatic Indexing

- You can choose if you want the collection to automatically index all documents or not
- By default, all documents are automatically indexed, but you can choose to turn it off
- When indexing is turned off, documents can be accessed only through their self-links or by queries using ID
  - With automatic indexing turned off, you can still selectively add only specific documents to the index
  - Conversely, you can leave automatic indexing on and selectively choose to exclude only specific document
- You can configure the default policy by specifying the value for the automatic property to be true or false.

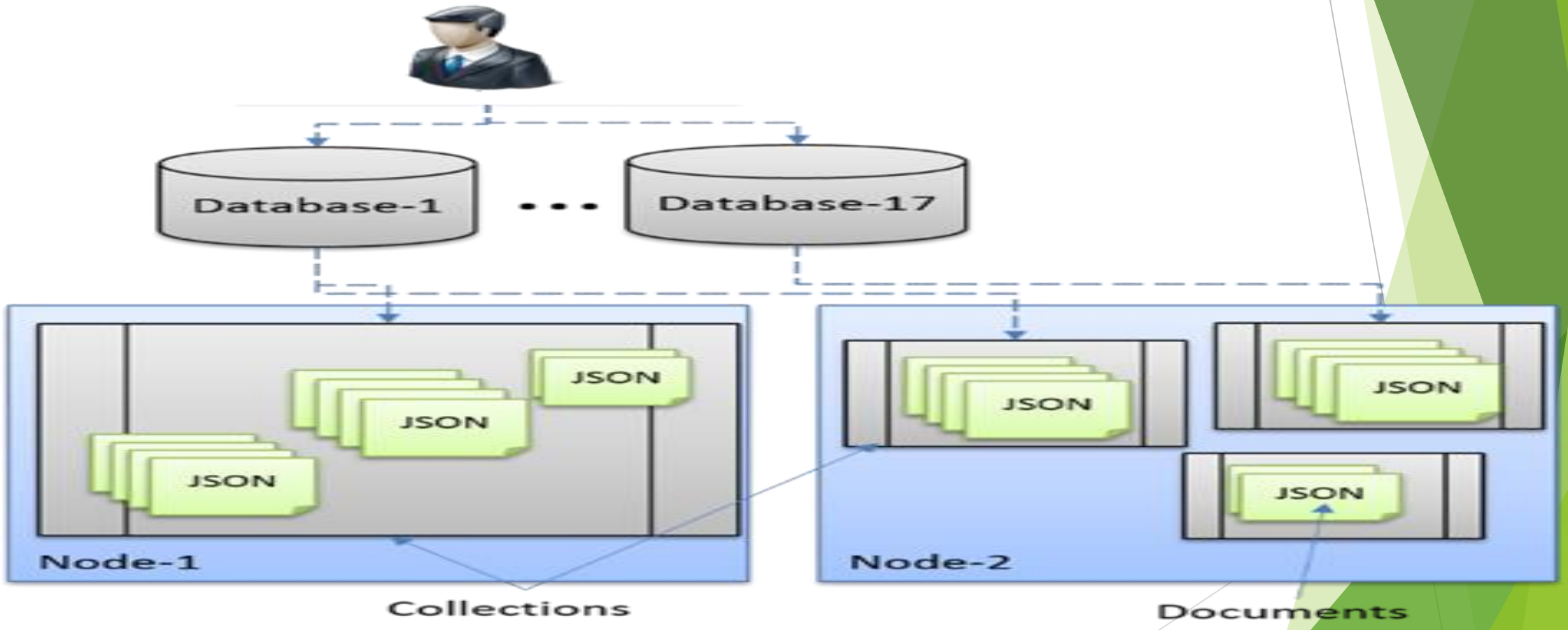
Indexing Modes

- You can choose between synchronous (**Consistent**) and asynchronous (**Lazy**) index updates
- By default, the index is updated synchronously on each insertion, replacement, or deletion action
  - This enables the queries to honor the same consistency level as that of the document reads without any delay for the index to catch up.
- You can also configure certain collections to update their index lazily.
  - Lazy indexing is great for scenarios where data is written in bursts, and you want to amortize the work required to index content over a longer period of time

Index Types and Precision

- There are two supported kinds of index types: Hash and Range
- Hash:
  - Choosing an index type of **Hash** enables efficient equality queries. For most use cases, hash indexes do not need a higher precision than the default value of 3 bytes.
- Range:
  - Choosing an index type of **Range** enables range queries (using >, <, >=, <=, !=).
  - For paths that have large ranges of values, it is recommended to use a higher precision like 6 bytes

Sharding with DocumentDB



Sharding with DocumentDB

- You can achieve a near infinite scale (in terms of storage and throughput) through DocumentDB by horizontally partitioning data, or sharding
- 3 common sharding patterns:
  - Range Partitioning
  - Lookup Partitioning
  - Hash Partitioning

Range Partitioning

- Partitions are assigned based on whether the partitioning key is inside a certain range.
- An example could be to partition data by timestamp or geography (e.g. zip code is between 30000 and 39999)

Lookup Partitioning

- Partitions are assigned based on a lookup directory of discrete values that map to a partition
- This is generally implemented by creating a lookup map that keeps track of which data is stored on which partition
  - An example could be to partition data by user

Hash Partitioning

- Partitions are based on the value of a hash function – allowing you to evenly distribute across n number of partitions
- An example could be to partition data by the hash code % 3 of the tenant to evenly distribute tenants across 3 partitions