Schema-Agnostic Indexing with Azure DocumentDB

DocumentDB was recently released for general availability to Azure developers. In this paper, we describe DocumentDB's indexing subsystem. The indexing subsystem needs to support automatic indexing of documents without requiring a schema or secondary indices, DocumentDB's query language, real-time, consistent queries in the face of sustained high document ingestion rates, and multitenancy under extremely frugal resource budgets while still providing predictable performance guarantees and remaining cost effective.

Overview of the Capabilities

DocumentDB is based on the JSON data model and JavaScript language directly within its database engine.

DocumentDB capabilities

The query language supports rich relational and hierarchical queries. It is rooted in JavaScript's type system, expression evaluation and function invocation model. Currently the query language is exposed to developers as a SQL dialect and language integrated JavaScript query , but other frontends are possible. The database engine is optimized to serve consistent queries in the face of sustained high volume document writes. By default, the database engine automatically indexes all documents without requiring schema or secondary indexes from developers.

Resource Model

A tenant of DocumentDB starts by provisioning a database account. A database account manages one or more DocumentDB databases. DocumentDB collection is a schema-agnostic container of arbitrary user generated documents. DocumentDB collection also manages stored procedures, triggers, user defined functions and attachments. Entities under the tenant's database account – databases, users, collections, documents etc. are referred to as resources.

SCHEMA AGNOSTIC INDEXING

- Documents as Trees: The technique which helps blurring the boundary between the schema of JSON documents and their instance values, is representing documents as trees
- Index as a Document: With automatic indexing, every path in a document tree is indexed unless the developer has explicitly configured the indexing policy to exclude certain path patterns.
- DocumentDB Queries: Despite being schema-agnostic, we wanted the query language to provide relational projections and filters, spatial queries, hierarchical navigation across documents, and invocation of UDFs written entirely in JavaScript.

Logical Index Organization

The index is a union of all the documents and is also represented as a tree. Each node of the index tree contains a list of document ids corresponding to the documents containing the given label value. The tree representation of documents and the index enables a schema-agnostic database engine.

Directed Paths as Terms

- Encoding Path Information : The number of segments in each term is an important choice in terms of the trade-off between query functionality, performance and indexing cost.
- Partial Forward Path Encoding Scheme : The partial forward path encoding involves parsing of the document from the root and selecting three suffix nodes successively to yield a distinct path consisting of exactly three segments.
- Partial Reverse Path Encoding Scheme : The partial reverse path encoding scheme is similar to the partial forward scheme, in that it selects three suffix nodes successively to yield a distinct path consisting of exactly three segments.

Bitmaps as Postings Lists

A postings list captures the document ids of all the documents which contain the given term. The size of the postings list is a function of the document frequency - the number of documents in the collection that contains a given term as well as the pattern of occurrence of document ids in the postings list.

- Partitioning a Postings List : Each insertion of a new document to a DocumentDB collection is assigned a monotonically increasing document id. To avoid static reservation of id space to store the postings list for a given range of document ids.
- Dynamic Encoding of Posting Entries: Depending on the distribution, postings words within a postings entry are encoded dynamically using a set of encoding schemes including various bitmap encoding schemes inspired primarily by WAH.

Customizing the Index

The default indexing policy automatically indexes all properties of all documents and provides consistent queries (meaning the index is updated synchronously with each document write). Developers can customize the trade-offs between storage, write/query performance, and query consistency, by overriding the default indexing policy on a DocumentDB collection and configuring the following aspects.

- Including/Excluding documents and paths to/from index.
- Configuring Various Index Types.
- Configuring Index Update Modes.

PHYSICAL INDEX ORGANIZATION

Consistent indexing in DocumentDB provides fresh query results in the face of sustained document ingestion.

Efficient and Consistent Index Updates

In a classical B+-tree, each such index update would be done as a read-modify-update. The Bw-Tree in DocumentDB was extended to support a new blind incremental update operation. This allows any record to be partially updated without accessing the existing value of the key and without requiring any coordination across multiple callers.

Lazy Index Updates with Invalidation Bitmap

Unlike consistent indexing, index maintenance of a DocumentDB collection configured with the lazy indexing mode is performed in the background, asynchronously with the incoming writes – usually when the replica is quiescent . The bitmap is consulted and updated to filter out the results from within the merge callback while serving a query.

Index Replication and Recovery

The primary considers the write operation successful if it is durably committed to local disk by a subset called the write quorum of replicas.

- Index Replication: A replica applying the terms to its database instance effectively provides the after image that will result in the creation of a series of delta updates.
- Index Recovery:
  The DocumentDB database engine starts a Bw-Tree checkpointing procedure to make all index updates stable up to a highest LSN corresponding to the document update.

Index Resource Governance

As a document database system, DocumentDB offers richer access functionality than a key-value store. Therefore, it needs to provide a normalized model for accounting, allocation and consumption of system resources for various kinds of access, request/response sizes, query operators etc. This is done in terms of an abstract rate based currency called a Request Unit, which encapsulates a chunk of CPU, memory and IOPS. It has roles with:

- CPU resources: All subsystems within DocumentDB are designed to be fully asynchronous and written to never block a thread which in-turn allows the number of threads in the thread pool to remain low
- Memory resources: An instance of DocumentDB’s database engine and its components including the Bw-Tree, operates within a given memory budget that can be adjusted dynamically. The memory limit for the Bw-Tree is maintained by swapping out memory cache resident Bw-Tree pages whenever memory pressure is detected.
- Storage IOPS resources: Because the Bw-Tree organizes storage in a log-structured manner, write I/Os are large and much fewer compared to read I/Os; hence, they are unlikely to create IOPS bottlenecks.
- On-disk storage: The size of a single consolidated logical index entry I(t), it's given by a formula.