Schema-Agnostic Indexing with Azure Document DB
Introduction

- Azure DocumentDB is Microsoft’s multi-tenant distributed database service for managing JSON documents at Internet scale
- **Multi-tenancy** is an architecture in which a single instance of a software application serves multiple customers
- A **distributed database** is a database in which storage devices are not all attached to a common processor. It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers.
- The **JSON** format is often **used** for serializing and transmitting structured data over a network connection. It is **used** primarily to transmit data between a server and web application, serving as an alternative to XML.
DocumentDB’s indexing subsystem needs to support

- (1) automatic indexing of documents without requiring a schema or secondary indices
- (2) DocumentDB’s query language
- (3) real-time, consistent queries in the face of sustained high document ingestion rates
- (4) multitenancy under extremely frugal resource budgets
- (5) still providing predictable performance guarantees and remaining cost effective
DocumentDB service manifests itself as an overlay network of machines, referred to as a federation which spans one or more clusters.

Each machine hosts replicas corresponding to various resource partitions within a fixed set of processes.

DocumentDB database engine consists of components including replicated state machine (RSM) for coordination, the JavaScript language runtime, the query processor, and the storage and indexing subsystems responsible for transactional storage and indexing of documents.
Design Goals for Indexing

- Automatic Indexing - By default, the indexing subsystem automatically indexes all documents without requiring developers to specify schema or secondary indices.

- Configurable storage/performance tradeoff - Although documents are automatically indexed by default, developers should be able to make fine grained tradeoffs between the storage overhead of index, query consistency and write/query performance using a custom indexing policy.

- Efficient, rich hierarchical and relational queries

- Consistent queries in face of sustained volume of document

- Multi-tenancy - index updates must be performed within the strict budget of system resources (CPU, memory, storage and IOPS) allocated per replica. For predictable placement and load balancing of replicas on a given machine, the worst case on-disk storage overhead of the index should be bounded and predictable.
No Schema, No Problem!

- The schema of a document describes the structure and the type system of the document independent of the document instance.

- With a goal to eliminate the impedance mismatch between the database and the application programming models, DocumentDB exploits the simplicity of JSON and its lack of a schema specification.

- JSON grammar, remaining agnostic to the concept of a document schema and blurring the boundary between the structure and instance values of documents.

- This, in turn, enables it to automatically index documents without requiring schema or secondary indices.
DocumentDB Queries

- Developers can query DocumentDB collections using queries written in SQL and JavaScript.
- Both SQL and JavaScript queries get translated to an internal intermediate query language called DocumentDB Query IL.
- The Query IL:
  - (a) is designed to exploit the JSON and JavaScript language integration inside DocumentDB’s database engine,
  - (b) is rooted in the JavaScript type system,
  - (c) follows the JavaScript language semantics for expression evaluation and function invocation and
  - (d) is designed to be a target of translation from multiple query language frontends (currently, SQL and JavaScript). The translated query eventually gets compiled into an imperative program and optimized using a rule-based optimizer.
LOGICAL INDEX ORGANIZATION

- The tree representation of documents and the index enables a schema-agnostic database engine.
- The logical representation of index can be viewed as an ordered set of key-value tuples, each is referred to as an index entry.
- The key consists of a term representing the encoded path information of the node in the index tree, and a PES that helps partition the postings horizontally.
- The value consists of postings list collectively representing the encoded document (or document fragment) ids.
• TERM
• POSTING LIST
Directed Paths as Terms

- A term represents a unique path (including both the position and label values) in the index tree.
- We need to consider the direction of the edges connecting the nodes of the document tree.
- For instance, forward path starting from each node in the tree to a leaf, or reverse path from leaf to the root etc.
- The direction of the path has associated tradeoffs including, (1) storage cost - measured by the number of paths generated for the index structure, (2) indexing maintenance cost - resources consumed for index maintenance corresponding to a batch of document writes, (3) cost of lookup queries
  1. Partial Forward Path Encoding Scheme - used to do range and spatial indexing
  2. Partial Reverse Path Encoding Scheme - used with wildcard queries like finding any node that contains the value "Athens" since the leaf node is the first segment
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.

We require a representation of a postings list that is dynamic (i.e. does not use a fixed sized/static scheme or pre-reserve space), compact (thereby minimizing storage overhead) and yet capable of computing fast set operations, e.g., to test for document presence during query processing.

1. Partitioning a Postings List - partition the postings list into postings entries
2. Dynamic Encoding of Posting Entries
Customizing the Index

- 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.

  1. Including/Excluding documents and paths to/from index
  2. Configuring Various Index Types
  3. Configuring Index Update Modes
     - Consistent - the queries on a given DocumentDB collection follow the same consistency level as specified for the point-read
     - Lazy - queries are eventually consistent
     - None - no index associated with it
Consistent indexing in DocumentDB provides fresh query results in the face of sustained document ingestion. This poses a challenge in a multi-tenant setting with frugal budgets for memory, CPU and IOPS. Index maintenance must be performed against the following constraints:

1. Index update performance must be a function of the arrival rate of the indexable paths
2. Index update cannot assume any path locality among the incoming documents
3. Index update for documents in a collection must be done within the CPU, memory and IOPS budget allocated per DocumentDB collection
4. Each index update should have the least possible write amplification
5. Each index update should incur minimal read amplification (ideally <= 1)
By extending the Bw-Tree we could meet the requirements

- The BwTree uses latch-free in-memory updates and log structured storage for persistence (as opposed to a B+ tree, developed by Microsoft)
- It exploits two trends in modern hardware:
  1. Multicore processors with multi-level memory/cache hierarchy
  2. Flash memory based SSDs with fast random reads (order of ~10-100 micro-sec)
- The latch-free property ensures that threads do not block and readers do not conflict with writers, thus supporting a high degree of concurrency
- The log-structured storage organization of the Bwtree is designed to work around inefficient random write performance on flash and is suitable for hard disks as well
Index Updates

- Document Analysis
- The first step in the index update is document analysis performed by the document analyzer in the indexing subsystem.
- The document analysis function $A$ takes the document content $D$ corresponding to a logical timestamp when it was last updated, and the indexing policy $I$ and yields a set of paths.
- $P. A (D, I) => P$
High Concurrency

- The Bw-Tree operates in a latch-free manner, allowing a high degree of concurrency in a natural manner. A modification to a page is done by appending a delta record on top of the existing portion of the page.

- In a classical B+-tree, each such index update would be done as a read-modify-update. This involves a read of the respective B-tree page, followed by modification in memory.
Index Updates

1. Document Analysis - The first step in the index update is document analysis performed by the document analyzer in the indexing subsystem

2. Efficient and Consistent Index Updates – achieved with blind incremental update operation which allows any record to be partially updated without accessing the existing value of the key and without requiring any coordination across multiple callers.
Index Replication and Recovery

- Index Replication
  1. The primary replica applies it to its database engine instance as well as sending the stream containing the terms to the secondary's
  2. Each secondary applies the terms to its local database instance

- Index Recovery
  1. In the first phase, the Bw-Tree is recovered. This restores a valid and consistent tree that is described by a root logical page ID and mapping table containing offsets to Bw-Tree pages on flash. This also recovers the highest stable LSN (call this LSN-S) up to which all updates have been made stable in the local database engine instance
  2. In the second phase, documents with updates higher than LSN-S are re-indexed by the indexing subsystem of the database engine and inserted into the Bw-Tree. This brings the index to a state that is consistent and up-to-date with documents
Index Resource Governance

- Need 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 (RU/second)

- Areas of governance
  1. CPU resources
  2. Memory resources
  3. Storage IOPS resources
  4. On-disk storage
INSIGHTS FROM THE PRODUCTION WORKLOADS

- Document Frequency Distribution - universally follow Zipf’s Law
- Schema Variance - use this insight (a) for designing the logical index layout including the partial forward and reverse paths and encoding of path segments and (b) for deduplication in the document ingestion path
- Query Performance
- On-Disk Index overhead - validates our design choice for making automatic indexing as the default option
- Blind Incremental Updates - it is extremely IO efficient even in the face of index growth
- Related Commercial Systems – no match for a fully resource governed and schema agnostic database engine to provide automatic indexing (without requiring schema or secondary indices) under sustained and rapid rates of document updates while still serving consistent queries, as described in the paper
The End