Schema-Agnostic Indexing with Azure DocumentDB

Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang

Microsoft Corporation

Sudipta Sengupta, Justin Levandoski, David Lomet

Microsoft Research

ABSTRACT

Azure DocumentDB is Microsoft's multi-tenant distributed database service for managing JSON documents at Internet scale. DocumentDB is now generally available to Azure developers. In this paper, we describe the DocumentDB indexing subsystem. DocumentDB indexing enables automatic indexing of documents without requiring a schema or secondary indices. Uniquely, DocumentDB provides real-time consistent queries in the face of very high rates of document updates. As a multi-tenant service, DocumentDB is designed to operate within extremely frugal resource budgets while providing predictable performance and robust resource isolation to its tenants. This paper describes the DocumentDB capabilities, including document representation, query language, document indexing approach, core index support, and early production experiences.

1. INTRODUCTION

Azure DocumentDB [1] is Microsoft's multi-tenant distributed database service for managing JSON [2] documents at Internet scale. Several large Microsoft applications, including Office, Skype, Active Directory, Xbox, and MSN, have been using DocumentDB, some since early 2012. 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 (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, and (4) multi-tenancy under extremely frugal resource budgets while (5) still providing predictable performance guarantees and remaining cost effective.

The paper is organized as follows: The rest of this section provides a short overview of DocumentDB's capabilities and architecture as well as, the design goals for indexing. Section 2 discusses schema-agnostic indexing. Section 3 describes the logical nature of DocumentDB's JSON derived index terms. Section 4 deals with the indexing method and discusses index maintenance, replication, recovery and considerations for effective resource governance. In Section 5, we substantiate design choices we have made with key metrics and insights harvested from our production clusters. Section 6 describes the related commercial systems, and Section 7 concludes the paper.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

1.1 Overview of the Capabilities

DocumentDB is based on the JSON data model [2] and JavaScript language [3] directly within its database engine. We believe this is crucial for eliminating the "impedance mismatch" between the application programming languages/type-systems and the database schema [4]. Specifically, this approach enables the following 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 (see [5]), 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.
- Transactional execution of application logic provided via stored procedures and triggers, authored entirely in JavaScript and executed directly inside DocumentDB's database engine. We exploit the native support for JSON values common to both the JavaScript language runtime and the database engine in a number of ways - e.g. by allowing the stored procedure to execute under an implicit database transaction, we allow the JavaScript *throw* keyword to model a transaction abort. The details of transactions are outside the scope of this paper and will be discussed in future papers.
- As a geo-distributed database system, DocumentDB offers well-defined and tunable consistency levels for developers to choose from (strong, bounded-staleness, session and eventual [6]) and corresponding performance guarantees [1, 7].
- As a fully-managed, multi-tenant cloud database service, all machine and resource management is abstracted from users. We offer tenants the ability to elastically scale both the throughput and SSD-backed document storage, and take full responsibility of resource management, cost effectively.

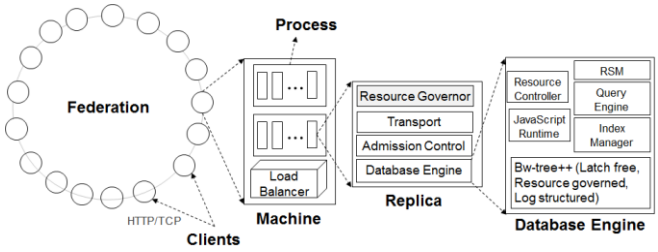

Figure 1. DocumentDB system topology and components.

1.2 Resource Model

A tenant of DocumentDB starts by provisioning a database account (using an Azure subscription). A database account manages one or more DocumentDB databases. A DocumentDB database in-turn manages a set of entities: users, permissions and collections. A DocumentDB collection is a schema-agnostic container of arbitrary user generated documents. In addition to documents, a DocumentDB collection also manages stored procedures, triggers, user defined functions (UDFs) and attachments. Entities under the tenant's database account – databases, users, collections, documents etc. are referred to as *resources*. Each resource is uniquely identified by a stable and logical URI and is represented as a JSON document. Developers can interact with resources via HTTP (and over a stateless TCP protocol) using the standard HTTP verbs for CRUD (*c*reate, *r*ead *u*pdate, *d*elete), queries and stored procedures. Tenants can elastically scale a resource of a given type by simply creating new resources which get placed across *resource partitions*. Each resource partition provides a single system image for the resource(s) it manages, allowing clients to interact with the resources within the partition using their stable, logical URIs. A resource partition is made highly available by a *replica set*.

1.3 System Topology

The DocumentDB service is deployed worldwide across multiple Azure regions [8]. We deploy and manage DocumentDB service on clusters of machines each with dedicated local SSDs. Upon deployment, the DocumentDB service manifests itself as an overlay network of machines, referred to as a *federation* (Figure 1) which spans one or more clusters. Each machine hosts replicas corresponding to various resource partitions within a fixed set of processes. Replicas corresponding to the resource partitions are placed and load balanced across machines in the federation. Each replica hosts an instance of the DocumentDB's *database engine*, which manages the resources (e.g. documents) as well as the associated index. The DocumentDB database engine in-turn 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.

To provide durability and high availability, DocumentDB's database engine persists data on local SSDs and replicates it among the database engine instances within the replica set respectively. Persistence, replication, recovery and resource governance are discussed in the context of indexing in Section 4.

1.4 Design Goals for Indexing

We designed the indexing subsystem of DocumentDB's database engine with the following goals:

- *Automatic indexing*: Documents within a DocumentDB collection could be based on arbitrary schemas. By default, the indexing subsystem *automatically* indexes all documents without requiring developers to specify schema or secondary indices.
- *Configurable storage/performance tradeoffs*: 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*. The index transformation resulting from a change in the indexing policy must be done *online* for availability and *in-place* for storage efficiency.
- *Efficient, rich hierarchical and relational queries*: The index should efficiently support the richness of DocumentDB's query APIs (currently, SQL and JavaScript [5]), including support for hierarchical and relational projections and composition with JavaScript UDFs.
- *Consistent queries in face of sustained volume of document writes*: For high write throughput workloads requiring consistent queries, the index needs to be updated efficiently and synchronously with the document writes. The crucial requirement here is that the queries must be served with the consistency level configured by the developer without violating performance guarantees offered to developers.
- *Multi-tenancy*: Multi-tenancy requires careful resource governance. Thus, 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.

Individually and collectively, each of the above goals pose significant technical challenges and require careful tradeoffs. We asked ourselves two crucial questions while considering the above goals: (1) what should be the logical and physical representations of the index? (2) what is the most efficient technique to build and maintain the index within a frugal budget of system resources in a multi-tenant environment? The rest of this paper discusses how we answered these questions when building the DocumentDB indexing subsystem. But first, we define what is meant by *schema-agnostic indexing*.

2. SCHEMA AGNOSTIC INDEXING

In this section, we explore the key insight to make the DocumentDB's database engine schema-agnostic, which in-turn is crucial for enabling automatic indexing and many other features.

2.1 No Schema, No Problem!

The schema of a document describes the structure and the type system of the document independent of the document instance. For example, the XML Schema specification [9] provides the language for representing schemas for XML documents [10]. Unlike XML, no such widely adopted schema standard exists for JSON. In contrast to XML, JSON's type system is a strict subset of the type systems of many modern programming languages, most notably JavaScript. The simplicity of the JSON grammar is one the reasons for its ubiquitous adoption despite the lack of a schema specification.

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. It makes no assumptions about the documents and

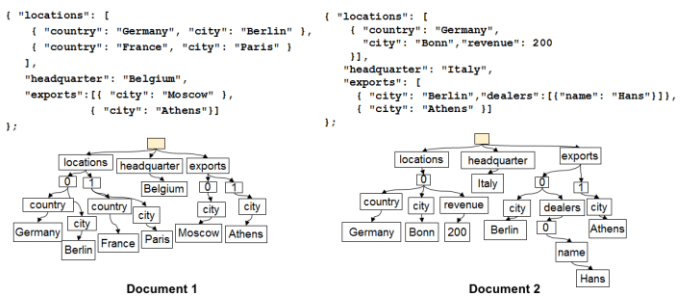

Figure 2. JSON documents as trees.

allows documents within a DocumentDB collection to vary in schema, in addition to the instance specific values. In contrast to other document databases, DocumentDB's database engine operates directly at the level of 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.

2.2 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. Representing JSON documents as trees in-turn normalizes both the structure and the instance values across documents into a unifying concept of a dynamically encoded *path* structure (see Figures 2 and 3; details are covered in Section 3). For representing a JSON document as a tree, each label (including the array indices) in a JSON document becomes a node of the tree. Both the property names and their values in a JSON document are all treated homogenously - as labels in the tree representation. We create a (pseudo) root node which parents the rest of the (actual) nodes corresponding to the labels in the document underneath. Figure 2 illustrates two example JSON documents and their corresponding tree representations. Notice that the two example documents vary in subtle but important ways in their schema. In practice, the documents within a DocumentDB collection can vary subtly or widely in both their structures and instance values.

2.3 Index as a Document

With automatic indexing, (1) every path in a document tree is indexed (unless the developer has explicitly configured the indexing policy to exclude certain path patterns). (2) Each update of a document to a DocumentDB collection leads to update of the structure of the index (i.e., causes addition or removal of nodes). One of the primary requirements of automatic indexing of documents is to ensure that the cost to index and query a document with deeply nested structure, say 10 levels, is the same as that of a flat JSON document consisting of key-value pairs just one level deep. Therefore a normalized path representation is the foundation upon which both automatic indexing and query subsystems are built.

There are two possible mappings of document and the paths: (a) *forward index mapping*, which keeps a map of (document id, path) tuples and (b) *inverted index mapping*, which keeps a map of (path, document id) tuples. Given the fact that the DocumentDB query language operates over paths of the document trees, the *inverted index* is a very efficient representation. An important implication of treating both the schema and instance values uniformly in terms of paths is that logically, just like the individual documents, the

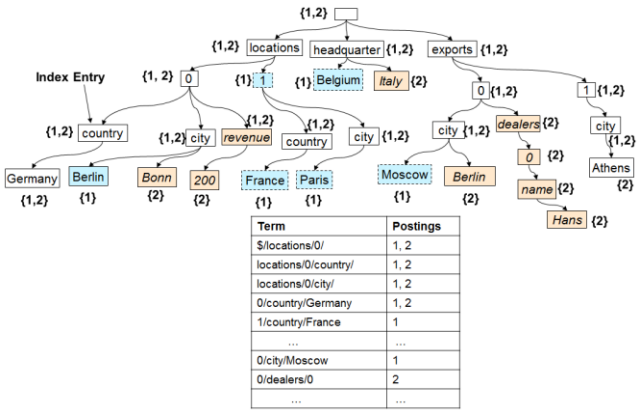

Figure 3. The resulting inverted index of the two documents from Figure 2 shown as a tree and path-to-document id map.

inverted index is also a tree and in fact, the index can be serialized to a valid JSON document! The index tree is a document which is constructed out of the union of all of the trees representing individual documents within the collection (Figure 3). The index tree grows over time as new documents get added or updated to the DocumentDB collection. Each node of the index tree is an *index entry* containing the label and position values (the *term*), and ids of the documents (or fragments of a document) containing the specific node (the *postings*). Notice from Figure 2 and Figure 3, that with the notable exception of arrays, the interior nodes represent the structure/schema of the document and the leaf nodes represent the values/instance. Both the size and number of index entries are a function of the variance contributed by the schema (interior nodes) and values (leaf nodes) among documents within a DocumentDB collection.

Having looked at how the index can be viewed as a union of tree representations of documents, let us look at how DocumentDB queries operate over the tree representation of documents.

2.4 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. Developers can query DocumentDB collections using queries written in SQL and JavaScript [5]. Both SQL and JavaScript queries get translated to an internal intermediate query language called DocumentDB *Query IL*. The Query IL supports projections, filters, aggregates, sort, flatten operators, expressions (arithmetic, logical, and various data transformations), system provided intrinsics and user defined functions (UDFs). 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 using an approach similar to Steno [13] and optimized using a rule based optimizer. The result of the compilation is an assembly of op-codes which ultimately get executed by a stack based virtual machine, which is a part of the DocumentDB's query engine.

One unique aspect of the DocumentDB's queries is that since they operate directly against the tree representation (instead of rows and columns in a relational table). They allow one to refer to properties

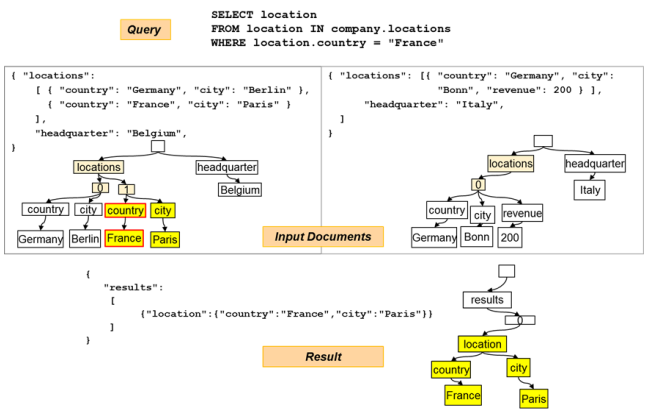

Figure 4. Point query example.

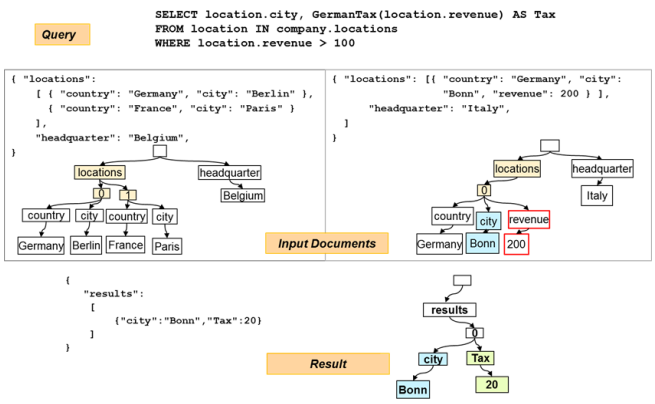

Figure 5. Range query example.

in JSON documents at any arbitrary depth, including wildcard paths such as "/location/*/France". Figure 4 provides an example of a point lookup query against a **`company`** collection consisting of two documents, which we saw earlier in Figure 2 and Figure 3. The query asks for the `location` (`city` and `country`) for all companies with a location "`France`". Note that this query navigates only the paths under the "`locations`" subtree of the input documents (through use of the **`IN company.locations`** clause). The query returns the resulting JSON document containing `country` ("`France`") and `city` ("`Paris`"). Figure 5 provides an example of a range query specifying a predicate for all locations with revenue greater than "100".

Notice here the query navigates only the "`locations`" subtree of the input documents. This query also invokes a UDF named `GermanTax` that calculates the tax on the revenue value for each valid location returned. This function is specified in the select clause and executed within the JavaScript language runtime hosted directly within the DocumentDB's database engine.

To illustrate the multiple query languages all getting translated to the Query IL, consider the following query expressed natively in JavaScript (inspired by underscore.js [11]):

```
function businessLogic() {
  var country = "Belgium";
  _.filter(function(x){
            return x.headquarter===country;});
}
```

The **`filter`** is a logical equivalent of the SQL WHERE clause with the implicit projection returning the entire document. Besides filter, DocumentDB's JavaScript query API [5] provides map, flatten, every, some, pluck, contains, sort, min, max, average, group-by, first, last, etc. Notice that the variable **`country`** is captured and is used using within the **`filter`**. The following JavaScript snippets are logically equivalent of the query in Figure 4.

```
function businessLogic() {
  _.chain(thisCollection().locations)
  .filter(function(location){
            return location.country==="France";})
  .map(function(location){return location;})
  .value();
}
```

The above examples were aimed to convey how the queries in DocumentDB operate on the tree representation of documents and that the multiple query language frontends are layered atop the query IL, which is rooted in JavaScript and JSON. We now discuss the details of the index organization.

3. LOGICAL INDEX ORGANIZATION

In Section 2 we looked at how the tree representation of the JSON documents allows the database engine to treat the structure of the document as well as the instance values homogeneously. The index is a union of all the documents and is also represented as a tree (Figure 3). 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. Finally we looked at how queries also operate against the index tree. For cost effective on-disk persistence, the index tree needs to be converted into a storage efficient representation. The logical representation of index (Figure 3) can be viewed as an ordered set of key-value tuples, each is referred to as an *index entry* (Figure 6). The *key* consists of a *term* representing the encoded path information of the node in the index tree, and a PES (posting entry selector, described in Section 3.2) that helps partition the *postings* horizontally. The *value* consists of *postings list* collectively representing the encoded document (or document fragment) ids.

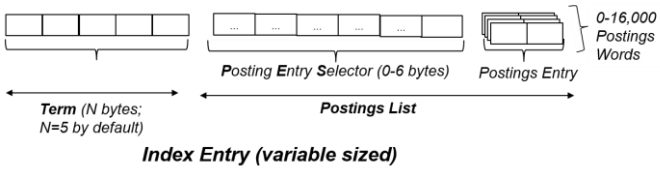

Figure 6. Index Entry.

3.1 Directed Paths as Terms

A term represents a unique path (including both the position and label values) in the index tree. So far we have assumed that the path representation in a document or index tree is undirected. For specifying the path information 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 - e.g.: **`SELECT * FROM root r WHERE r.location[0].country = "France"`**, (4) cost of wildcard lookup queries - e.g.: **`SELECT c FROM c JOIN w IN c.location WHERE w = "France"`**, and, (5) cost of range queries - e.g.: **`SELECT * FROM root r WHERE r.Country < "Germany"`**.

Full Forward Path
```
$/location/0/country/
$/location/0/country/Germany
$/location/0/country/France
$/location/0/city/
```

Partial Forward Path
```
$/location/0/
location/0/country/
location/0/city/
0/country/Germany
0/country/France
```

Full Reverse Path
```
France/country/0/location/$
Germany/country/0/location/$
city/0/location/$
Country/0/location/$
```

Partial Reverse Path
```
0/location/$
country/0/location/
city/0/location/
Germany/country/0
France/country/0
```

Figure 7. Various path representations.

Figure 7 shows four path representations that we evaluated and their tradeoffs. We settled on a combination of *partial forward path* representation for paths where we need range support while following ***partial reverse path*** representation for paths needing equality (hash) support.

3.1.1 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. We default to three segments for two primary reasons: (1) most of the JSON documents have root, a key and a value in most of the paths which fit in three segments and (2) The choice of three segments helps distinguish various paths from each other and yet keep each path small enough to be able to reduce storage cost. Empirically this choice has helped us strike a good balance between storage cost and query performance. We do provide a way to dynamically choose the number of segments to reduce the storage at the cost of query features like wild card searches. The choice of encoding scheme for the path information significantly influences the storage size of the terms and consequently the overall index. By default, we use a five byte path encoding scheme for both styles of partial forward and partial reverse paths, wherein we use one byte each for grand-parent & parent segment, while using three bytes for the leaf node, as depicted in Figure 8.

3.1.2 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. We use separate encoding functions for each of these segments to maximize uniqueness across all paths. This scheme is used to do range and spatial indexing. Query features like inequality filter search and ORDER BY need this scheme to efficiently serve the results. The encoding of a segment is done differently for numeric and non-numeric labels. For non-numeric values, each of the three segments are encoded based on all the characters. The least significant byte of the resultant hash is assigned for the first and second segments. For the last segment, lexicographical order is preserved by storing the full string or a smaller prefix based on the precision specified for the path. For the numeric segment appearing as the first or second segments, we apply a special hash function which optimizes for the non-leaf numeric values. The hash function exploits the fact that most non-leaf numeric values (e.g. enumerations, array indices etc.) are frequently concentrated between 0-100 and rarely contain negative or large values. The hashing yields highly precise values for the commonly occurring numbers and progressively lower precision for larger values. A numeric segment occurring in the third position is treated similar to any non-numeric segment appearing in the third position – the most significant *n* bytes (n is the numeric precision specified for the path) of the 8 byte hash are applied, to preserve order (see Figure 8, right).

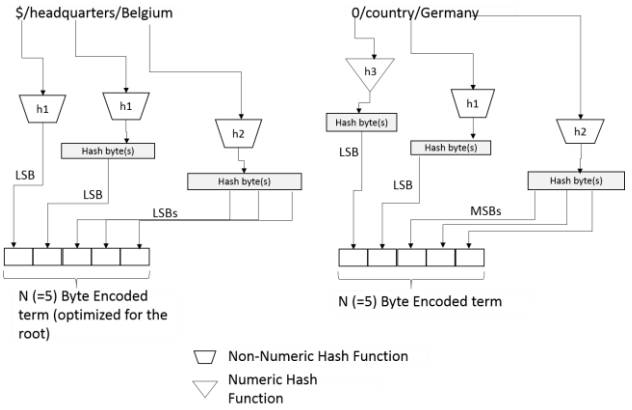

Figure 8. Encoding path segments into terms.

The encoding we use for numbers is based on the IEEE754 encoding format.

3.1.3 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. The term generated is, however, in the reverse order, with the leaf having higher number of bits in the term, placed first. This scheme is suitable for point query performance. This scheme also serves wildcard queries like finding any node that contains the value "`Athens`" since the leaf node is the first segment. The key thing to note is that the intermediate nodes of each path, across all the documents are generally common while the leaves of the paths tend to be unique. In the inverted index, there are fewer terms corresponding to the interior nodes of the index tree with dense postings. The terms mapping the last suffix path (containing leaves) tend to contain relatively fewer postings. The index exploits this behavior to contain the explosion in storage corresponding to, the many suffix paths.

3.2 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. As an example, a document id space of 8 bytes allows for up to 2 ^ 64 documents in a collection. A fixed sized/static scheme will require 8 bytes to represent a single posting and 8 x document frequency (*t*) to represent a single index entry for a term *t*! 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. To this end, we apply two techniques:

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, we partition the postings list into *postings entries.* Additionally, partitioning also helps, to determine the maximum size of pages and split policy of the B+-tree [14] page sizes used as the physical access method. A postings entry is an ordered set of one or more postings words of documents *within a specific document id range*. A single postings entry represents up to 16K consecutive posting ids. For instance, a

posting list can be easily represented as an ordered list of integers (2 byte words) or as bit array with a length of 16K bits. The postings list for a given term consists of a variable length collection of postings entries partitioned by postings entry selector (PES). A PES is a variable length (1-7 bytes), offset into the postings entry. The number of postings entries for a given size of a PES is a function of document frequency for the document id range which falls within the PES range. Document ids within 0-16K will use the first postings entry, document ids from 16K-4M will use the next 256 posting entries, document ids from 4M-1B will use the next 64K postings entries and so on. The number of PES bytes is a function of the number of documents in a collection. For instance, a collection with 2M documents will not use more than 1 byte of PES and will only ever use up to 128 postings entries within a postings list.

Dynamic Encoding of Posting Entries. Within a single partition (pointed by a PES), each document needs only 14 bits which can be captured with a short word. This marks the upper bound on the postings list within a bucket: we should never need more than 32KB to capture a bucket that is densely packed with all possible ids mapped to the bucket. However, to spend two bytes for each id is still expensive, especially given DocumentDB's goal to index all (or most) paths of all documents. Depending on the distribution, postings words within a postings entry are encoded dynamically using a set of encoding schemes including (but not restricted to) various bitmap encoding schemes inspired primarily by WAH (Word-Aligned Hybrid) [15]. The core idea is to preserve the best encoding for dense distributions (like WAH) but to efficiently work for sparse distributions (unlike WAH).

3.3 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. Developers can choose certain documents to be excluded or included in the index at the time of inserting or replacing them to the collection. Developers can also choose to include or exclude certain paths (including wildcard patterns) to be indexed across documents which are included in an index.

Configuring Various Index Types. We have designed the index to support four different indexing types: hash, range, spatial, and text. For each of the included paths, developers can also specify (a) the *type* of index they require over a collection based on their data and expected query workload and (b) the numeric/string "precision" used to specify the number of bytes used for encoding each path into a term. The storage overhead associated with precise hashing of paths may not be desirable if the application is not going to query a particular path.

Configuring Index Update Modes. DocumentDB supports three indexing modes which can be configured via the indexing policy on a DocumentDB collection: 1) ***Consistent***. If a DocumentDB collection's policy is designated as "consistent", the queries on a given DocumentDB collection follow the same consistency level as specified for the point-reads (i.e. strong, bounded-staleness, session or eventual). The index is updated *synchronously* as part of the document update (i.e. insert, replace, update, and delete of a document in a DocumentDB collection). Consistent indexing supports consistent queries at the cost of possible reduction in write throughput. This reduction is a function of the unique paths that need to be indexed and the "consistency level". The "consistent" indexing mode is designed for "write quickly, query immediately" workloads. 2) ***Lazy***. To allow maximum document ingestion throughput, a DocumentDB collection can be configured with lazy consistency; meaning queries are eventually consistent. The index is updated *asynchronously* when a given replica of a DocumentDB collection's partition is quiescent (i.e. resources are available to index the documents in a rate limited manner without affecting the performance guarantees offered for the user requests). For "ingest now, query later" workloads requiring unhindered document ingestion, "lazy" indexing mode may be suitable. 3) ***None***. A collection marked with index mode of "None" has no index associated with it. Configuring the indexing policy with "None" has the side effect of dropping any existing index.

A change in indexing policy on a DocumentDB collection can lead to a complete change in the shape of the logical index including the paths can be indexed, their precision, as well as the consistency model of the index itself. Thus a change in indexing policy, effectively requires a complete transformation of the old index into a new one. The index transformation is done both, *online* and *in-situ* without requiring an additional "shadow" on-disk storage. We will cover the design of index transformation in a future paper.

4. PHYSICAL INDEX ORGANIZATION

Having looked at the "logical" organization of the index and various aspects of the index that developers can customize, we now discuss the "physical" organization of the index, both in-memory and on-disk.

4.1 The "Write" Data Structure

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 index-able paths.
2. Index update cannot assume any path locality among the incoming documents. In fact, our experience of running DocumentDB service for several first-party applications has proven that the probability of paths across documents being localized on a single page on the disk is extremely rare especially for the paths containing leaf nodes.
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 (ideally <= 1).
5. Each index update should incur minimal read amplification (ideally <= 1). By its nature, an update to the inverted index requires merging of postings list for the term. This implies that a naïve solution, would result into a read IO for every term update!

Early on, we learnt that the use of a classical B+ tree was hopelessly inefficient for meeting any of the aforementioned constraints. Efficient maintenance of the document index without any prior knowledge of document schemas is dependent on choosing the most "write-efficient" data structure to manage the index entries. In addition to the above requirements for the index update, the index data structure should be able to serve point lookup, range and wildcard queries efficiently – all of which are key to the DocumentDB query language.

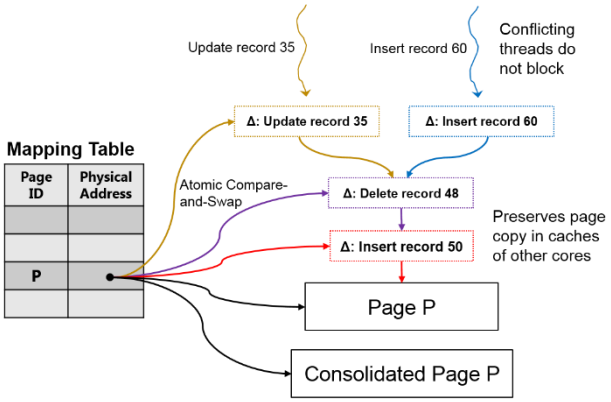

Figure 9. Latch-free and cache-friendly operations.

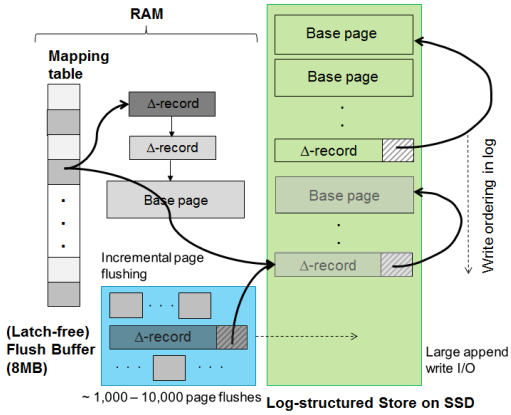

Figure 10. Incremental page flushing to the Bw-Tree log structured storage.

4.2 The Bw-Tree for DocumentDB

After several attempts, we eventually concluded that by extending the Bw-Tree [16, 17] we could meet the requirements we described earlier. This implementation is used as the foundational component of DocumentDB's database engine for several reasons. The Bw-Tree uses latch-free in-memory updates and log structured storage for persistence. It exploits two trends in modern hardware: (i) multi-core processors with multi-level memory/cache hierarchy, and (ii) 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. In memory it is up to 4x faster than latch-free skiplists (see [16]), a competitive state-of-the-art range index solution. The log-structured storage organization of the Bw-tree [17] is designed to work around inefficient random write performance on flash and is suitable for hard disks as well. This technique is similar to that proposed for file systems [18], but with a crucial difference – unlike the log structured file systems, the Bw-Tree completely decouples the logical pages from their physical counterparts. This enables numerous optimizations including reduction in the write amplification. Its technique of updating pages by prepending delta records avoids "in-place updates" and harvests benefits across both memory and flash – (a) it reduces cache invalidation in the memory hierarchy, and (b) it reduces write amplification on flash.

The original Bw-Tree design was extended in numerous ways to facilitate DocumentDB specific requirements. (1) To deliver sustained rapid writes for DocumentDB, the Bw-Tree was extended to support *blind incremental updates* which allows DocumentDB's database engine to utilize full storage write bandwidth for index updates (i.e., writes are not slowed down by reads) and is described in Section 4.3.2. (2) Efficient index recovery in DocumentDB required a CPU and IOPS efficient restart of the "cold" tree, as well as first class support for "streaming" backup/restore of the Bw-Tree. (3) DocumentDB's database engine also needed first class support for flexible resource governance in a multi-tenant setting that plugs into the overall DocumentDB architecture. To that end, many changes were made including (a) the Bw-tree's LSS (log structured store) subsystem needed to support *dynamic* resizing of its secondary storage file based on the accurate calculation of index size, (b) rate limited log flushing to prevent write stalls even under extremely low resource situations and (c) a new CPU efficient cooperative page consolidation algorithm using leases to avoid any redundant consolidation work across threads.

4.2.1 High Concurrency

In a classical B+-Tree, a page is (read or write) latched before access. A write latch does not allow concurrent reads or writes to the page, hence threads that need to do conflicting operations on the page block. Also, acquiring and releasing a latch involves two expensive operations. Moreover, an update to the page is done in-place; this is not cache-friendly in a multi-core environment as it invalidates the copy of the page in the caches of other cores. 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, as shown in Figure 9. This requires the starting location of the page to change after every update. For this reason, all page references use a level of indirection through the *mapping table*. The mapping table provides the translation from logical page ID to physical page location. It also serves as the central data structure for concurrency control. Delta record updates to the page are installed using a compare-and-swap (CAS) operation, which is a single expensive operation (versus two in the latched case). When multiple concurrent threads attempt to append delta records to the same (prior) state of the page, exactly one thread wins and the others have to retry. Thus, threads doing conflicting updates to the page do not block. Moreover, the delta updating methodology preserves the (unmodified) portion of the page in the caches of other cores. When delta chains get large (beyond some threshold), page access efficiency suffers. The page is then consolidated, applying the deltas to produce a new optimized base page. Because such a reconfiguration occurs in batch instead of after every update to the page, it is much more efficient than in classical B+-trees. The consolidated page is also installed in the mapping table using the same CAS mechanism.

4.2.2 Write Optimized Storage Organization

In a classical B+-Tree, storage is organized into fixed size pages (say, ~8KB-32KB). When a page needs to be updated, it is read into memory, updated in-place, and subsequently written back to storage (in-whole). When the insert workload has no locality in the key space, and the size of the index is larger than a replica's fixed and small memory budget, as is the case with DocumentDB, this leads to many random read and write I/Os. This slows down the rate of insertions. Moreover, update-in-place mechanisms increase write amplification on flash based SSDs, slowing down the device due to background garbage collection activity. This also reduces the lifetime of the device. The Bw-tree addresses this issue from

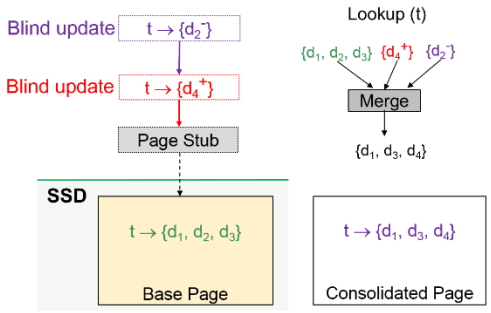

Figure 11. The new Bw-Tree access method for DocumentDB: Blind Incremental Updates.

two aspects: (1) Bw-tree storage is organized in a log-structured manner; and (2) Bw-tree pages are flushed in an incremental manner. On flash, the portions of a page are linked backward in the log, as shown in Figure 10, with the most recently flush delta record appearing later in the log and pointing to the chain of previously flushed delta records. The mapping table also serves as the data structure for recording the starting offset of a page on flash. When a page on flash needs to be updated, and is not already in the main memory, the Bw-Tree reads it into memory and prepends a delta record to it. When the Bw-Tree flushes this page, only the unflushed portion of the page, consisting of possibly multiple delta records, is copied as a single contiguous delta record (C-delta) into a flush buffer. Two flush buffers are used in a ping-pong manner and are maintained in a latch-free manner. A flush buffer is large (of the order of 4 MB). When full, it typically contains 1,000 – 10,000 page flushes and is appended to the end of the log on flash with a single write I/O. This is key to achieving write efficiency.

4.3 Index Updates

The following sections looks at various aspects of updating the index starting with document analysis and describing the two types of (*consistent* and *lazy*) of index updates.

4.3.1 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*.

$$A\ (D,\ I) => P$$

The document analyzer provides basic operators to add two document instances:

$$A_1\ (D_1,\ I_1) + A_2\ (D_2,\ I_2) => P_{1+2}$$

As well as, subtract two document instances

$$A_1\ (D_1,\ I_1) - A_2\ (D_2,\ I_2) => P_{1-2}$$

These operators are extremely powerful and provide the foundation for index maintenance in DocumentDB. Specifically, given that the consistent indexing is done synchronously with the incoming document writes, during the index update, the diff of the index terms corresponding to the older versions of the deleted or replaced documents is still available. The document analyzer supports the "minus" operator to create the diff of the before and after images of the paths in the two documents. This vastly simplifies the processing of delete and replace operations in consistent indexing.

4.3.2 Efficient and Consistent Index Updates

Recall that an index stores term-to-postings list mappings, where a postings list is a set of document (or document fragment) ids. Thus, a new document insertion (or, deletion) requires updating of the postings lists for all terms in that document.

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. Because the terms in a document have no locality pattern and because memory budget is meagre, this would almost always require a read I/O for every term update in the index. To make room for the read pages, some updated pages would need to be flushed from the cache. For document indexing, we observe that such an index update only adds (deletes) a document id to (from) the existing postings list. Hence, these updates could be done, *logically*, without knowing the existing value (postings list) of the key (term). To achieve this, 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.

When a page is swapped out (e.g., to adhere to a memory budget), a slim page stub is left behind in memory that describes the start offset of the page on flash and some other metadata (e.g., high key, side pointer) so as to facilitate key lookups). A blind incremental update of a key prepends a delta record that describes the update to the relevant page. In the case of document ingestion in DocumentDB, such a delta record describes the mapping t→d+, where d is the document id, t is a term, and "+" denotes addition (similarly, "-" would denote deletion). This delta record append operation does not involve any read I/O to retrieve the page from storage. The blind incremental process is depicted in Figure 11.

When a lookup comes to a Bw-Tree page on a given key k, the whole page is read from storage and the multiple fragments of the page describing base value and updates to key k (on base page and delta records) are combined using a *merge* callback function to obtain the final value that is returned.

DocumentDB's database engine uses the same merge function to consolidate Bw-Tree pages by combining fragments of key values across base page and delta records. Note that due to the latch free nature of the Bw-Tree, the postings (value) for a given term (key) can get updated out of order. Therefore, the merge callback needs to provide commutative (and idempotent) merge of the delta values for a given key in the face of out of order updates for a given key. To illustrate this, consider the document with a lone property called "status" which can have two possible values, "on" or "off": `{"status":"on"}`. For the purposes of this example, assume that the document gets updated concurrently by multiple users with the "status" toggled between "on" or "off" in rapid succession with the final value as "off". If the updates to the keys were sequential, the transient and final values corresponding to the key "$/status/on" would have been {id+, id-, id+, id-} and {} respectively. Similarly, the transient and the final values for the "$/status/off" would be {id+, id-, id+} and {id+} respectively. However, latch free updates may lead to transient values for the key "status/on" as {id+, id-, id+, id-} and "$/status/off" as {id+, id+, id-}. To complicate the matters, the merge callback may get dispatched with partial values of the delta updates – e.g. "$/status/on" with {id+, id-, id-} or "$/status/off" with {id+, id+}. The merge callback therefore needs to detect the out of order delivery of the delta values by inspecting the polarity of each delta value and applying cancelation of opposite polarities. However, since the merge callback can get invoked with partial delta values, the callback needs to maintain a "holdout" (which is also encoded as a delta value) of the non-cancellable deltas for a future merge. The polarity based merge of the postings with the holdout is crucial for accurate computation of

the postings list and the query accuracy in the face of allowing latch free updates.

4.3.3 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 (e.g. either when there is an absence of user requests or sufficient surplus resources available to the indexing subsystem). Since we do not maintain *previous* document images for deleted/replaced documents and since the index maintenance in case of lazy indexing is done asynchronously - we cannot assume the availability of the before and after images of the terms which are being indexed. This has two downsides: (1) the indexing subsystem can return false positives, causing an additional I/O penalty to serve a query and (2) Bw-Tree pages accumulate stale entries for the deleted/replaced documents in the postings lists and bloat memory and on-disk layout. To avoid these, we maintain a *counting invalidation bitmap* which is a bitmap representing the document ids corresponding to the deleted and replaced document images and a count representing the number of times the document update has been recorded. The bitmap is consulted and updated to filter out the results from within the merge callback while serving a query. The bitmap is also consulted and updated during the invocation of the merge function during page consolidation.

For the "cold terms" on the pages which lay on disk waiting to be consolidated or for which the queries were never issued, we schedule a "compaction scan" for the leaf pages in the background in a rate limited manner. The compaction cycle is triggered when either the invalidation bitmap or the on-disk Bw-Tree file size have reached a configurable threshold. The compaction scan ensures that all delta updates as of some point in logical time (corresponding to a document update) to Bw-Tree pages are consolidated into new compact base pages. Doing this ensures that the merge callback function (used to merge the deltas into its base pages) has seen the appropriate invalidation bitmap entries. Once the compaction scan completes, memory for the old invalidation bitmap entries can be re-used (or de-allocated).

4.4 Index Replication and Recovery

DocumentDB follows a single master model for writes; clients issue writes against the distinguished primary replica of the replica set, which in-turn propagates the client's request guaranteeing a total order to the secondary replicas in the set. The primary considers the write operation successful if it is durably committed to local disk by a subset called the write quorum (W) of replicas. Similarly, for a read/query operation the client contacts the subset of replicas, called the read quorum (R) to determine the correct version of the resource; the exact size of read quorum depends on the default consistency policy configured by the tenant for the database account (which can be overridden on a per request basis).

4.4.1 Index Replication

During the steady state, the primary replica receiving the writes analyzes the document and generates the terms. The primary replica applies it to its database engine instance as well as sending the stream containing the terms to the secondaries. Each secondary applies the terms to its local database instance. A replica (primary or secondary) applying the terms to its database instance effectively provides the after image that will result in the creation of a series of delta updates. These in turn will eventually be reconciled with the before image of the terms when the merge callback is invoked. The DocumentDB resource governance model divides resource budgets among the primary and secondaries in a non-uniform manner with the primary carrying the bulk of the write responsibilities and secondaries serving the reads/queries; the cost of analyzing the document is paid only on the primary. After a failover, when the new replica joining the quorum needs to be rebuilt from scratch, the primary sends multiple physical streams to the secondary directly from the Bw-Tree LSS. On the other hand, if the newly joining replica needs to catch-up with the existing primary by only a few documents, the secondary simply analyzes and regenerates the terms locally and applies them to its database instance - in this particular case, the cost of local term generation for a small number of documents is cheaper compared to the coordination, IO and transmission overhead needed to fully rebuild a replica.

4.4.2 Index Recovery

The Bw-Tree exposes an API that allows the upper layer in the indexing subsystem to indicate that all index updates below some LSN should be made stable. Since the recovery time can adversely influence the service level agreement (SLA) for availability, the index checkpointing design is optimized to ensure that the crash recovery requires minimum amount of index to be rebuilt (if at all). Periodically, 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. This process involves scanning the mapping table and incrementally flushing pages to flush buffers (if needed). The highest checkpointed LSN corresponding to the document update is persisted in flush buffer headers. Additionally, The Bw-Tree log-structured storage (LSS) layer checkpoints at configurable intervals of log size growth [17]. The aim of this approach is to limit the amount of the log that needs to be scanned during Bw-Tree recovery, and hence time.

End-to-end recovery of the index happens in two phases: 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. 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. During this phase, the index updates are applied in an idempotent fashion and applying the invalidation bitmap technique similar to the one explained in 4.3.3. Finally, to support replica rebuild as well as disaster recovery scenarios, the Bw-Tree in DocumentDB is extended to support online streaming. For both, efficient network usage and storage efficiency, the backup stream produced by the Bw-Tree includes only the active portion of the log (approximately 75% of the on-disk file on the primary site). The streaming support is designed to provide a consistent snapshot in presence of writes and while the physical file size is undergoing changes.

4.5 Index Resource Governance

As a document database system, DocumentDB offers richer access functionality than a key-value store (e.g. get/put). 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* (RU/second), which encapsulates a chunk of CPU, memory and IOPS. Correspondingly, RU/second provides the normalized unit for accounting, provisioning, allocating and consuming throughput guarantees. The system must ensure that it can provide the throughput that was configured for a given DocumentDB collection. We learned early on that the key to providing predictable

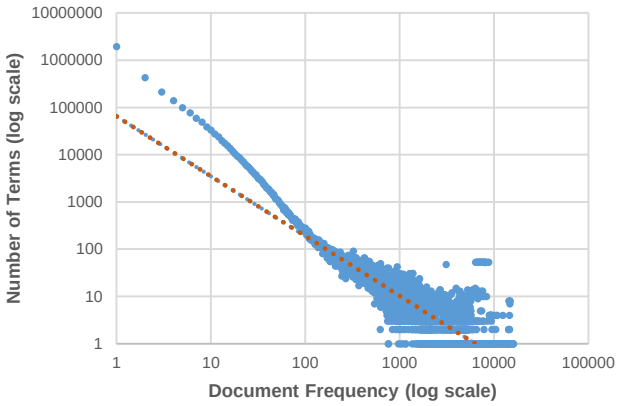

Figure 12. Document Frequency and Unique path lengths across collections.

performance guarantees is to build the entire service with resource governance from the ground up. A DocumentDB replica uniquely belongs to a tenant and is designed to operate within a fixed budget of system resources in terms of RU/second. Each DocumentDB process may host database engine instances corresponding to the replicas belonging to various tenants, and is monitored for CPU, IOPS and memory consumption. Further, the database engine instance corresponding to a replica within the process manages its own thread-pool, memory and IO scheduling. All subsystems hosted within the database engine instance are allocated fixed budgets of memory, CPU and IOPS from the replica's overall resource budget for a given time slice. The budget corresponds to the RUs/sec rate assigned to a replica to meet the performance guarantees promised to the tenant.

4.5.1 Index Resource Governance

All operations within a DocumentDB's database engine are performed within the quota allocated for CPU, memory, storage IOPS and the on-disk storage. Additionally, like all DocumentDB components, the database engine honors the throttling signals from the resource governor. Inside DocumentDB, the Bw-Tree GC operates in a rate limited manner with short and frequent cycles within the pre-allocated budget of IOPS, CPU, storage and memory.

CPU resources. As described previously, a DocumentDB database engine instance manages its thread scheduler. 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 (e.g., equal to the number of cores on the machine). Since the in-memory operations of the Bw-Tree are completely latch-free, the synchronous path is very efficient since a thread traversing the index or updating a page in memory will never run into a lock (or latch). In fact, except for the following three cases of asynchronous IO, all of Bw-Tree operations complete synchronously: (1) Reading a page not in memory, (2) Writing a sealed flush buffer and awaiting its IO completion. (3) Waiting for the LSS garbage collector to free the storage space. In all three cases, the usage of continuation style programming model within the database engine implementation allows for making progress without blocking any threads.

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. Pages are selected for swapout using a variant of the LRU cache eviction algorithm called CLOCK [19]. Page swapout functionality is distributed across threads (using latch-free techniques) so that it scales with concurrent index activity. Every thread that accesses the Bw-Tree first performs some page swapout work (if needed) before executing the actual access method operation.

Storage IOPS resources. A DocumentDB database engine instance needs to operate within a given IOPS budget. Its Bw-Tree maintains a running average of IOPS usage over time. Just before issuing an I/O, it checks whether the I/O would lead to IOPS budget violation. If so, the I/O is delayed and then attempted again after a computed time interval. 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 (but may create storage bandwidth bottlenecks). Moreover, flush buffer writes are necessary for progress in many parts of the system (e.g., page swapout, checkpointing, garbage collection). Hence, write I/Os are not subject to IOPS resource governance. The Bw-Tree propagates internal resource usage levels upward to the resource controller of the database engine so that when budget violation is imminent, request throttling can happen further upstream in the database engine.

On-disk storage. The size of a single *consolidated* logical index entry I(t), is given by the following formula:

$$I(t) = \left(\underbrace{N}_{Term} + \underbrace{\{1-7\} \times 8}_{PES} + \underbrace{C \times d(t)}_{Postings\ Entry} \right)$$

where N as the number of bytes per term (default is 3 bytes), the *Postings Entry Selector* (PES) can vary from 1 to 7 bytes (that is where 8 in the above formula comes from), *d (t)* is the document frequency, defined as the number of documents in the collection containing term *t*, *C* is the *compression factor* and represents the average number of bytes required to encode one posting of the *d (t) and I (t)* is the size of a single index entry. In practice, index entries are unconsolidated and the size of a single unconsolidated logical index entry *I'(t)* includes this transient overhead due to *D* delta updates.

$$I'(t) = \sum_{i=0}^{i=D} I\ (t)$$

Eventually, when the delta updates for the term *t* gets consolidated (when the postings for a given term *t* across the delta updates get merged via the merge callback invoked as part of page consolidation), *I'(t)* will become *I(t)*. The cumulative cost of the entire *logical* index *S* for *K* unique index terms within a collection, is calculated by the following formula

$$S = \sum_{t=0}^{t=K} (I'(t))$$

The biggest contributor of the overall logical index size is the number of unique terms *K* among the documents within a collection. The value of *K* in-turn is a function of the variance among the schemas (interior nodes) and values (leaf nodes) all the documents in a collection. In general, the variance among documents is maximum among the leaf nodes (property values) and progressively diminishes to zero as we approach the root node. The on-disk index size is *directly* proportional to S plus a fixed additional headroom for GC run effectively (currently set to 66.6% of S, making the on-disk file size as *1.5xS*). For efficient storage

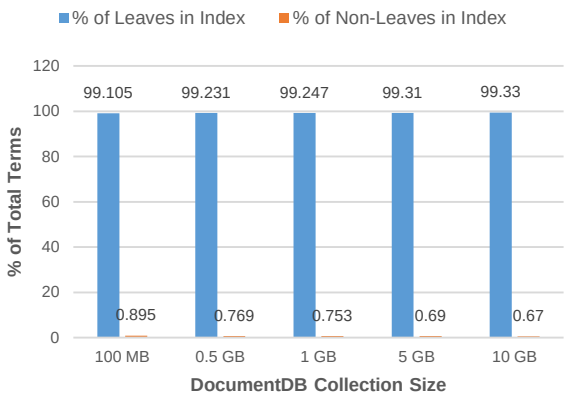

Figure 13. Schema Variance across DocumentDB collections of various sizes.

utilization across multiple replicas on the machine (and for load balancing across the federation), the on-disk Bw-Tree file for a given database instance is required to dynamically grow and shrink proportional to the logical index size *S*.

5. INSIGHTS FROM THE PRODUCTION WORKLOADS

Over the past two years, we have been continuously validating and making improvements to our logical index organization and maintenance based on the telemetry collected from the DocumentDB service clusters running production workloads for Microsoft (MSN, Xbox, Skype and others) and third party applications across six different geographical regions. While we collect and analyze numerous indexing related metrics from DocumentDB service deployed worldwide in Azure datacenters, here we present a selected set of metrics which guided some of the design decisions we presented earlier in the paper.

5.1 Document Frequency Distribution

Across all our clusters, regardless of the workloads and the sizes of documents or DocumentDB collections, we observe that document frequency distribution for the unique terms universally follow Zipf's Law [20] (Figure 12). While this fact has been universally observed for document corpora in various IR systems, it validates many of the decisions we have made for the DocumentDB's logical index organization in the context of JSON documents. Specifically, the design of logical index entry and the byte allocation for PES boundary, choice of various encoding schemes for postings and the ability to dynamically change the encoding based on the changes to distribution, as well as, application of semantic aware deduplication techniques, were all influenced by the observation.

5.2 Schema Variance

Recall that we represent JSON documents as trees, with schema represented by the interior nodes and the instance values represented by the leaves. Figure 13 illustrates this across a random sample of DocumentDB collections grouped by their sizes (100MB to 10GB) and containing documents ranging from 1KB to 1MB. As shown in the graph, regardless of the workload, document or collection size, the number of unique leaf nodes (instance values) completely dwarf the number of interior nodes (schema). We 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.

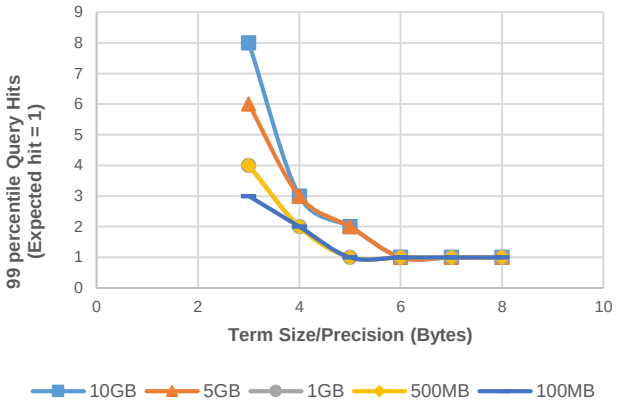

Figure 14. Query Performance vs. Index Size.

5.3 Query Performance

We define query precision in terms of the number of false positives in postings for a given term lookup. Query precision therefore is a good proxy for query performance. The highest value of query precision, we can ever get is 1 – this is when we load only those documents which contain the results of the query. Put differently, if the terms in the logical index were hashed perfectly, the query precision will be 1. As depicted in Figure 14, for the same number of bytes allocated to a term, the query precision decreases as the size of the DocumentDB collection grows. For a 10GB collection, 5-6 bytes yields the best query precision/performance. Figure 14 also illustrates the storage size (contributed by the bytes allocated to hash a term) to query performance tradeoff.

5.4 On-Disk Index overhead

Figure 15 validates our design choice for making automatic indexing as the default option. The graph shows the average and 99nth percentile *index overhead* (*uncompressed* logical index size *S* compared to size of documents) across DocumentDB tenants in various datacenters.

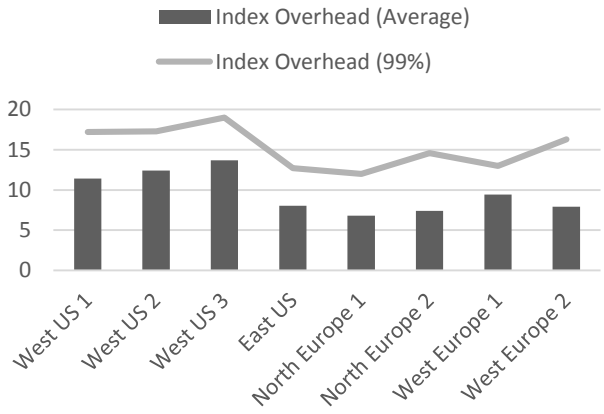

Figure 15. Index Overhead In Practice.

5.5 Blind Incremental Updates

Doing highly performant index updates within an extremely frugal memory and IOPS budget is the key reason behind the blind incremental update access method. This is evident in Figure 16 which shows the IO efficiency (defined as, *the total number of IOs issued for updating index of a given size*) of the blind incremental

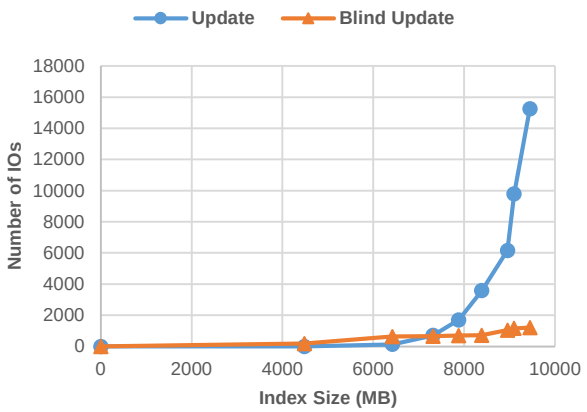

Figure 16. IO efficiency was the main reason for inventing the Blind Incremental Update access method in DocumentDB.

update access method for updating the index during document ingestion of documents varying from 1KB-10KB, each containing 10-20 unique terms and within a strict memory budget of 4MB! Note that in case of blind incremental updates, the read IOs completely dominate the total number of IOs. This is because due to the large and elastic flushes, the write IOs contribute a tiny and constant percentage of the overall IOs issued. In contrast, with the classical update method and without any term locality, each write has an associated read. The key reason for introducing the new blind incremental updates access methods should be evident from Figure 16: compared to the classical update method, it is extremely IO efficient even in the face of index growth.

6. Related Commercial Systems

Currently, there are two types of commercial NoSQL storage systems broadly available in the market:

1. Non-document oriented NoSQL storage systems (e.g. [21, 22]) which are designed and operated as multi-tenant cloud services with SLAs.
2. Document databases (e.g. [23, 24]) designed as single tenant systems which run either on-premises or on dedicated VMs in the cloud. These databases have the luxury of using all of the available resources of a (physical or virtual) machine. These systems were neither designed to be operated as a fully-managed, multi-tenant cloud services in a cost effective manner nor designed to provide SLAs (for consistency, durability, availability and performance). Hence, operating the database engine and offering database capabilities under a frugal amount of system resources while still providing performance isolation across heterogeneous tenants is not even a concern of these systems.

We believe that the design constraints under which DocumentDB is designed to operate and the capabilities it offers (see Section 1.1) are fundamentally different than either of the two types of systems above. Neither of the two types of systems above provide 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.

7. CONCLUSION

This paper described the design and implementation of the indexing subsystem of DocumentDB, a multi-tenant distributed document database service for managing JSON documents at massive scale. We designed our database engine to be schema-agnostic by representing documents as trees. We support automatic indexing of documents, serve consistent queries in the face of sustained write volumes under an extremely frugal resource budget in a multi-tenant environment. Our novel logical index layout and a latch-free, log-structured storage with blind incremental updates are key to meet the stringent requirements of performance and cost effectiveness. DocumentDB is also capable of performing online and in-situ index transformations, as well as handle index replication and recovery in DocumentDB's distributed architecture. The indexing subsystem described here is currently supporting production workloads for several consumer scale applications worldwide.

8. REFERENCES

[1] Azure DocumentDB Documentation. http://azure.microsoft.com/en-us/documentation/services/documentdb/

[2] Javascript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt

[3] ECMAScript Language Specification, http://www.ecma-international.org/publications/standards/Ecma-262.htm

[4] T. Neward. The Vietnam of Computer Science. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

[5] DocumentDB Query Language. http://msdn.microsoft.com/en-us/library/dn782250.aspx

[6] D. B. Terry. Replicated Data Consistency Explained Through Baseball. *Commun. ACM*, 56(12): 82-89, 2013.

[7] D. Abadi. Consistency Tradeoffs in Modern Distributed Database Systems Design: CAP is Only Part of the Story. *IEEE Computer*, 45(2): 37-42, 2012.

[8] Microsoft Azure. http://www.windowsazure.com/en-us/

[9] XML Schema specification. http://www.w3.org/XML/Schema

[10] XML Infoset. http://www.w3.org/TR/xml-infoset/

[11] Underscore.js. http://underscorejs.org

[12] LINQ (Language Integrated Query). http://msdn.microsoft.com/en-us/library/bb397926.aspx

[13] D. G, Murray M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In *PLDI*, pages 121-131, 2011.

[14] D. Comer. The Ubiquitous B-tree. *ACM Comput. Surv.*, 11(2): 121-137, 1979.

[15] K. Wu, K. Stockinger, and A. Shoshani. Breaking the Curse of Cardinality on Bitmap Indexes. In *SSDBM*, pages 348–365, 2008.

[16] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013, pages 302-313, 2013.

[17] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10): 877-888, 2013.

[18] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log Structured File System. *ACM TOCS*, 10(1): 26-52, 1992.

[19] F. J. Corbato. A Paging Experiment with the Multics System. *MIT Project MAC Report* MAC-M-384, May, 1968.

[20] Zipf's Law, http://en.wikipedia.org/wiki/Zipf%27s_law

[21] Amazon DynamoDB. http://aws.amazon.com/dynamodb/

[22] Google Cloud Datastore. https://cloud.google.com/datastore/

[23] MongoDB. http://mongodb.com/

[24] Couchbase. http://couchbase.com/