HybridRow

Jason Hunter

# Abstract

HybridRow is a bespoke database row format, schema-based type system, serialization library, and transport encoding developed to meet the unique needs of Cosmos DB’s multi-model and distributed partitioning requirements.

In this paper we first provide a brief overview of the Cosmos DB architecture that led to the development of HybridRow. We then provide a description of the design of HybridRow. We discuss HybridRow Schema and how it interacts with the distributed system. Lastly, we provide some performance data on HybridRow serialization.

# Cosmos DB Architecture

Azure Cosmos DB is a globally distributed, highly available, elastic, multi-model database service. Cosmos DB is both locally partitioned, and geographically replicated. Partitioning and local replication enable unlimited elasticity and high per-partition availability. It is common for Cosmos DB workloads to include datasets of 10’s of TB, partitioned across 1000’s of nodes. Additionally, geo-replication and multi-master ensure copies of the data simultaneously exist in 10’s of regions around the world, while allowing simultaneous writes even in the face of transient global network interruptions.

On top of this robust distributed storage engine, Cosmos DB embodies a multi-model approach to programmability. Cosmos DB exposes multiple different API surface areas that allow existing applications to leverage the system’s capabilities using the programming model best suited to their development. API interoperability is available for a wide selection of NoSQL and open source programming models including Cassandra, MongoDB, Gremlin (Graph), etcd, Spark, and SQL, among others. Applications can leverage the system using either existing 3rd party client libraries provided by external API ecosystems or via drop-in replacement enlightened client drivers that leverage Cosmos DB technology directly.

## Tiered System Architecture

Cosmos DB operates as either a 2-tier or 3-tier system depending on which client drivers are used. When leveraging existing 3rd party drivers, or when Cosmos DB drivers are operating in gateway mode, the system operates in three logical layers:

**Client Drivers**: Formulate and issue both DDL (metadata operations), DML (writes), and Queries (reads) in the native APIs representation.

**Frontends**: Translate application intent expressed in some native API into the Cosmos DB Intermediary Language (IL) and forward the request to the appropriate Backend(s) for processing.

**Backends**: implement all relational and storage functionality for an individual partition including transaction processing, replication, index maintenance, query execution, backups and recovery.

When Cosmos DB’s enlightened drivers are in use, direct mode enables some aspects of the Frontends to be incorporated into the clients which then communicate in IL directly to the Backends via Cosmos DB’s RPC protocol. This is the same on-wire interaction the Frontends use to communicate with the Backends in the 3-tier topology.

## Universal Store

Though Cosmos DB offers a range of programmability APIs, the Backend implements a single universal storage design. The platform exposes an Intermediary Language (IL) and query semantics with sufficient fidelity to implement the capabilities of all externally accessible APIs. To achieve this the database design must meet the following requirements:

* Capable of storing row data across all models.
* Be competitive in storage size across all models.
* Be competitive in access performance for both reads and writes across all models.
* Offer schema definition that encompasses all models.
* Provide for Schema evolution that aligns with a loosely coupled scalable distributed system.

What makes this list of requirements particularly challenging is that the set of programming models supported covers a wide spectrum in both schematization and related storage efficiency. Cassandra, for example, is a fully schematized environment with a highly specialized row compression design that both leverages the existence of known schema and knowledge of typical Cassandra update patterns to store large datasets with low storage overhead. In contrast to this highly structured environment, Cosmos DB’s SQL API provides a fully unschematized, highly flexible, JSON document store. JSON documents exhibit a high degree of heterogeneity across rows, with no requirements for common schema adherence. Cosmos DB provides for schema-less auto-indexing and robust cross-datatype query. As a final example, MongoDB lands somewhere between these two extremes, offering a richer type system than JSON, partial schematization coupled with unschematized semi-structured data, explicit indexes, and a rich query model.

## Distributed Database

In addition to the demand for a highly flexible schema design, a scale-out distributed system must deal with the fundamental property that the propagation of information is not instantaneous. To achieve infinite elasticity, no system operation can be proportional to the total number of partitions. At steady state, all partitions can operate in parallel independently, distributing the total workload evenly. Implementing synchronous schema evolution operations (DDL) would, however, require synchronous coordination among all partitions thus impacting availability and scalability.

To avoid this, all schema evolution operations in Cosmos DB propagate asynchronously. Once a master partition has committed a schema evolution, the knowledge of the new schema propagates asynchronously both among the Backends and to the Frontends/Clients. This has three main implications:

**No Size-Of-Data Evolutions**: A table may have 100’s of TB’s of existing data distributed across 1000’s of nodes at the time of a schema change. Committing a schema change at a particular Backend must not require changing existing data. Existing data must be interpretable in the context of the new schema definition without modification. The process of interpreting a down-level row is called *row upgrade*.

**Backends Are Ahead**: A Backend may receive requests from Frontends/Clients who do not yet know the latest schema. Such requests should never be rejected as long as the incoming data is compatible with the view of the schema that the Backend knows about. The Backend must interpret the request through *row upgrade* in the same way it would have, had the data been already present in the database when the schema change arrived.

**Backends Are Behind**: A Backend may receive requests from Frontends/Clients who have already learned of a schema change the Backend does not yet know about. Backends must be able to distinguish these requests efficiently and either wait for propagation to occur or reject the request with a retriable client error allowing the Frontend/Client to transparently reissue the request once propagation has been completed.

## Efficient Updates

The bread and butter of NoSQL systems are simple CRUD operations and partition-aligned queries. In GET and PUT operations the interactions deal with whole entities. The performance of whole-row serialization is a key metric. Cosmos DB, like other database systems, must also provide efficient operations that fall into three other categories:

**Patch**: Patch or a *partial update* is a mutation that affects only a small fraction of a larger row. In many cases only a single cell within the row is updated. The update may be expressed as a function of an existing value (e.g. increment an integer field by 1, append an item to a list). Patch is most efficient if it can be accomplished without materializing the parts of the row that are not being read or modified.

**Set-Based Updates**: A set based update performs the same patch, i.e. a partial read followed by a partial update, across an entire set of rows instead of on just a single row.

**Batch Operations**: A batch operation performs multiple distinct operations to different rows either for transactional purposes (atomically), or in parallel for bulk efficiency.

# HybridRow design

HybridRow is a bespoke encoding designed specifically for Cosmos DB. It derives its name from the hybridization of traditional relational database row layouts like those used by Microsoft SQL Server, and extensible transport encodings like BSON or Protobuf.

The HybridRow format is used for three purposes:

**Database Row Format**: it defines the logical structure of user content stored within the database. The HybridRow type system drives schema definition and provides for strongly typed indexing and query.

**Storage Format**: it defines a storage efficient physical encoding of data in the storage system. Allows for partial materialization, partial update, and deferred row upgrade.

**Transport Format**: it provides a library for fast serialization and deserialization of user content and protocol structures.

## Tables

A *Table* is defined as a sequence of *Rows*. Associated with a table is a *Namespace* of *Schema* versions. Each namespace contains one or more schema, and for each schema one or more versions forming an ordered sequence of schema evolutions. Every table has one distinguished schema, called the *Table Schema*, that defines the structure of the table’s rows. The namespace may also contain additional schema defining nested structures within a row.

Every existing row within a table conforms to exactly one version of the Table Schema. The version of the Table Schema used to encode a row is called the *Row Version*. The most recent version of the Table Schema is called the *Latest Version*. A row may be converted at any time from its current Row Version to the Latest Version through a deterministic process called *row upgrade*. All of the rows within a table need not have the same row version, as long as they all conform to some row version known to the Backend where they are stored. A Backend decides when to perform a persistent row upgrade on old rows. Such an upgrade may occur during insertion, updates, query, or as part of periodic background operations such as index maintenance or asynchronous schema propagation.

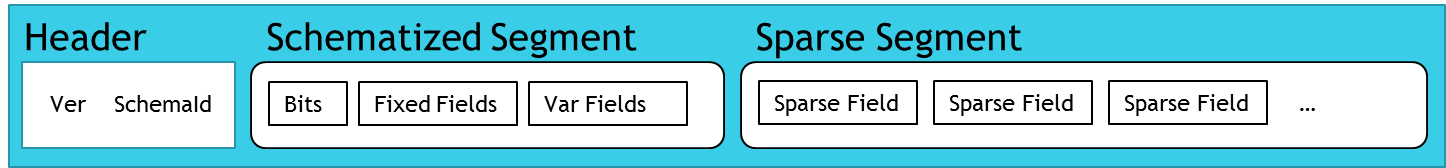


Figure 1: Hybrid Row Layout

## Rows

An individual row is defined by a specific version of a schema, which in turn optionally defines a named set of typed fields. Each row includes a header and encodes its fields across two storage segments as depicted in Figure 1.

### Header

The header defines the version of the HybridRow encoding used to create this row. A single table can include rows encoded across multiple versions of the HybridRow specification as long as each version is known to the Backend where the rows are stored. The *schema id* is a 4-byte integer, unique within the namespace associated with a table, that identifies the schema version used to create the row. The namespace may contain multiple versions of the Table Schema, and the schema id within any given row need not be the Latest Version. The existence of a schema id within the table’s namespace ensures recursively that all schema necessary to interpret this row are also defined in the namespace. This property is discussed in greater detail in the section Schema Versioning below.

### Schematized Segment

Placement of fields within the *Schematized Segment* versus the *Sparse Segment* is controlled by the storage specification of the field within the schema. The two segments offer different modeling choices to APIs with tradeoffs in storage efficiency and update performance.

The Schematized Segment allows the encoding to leverage the schema to elide field metadata in favor of a more compact representation. This segment is the direct analog of traditional relational database layouts. It is comprised of three parts:

**Bits**: indicate the presence of nullable fields. Bits also store the values of Boolean fields.

**Fixed Fields**: reserve storage in-row for fixed-size values. Only the data itself is stored, no metadata.

**Variable Fields**: store raw fixed-size or variable-length values, but only when present. Again, only the data itself is stored, no metadata.

Because the Schematized Segment stores only raw values without either name or type metadata, the corresponding schema must be available to interpret the contents.

### Sparse Segment

In contrast to the Schematized Segment, the *Sparse Segment* is fully self-describing and does not require the availability of schema to interpret. The Sparse Segment is composed of a sequence of zero or more *Sparse Fields*. As shown in Figure 2, each sparse field is a 3-tuple.



Figure 2: Sparse Field

* **Type Code**: type metadata that indicates how to interpret the value.
* **Path**: scope-relative metadata names the field. Field names do NOT need to be unique.
* **Value**: byte sequences store raw, optionally size-prefixed, field values.

Because the Sparse Segment is self-describing it can be used to store fields that either conform or diverge from the field definition within the schema. This enables partial schematization discussed in greater detail in the section Partial Schema below.

### Storage Tradeoffs

When modelling data within HybridRow, the various storage options available allow design-time tradeoffs for maximizing individual workloads.

Fixed fields use reservations and so always take up storage, but they provide the fastest possible access and partial updates. They can also achieve the highest density since they require zero overhead for always present values. We leverage fixed fields in Cassandra, for example, to store the partition and cluster key columns which are present in every row.

Variable and nullable fields also have very high density requiring only a single bit of overhead for nullable values, and a varint-encoded length prefix for variable length values. Variable fields provide a good balance of both storage size and metadata overhead. It is the preferred storage class for general purpose schematized fields in Cassandra, MongoDB and SQL. Variable fields require linked-list traversal to access, and block-copy operations to perform in-place updates, so their access methods are slightly more expensive than fixed fields.

Sparse fields provide the greatest flexibility but at a corresponding cost in access time and storage overhead. Sparse fields can be schematized or not, and in-row values may be constrained to match the schema or allowed to diverge, depending on the needs of the workload. Fully sparse rows can even be written without a schema at all by leveraging the HybridRow system schemas. These features are key to the implementation of partial schematization and schema inference discussed in detail in Section 3.

Sparse fields encode both name and type metadata. Duplicates are allowed, if desired. Top-level sparse fields take up no storage at all when not present, enabling the efficient modelling of very wide sparse tables such as those commonly found in Cassandra and Graph. Sparse segments can be arbitrarily divided at field boundaries allowing partial field decomposition and inexpensive reassembly. This feature is used to avoid storage duplication when building multiple indexes and to implement transparent normalization techniques like Cassandra’s Static Columns.

Sparse fields also support complex content such as collections, UDTs, and fully nested tables (as discussed in more detail in the section Type System). Nested content makes it possible to model Cassandra’s collections and UDTs. JSON and BSON’s deeply nested object/array structures map well to nested sparse collections, and HybridRow yields additional benefits in storage savings from typed collections and UDTs where possible. Nested tables allow designers to make tradeoffs between normalization and collocation. These are common challenges in NoSQL and scale-out relational workloads.

## Type System

The HybridRow type system includes a broad set of types including a variety of primitive scalar types, collection types, and user defined records.

### Primitives

HybridRow provides support for over 20 primitive types including various signed and unsigned precise numeric types up to 64 bits, IEEE floating point types up to 128 bits, variable length encoded integers (varints), and both precise and granular timestamps. In addition, the type system includes a set of query critical primitive types that often appear in databases but are less commonly included natively in serialization formats. DateTime, Guid, and Decimal (for currency operations) are available. Variable length types include UTF8 strings and binary byte sequences which support efficient in-row storage of blobs like images and data tables.

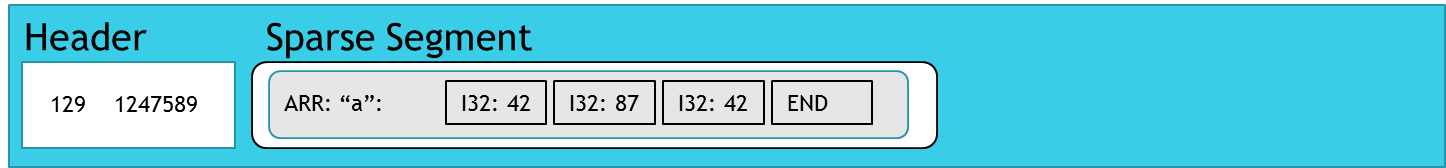
All types support optional nullability.

### Collections

In addition to scalars, sparse fields can include collections. Collections allow multiple cells to be stored in a single field. Collections can nest to any depth. The cell format within a collection is the same as a sparse field itself which enables full composability. The collection type system offers sufficient flexibility to model both the JSON/BSON data model and Cassandra’s UDTs.

Collection types come in four categories:

**Sparse Collections**: Sparse collections, including array and object, are the most general of the collection types. Sparse collections contain zero or more sparse fields. Their elements may be heterogenous. Different rows conforming to the same schema need not have either the same number or types of elements. Some examples of sparse collection encodings can be seen in Figure 3.



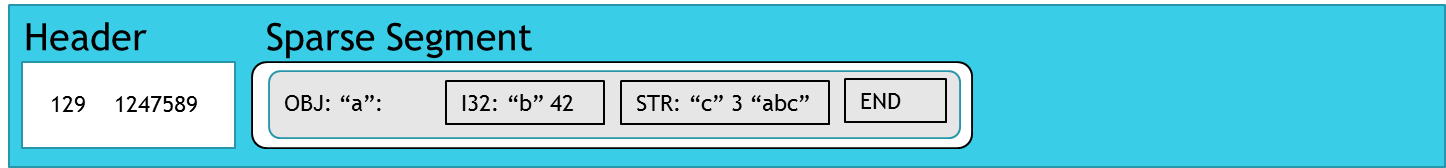


Figure 3: Sparse Collection Examples

**Typed Collections**: Typed collections, which include array<T> and tuple<T, …>, extend upon sparse collections and optimize the encoding format for the common case of homogenous elements. A typed collection encodes its values in a densely packed layout while representing the common item metadata as a generic parameter to the collection type itself. Figure 4 shows an example of a densely packed array<int32>.

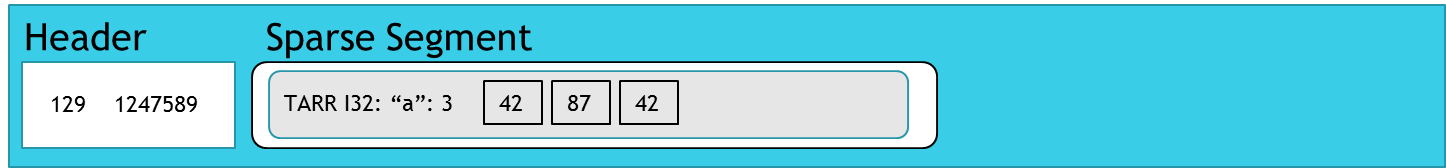


Figure 4: Typed Collection Example

**Indexed Collections**: Indexed collections, such as set<T> and map<K, V>, further extend typed collections with the addition of clustered indexes that ensure sort order and uniqueness. Equality is defined by a cross-platform binary collation which is aware of the sparse field encoding. This ensures that equality is defined at the encoding layer and not subject to language-specific comparator logic that might differ between sender and receiver. Additionally, equality is well-defined for all arbitrarily nested sparse content including, for example, set<map<array, string> where the map key is itself a sparse collection whose elements could be any type. Figure 5 shows an example of a map with unique integer keys.

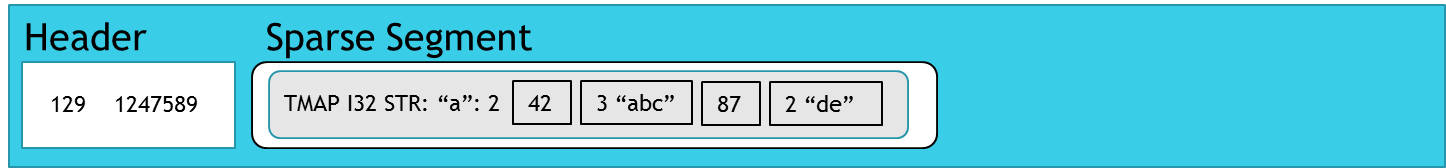


Figure 5: Indexed Collection Example

**User Defined Types:** User defined record types (UDTs) complete the composability story. UDTs are embedded rows defined by their own schema, independent from their containing row. UDT values recursively define a new HybridRow within a cell comprised of its own Schematized and Sparse Segments. UDTs allow the sharing of structured type definitions across tables within a database with correlated evolution.

When coupled with other collections, UDTs allow for the definition of fully nested tables within a row. Cosmos DB’s query runtime supports querying nested tables, UNNEST and IN operations, and subqueries. Nested tables can be indexed. Figure 6 show an example of a nested table containing two rows.

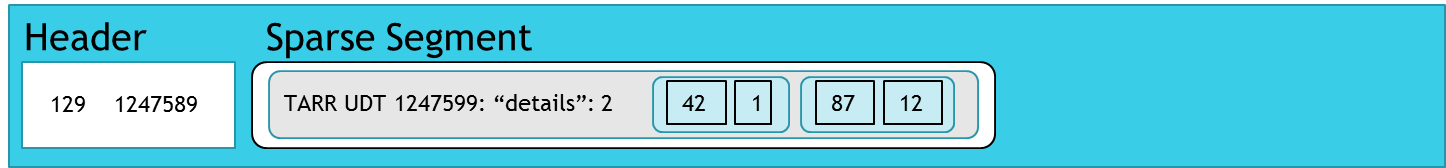


Figure 6: Nested Table Example

## APIs and Languages

HybridRow participates in all tiers of Cosmos DB. Its native C++ implementation is used in the Backends. It is incorporated into all language SDKs for which Cosmos DB provides an enlightened client driver including C# and Java.

Though primarily designed as a database row format, the necessity for communicating whole or partial row content as the payload of network messages drove the need for HybridRow’s use also as a transport format. Additionally, database exports, test datasets, and modeling have given rise to its use as a file serialization format.

The HybridRow library provides direct support for three patterns of use:

**Streaming**: When reading or writing whole rows from left to right, the forward‑only streaming API provides the best performance. It is used during inserts and bulk operations, in formulating response rows during query projection, and during streaming transformations.

**RowCursor**: When reading or updating individual cells within an existing row, the row cursor API provides optimal access methods. It supports seeking to individual cells without materializing other fields and allows reads and in-place updates of all field types including collections. RowCursor is used to implement patch operations, set-based updates, and value extraction for query projection and index maintenance.

**Generated Code**: When dealing with RPC transmissions or fixed file formats where the schema of the dataset is known at compile-time, the HybridRow library includes code generation tools that produce structs and optimized encoders and decoders.

# HybridRow Schema

HybridRow describes structure through a high-level *Schema Definition Language* (SDL). Figure 7 shows an example schema *Namespace*. Like Avro and Protobuf, HybridRow’s SDL is centered around the definition of types called *schema*. Each type defines zero or more schematized fields. Field definitions include a name, a field type, traits like nullability, and an optional storage class. Field types may be scalar or complex. Field types may refer to built-in types or reference another schema defined within the same namespace. User defined types are referenced by name and schema id.

## Schema Compiler

Schema namespaces can be created by authoring the textual SDL or using the programmable object model. Schema may be stored as text or as binary (using its own HybridRow schema). A namespace definition, however, is not consumed directly by the serialization system. Instead it must first pass through the *schema compiler*. The schema compiler produces a *layout* which is a data structure containing encoding offsets, tokenization tables, and other metadata that drives encoding and decoding. The schema compiler can also optionally generate optimized code for an encoder and decoder specific to a particular schema where the layout’s numerics become compile-time constants allowing code inlining and other optimizations.

Both the Frontends and Backends are multi-tenant environments in Cosmos DB and so leverage dynamic schema compilation and dynamic layouts, rather than generated code, for customer schemas. Schema namespaces are compiled on-demand when partitions are loaded and freed when partitions are moved. By contrast, transport messages such as patch and batch use code-generation based encoding and decoding for improved performance.

## Partial Schema

The schema in HybridRow drives many storage compression and performance benefits. Schematized fields elide metadata that is redundantly expressed in the layout. String tokenization tables in the layout compress strings within the row. Typed collections lead to more packed representations of the same data. Fixed and variable fields use compile-time computed offsets to speed encoding and decoding.

In addition to the obvious performance benefits of schema, traditionally the primary function of schema is structural enforcement. However, in Cosmos DB, strong schema adherence is neither required nor even desirable for some workloads. To support the spectrum of model requirements HybridRow schema support three modes of structural enforcement (specified as a schema trait):

**Full Schema**: Fully schematized environments, like Cassandra, both express schema for all fields and require that all values conform to the schema. In full schema mode, HybridRow behaves like a traditional relational system. When writing a field or cell, the value must confirm to all constraints specified within the schema or an error is raised. Data in this environment is always stored optimally, and the layout of the row is determined at schema compilation. The evolution of types in subsequent versions is constrained to deterministic mutations.

|  |
| --- |
| {      name: "Microsoft.Azure.Cosmos.Serialization.HybridRow.Tests.Perf.Hotel",      schemas: [      {        name: "PostalCode",        id: 1,        type: "schema",        fields: [          {path: "zip", type: {type: "int32", storage: "fixed"}},          {path: "plus4", type: {type: "int16", storage: "sparse"}}        ]      },      {        name: "Address",        id: 2,        type: "schema",        fields: [          {path: "street", type: {type: "utf8", storage: "variable"}},          {path: "city", type: {type: "utf8", storage: "variable"}},          {path: "state", type: {type: "utf8", storage: "fixed", length: 2}},          {path: "postal\_code", type: {type: "schema", name: "PostalCode", id: 1}}        ]      },      {        name: "Hotels",        id: 3,        type: "schema",        partitionkeys: [{path: "hotel\_id"}],        fields: [          {path: "hotel\_id", type: {type: "utf8", storage: "fixed", length: 8}},          {path: "name", type: {type: "utf8", storage: "variable"}},          {path: "phone", type: {type: "utf8", storage: "variable"}},          {path: "address", type: {type: "schema", name: "Address", id: 2, immutable: true}}        ]      },      {        name: "Guests",        id: 5,        type: "schema",        partitionkeys: [{path: "guest\_id"}],        primarykeys: [{path: "first\_name"}, {path: "phone\_numbers", direction: "desc"}],        fields: [          {path: "guest\_id", type: {type: "guid", storage: "fixed"}},          {path: "first\_name", type: {type: "utf8", storage: "variable"}},          {path: "last\_name", type: {type: "utf8", storage: "variable"}},          {path: "title", type: {type: "utf8", storage: "variable", length: 20}},          {path: "emails", type: {type: "array", items: {type: "utf8", nullable: false}}},          {path: "phone", type: {type: "array", items: {type: "utf8", nullable: false}}},          {            path: "addresses",            type: {              type: "map",              keys: {type: "utf8", nullable: false},              values: {type: "schema", name: "Address", id: 2, nullable: false}            }          },          {path: "confirm\_number", type: {type: "utf8", storage: "variable"}}        ]      }  } |

Figure 7: Schema Namespace Example

**Partial Schema**: The partial schema mode allows the HybridRow library to speculatively take advantage of size compression and performance optimizations for those fields which are specified while still allowing non-conforming values. When values agree, they are written to the row as specified by the schema. When values diverge, they are written to the row as sparse fields. Partial schema is the default mode for document store models like the SQL API where there is very high diversity between rows. It is not necessary to define a single schema that encompasses all rows. Defining an approximate schema that covers the most important fields while allowing other content to naturally flow to sparse fields achieves the benefits of schematization without any of the evolutionary or versioning difficulties.

Partial schema makes schema evolution easy by eliminating the race conditions between schema upgrade and application deployment. Because partial schema is essentially a performance optimization, new schema can be rolled out at the same time as code that consumes it. If the new code attempts to write a row before the new schema has propagated, no data will be lost. When the new schema arrives, the old row will simply be subject to *row upgrade*.

Partial schema also enables the imprecise process of schema inference. Because it is not necessary for all rows to conform to the schema, it is possible for a background machine learning process to sample data in the table and *guess* at a best-fit schema. Even if the guess is not ideal, no data is ever lost, and correctness is maintained at all times. Partial benefits can still be achieved for the subset of fields identified by the schema inference model. The inference system can run again, perhaps when more data is available, or when a new machine learning model is trained, to produce better results.

**Partial Enforcement**: Lastly, the partial enforcement mode offers the benefits of both the full and partial modes. Like MongoDB, in this mode HybridRow can be directed to enforce the structure of schema specified fields while still allowing partial schema semantics for unspecified fields. This mode ensures a high degree of conformance for query-critical fields while leveraging the sparse segment capabilities to store semi-structured data that may differ from row to row.

## Query Runtime

HybridRow interacts with the query runtime in many different ways.

Rows arrive as HybridRow payloads from RPC interactions where they are stored directly in HybridRow format in the table’s clustered index. Depending on the indexing policy, a subset of the fields is extracted from the row via a process called *term generation* using RowCursor. The terms, along with their type information, are written into secondary indexes using the BwTree.

Cosmos DB’s query syntax supports robust query specifications including filters, aggregations, subqueries, UNNEST and IN operators, and projections. Query execution is type-aware and expressed over the unified HybridRow type system. Projections result in a dynamic schema specification for the result set where each output column’s type is the result of strongly typed expression compilation during query evaluation. Query result sets are returned on the wire via a streaming self-describing HybridRow *RecordIO* format which optionally includes dynamic result set schema definitions, and streaming row generation. Each streaming result record is a separate HybridRow. In SELECT \* queries, the dynamic schema is elided, being the same as the Latest Version of the Table Schema, and the row buffers are block copied directly from the clustered index to achieve optimal throughput.

## Schema Versioning

At the time of table creation, the table is assigned a schema *Namespace* which defines one distinguished schema to act as the *Table Schema*. The Table Schema may reference other schemas both directly and indirectly via UDT field definitions. All schema references must be resolvable within the namespace or a schema compilation error is raised. In Figure 7, the Guest:5 schema refers directly to Address:2 which in turn references PostalCode:1 resulting in the schema dependency graph show in Figure 8. The *schema id* of the Table Schema is written into the header of every row inserted into the table.

Over time, schemas change. Schema modifications (DDLs) are submitted by a client to a table’s master partition which holds the authoritative copy of the table’s schema namespace. The proposed modification is merged with the existing namespace to form a new namespace which is compiled and checked for correctness and evolutionary constraints. If the new namespace passes these checks the DDL is committed at the master. Knowledge of the change is then asynchronous propagated to Backends, Frontends, and clients in parallel.

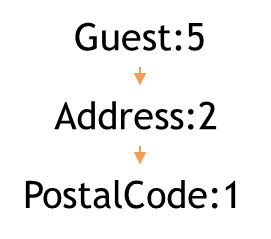


Figure 8: Dependency Graph

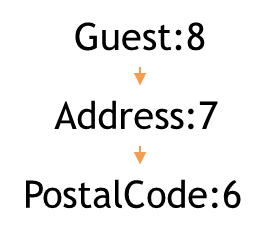


Figure 9: Dependency Graph After ALTER UDT

When altering the Table Schema, a new schema id unique within the namespace is assigned to the new version. This id is required to be monotonically increasing relative to the immediately previous version, resulting in a totally ordered sequence of schema versions within the namespace. New rows written after the master commits may logically include any valid schema id up to this new Latest Version. A Backend will only accept or produce rows using the new schema id once knowledge of it has successfully propagated to that Backend. However, a Backend will continue to accept rows written with previous versions allowing propagation to both Frontends and clients independently.

When altering a UDT schema embedded within a table, the UDT schema like the Table Schema is assigned a new schema id. When the UDT’s modification is merged to form the new namespace, new versions of schema for which the modified UDT appears in its dependency graph are dynamically generated. Figure 9 shows the new state of the Guest:8 dependency graph after a UDT modification to PostalCode:1 has been committed resulting in PostalCode:6.

This schema id roll-up behavior allows rows requiring a *row upgrade* to be identified efficiently. Only the row header at the beginning of the row must be examined to quickly identify all schemas needed to process the row. The row’s internal structure does not need to be parsed. If the row’s version is known to the Backend, Frontend or client, then subsequently all schema referenced within the row’s encoding must also be known.

The need for row upgrade can arise at any tier within the system. A Backend may perform a row upgrade when receiving a row before storing it, when reading a row for query projection, when performing term generation for index maintenance or during an index policy change. Additionally, a background maintenance job called RowRewriter may periodically sweep over old rows to perform integrity checking and to upgrade rows to a newer schema version. Only after a fully completed sweep of RowRewriter across all partitions of a table is it possible to garbage collect old schema versions and to prune the namespace. This process runs asynchronously and lazily at each Backend that holds partitions for a table. The global state of the sweep is maintained by the master partition.

Frontends and clients in direct mode may also perform row upgrades when processing a result set. This can arise either because the Frontend compiled the incoming request against a more recent version of the schema than that known by the Backend from which the row was returned, or because the Backend chose to block-copy an existing row buffer from the clustered index for performance. In the latter case the Backend need not know which version of the schema the Frontend will eventually require and so it is optimal to defer the row upgrade as late as possible. Clients may even further defer this process by passing the row buffer unprocessed downstream to another consumer.

# Performance Analysis

In this section we provide a brief summary of some key performance metrics. All tests were run using the C# versions of each library, on the same hardware, using the same datasets. Three different sample schemas were used:

**Room**: A very small row using only fixed or variable types.

**Hotel**: A medium size row with a mix of types including some simple nested UDTs.

**Guest**: A large size row with heavy use of collections, UDTs, and complex nested content.

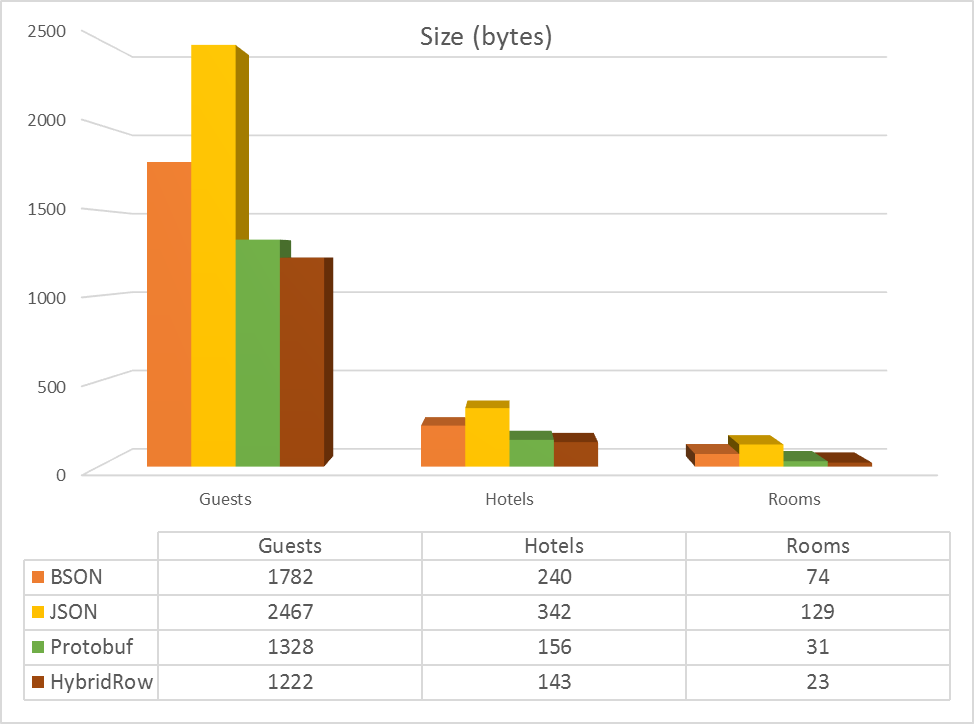


Figure 10: Size

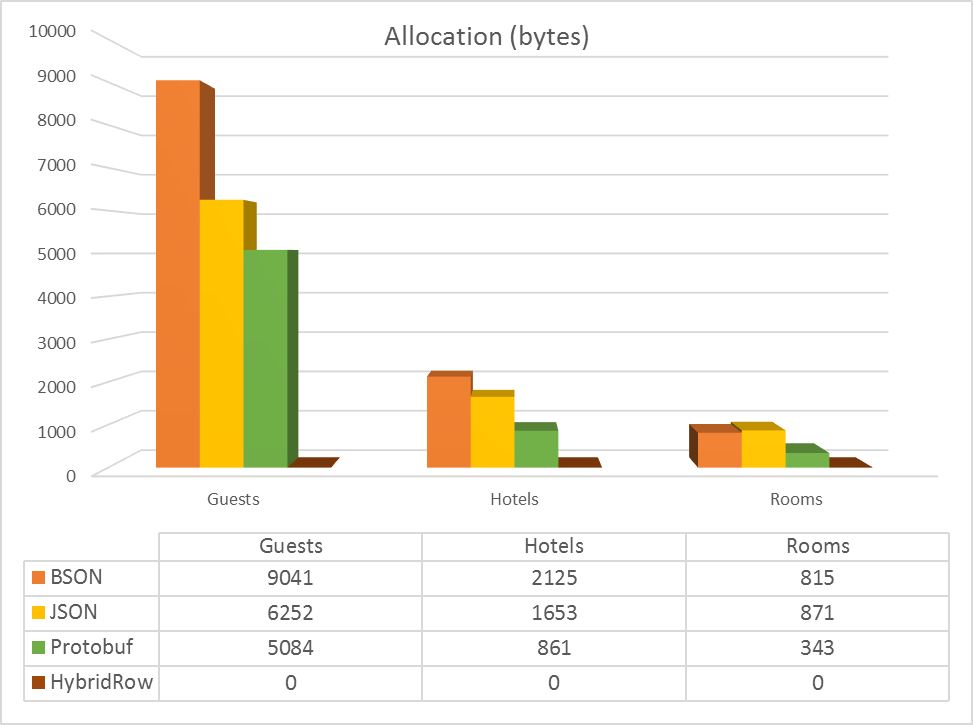


Figure 11:Allocations

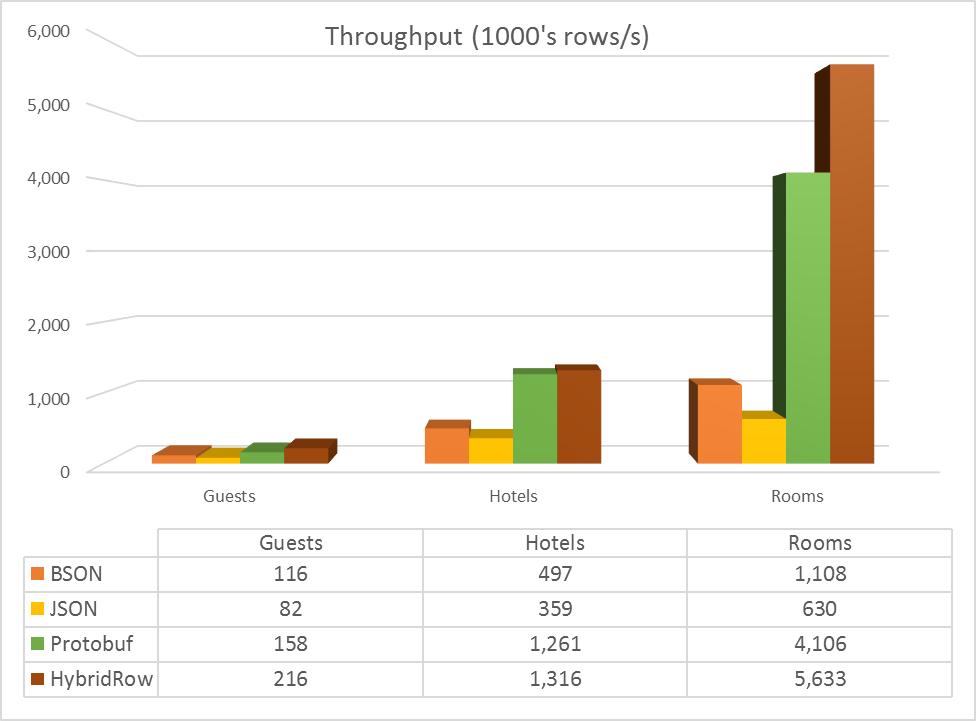


Figure 12: Read Throughput

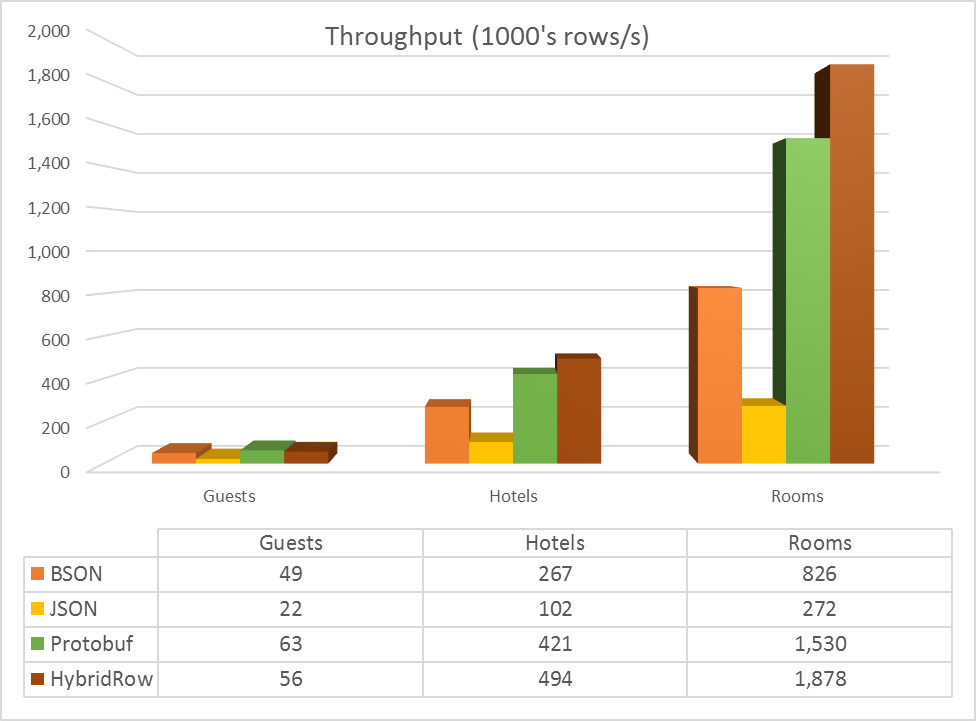


Figure 13: Write Throughput

## Size

Figure 10 shows a comparison of the average encoded size across a set of representative rows in each sample dataset. HybridRow’s encoding is very competitive when compared to other common fully structured and semi-structured formats. In our observations string tokenization, native support for binary sequences, precise integers, and packed arrays provide for the largest gains in most workloads.

## Throughput

Serialization performance is critical to all database operations. Figure 12 shows a comparison of read throughput as a function of how many rows can be decoded per second (in thousands). Both decoding and encoding time is proportional to the size and complexity of the input. As expected, schematized fields benefit the most from very fast access methods. Figure 13 compares write throughput as function of rows encoded per second (in thousands).

## Allocations

Both the Frontends and the Backends in Cosmos DB are long running multi-tenant processes. Additionally, most clients of Cosmos DB are also long-running services. Memory management and efficient reuse is a key performance metric for healthy server operation. Figure 11 shows the memory use of each library at steady state, assuming the row buffer is reused across rows in a result set. Because HybridRow performs its operations in-place over the serialized form directly, it allocates little or no additional memory during reading and writing.