Performance Benchmarking

- Status: Accepted
- Minimum Server Version: N/A

Abstract

This document describes a standard benchmarking suite for MongoDB drivers.

Overview

Name and purpose

Driver performance will be measured by the MongoDB Driver Performance Benchmark (AKA "DriverBench"). It will provide both "horizontal" insights into how individual language driver performance evolves over time and "vertical" insights into relative performance of different drivers.

We do expect substantial performance differences between language families (e.g. static vs. dynamic or compiled vs. virtual-machine-based). However we still expect "vertical" comparison within language families to expose outlier behavior that might be amenable to optimization.

Task Hierarchy

The benchmark consists of a number of micro-benchmarks tasks arranged into groups of increasing complexity. This allows us to better isolate areas within drivers that are faster or slower.

- BSON -- BSON encoding/decoding tasks, to explore BSON codec efficiency
- Single-Doc -- single-document insertion and query tasks, to explore basic wire protocol efficiency
- Multi-Doc -- multi-document insertion and query tasks, to explore batch-write and cursor chunking efficiency
- Parallel -- multi-process/thread ETL tasks, to explore concurrent operation efficiency

Measurement

In addition to timing data, all micro-benchmark tasks will be measured in terms of "megabytes/second" (MB/s) of documents processed, with higher scores being better. (In this document, "megabyte" refers to the SI decimal unit, i.e. 1,000,000 bytes.) This makes cross-benchmark comparisons easier.

To avoid various types of measurement skew, tasks will be measured over numerous iterations. Each iteration will have a "scale" -- the number of similar operations performed -- that will vary by task. The final score for a task will be the median score of the iterations. Other quantiles will be recorded for diagnostic analysis.

Data sets

Data sets will vary by micro-benchmark. In some cases, they it will be a synthetically generated document inserted repeatedly (with different `_id` fields) to construct an overall corpus of documents. In other cases, data sets will be synthetic line-delimited JSON files or mock binary files.

Composite scores

Micro-benchmark scores will be combined into a composite for each weight class ("BSONBench", "SingleBench", etc.) and for read and write operations ("ReadBench" and "WriteBench"). The read and write scores will be combined into an aggregate composite score ("DriverBench"). The compositing formula in the DriverBench uses simple averages with equal weighting.

Versioning

DriverBench will have vX.Y versioning. Minor updates and clarifications will increment "Y" and should have little impact on score comparison. Major changes, such as changing score weights, MongoDB version tested against, or hardware used, will increment "X" to indicate that older version scores are unlikely to be comparable.

Benchmark execution phases and measurement

All micro-benchmark tasks will be conducted via a number of iterations. Each iteration will be timed and will generally include a large number of individual driver operations.

We break up the measurement this way to better isolate the benchmark from external volatility. If we consider the problem of benchmarking an operation over many iterations, such as 100,000 document insertions, we want to avoid two extreme forms of measurement:

- measuring a single insertion 100,000 times -- in this case, the timing code is likely to be a greater proportion of executed code, which could routinely evict the insertion code from CPU caches or mislead a JIT optimizer and throw off results
- measuring 100,000 insertions one time -- in this case, the longer the timer runs, the higher the likelihood that an external event occurs that affects the time of the run

Therefore, we choose a middle ground:

- measuring the same 1000 insertions over 100 iterations -- each timing run includes enough operations that insertion code dominates timing code; unusual system events are likely to affect only a fraction of the 100 timing measurements

With 100 timings of inserting the same 1000 documents, we build up a statistical distribution of the operation timing, allowing a more robust estimate of performance than a single measurement. (In practice, the number of iterations could exceed 100, but 100 is a reasonable minimum goal.)

Because a timing distribution is bounded by zero on one side, taking the mean would allow large positive outlier measurements to skew the result substantially. Therefore, for the benchmark score, we use the median timing measurement, which is robust in the face of outliers.

Each benchmark is structured into discrete setup/execute/teardown phases. Phases are as follows, with specific details given in a subsequent section:

- setup -- (ONCE PER MICRO-BENCHMARK) something to do once before any benchmarking, e.g. construct a client object, load test data, insert data into a collection, etc.
- before task -- (ONCE PER ITERATION) something to do before every task iteration, e.g. drop a collection, or reload test data (if the test run modifies it), etc.

- do task -- (ONCE PER ITERATION) smallest amount of code necessary to execute the task; e.g. insert 1000 documents one by one into the database, or retrieve 1000 document of test data from the database, etc.
- after task -- (ONCE PER ITERATION) something to do after every task iteration (if necessary)
- teardown -- (ONCE PER MICRO-BENCHMARK) something done once after all benchmarking is complete (if necessary); e.g. drop the test database

The wall-clock execution time of each "do task" phase will be recorded. We use wall clock time to model user experience and as a lowest-common denominator across languages and threading models. Iteration timing should be done with a high-resolution monotonic timer (or best language approximation).

Unless otherwise specified, the number of iterations to measure per micro-benchmark is variable:

- iterations should loop for at least 1 minute cumulative execution time
- iterations should stop after 100 iterations or 5 minutes cumulative execution time, whichever is shorter

This balances measurement stability with a timing cap to ensure all micro-benchmarks can complete in a reasonable time. Languages with JIT compilers may do warm up iterations for which timings are discarded.

For each micro-benchmark, the 10th, 25th, 50th, 75th, 90th, 95th, 98th and 99th percentiles will be recorded using the following algorithm:

- Given a 0-indexed array A of N iteration wall clock times
- Sort the array into ascending order (i.e. shortest time first)
- Let the index i for percentile p in the range [1,100] be defined as: `i = int(N * p / 100) - 1`

N.B. This is the Nearest Rank algorithm, chosen for its utter simplicity given that it needs to be implemented identically across multiple languages for every driver.

The 50th percentile (i.e. the median) will be used for score composition. Other percentiles will be stored for visualizations and analysis (e.g. a "candlestick" chart showing benchmark volatility over time).

Each task will have defined for it an associated size in megabytes (MB). The score for micro-benchmark composition will be the task size in MB divided by the median wall clock time.

Micro-benchmark definitions

Datasets are available in the `data` directory adjacent to this spec.

Note: The term "LDJSON" means "line-delimited JSON", which should be understood to mean a collection of UTF-8 encoded JSON documents (without embedded CR or LF characters), separated by a single LF character. (Some Internet definition of line-delimited JSON use CRLF delimiters, but this benchmark uses only LF.)

BSON micro-benchmarks

Datasets are in the `extended_bson` tarball.

BSON tests focus on BSON encoding and decoding; they are client-side only and do not involve any transmission of data to or from the benchmark server. When appropriate, data sets will be stored on disk as

extended strict JSON. For drivers that don't support extended JSON, a BSON analogue will be provided as well.

BSON micro-benchmarks include:

- Flat BSON Encoding and Flat BSON Decoding -- shallow documents with only common BSON field types
- Deep BSON Encoding and Deep BSON Decoding -- deeply nested documents with only common BSON field types
- Full BSON Encoding and Full BSON Decoding -- shallow documents with all possible BSON field types

Flat BSON Encoding

Summary: This benchmark tests driver performance encoding documents with top level key/value pairs involving the most commonly-used BSON types.

Dataset: The dataset, designated FLAT_BSON (ftnt4 Disk file `flat_bson.json`), will be synthetically generated and consist of an extended JSON document with a single `_id` key with an object ID value plus 24 top level keys/value pairs of the following types: string, Int32, Int64, Double, Boolean. (121 total key/value pairs) Keys will be random ASCII strings of length 8. String data will be random ASCII strings of length 80.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (7531 bytes) times 10,000 operations, which equals 75,310,000 bytes or 75.31 MB.

Phase	Description
Setup	Load the FLAT_BSON dataset into memory as a language-appropriate document types. For languages like C without a document type, the raw JSON string for each document should be used instead.
Before task	n/a
Do task	Encode the FLAT_BSON document to a BSON byte-string. Repeat this 10,000 times.
After task	n/a
Teardown	n/a

Flat BSON Decoding

Summary: This benchmark tests driver performance decoding documents with top level key/value pairs involving the most commonly-used BSON types.

Dataset: The dataset, designated FLAT_BSON, will be synthetically generated and consist of an extended JSON document with a single `_id` key with an object ID value plus 24 top level keys/value pairs of each of the following types: string, Int32, Int64, Double, Boolean. (121 total key/value pairs) Keys will be random ASCII strings of length 8. String data will be random ASCII strings of length 80.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (7531 bytes) times 10,000 operations, which equals 75,310,000 bytes or 75.31 MB.

Phase	Description
Setup	Load the FLAT_BSON dataset into memory as a language-appropriate document types. For languages like C without a document type, the raw JSON string for each document should be used instead. Encode it to a BSON byte-string.
Before task	n/a
Do task	Decode the BSON byte-string to a language-appropriate document type. Repeat this 10,000 times. For languages like C without a document type, decode to extended JSON instead.
After task	n/a
Teardown	n/a

Deep BSON Encoding

Summary: This benchmark tests driver performance encoding documents with deeply nested key/value pairs involving subdocuments, strings, integers, doubles and booleans.

Dataset: The dataset, designated DEEP_BSON (disk file `deep_bson.json`), will be synthetically generated and consist of an extended JSON document representing a balanced binary tree of depth 6, with "left" and "right" keys at each level containing a sub-document until the final level, which will contain a random ASCII string of length 8 (126 total key/value pairs).

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (1964 bytes) times 10,000 operations, which equals 19,640,000 bytes or 19.64 MB.

Phase	Description
Setup	Load the DEEP_BSON dataset into memory as a language-appropriate document type. For languages like C without a document type, the raw JSON string for each document should be used instead.
Before task	n/a
Do task	Encode the DEEP_BSON document to a BSON byte-string. Repeat this 10,000 times.
After task	n/a
Teardown	n/a

Deep BSON Decoding

Summary: This benchmark tests driver performance decoding documents with deeply nested key/value pairs involving subdocuments, strings, integers, doubles and booleans.

Dataset: The dataset, designated DEEP_BSON, will be synthetically generated and consist of an extended JSON document representing a balanced binary tree of depth 6, with "left" and "right" keys at each level containing a sub-document until the final level, which will contain a random ASCII string of length 8 (126 total key/value pairs).

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (1964 bytes) times 10,000 operations, which equals 19,640,000 bytes or 19.64 MB.

Phase	Description
Setup	Load the DEEP_BSON dataset into memory as a language-appropriate document types. For languages like C without a document type, the raw JSON string for each document should be used instead. Encode it to a BSON byte-string.
Before task	n/a
Do task	Decode the BSON byte-string to a language-appropriate document type. Repeat this 10,000 times. For languages like C without a document type, decode to extended JSON instead.
After task	n/a
Teardown	n/a

Full BSON Encoding

Summary: This benchmark tests driver performance encoding documents with top level key/value pairs involving the full range of BSON types.

Dataset: The dataset, designated FULL_BSON (disk file `full_bson.json`), will be synthetically generated and consist of an extended JSON document with a single `_id` key with an object ID value plus 6 each of the following types: string, double, Int64, Int32, boolean, minkey, maxkey, array, binary data, UTC datetime, regular expression, Javascript code, Javascript code with context, and timestamp. (91 total keys.) Keys (other than `_id`) will be random ASCII strings of length 8. Strings values will be random ASCII strings with length 80.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (5734 bytes) times 10,000 operations, which equals 57,340,000 bytes or 57.34 MB.

Phase	Description
Setup	Load the FULL_BSON dataset into memory as a language-appropriate document type. For languages like C without a document type, the raw JSON string for each document should be used instead.
Before task	n/a
Do task	Encode the FULL_BSON document to a BSON byte-string. Repeat this 10,000 times.
After task	n/a

Phase	Description
Teardown	n/a

Full BSON Decoding

Summary: This benchmark tests driver performance decoding documents with top level key/value pairs involving the full range of BSON types.

Dataset: The dataset, designated FULL_BSON, will be synthetically generated and consist of an extended JSON document with a single `_id` key with an object ID value plus 6 each of the following types: string, double, Int64, Int32, boolean, minkey, maxkey, array, binary data, UTC datetime, regular expression, Javascript code, Javascript code with context, and timestamp. (91 total keys.) Keys (other than `_id`) will be random ASCII strings of length 8. Strings values will be random ASCII strings with length 80.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (5734 bytes) times 10,000 operations, which equals 57,340,000 bytes or 57.34 MB.

Phase	Description
Setup	Load the FULL_BSON dataset into memory as a language-appropriate document types. For languages like C without a document type, the raw JSON string for each document should be used instead. Encode it to a BSON byte-string.
Before task	n/a
Do task	Decode the BSON byte-string to a language-appropriate document type. Repeat this 10,000 times. For languages like C without a document type, decode to extended JSON instead.
After task	n/a
Teardown	n/a

Single-Doc Benchmarks

Datasets are in the `single_and_multi_document` tarball.

Single-doc tests focus on single-document read and write operations. They are designed to give insights into the efficiency of the driver's implementation of the basic wire protocol.

The data will be stored as strict JSON with no extended types.

Single-doc micro-benchmarks include:

- Run command
- Find one by ID
- Small doc insertOne
- Large doc insertOne

Run command

Summary: This benchmark tests driver performance sending a command to the database and reading a response.

Dataset: n/a

Dataset size: While there is no external dataset, for score calculation purposes use 130,000 bytes (10,000 x the size of a BSON {hello:true} command).

N.B. We use {hello:true} rather than {hello:1} to ensure a consistent command size.

Phase	Description
Setup	Construct a MongoClient object. Construct whatever language-appropriate objects (Database, etc.) would be required to send a command.
Before task	n/a
Do task	Run the command {hello:true} 10,000 times, reading (and discarding) the result each time.
After task	n/a
Teardown	n/a

Find one by ID

Summary: This benchmark tests driver performance sending an indexed query to the database and reading a single document in response.

Dataset: The dataset, designated TWEET (disk file `tweet.json`), consists of a sample tweet stored as strict JSON.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (1622 bytes) times 10,000 operations, which equals 16,220,000 bytes or 16.22 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the TWEET document into memory as a language-appropriate document type (or JSON string for C). Construct a Collection object for the `corpus` collection to use for querying. Insert the document 10,000 times to the `perftest` database in the `corpus` collection using sequential `_id` values. (1 to 10,000)
Before task	n/a
Do task	For each of the 10,000 sequential `_id` numbers, issue a find command for that `_id` on the `corpus` collection and retrieve the single-document result.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc insertOne

Summary: This benchmark tests driver performance inserting a single, small document to the database.

Dataset: The dataset, designated SMALL_DOC (disk file `small_doc.json`), consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 10,000 operations, which equals 2,750,000 bytes or 2.75 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C).
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Insert the document with the insertOne CRUD method. DO NOT manually add an `_id` field; leave it to the driver or database. Repeat this 10,000 times.
After task	n/a
Teardown	Drop the `perftest` database.

Large doc insertOne

Summary: This benchmark tests driver performance inserting a single, large document to the database.

Dataset: The dataset, designated LARGE_DOC (disk file `large_doc.json`), consists of a JSON document with an encoded length of approximately 2,500,000 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (2,731,089 bytes) times 10 operations, which equals 27,310,890 bytes or 27.31 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the LARGE_DOC dataset into memory as a language-appropriate document type (or JSON string for C).
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Insert the document with the insertOne CRUD method. DO NOT manually add an `_id` field; leave it to the driver or database. Repeat this 10 times.
After task	n/a
Teardown	Drop the `perftest` database.

Multi-Doc Benchmarks

Datasets are in the `single_and_multi_document` tarball.

Multi-doc benchmarks focus on multiple-document read and write operations. They are designed to give insight into the efficiency of the driver's implementation of bulk/batch operations such as bulk writes and cursor reads.

Multi-doc micro-benchmarks include:

- Find many and empty the cursor
- Small doc bulk insert
- Large doc bulk insert
- Small doc Collection BulkWrite insert
- Large doc Collection BulkWrite insert
- Small doc Client BulkWrite insert
- Large doc Client BulkWrite insert
- Small doc Client BulkWrite Mixed Operations
- Small doc Collection BulkWrite Mixed Operations
- GridFS upload
- GridFS download

Find many and empty the cursor

Summary: This benchmark tests driver performance retrieving multiple documents from a query.

Dataset: The dataset, designated TWEET consists of a sample tweet stored as strict JSON.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (1622 bytes) times 10,000 operations, which equals 16,220,000 bytes or 16.22 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the TWEET dataset into memory as a language-appropriate document type (or JSON string for C). Construct a Collection object for the `corpus` collection to use for querying. Insert the document 10,000 times to the `perftest` database in the `corpus` collection. (Let the driver generate `_id`s).
Before task	n/a
Do task	Issue a find command on the `corpus` collection with an empty filter expression. Retrieve (and discard) all documents from the cursor.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc bulk insert

Summary: This benchmark tests driver performance inserting multiple small documents to the database using `insertMany`.

Dataset: The dataset, designated SMALL_DOC consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 10,000 operations, which equals 2,750,000 bytes or 2.75 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C).
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `insertMany` with 10,000 copies of the document. DO NOT manually add an `_id` field; leave it to the driver or database.
After task	n/a
Teardown	Drop the `perftest` database.

Large doc bulk insert

Summary: This benchmark tests driver performance inserting multiple large documents to the database using `insertMany`.

Dataset: The dataset, designated LARGE_DOC consists of a JSON document with an encoded length of approximately 2,500,000 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (2,731,089 bytes) times 10 operations, which equals 27,310,890 bytes or 27.31 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the LARGE_DOC dataset into memory as a language-appropriate document type (or JSON string for C).
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `insertMany` with 10 copies of the document. DO NOT manually add an `_id` field; leave it to the driver or database.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc Collection BulkWrite insert

Summary: This benchmark tests driver performance inserting multiple small documents to the database using the `Collection::bulkWrite` operation.

Dataset: The dataset, designated SMALL_DOC consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 10,000 operations, which equals 2,750,000 bytes or 2.75 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of insert models for 10,000 copies of the document, allowing the driver or database to auto-assign the `_id` field.
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `Collection::bulkWrite` with the list of insert models.
After task	n/a
Teardown	Drop the `perftest` database.

Large doc Collection BulkWrite insert

Summary: This benchmark tests driver performance inserting multiple large documents to the database using the `Collection::bulkWrite` operation.

Dataset: The dataset, designated LARGE_DOC consists of a JSON document with an encoded length of approximately 2,500,000 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (2,731,089 bytes) times 10 operations, which equals 27,310,890 bytes or 27.31 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the LARGE_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of insert models for 10 copies of the document, allowing the driver or database to auto-assign the `_id` field.
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `Collection::bulkWrite` with the list of insert models.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc Client BulkWrite insert

Summary: This benchmark tests driver performance inserting multiple small documents to the database using the `Client::bulkWrite` operation.

Dataset: The dataset, designated SMALL_DOC consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 10,000 operations, which equals 2,750,000 bytes or 2.75 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of insert models for 10,000 copies of the document in the same namespace ("perftest.corpus"), allowing the driver or database to auto-assign the `_id` field.
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `Client::bulkWrite` with the list of insert models.
After task	n/a
Teardown	Drop the `perftest` database.

Large doc Client BulkWrite insert

Summary: This benchmark tests driver performance inserting multiple large documents to the database using the `Client::bulkWrite` operation.

Dataset: The dataset, designated LARGE_DOC consists of a JSON document with an encoded length of approximately 2,500,000 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (2,731,089 bytes) times 10 operations, which equals 27,310,890 bytes or 27.31 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the LARGE_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of insert models for 10 copies of the document in the same namespace ("perftest.corpus"), allowing the driver or database to auto-assign the `_id` field.
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `Client::bulkWrite` with the list of insert models.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc Client BulkWrite Mixed Operations

Summary: This benchmark tests driver performance of a `Client::bulkWrite` operation with small documents and mixed operations (e.g. insert, replace and delete).

Dataset: The dataset, designated SMALL_DOC consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 20,000 operations (insert + replace), which equals 5,500,000 bytes or 5.5 MB.

Phase	Description
Setup	Construct a MongoClient object. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of 10 numbered collection names (`corpus_1`, `corpus_2`, ...) Create a list of write models for 10,000 document copies. For each document (where i goes from 0 to 9,999), create three write models with the namespace set to the collection name found at index `i % 10` in your collection names list: 1. An insert model. 2. A replace model (using an empty filter and a copy of the document). 3. A delete model (using an empty filter). Notes: - DO NOT manually add an `_id` field; leave it to the driver or database. You should have 30,000 write models in your list.
Before task	Drop the `perftest` database, create it again and create the collections found in your numbered collection names list.
Do task	Do an ordered `Client::bulkWrite` with the list of write models.
After task	n/a
Teardown	Drop the `perftest` database.

Small doc Collection BulkWrite Mixed Operations

Summary: This benchmark tests driver performance of a `Collection::bulkWrite` operation with small documents and mixed operations (e.g. insert, replace and delete).

Dataset: The dataset, designated SMALL_DOC consists of a JSON document with an encoded length of approximately 250 bytes.

Dataset size: For score purposes, the dataset size for a task is the size of the single-document source file (275 bytes) times 20,000 operations (insert + replace), which equals 5,500,000 bytes or 5.5 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Load the SMALL_DOC dataset into memory as a language-appropriate document type (or JSON string for C). Create a list of write models for 10,000 copies of the document where for each copy of the document add the following models: 1. insert model. 2. replace model with a copy and using an empty filter. 3. delete model using empty filter. DO NOT manually add an `_id` field; leave it to the driver or database. You should have 30,000 write models in your list.

Phase	Description
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command. Construct a Collection object for the `corpus` collection to use for insertion.
Do task	Do an ordered `Collection::bulkWrite` with the list of write models.
After task	n/a
Teardown	Drop the `perftest` database.

GridFS upload

Summary: This benchmark tests driver performance uploading a GridFS file from memory.

Dataset: The dataset, designated GRIDFS_LARGE (disk file `gridfs_large.bin`), consists of a single file containing about 50 MB of random data. We use a large file to ensure multiple database round-trips even if chunks are are sent in batches.

Dataset size: For score purposes, the dataset size for a task is the size of the source file (52,428,800 bytes) times 1 operation or 52.43 MB.

Phase	Description
Setup	Construct a MongoClient object. rop the `perftest` database. Load the GRIDFS_LARGE file as a string or other language-appropriate type for binary octet data.
Before task	Drop the default GridFS bucket. Insert a 1-byte file into the bucket. (This ensures the bucket collections and indices have been created.) Construct a GridFSBucket object to use for uploads.
Do task	Upload the GRIDFS_LARGE data as a GridFS file. Use whatever upload API is most natural for each language (e.g. open_upload_stream(), write the data to the stream and close the stream).
After task	n/a
Teardown	Drop the `perftest` database.

GridFS download

Summary: This benchmark tests driver performance downloading a GridFS file to memory.

Dataset: The dataset, designated GRIDFS_LARGE, consists of a single file containing about 50 MB of random data. We use a large file to ensure multiple database round-trips even if chunks are are sent in batches.

Dataset size: For score purposes, the dataset size for a task is the size of the source file (52,428,800 bytes) times 1 operation or 52.43 MB.

Phase	Description

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Upload the GRIDFS_LARGE file to the default gridFS bucket with the name "gridfstest". Record the `_id` of the uploaded file.
Before task	Construct a GridFSBucket object to use for downloads.
Do task	Download the "gridfstest" file by its `_id`. Use whatever download API is most natural for each language (e.g. open_download_stream(), read from the stream into a variable). Discard the downloaded data.
After task	n/a
Teardown	Drop the `perftest` database.

Parallel

Datasets are in the `parallel` tarball.

Parallel tests simulate ETL operations from disk to database or vice-versa. They are designed to be implemented using a language's preferred approach to concurrency and thus stress how drivers handle concurrency. These intentionally involve overhead above and beyond the driver itself to simulate -- however loosely -- the sort of "real-world" pressures that a drivers would be under during concurrent operation.

They are intended for directional indication of which languages perform best for this sort of pseudo-real-world activity, but are not intended to represent real-world performance claims.

Drivers teams are expected to treat these as a competitive "shoot-out" to surface optimal ETL patterns for each language (e.g. multi-thread, multi-process, asynchronous I/O, etc.).

Parallel micro-benchmarks include:

- LDJSON multi-file import
- LDJSON multi-file export
- GridFS multi-file upload
- GridFS multi-file download

LDJSON multi-file import

Summary: This benchmark tests driver performance importing documents from a set of LDJSON files.

Dataset: The dataset, designated LDJSON_MULTI (disk directory `ldjson_multi`), consists of 100 LDJSON files, each containing 5,000 JSON documents. Each document should be about 1000 bytes.

Dataset size: For score purposes, the dataset size for a task is the total size of all source files: 565,000,000 bytes or 565 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database.

Phase	Description
Before task	Drop the `corpus` collection. Create an empty `corpus` collection with the `create` command.
Do task	Do an unordered insert of all 500,000 documents in the dataset into the `corpus` collection as fast as possible. Data must be loaded from disk during this phase. Concurrency is encouraged.
After task	n/a
Teardown	Drop the `perftest` database.

LDJSON multi-file export

Summary: This benchmark tests driver performance exporting documents to a set of LDJSON files.

Dataset: The dataset, designated LDJSON_MULTI, consists of 100 LDJSON files, each containing 5,000 JSON documents. Each document should be about 1000 bytes.

Dataset size: For score purposes, the dataset size for a task is the total size of all source files: 565,000,000 bytes or 565 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Drop the `corpus` collection. Do an unordered insert of all 500,000 documents in the dataset into the `corpus` collection.
Before task	Construct whatever objects, threads, etc. are required for exporting the dataset.
Do task	Dump all 500,000 documents in the dataset into 100 LDJSON files of 5,000 documents each as fast as possible. Data must be completely written/flushed to disk during this phase. Concurrency is encouraged. The order and distribution of documents across files does not need to match the original LDJSON_MULTI files.
After task	n/a
Teardown	Drop the `perftest` database.

GridFS multi-file upload

Summary: This benchmark tests driver performance uploading files from disk to GridFS.

Dataset: The dataset, designated GRIDFS_MULTI (disk directory `gridfs_multi`), consists of 50 files, each of 5MB. This file size corresponds roughly to the output of a (slightly dated) digital camera. Thus the task approximates uploading 50 "photos".

Dataset size: For score purposes, the dataset size for a task is the total size of all source files: 262,144,000 bytes or 262.144 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database.
Before task	Drop the default GridFS bucket in the `perftest` database. Construct a GridFSBucket object for the default bucket in `perftest` to use for uploads. Insert a 1-byte file into the bucket (to initialize indexes).
Do task	Upload all 50 files in the GRIDFS_MULTI dataset (reading each from disk). Concurrency is encouraged.
After task	n/a
Teardown	Drop the `perftest` database.

GridFS multi-file download

Summary: This benchmark tests driver performance downloading files from GridFS to disk.

Dataset: The dataset, designated GRIDFS_MULTI, consists of 50 files, each of 5MB. This file size corresponds roughly to the output of a (slightly dated) digital camera. Thus the task approximates downloading 50 "photos".

Dataset size: For score purposes, the dataset size for a task is the total size of all source files: 262,144,000 bytes or 262.144 MB.

Phase	Description
Setup	Construct a MongoClient object. Drop the `perftest` database. Construct a temporary directory for holding downloads. Drop the default GridFS bucket in the `perftest` database. Upload the 50 file dataset to the default GridFS bucket in `perftest`.
Before task	Delete all files in the temporary folder for downloads. Construct a GridFSBucket object to use for downloads from the default bucket in `perftest`.
Do task	Download all 50 files in the GRIDFS_MULTI dataset, saving each to a file in the temporary folder for downloads. Data must be completely written/flushed to disk during this phase. Concurrency is encouraged.
After task	n/a
Teardown	Drop the `perftest` database.

Composite score calculation

Every micro-benchmark has a score equal to the 50th percentile (median) of sampled timings, expressed as Megabytes (1,000,000 bytes) per second where the micro-benchmark "database size" given in each section above is divided by the 50th percentile of measured wall-clock times.

From these micro-benchmarks, the following composite scores must be calculated:

Composite Name	Compositing formula
BSONBench	Average of all BSON micro-benchmarks
SingleBench	Average of all Single-doc micro-benchmarks, except "Run Command"
MultiBench	Average of all Multi-doc micro-benchmarks
ParallelBench	Average of all Parallel micro-benchmarks
ReadBench	Average of "Find one", "Find many and empty cursor", "GridFS download", "LDJSON multi-file export", and "GridFS multi-file download" microbenchmarks
WriteBench	Average of "Small doc insertOne", "Large doc insertOne", "Small doc bulk insert", "Large doc bulk insert", "Small doc Collection BulkWrite insert", "Large doc Collection BulkWrite insert", "Small doc Client BulkWrite insert", "Large doc Client BulkWrite insert", "Small doc Client BulkWrite Mixed Operations", "Small doc Collection BulkWrite Mixed Operations", "GridFS upload", "LDJSON multi-file import", and "GridFS multi-file upload" micro-benchmarks
DriverBench	Average of ReadBench and WriteBench

At least for this first DriverBench version, scores are combined with simple averages. In addition, the BSONBench scores do not factor into the overall DriverBench scores, as encoding and decoding are inherent in all other tasks.

Benchmark platform, configuration and environments

Benchmark Client

TBD: spec Amazon instance size; describe in general terms how language clients will be run independently; same AWS zone as server

All operations must be run with write concern "w:1".

Benchmark Server

TBD: spec Amazon instance size; describe configuration (e.g. no auth, journal, pre-alloc sizes?, WT with compression to minimize disk I/O impact?); same AWS zone as client

Score Server

TBD: spec system to hold scores over time

Datasets

TBD: generated datasets should be park in S3 or somewhere for retrieval by URL

Changelog

- 2024-12-23: Add Client and Collection BulkWrite benchmarks

- 2024-01-22: Migrated from reStructuredText to Markdown.
- 2022-10-05: Remove spec front matter and reformat changelog.
- 2021-04-06: Update run command test to use `hello` command
- 2016-08-13:
  - Update corpus files to allow much greater compression of data
  - Updated LDJSON corpus size to reflect revisions to the test data
  - Published data files on GitHub and updated instructions on how to find datasets
  - RunCommand and query benchmark can create collection objects during setup rather than before task. (No change on actual benchmark.)
- 2016-01-06:
  - Clarify that `bulk insert` means `insert_many`
  - Clarify that "create a collection" means using the `create` command
  - Add omitted "upload files" step to setup for GridFS multi-file download; also clarify that steps should be using the default bucket in the `perftest` database
- 2015-12-23:
  - Rename benchmark names away from MMA/weight class names
  - Split BSON encoding and decoding micro-benchmarks
  - Rename BSON micro-benchmarks to better match dataset names
  - Move "Run Command" micro-benchmark out of composite
  - Reduced amount of data held in memory and sent to/from the server to decrease memory pressure and increase number of iterations in a reasonable time (e.g. file sizes and number of documents in certain datasets changed)
  - Create empty collections/indexes during the `before` phase when appropriate
  - Updated data set sizes to account for changes in the source file structure/size