MongoDB Internals

- BSON
- WiredTiger
- Journaling

BSON Overview

- BSON is Binary JavaScript Object Notation
- Allows serialization of data more efficiently than JSON
- Contains type and length information for efficient machine parsing
- Used for transmitting and storing data across web-based applications
- Provides flexibility and support for various data types and programming languages
- Contains metadata to facilitate parsing in scenarios where data structure is not predefined

BSON Types with Metadata (Simple Primitives)

BSON Type	Metadata (Hex)	Explanation
Int32	0x10	32-bit signed integer
Int64	0x12	64-bit signed integer
Double	0x01	Double-precision floating-point number
Decimal128	TBD	Decimal floating-point number
String	0x02	UTF-8 string
Object	0x03	Embedded document
Array	0x04	List or array
Binary	0x05	Binary data
Boolean	0x08	Boolean value
Date	0x09	UTC datetime
Timestamp	0x11	Special BSON timestamp

BSON Types with Metadata (Extended Primitives)

BSON Type	Metadata (Hex)	Explanation
RegEx	0x0B	Regular expression
JavaScript	0x0D	JavaScript code
Symbol	0x0E	Symbol
MinKey	0xFF	Minimum key value
MaxKey	0x7F	Maximum key value

BSON Example

```
1 \x16\x00\x00\x00                  // total document size
2 \x02                              // 0x02 = type String
3 hello\x00                         // field name
4 \x06\x00\x00\x00world\x00         // field value
5 \x00                              // 0x00 = type EOO ('end of
      object')
```

BSON: More Compact Numbers

```
db.numbers.insertMany([
    {a: 1, b:1.5, c: BigInt("12345679012345678")},
    {x: Decimal128("1234567890123456789O.12345e+1000")},
    {a: 2, b:3.5, c: BigInt("12345679012345678")},
    {x: Decimal128("91234567890123456789.12345e+500")}
])
```

Binary Dump of BSON

```
33 00 00 00 07 5f 69 64 00 66 15 b3 33 55 8c 65 3....._id.f..3U.e
a8 a8 7b 2d a9 10 61 00 01 00 00 00 01 62 00 00 ..{-..a......b..
00 00 00 00 00 f8 3f 12 63 00 4e b7 0a 64 54 dc ......?.c.N..dT.
2b 00 00 29 00 00 00 07 5f 69 64 00 66 15 b3 33 +..)....._id.f..3
55 8c 65 a8 a8 7b 2d aa 13 78 00 79 df e2 3d 44 U.e..{-..x.y..=D
a6 36 0f 6e 05 01 00 00 00 06 38 00 33 00 00 00 .6.n......8.3...
07 5f 69 64 00 66 15 b3 33 55 8c 65 a8 a8 7b 2d ._id.f..3U.e..{-
ab 10 61 00 02 00 00 00 01 62 00 00 00 00 00 00 ..a......b......
00 0c 40 12 63 00 4e b7 0a 64 54 dc 2b 00 00 29 ..@.c.N..dT.+..)
00 00 00 07 5f 69 64 00 66 15 b3 33 55 8c 65 a8 ...._id.f..3U.e.
a8 7b 2d ac 13 78 00 59 db 96 15 fb 44 64 95 f7 .{-..x.Y....Dd..
8b 07 00 00 00 1e 34 00                         ......4.
```

Importing and Exporting BSON Files in MongoDB

Command	Description
`mongodump collection.bson`	Dump binary BSON to a file
`mongodump -outFile=collection.json collection.bson`	Dump a collection to a JSON file
`mongorestore -d db_name /path/file.bson`	Import a BSON file to a database
`mongorestore -drop -d db_name -c collection_name /path/file.bson`	Drop collection and import from BSON
`mongorestore -d db_name /path/`	Restore a folder exported by `mongodump` to a database
`bsondump collection.bson`	Dump text of BSON file to console
`bsondump -outFile=collection.json collection.bson`	Convert JSON file to BSON

Comparing: text vs. BSON

- BSON: 184 bytes
- text: 240 bytes
- This is not a huge improvement, why?
- Metadata

Efficient Binary Transport of Data

- To efficiently send data, metadata must be defined once
- This requires that each row has the same structure
- SQL has the same structure, but still transports data as text!
- Most web applications use JSON
- Let's examine the overhead of text vs. binary transport

Client/Server Communication

Overhead of XML and JSON

XML is 113 bytes (could be streamlined)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <point>
3   <x>123.45678</x>
4   <y>123.45678</y>
5   <z>123.45678</z>
6 </point>
```

JSON is 63 bytes

```
1 {
2     "x": 123.45678,
3     "y": 123.45678,
4     "z": 123.45678
5 }
```

Overhead of ASCII vs Binary

- Text has no metadata (cryptic but 28 bytes)
- Binary is the most cryptic but only 12 bytes

```
123.45678,123.45678,123.45678
```

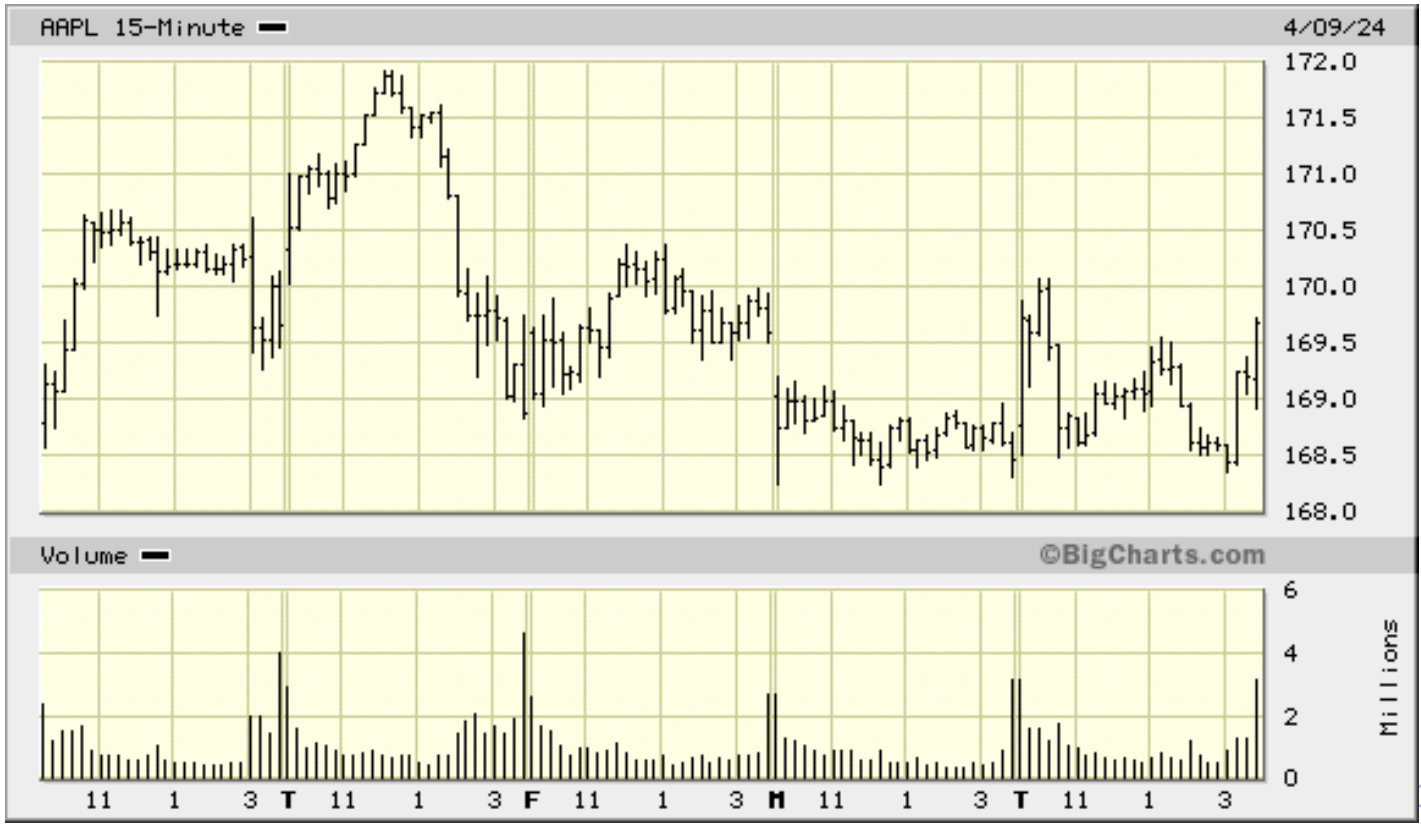
Key: Metadata only once, Regular Structure

- Consider a stock quotes example
- Each day has high, low, open, close and volume
- Each row has the same structure – so why repeat metadata?
- Even this, ASCII has significant overhead
- Binary data is fixed in size
- This means for binary we can jump to any desired record
- For text, we would need to maintain an index

```
Date,Open,High,Low,Close,Volume,OpenInt
1984-09-07,0.42388,0.42902,0.41874,0.42388,23220030
1984-09-10,0.42388,0.42516,0.41366,0.42134,18022532
1984-09-11,0.42516,0.43668,0.42516,0.42902,42498199
1984-10-10,0.39316,0.39316,0.38164,0.38164,101750813
2017-11-01,169.26,169.33,165.02,166.29,33758685
```

Compression of Highly Regular Data

- Dates are highly predictable, skipping weekends and occasional holidays
- Every date is the same, why send it for multiple stocks?
- Prices are correlated $high > low$, usually all are very similar
- There are days when the stock price swings wildly though
- Compression must be efficient or not worth it



Random Access or Sequential?

- If sending single values, then compression is only reducing resolution
- How accurate are prices? How many digits needed?
- Unfortunately, there is wide variation
- AAPL: 176.52 (5 significant digits)
- BRK.A $626,900.00$ (10 significant digits)
- A database must handle **all** cases, not just the average case

Single Numbers: Binary Fallback

- Two record types: average, and extreme case
- Average: encode as 16-bit integer (4.5 digits $0..65535$)
- Extreme: encode as 32-bit float (8 digits, live with rounding errors?)
- Volume is an interesting example: we really don't need full accuracy
- Price? People want to know exactly...
- 1 byte of overhead, and the average case is shorter (but requires work!)

```
type
01   [hi=16-bit] [low=16-bit] [[open=16-bit] [close=16-bit] [
02   [hi=32-bit] [low=32-bit] [[open=32-bit] [close=32-bit] [
```

Sending a Time Sequence

- The usual use-case is sending a time series of prices
- Everyone wants to know price for
- 1, 2, 5 years, 1, 2, 3, 6 months
- In this case, compression is highly advantageous
- Compress once, keep data set, give it to everyone
- First: delta encode (numbers get much smaller)
- More zero bits in the numbers
- Compress using LZMA (Lempel-Ziv-Markov algorithm)

LZMA

Compressing a Read-Mostly Dataset

- Efficient because read-mostly
- Compression can be done in advance
- AAPL stock quotes from start to 2017: 8397 records

Case	Size
CSV	428.3kB
Strip Date	319.5kB
Strip date, LZMA	97.2kB
binary lossless	201.5kB
binary lossless LZMA	90.9kB
delta encoded bin	201.5kB
delta encoded LZMA	???6.kB

The final compressed delta can be astonishingly good

Variable-Length Data is Trouble

- Variable length data can be very large or very short
- Strings
- Lists
- This makes it hard to index records
- One way is to have the fixed size in one chunk
- Pointers to the variable size

For Constant Data, Turn String into a Hash Value

- Each string can be hashed to a unique number
- Numbers can be in sorted order and sorted
- Problem: Every time the numbers change, data everywhere must be rebuilt

WiredTiger

- Modern storage engine for MongoDB.
- Developed by WiredTiger, Inc. (acquired by MongoDB, Inc. in 2014).
- Designed for high performance, scalability, and efficiency.
- Features document-level concurrency control.
- Supports compression and encryption.
- Implements a B-tree based storage architecture.
- Efficiently handles both read and write operations.
- Can handle databases ranging from gigabytes to terabytes in size.
- Suitable for a wide range of workloads, including transactional and analytical applications.

WiredTiger Docs WiredTiger Source

WiredTiger Format

- Low latency and high throughput (in-cache reads no, writes 1 latch)
- Handles data sets much larger than RAM without performance or resource degradation,
- Predictable behavior under heavy access and large volumes of data
- Transactional semantics without blocking (but distributed failures possible?)
- Stores are not corrupted by torn writes, reverting to the last checkpoint after system failure,
- Supports both checkpoint-level and commit-level durability
- Additional timestamp mode (fine-grained control over durability
- Petabyte tables
- records up to 4GB
- Record numbers up to 64-bits.

Conclusion

- The document model allows different structures than SQL tables, but at a cost in performance
- Both SQL and MongoDB suffer from inefficient data transmission