Object serialization vs relational data modelling in Apache Cassandra: a performance evaluation

Valdemar Johansen

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Computer Science. The thesis is equivalent to ten weeks of full time studies.

Contact Information:
Author(s):
Valdemar Johansen
E-mail: valdemarjohansen@gmail.com

External advisor:
Jim Håkansson, Ericsson Karlskrona

University advisor:
Professor Håkan Grahn
Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
ABSTRACT

Context. In newer database solutions designed for large-scale, cloud-based services, database performance is of particular concern as these services face scalability challenges due to I/O bottlenecks. These issues can be alleviated through various data model optimizations that reduce I/O loads. Object serialization is one such approach.

Objectives. This study investigates the performance of serialization using the Apache Avro library in the Cassandra database. Two different serialized data models are compared with a traditional relational database model.

Methods. This study uses an experimental approach that compares read and write latency using Twitter data in JSON format.

Results. Avro serialization is found to improve performance. However, the extent of the performance benefit is found to be highly dependent on the serialization granularity defined by the data model.

Conclusions. The study concludes that developers seeking to improve database throughput in Cassandra through serialization should prioritize data model optimization as serialization by itself will not outperform relational modelling in all use cases. The study also recommends that further work is done to investigate additional use cases, as there are potential performance issues with serialization that are not covered in this study.

Keywords: Distributed systems organizing principles, information storage technologies, data structures and algorithms for data management
# CONTENTS

ABSTRACT .................................................................................................................... 1

CONTENTS ..................................................................................................................... 2

1 INTRODUCTION ....................................................................................................... 3

2 SCOPE AND RESEARCH QUESTIONS ....................................................................... 4
  2.1 AIM AND SCOPE .................................................................................................. 4
  2.2 RESEARCH QUESTIONS .................................................................................... 4
  2.3 LIMITATIONS ................................................................................................... 4

3 BACKGROUND .......................................................................................................... 5
  3.1 APACHE CASSANDRA ...................................................................................... 5
  3.2 SERIALIZATION .................................................................................................. 5
  3.2.1 General ......................................................................................................... 5
  3.2.2 Apache Avro .................................................................................................. 6

4 RELATED WORK ....................................................................................................... 7

5 METHOD ...................................................................................................................... 9
  5.1 OVERVIEW ......................................................................................................... 9
  5.2 TEST DATA ........................................................................................................ 9
  5.3 DATA MODELLING ............................................................................................ 10
    5.3.1 CQL data modelling ................................................................................... 10
    5.3.2 BLOB data modelling ................................................................................ 11
  5.4 WRITE OPERATIONS ........................................................................................ 12
    5.4.1 Definition ..................................................................................................... 12
    5.4.2 Writing CQL ................................................................................................ 13
    5.4.3 Writing BLOBs ............................................................................................ 13
  5.5 READ OPERATIONS ........................................................................................... 13
  5.6 EXPERIMENTS .................................................................................................... 13
    5.6.1 Read/write instructions ............................................................................... 13
    5.6.2 Experiment threading ................................................................................ 14
    5.6.3 Metrics collection ....................................................................................... 14
    5.6.4 Test environment ....................................................................................... 14
    5.6.5 Workloads .................................................................................................. 15
    5.6.6 Metrics ........................................................................................................ 17

6 RESULTS AND ANALYSIS ....................................................................................... 18
  6.1 WRITE OPERATIONS .......................................................................................... 18
    6.1.1 Overview ...................................................................................................... 18
    6.1.2 Latency distribution .................................................................................... 19
    6.1.3 Latency/operation complexity breakdown ............................................... 20
  6.2 READ OPERATIONS ............................................................................................ 22
    6.2.1 Overview ...................................................................................................... 22
    6.2.2 Latency distribution .................................................................................... 23
    6.2.3 Latency/operation complexity breakdown ............................................... 23

7 CONCLUSIONS AND FUTURE WORK ................................................................. 26

8 REFERENCES ............................................................................................................. 28

APPENDICES .................................................................................................................. 29
  8.1 A. TEST DATA ..................................................................................................... 29
    8.1.1 A.1 Small object .......................................................................................... 29
    8.1.2 A.2 Large object ......................................................................................... 31
1 INTRODUCTION

The current information landscape is characterized by large data volumes and high data access rates [2]. While client applications are steadily proliferating and improving their processing power, improvements in I/O performance are not keeping pace with this trend [14], giving rise to scalability issues that traditional RDBMS (relational database management systems) were not designed to handle. To address these challenges, so-called NoSQL (‘Not Only Structured Query Language’) DBMS (database management systems) have been developed that sacrifice some features prevalent in RDBMS solutions—such as join operations, constraints, and transactions—in order that improved scalability, faster performance and higher availability may be attained [3][10]. These compromises have not, however, eliminated the need for the functionality offered by RDBMS solutions but have instead created demand for systems that replace I/O intensive, throughput-reducing composite queries and intricate table schemata with client-side approaches that can support complex data structures.

One approach for achieving this is to serialize data prior to database statement execution and handling relational logic at the application layer, thereby replacing some of the server-side I/O loads with client-side CPU loads. This paper evaluates an alternative storage structure which achieves high performance in the Apache Cassandra DBMS by storing data in serialized form as BLOBs (‘Binary Large Objects’) while still supporting well-defined data structures through use of the Apache Avro serialization library [1]. Apache Cassandra is a linearly scalable, no single point of failure, low latency NoSQL database [6]. Among NoSQL solutions, Apache Cassandra is one of the top performing DBMSs and the “clear winner” in terms of scalability [12]. The Apache Avro serialization library features support for complex nested data structures, version control, backwards compatibility and interoperability between application environments. Avro also enables developers to define multiple read and write schemata for the same object types, allowing parallel representations of the same data to coexist at the application level, analogous to views in RDBMS systems, salvaging some of the functionality lost in many NoSQL implementations [4][14].

While data model simplification can significantly improve database I/O performance [13] it also imposes the limitation that the entire object must be deserialized in order to access or modify a single column. Hence not all applications will benefit from serialization. It is therefore of interest to compare the performance of unserialized relational data modelling with serialization strategies. This paper compares the read and write performance in Cassandra of unserialized SQL-style relational modelling with database models that implement Avro serialization.
2 SCOPE AND RESEARCH QUESTIONS

2.1 Aim and scope

This paper compares different data modelling strategies in the Apache Cassandra DBMS. The results presented here should not be understood to apply to traditional RDBMS as these are optimized for different use cases than NoSQL DBMS. The Cassandra Query Language is not a fully-fledged RDBMS query language as it does not feature the type of nested select queries that are common to RDBMS. Similar performance measures to the ones presented here may be observed in other NoSQL DBMS configurations but these are not the subject of this investigation.

2.2 Research questions

RQ1: How do read and write operations in Cassandra perform when storing objects in relational tables compared to using Apache Avro serialization?
RQ2: How does serialization granularity affect latency?
RQ3: How does object complexity relate to read/write latency?
RQ4: How significant is the client-side latency for Avro serialization?

2.3 Limitations

The software developed for the experiments conducted in this study was designed to discern the overall performance difference in basic use cases (pure read and pure write workloads). Mixed read/write workloads are not considered in this study, but only the extremes of pure read and pure write workloads. Further improvements and optimizations could be made to the methods herein described, such as batch inserts, index optimization, threading and Cassandra cache configuration.

The paper does not compare the more advanced functional aspects of the Avro library but strictly investigates the performance of its serialization utility. Two levels of serialization granularity are investigated in order to possibly derive useful principles for performance considerations but this investigation does not attempt to discover optimal granularity for specific use cases.
3 Background

3.1 Apache Cassandra

Apache Cassandra [11] is a distributed storage system designed to handle large data volumes spread across multiple servers. A Cassandra database is usually referred to as a cluster consisting of nodes where each node is a Cassandra process instance, typically running on a separate machine.

Cassandra’s durable, distributed design relies on the use of data replication, meaning multiple copies of the same data will exist on different nodes in the cluster. For instance, with a replication factor of three, three copies of the data will be distributed across three different nodes in the cluster.

Unlike most RDBMS systems, as well as most NoSQL solutions, Cassandra is not built on a master/slave hierarchy but is instead designed to have no single point of failure; all nodes in a Cassandra cluster are equal. This decentralized hierarchy combined with data replication allows the system to stay up when nodes go down, a frequent occurrence in large clusters consisting of hundreds of nodes. For instance, if the replication factor is set to three, only two nodes need to be available for a read operation to be successful, increasing the system’s availability.

Cassandra is designed to be elastically scalable, meaning database throughput increases linearly as more nodes are added to the cluster. Cassandra achieves low write latency by having all write operations be sequential on disk (appended), reducing movement of the disk read-and-write-head.

3.2 Serialization

3.2.1 General

There are several issues with conventional SQL-style data modelling that make it a poor strategy for NoSQL approaches. In the SQL data definition model (used by most RDBMS systems) data is stored in tables made up by rows and columns. Relationships between columns in different tables may be defined, as well as more complex functionality such as stored procedures. The more complex the design, the more I/O intensive the query is likely to be as different tables are stored in physically different locations on disk.

SQL-style data modelling can produce maintenance problems, since changes in the data structure require reciprocal changes in the table definition. In cloud-based applications, with enormous data volumes such a rigid design can become unmaintainable, as millions of concurrent users try to access a service using a broad array of devices running different software that would all need to be updated to access the new format.

Data serialization aims to address both the performance and maintainability issues by simplifying the data model. The serialization approach essentially treats the database as a basic file storage system where data structures are stored in binary format in single
columns and uniquely identified by id columns. This way, queries can be simplified by reading fewer columns, reducing I/O loads. In combination with a feature-rich serialization library, logic that in an SQL-style system would be handled in the database itself can be decentralized to object serialization schemata and client-side application logic.

3.2.2 Apache Avro

Apache Avro [1] is a serialization library with full APIs support for many languages (including Java, C, C++, C#, Python). Avro allows users to define data definition structures to be used in serialization. The structures are called schemas and are expressed in JSON (JavaScript Object Notation). These schemas are then used to serialize and deserialize the data. Serialization converts the structure into binary form and deserialization using the schema converts the object back into readable form. Unlike similar systems, like Thrift and Protocol Buffers, Avro does not require a code generation step.

The use of JSON as the definition language facilitates data exchange between many different languages. Avro supports evolving schemas: it can handle dropped fields, added fields and changed fields. This enables older applications to read data serialized in new schema versions and new applications to read data serialized in old schema versions.
4 RELATED WORK

Previous work has identified challenges associated with supporting complex data structures in NoSQL systems while maintaining high performance [6][7][9][16]. I/O overhead is invariably identified as the main performance bottleneck but few studies have thus far analyzed cohesive solutions to the problem.

Zou et al [14] investigate the use of data compression after in-situ analysis to reduce transfer overhead in large-scale scientific data analytics applications, showing experimentally that end-to-end transfer performance can be improved with such an approach, but also observe that the optimal choice of compression method (in terms of performance) is highly situational. The authors therefore advocate “a latency based quantitative model” to dynamically determine whether compression should be used for a given input and, if so, what compression algorithm would offer the best performance.

Raichand and Rani [13] compare performance of compressed and uncompressed tables in the Hadoop database. They analyze three compression codecs—Snappy, LZO and GZIP—and show that compression greatly improves query execution performance over relational modelling, though without addressing the issue of query complexity.

Lombardo et al [9] propose a method of achieving the expressiveness of RDBMS interfaces in NoSQL applications by storing nested structures in XML format in single columns. Though this effectively simplifies the table structure while preserving object complexity, complex queries (e.g. nested selects) would require excessive XML parsing for key extraction with a negative performance impact on query execution. To address this concern, the authors propose generating additional tables with (overlapping) index data to mitigate the performance penalty incurred by XML parsing, though they provide no experimental data.

Klein et al. [8] use a test client based on the YCSB (Yahoo Cloud Service Benchmarking) framework to compare the performance of nested data structures in three NoSQL DBMS solutions (Cassandra, MongoDB and Risk) using a dataset of one million patient records associated—in a one-to-many relationship—with patient test result records. They find that Cassandra outperforms MongoDB and Risk in both read, write and mixed workloads.

Rabl, Tilmann, et al. [12] find that among NoSQL solutions, Apache Cassandra is the most scalable DBMS, though this comes at the expense of relatively high read and write latencies.

Maeda [10] compares the performance of twelve serialization libraries: Avro, Flex, Gson, Jackson, Jsonic, JsonLib, JsonMarshaller, JsonSmart, Java Serializable, Protobuff, ProtoStuff, Thrift and XStream. The study finds that out of the analyzed libraries binary serialization in Avro was the most compact in terms of file size, and among the best performing in both the serialization and deserialization phases.

This paper differs from related work in three main ways:

1. It compares unserialized data modelling (relational tables) with two levels of table serialization (fully and partially serialized).
2. It investigates the convergent performance of two of the best performing and feature-rich NoSQL tools—Apache Avro and Apache Cassandra—which has thus far not been explored by previous research.

3. It uses real application data of different levels of complexity to investigate variations in query execution performance over a range of object complexity levels for three different data models.
5 METHOD

5.1 Overview

This paper compares read and write performance in Cassandra using three different data modelling methods:

- The CQL data model employs a traditional, unserialized, relational data model where the table definitions in the database directly mirror the structure of the input data.
- The BLOB-1 data model serializes the input data to a byte array and stores it in a single column along with a primary key column.
- BLOB-4 represents an intermediate case. It stores the byte array in the same way as in the BLOB-1 method, but divides the input data into four classes which are serialized separately and stored in different tables.

The experiments consist of reading and writing a large number of objects to a Cassandra cluster. The performance of the three data models are then logged and analyzed.

5.2 Test data

Twitter data was chosen as test data as it has a nested data structure (appropriate for serialization), is easily obtainable, and represents a real use case. The test data was collected by accessing the public Twitter sample stream via the twitter4j API. The API exposes user status updates (tweets) in the form of twitter4j.Status objects. These objects contain data directly pertaining to the tweet itself—its text content, its timestamp, its unique id, etc—but may also contain data not directly pertinent to the status update, such as a copy of the user’s account data, containing details like the number of followers the user has, the user’s profile image url, etc. If the tweet is a retweet, the status object will contain a complete duplicate copy of the parent tweet including its entire nested structure (all sub-objects). A retweet is therefore on average twice the size of a regular tweet.

Status objects can contain objects and arrays. In JSON notation, objects are structures contained within ‘{}’ braces and arrays are structures contained within ‘[]’ brackets (see figure 1). Objects may in turn contain arrays of nested objects. The complexity of a status object can thus vary significantly, and may range in size from two objects to over ninety. This investigation found that the average object complexity was 8.5 objects per status record. This figure may change in the future as more users add location information to their account. The least complex status objects contained about 90 member variables while more deeply nested objects could exceed 400 members. (See Appendix A for complete listings of object examples in the form analyzed here.)

The twitter4j Status objects were converted into JSON format using the codehaus.jackson library and written to file so that the same dataset could be reused in all experiments. The Jackson library [5] was used for the conversion because the Twitter API enforces rate limits on getAsRawJson() invocations for Status objects. Figure 1 shows a simplified version of a Status object:
Figure 1. Simplified example of a twitter4j.Status object in JSON format, showing its nested structure

5.3 Data modelling

On analyzing the test data, it was found that the ‘place’ objects were present in fewer than 2% of the sampled tweets. Objects of the ‘symbolEntities’ and ‘contributors’ classes were even rarer. It was decided not to include these objects in the data experiments as they could skew the results making them more difficult to interpret. The self-referencing composition arrow for the Status objects present in all three data models below (figures 2-4) signifies that a Status object can be related to another object of the same type. Such a relationship represents a retweet. Retweets are written as separate Status objects.

5.3.1 CQL data modelling

In the CQL data model, column families were defined for each object. To avoid key collisions, primary keys were generated for complex structures (objects consisting of more than one member) that lacked unique keys. Figure 2 represents the data model for the CQL data model.
5.3.2 BLOB data modelling

Two modelling approaches were implemented for Avro-serialization so that the performance of different composition strategies and serialization granularities could be compared and contrasted.

5.3.2.1 BLOB-4

The BLOB-4 strategy splits the twitter4j.Status object into four entities, as shown in figure 3. All objects possessing unique ids were serialized and written as separate Avro objects in their own tables. These were: user, userMentionEntities, mediaEntities, and extendedMediaEntities (see appendix A for sample objects). Nested objects that did not possess a unique id in the JSON object (urlentity, hashtagentities, size, sizes) were nested within their parents and serialized together as a single object.

‘Status’ BLOBs were thus serialized with the following objects present in the JSON objects: usermentionentities, hashtagentities and urlentity. ‘User’ BLOBs were serialized...
with urlentity, and ‘MediaEntity’ and ‘ExtendedMediaEntity’ BLOBs were serialized with size objects arrays. See Appendix A1 and A2 for examples.

5.3.2.2 BLOB-1
The BLOB-1 strategy (Figure 3) stores the entire Status structure in one serialized object which is stored in a single column. Its data model is thus very simple:

![Figure 4. The BLOB-1 data model.](image)

Retweets are the only type of objects that can produce two rows in the BLOB1 model. In the test data 32.1% of the objects were found to be retweets.

Table 1 shows the table output for a BLOB-1 or BLOB-4 table when viewed in Cassandra’s cqlsh CLI:

**Table 1. Example of table output for BLOB-1 and BLOB-4**

<table>
<thead>
<tr>
<th>Key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>416501001743785984</td>
<td>0x3402b0c996b6e650027e4d792062657374206672 (...)</td>
</tr>
</tbody>
</table>

5.4 Write operations

5.4.1 Definition
CQL write operations are defined as the insertion of all parameters from all objects contained within the JSON representation of a twitter4j.Status object into their corresponding column families and columns (Figure 5).

BLOB write operations (Figure 5) are defined as the following procedure:

1. The construction of an object (generated from an Avro schema) for each object contained within the JSON representation of the twitter4j.Status object.
2. A key is then extracted from the JSON object to uniquely identify the row in the database.
3. The object is serialized to a byte array using the org.apache.avro.io.DatumWriter library.
4. The object is written to Cassandra.

![Figure 5. Steps in CQL (blue) and BLOB (green) write operations.](image)
Since the twitter4j.Status data structure contains a lot of duplicate values, duplicate objects could cause write collisions. This is particularly prevalent for the sizes array nested within MediaEntity objects. In the BLOB methods, these will simply be nested within the parent object’s BLOB.

5.4.2 Writing CQL
In the CQL method, prepared statements were defined for each table corresponding to objects within the JSON structure. Prepared statements were initialized using the parameters from the JSON object and executed. No statements were generated for sub-objects that were null in the JSON object.

5.4.3 Writing BLOBs
In the BLOB methods, parameters parsed from the JSON input were used to build Avro objects using the newBuild() invocation. This method was chosen rather than class constructors in order to allow null parameters. Objects were then serialized to byte arrays and written in string form to Cassandra in the format depicted in Table 1.

5.5 Read operations
Read operations were designed to resemble a typical use case of fetching an entire record (a Status object and all its sub-objects) from the database.

A CQL read operation is defined as a sequence of CQL SELECT (prepared) statements that fetch all rows associated with a status object’s id field and the id fields of its sub-structures. Since Cassandra does not support nested queries the objects were read via successive executions of (prepared) CQL SELECT statements defined for each respective table, e.g. "SELECT * FROM status WHERE id = ?;" followed by "SELECT * FROM user WHERE id = ?" using the id parameters extracted from previous calls in subsequent calls. A CQL read operation was considered complete when its return value was received.

A BLOB-4 read operation is a sequence of get statements that select the id of the status object, deserializes the returned object, extracts the id fields of its embedded objects (User, MediaEntity, ExtendedMediaEntity), fetches them via more get invocations and deserializes them. A BLOB-4 operation is considered complete when the last object in the chain has been deserialized. A BLOB-1 read operation simply fetches the entire object based on its id and deserializes it.

5.6 Experiments
Experiments consisted of write operations (5.4), read operations (5.5). Workloads (5.6.7) were defined as a list of 100 000 read or write instructions saved to a file. Each instruction resulted in the execution of either a read operation or a write operation.

5.6.1 Read/write instructions
Read instructions were represented by a twitter4j.Status object’s id field (a long integer in string form). These were used as primary keys for tweets in the database. A dequeued read instruction would result in a read operation.
Write instructions were represented as the JSON representation of the twitter4j.Status object (see appendix A for examples).

### 5.6.2 Experiment threading
A threaded implementation (Figure 6) was used to cache instructions read from file in a queue to ensure a constant flow of instruction with minimal I/O overhead and to allow large datasets without overflowing the process stack. In all experiments, only a single thread was dedicated for database statement executions in order to make the different implementations more comparable. The thread was never starved of content during the workload, but refilled whenever the queue size dipped below 25% of maximum capacity.

![Figure 6. Threaded test case execution.](image)

### 5.6.3 Metrics collection
Dequeued read operations executed a get() method call with a tweet id as its calling parameter. The return value was then processed and the primary key fields of the other tables associated with the status id were extracted (figures 2, 3 and 4). Additional get() calls were then executed for these fields to fetch the entire data structure (all tables). For each instruction, the number of get() calls and the duration of each get() method—$\Delta t$ before and after the method call—were logged to a csv file. Note that a BLOB-1 read operation could at most result in one recursive call for the retweeted ‘parent’ status object; the rest of the structure was entirely contained within the BLOB.

Dequeued write operations were parsed from string format to a JSON object and a put() method call was executed. The JSON object’s fields were processed to build the insert query. As with read operations, for each put() call, the number of calls and the duration of each call—$\Delta t$ before and after the method call—were logged to a csv file.

### 5.6.4 Test environment
The client ran on a standalone 32-core Intel Xeon machine (CPU E5-2690 0 @ 2.90GHz) with 64 GB RAM running Red Hat Enterprise Linux Server release 6.4 (Santiago) with Hugepagesize enabled at 2048 kB.

The Cassandra cluster ran in a 32 machine DRS cluster consisting of 32 8-core Intel Xeon 2.60 GHz Dell machines with round-robin path selection, each with 32 logical processors available via hyperthreading. Each Cassandra node was allotted 8 single core virtual sockets and 32GB of RAM.

The nodes test keyspaces were configured WITH replication = { 'class': 'NetworkTopologyStrategy', 'DC1': '3' } AND durable_writes = true. The test application was written in Java and used on demand JVM mapping. The average network latency was measured to 0.28 ms. The network bandwidth was measured to be in excess of 15 MB/s.
5.6.5 Workloads

A test data set of 100,000 objects (file size 530 MB) was used in all experiments, obtained from the public stream in May 2015. This size was chosen to be large enough to reduce the impact of anomalous system overheads while still being small enough to allow several runs to be made in the same time frame. Each workload was run five times and ranked according to total test duration. The median read and write cases for each data model were then selected for analysis.

Figure 8 shows the distribution of object complexity within the test dataset of 100,000 JSON objects.

The dataset contained mostly objects of simple complexity (i.e. composed of less than a dozen nested sub-objects) but there were a significant number of objects in the complexity range 27-39. To account for this uneven distribution, the object complexity level for each operation was logged in csv format along with their individual read/write
latencies. This enabled graphing the relationship between object complexity and total operational latency (see section 6, Results).

The workloads (Table 2) were inspired by the YCSB workload, as described in Klein et al [8], however the YCSB framework was not used in this investigation. Read operations were non-sequential, i.e. done in random order and not in the order the objects were written. 100% read and 100% write operations were chosen as opposed to mixed workloads in order to test the two extreme cases.

*Table 2. Workloads*

<table>
<thead>
<tr>
<th>Label</th>
<th>Operations</th>
<th>Application example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A— Insert</td>
<td>Insert: 100%</td>
<td>Uploading records used during a workday to the centralized data store.</td>
</tr>
<tr>
<td>B— Read</td>
<td>Read: 100%</td>
<td>Downloading (locally caching) records scheduled to be used that day from a centralized primary data store.</td>
</tr>
</tbody>
</table>
5.6.6 Metrics

The table below defines the metrics used in the experiments.

*Table 3. Metrics*

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server latency</td>
<td>Time from when the request is sent to the database until the response is received back from the database</td>
<td>Measuring the latency of get() and put() operations and deducting this time from the total operation time allows an indirect measure of the client-side latency (building prepared statements, serializing and deserializing).</td>
</tr>
<tr>
<td>Client-side latency</td>
<td>Total latency minus server latency</td>
<td>A measure of the latency overhead incurred in the test application developed for the experiments.</td>
</tr>
<tr>
<td>Latency distribution</td>
<td>95th percentile, the most extreme 5% of the values are ignored</td>
<td>A measure of performance variation.</td>
</tr>
<tr>
<td>Throughput</td>
<td>Operations per second; the number of complete status objects inserted to or read from Cassandra per second.</td>
<td>For large enough datasets the average operations per second provides a useful indicator of the performance difference between different data models.</td>
</tr>
<tr>
<td>Hits per second</td>
<td>The number of cache hits per second, as registered by the Cassandra process</td>
<td>Duplicates of recent read operations will result in cache hits that eliminate I/O for that operation since it is read from the cache instead of from disk. This can skew average read performance measurements.</td>
</tr>
</tbody>
</table>
6 RESULTS AND ANALYSIS

All results refer to the test case of 100 000 objects represented in figure 8. All results ignore the 5% most extreme latency measurements obtained.

6.1 Write operations

100 000 JSON objects were saved on file and written to the database according to the method described in section 5.4.

6.1.1 Overview

Figure 9 shows the average write latency for all three data models and the average number of statements executed for each model.

![Figure 9.](image)

The CQL data model achieved an average write latency of 9.41 milliseconds per operation. The corresponding result for the BLOB-4 method was 5.09 milliseconds and 2.26 milliseconds for the BLOB-1 method. Table 4 list the results table form:

<table>
<thead>
<tr>
<th></th>
<th>CQL</th>
<th>BLOB-4</th>
<th>BLOB-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average latency</td>
<td>9.41</td>
<td>5.09</td>
<td>2.26</td>
</tr>
<tr>
<td>Average number of statements</td>
<td>8.82</td>
<td>3.30</td>
<td>1.32</td>
</tr>
<tr>
<td>Average latency per statement (ms)</td>
<td>1.07</td>
<td>1.54</td>
<td>1.71</td>
</tr>
</tbody>
</table>
On average, the CQL method used 8.82 database statements to write an object, BLOB-4 used 3.30, and BLOB-1 used 1.32 statements. The latency measurements for each model seem in proportion to the number of statements executed for that model, indicating a relationship between write latency and the number of statements used to write an entire object.

The average latency per statement is higher in the BLOB methods than in the CQL methods. This is consistent with their respective data models, as more work is performed per statement in the BLOB models since they consist of fewer, larger objects.

6.1.2 Latency distribution

Figure 10 shows the latency distribution of all three data models. Each graph represents the latency measurements for its respective method sorted in ascending order where a data point’s position on the horizontal axis represents its latency ranking.

The left-most section of figure 10 shows that the CQL data model has a lower best case latency than BLOB-4 yet its worst case latency is approximately double that observed for BLOB-4. BLOB-1 achieves the best overall performance and the smallest performance variation.

Analysis
That the CQL data model would exhibit a best-case performance comparable to that of BLOB-4 is not surprising given the dataset’s complexity distribution (illustrated in figure 8). The dataset is heavily skewed towards simple objects composed of three or four sub-objects—such objects comprise nearly 50% of the dataset. These simple objects only require the execution of one additional insert statement in the BLOB-4 method than in the CQL method. Since the BLOB-4 method also requires an extra serialization step, some degree of latency overlap for the simplest objects would be expected.
Similarly, the BLOB-1 data model’s relatively low and consistent latency results compared to the other two methods should be expected since it executes fewer statements. The increase in average latency for BLOB-1 visible in the upper third of the graph is approximately consistent with the proportion of retweets in the dataset (32.1%). Retweets require two statements executions in BLOB-1 whereas all non-retweets are written in a single statement.

6.1.3 Latency/operation complexity breakdown

Figure 11 further indicates a relationship between write latency and the number of statements executed. For the BLOB-1 and BLOB-4 data models, the relationship is closer than for the CQL method which exhibits a curious latency drop at object complexity 28. Figure 11 also shows that BLOB-1 never executes more than 2 statements and that its write latency is relatively unaffected by increases in object complexity compared to the BLOB-4 and CQL data models.

The noticeable decrease in average latency relative to the number of statements executed at object complexity 28 for the CQL data model can be explained by certain peculiarities of the test data used (twitter4j.Status objects) and the CQL data model as defined in section 5.3.1. In the dataset, Status objects often contain an array of sub-objects—arrays of extendedMediaEntities objects are particularly common—and the retweeted objects often contain a duplicate set of the same objects. This is the case, for instance, in the object provided in Appendix A2, where the MediEntities and ExtendedMediEntities objects have the same id in both the Status object and the retweetedStatus object.

Often, ExtendedMediaEntities will appear in arrays of four where each element in turn typically contains an array of four Size objects that are also duplicated in the retweet. In the CQL data model, this results in key collisions when the object is written (since the primary key is generated from the object id) and each time an INSERT statement is executed on an existing key in Cassandra it will result in an UPDATE statement being executed instead.
In the BLOB-4 model, the mediaEntities and extendedMediaEntities are likely to be duplicated in retweets, but their Size arrays are serialized and contained within their respective parent objects and not written as separate objects as in the CQL data model. This is consistent with the less noticeable performance improvement at complexity 28 for BLOB-4 than for CQL.

Table 5 shows the client/server write latencies. ‘Put latency’ denotes non-client-side latency for database statement execution.

<table>
<thead>
<tr>
<th></th>
<th>CQL</th>
<th>BLOB-4</th>
<th>BLOB-1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total latency (ms)</strong></td>
<td>7.505</td>
<td>2.256</td>
<td>5.090</td>
</tr>
<tr>
<td><strong>Put latency (ms)</strong></td>
<td>7.386</td>
<td>1.987</td>
<td>4.786</td>
</tr>
<tr>
<td><strong>Client-side latency (ms)</strong> (Total – Put)</td>
<td>0.119</td>
<td>0.269</td>
<td>0.304</td>
</tr>
</tbody>
</table>

The table shows that the client-side latency accounts for a small portion of the overall operation latency relative to the put latency.
6.2 **Read operations**

6.2.1 **Overview**

Figure 16 shows the average read latency for all three data models and the average number of statements executed per read operation.

![Figure 12. Workload B overview.](image)

Figure 12 reveals a similar pattern for read operations as the one observed for write operations (fig. 9). The BLOB-1 data model again outperforms the CQL and BLOB-4 data models, but here the latency difference between CQL and BLOB-4 is not as pronounced as in the write workload.

Compared to the write workload, CQL read operations on average require 7.21 statement executions while CQL write operations on average required 8.82 statements. The number of read statements executed for the BLOB-1 and BLOB-4 data models differs less from the write workload results: 1.36 read statements vs 1.32 write statements for BLOB-1, and 3.72 read statements vs 3.30 write statements for BLOB-4.

Table 6 shows the graph data in table form:

<table>
<thead>
<tr>
<th></th>
<th>CQL</th>
<th>BLOB-4</th>
<th>BLOB-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average read latency</td>
<td>7.18</td>
<td>5.31</td>
<td>2.11</td>
</tr>
<tr>
<td>Average number of statements executed</td>
<td>7.21</td>
<td>3.72</td>
<td>1.36</td>
</tr>
<tr>
<td>Average total latency per statement (ms)</td>
<td>0.99</td>
<td>1.43</td>
<td>1.55</td>
</tr>
</tbody>
</table>
As in the write workload, while the BLOB-1 and BLOB-4 methods complete the read operations with lower average latency than the CQL model, they have higher latency on a per-statement basis which is to be expected since they perform more work per statement.

The lower average latency of CQL reads (7.18 ms) than CQL writes (9.41 ms) is proportional to the difference in the number of statements executed (an average of 7.21 read operations versus an average of 8.82 write operations), indicating a linear relationship between read latency and the number of statements executed.

Read operations require fewer statements than write operations in CQL data modelling since multiple objects can be fetched with a single query (e.g. `SELECT *`). For example, objects of the extendedMediaEntities and Size classes often appear in arrays of four and can therefore be read in one statement.

### 6.2.2 Latency distribution

Figure 13 shows the latency distribution for read operations.

![Read latency distribution](image)

*Figure 13. The read latency distribution (workload B).*

Here, the latency distribution pattern is similar to that seen in the write workload, but the difference between the CQL and BLOB-4 methods is less pronounced.

### 6.2.3 Latency/operation complexity breakdown

Figure 14 shows read latency at the different levels of object complexity and the average number of database statements executed to read the objects.
The most conspicuous detail about these results when contrasted with figure 11 is that the CQL reads require relatively fewer statements than CQL writes as the object complexity increases, observable in the relatively smaller length differential between the blue and red bars in figure 14 than in figure 11.

The CQL graph shows a pattern resembling that of the write workload. As in the write workload, a drop in average latency relative to the number of statements executed is found at around object complexity 27, however in this case the graph plateaus after this point and intersects with the BLOB-4’s average latency graph at object complexity 47, after which point the CQL method actually achieves a lower average read latency than BLOB-4 despite executing more statements.

As in the write workload, the average latency for BLOB-1 and BLOB-4 appears proportional to the number of statements executed. Here, too, the BLOB-1 method’s average latency seems little affected by increasing object complexity, but seems proportional to the number of statements executed.

**Analysis**

The CQL performance improvement noticeable after object complexity level 27 is likely due to cache hits occasioned by successive reads of the same key due to the data duplication present in retweets. Cache hit rate metrics are available as managed Java beans (MBeans) that the Cassandra process exposes via Java Management Extensions (JMX). Table 7 shows the stable value of the OneMinuteRate attribute of the MBean org.apache.cassandra.metrics:type=Cache,scope=KeyCache,name=Hits collected while running the read workload:

<table>
<thead>
<tr>
<th>Get statements executed per second</th>
<th>CQL</th>
<th>BLOB-1</th>
<th>BLOB-4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>959.9</td>
<td>544.2</td>
<td>597.8</td>
</tr>
</tbody>
</table>
The figures confirm that cache hit ratio in the CQL data model is relatively high compared to the BLOB models, comprising 14.8% of the statements. This could explain why read operations on more complex objects (fig. 18) do not exhibit higher average latency despite executing more statements, since for these objects the percentage would be even higher as they contain more duplicated content. Duplicate read operations would already reside in the Cassandra process’ cache and would not require any I/O, thereby reducing read latency.

The difference in cache hit ratio recorded for the BLOB-1 and BLOB-4 models is possibly due to Cassandra pre-fetching being more efficient in the BLOB-1 model since its rows are all stored in the same column family.

Table 8 shows that client-side latency accounts for a small proportion of overall latency, as in the write workload, indicating that the difference in latency depends on the specifics of the data modelling and Cassandra’s internal implementation details rather than client-side factors.

<table>
<thead>
<tr>
<th></th>
<th>CQL</th>
<th>BLOB-1</th>
<th>BLOB-4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total latency (ms)</strong></td>
<td>6.294</td>
<td>1.974</td>
<td>4.675</td>
</tr>
<tr>
<td><strong>Get latency (ms)</strong></td>
<td>6.273</td>
<td>1.813</td>
<td>4.447</td>
</tr>
<tr>
<td><strong>Client-side latency (ms)</strong> (Total – Get)</td>
<td>0.021</td>
<td>0.161</td>
<td>0.228</td>
</tr>
</tbody>
</table>
7 CONCLUSIONS AND FUTURE WORK

The results show that object serialization and data modelling can have a major impact on latency in Cassandra. In answer to RQ1, the study finds that the two Avro-serialized data models BLOB-1 and BLOB-4 overall achieved lower latency than CQL data modelling for both read and write workloads. The BLOB-4 data model achieved a 45.9% reduction in average write latency over the CQL data model, and the BLOB-1 data model achieved an average write latency reduction of 76.0% over CQL and a 55.6% reduction over BLOB-4. For read operations, the BLOB-4 data model achieved a 26.0% average latency reduction over CQL, BLOB-1 achieved a reduction of 70.6% over CQL and a 60% reduction over BLOB-4.

For all three methods, latency was found to increase relative to the number of statement executions needed to perform an operation, though, interestingly, the CQL latency results relative to statement count were skewed—especially in the read workload—by the high rate of member duplication in the Twitter test data resulting in key duplication in Cassandra, an unexpected finding which further emphasizes the performance impact of data modelling. The highest average latencies were nonetheless recorded in the CQL data model which was also the most statement-intensive of the methods studied. BLOB-1 was both the least statement-intensive and the best performing by a wide margin in both the read and write workloads. BLOB-1 latency showed little performance variation despite large variations in object complexity. The slight latency spike noticeable for BLOB-1 was consistent with the percentage of retweets (32%) and the number of statements executed. The number of statement executions was thus identified as the most important factor influencing performance. In answer to RQ2 and RQ3, the study therefore finds that object complexity has a negligible effect if handled client-side (BLOB-1) but has a major effect if it results in the execution of more statements, which is achieved with lower serialization granularity. Developers seeking to increase Cassandra throughput should therefore opt for a data model that processes objects in as few statements as possible. With regards to RQ4, tables 5 and 8 show that client-side latency has a negligible effect on total latency.

This paper has investigated the performance benefits Avro serialization can offer in simple read and write operations. A limitation with serialization in more complex use cases, however, is that it prevents access to individual object members that would be possible in relational model. With the serialization approach investigated here, entire BLOBs must be read, deserialized, changed, serialized, and re-inserted for an update to take place. Yet NoSQL solutions typically want to forgo supporting intricate relational logic from residing in the DBMS. Client-side logical layers therefore become increasingly important with serialization in NoSQL. Further studies should investigate such utility aspects of serialization, as well as other complimentary client-side solutions that can offer some of the functionality present in RDBMS while still maintaining the performance benefits of serialization demonstrated in this paper. The performance of secondary overlapping indices could provide one such case study, as recommended by Lombardo et al [9]. These would be separate tables with index information that define relations between objects, circumventing the need to deserialize objects for metadata extraction. This approach could then be compared with solutions where the metadata (e.g. foreign keys) is stored within the BLOB. Another interesting study could be to compare CQL and BLOB-based data modelling with data compression or data models.
that use the composite column feature recently added to Cassandra as an alternative data modelling approach to serialization.
8 REFERENCES


APPENDICES

8.1 A. Test data

A small and a large twitter4j.Status object converted into JSON format using the codehaus.jackson library. Objects of this type were used as test data.

8.1.1 A.1 Small object

Table 9 – Mapping of this object in each data model

<table>
<thead>
<tr>
<th>Data modelling</th>
<th>Number of objects</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQL</td>
<td>4</td>
<td>1 x status, 1 x user, 1 x usermentionentity, 1 x user_urlentity</td>
</tr>
<tr>
<td>BLOB-1</td>
<td>1</td>
<td>1 x status</td>
</tr>
<tr>
<td>BLOB-4</td>
<td>2</td>
<td>1 x status, 1 x user</td>
</tr>
</tbody>
</table>

```
{
  "createdAt": 1427365709000,
  "id": 581840015951695000,
  "text": "@edsheeran aw that must have been azming to do 😃😃😃
  "source": "<a href="https://mobile.twitter.com" rel="nofollow">Mobile Web (M5)</a>",
  "inReplyToStatusId": 580923223967182800,
  "inReplyTouserId": 85452649,
  "favoriteCount": 0,
  "inReplyToScreenName": "edsheeran",
  "geolocation": null,
  "place": null,
  "retweetCount": 0,
  "lang": "en",
  "retweetedStatus": null,
  "userMentionEntities": [
    {
      "start": 0,
      "end": 10,
      "name": "Ed Sheeran",
      "screenName": "edsheeran",
      "id": 85452649,
      "text": "edsheeran"
    }
  ],
  "hashtagEntities": [],
  "mediaEntities": [],
  "extendedMediaEntities": [],
  "symbolEntities": [],
  "currentUserRetweetId": -1,
  "scopes": null,
  "user": {
    "id": 2814798077,
    "name": "Allmyheart",
    "screenName": "heartstogether1",
```
8.1.2 A.2 Large object

This object contains a retweetedStatus object and would produce two write operations in the BLOB-1 method.

Table 10 – Mapping of this object in each data model

<table>
<thead>
<tr>
<th>Data modelling</th>
<th>Number of objects</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQL</td>
<td>27</td>
<td>2 x status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 x usermentionentity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x mediaentity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x extendedmediaentity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 x sizes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x user_urlentity</td>
</tr>
<tr>
<td>BLOB-1</td>
<td>2</td>
<td>2 x status</td>
</tr>
<tr>
<td>BLOB-4</td>
<td>8</td>
<td>2 x status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x media</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x extendedmedia</td>
</tr>
</tbody>
</table>

"retweetCount": 19,
"lang": "en",
"retweetedStatus": null,
"userMentionEntities": [],
"hashtagEntities": [],
"mediaEntities": [
  {
    "start": 53,
    "end": 75,
    "id": 58099407572684800,
    "url": "http://t.co/Lr3LXAFIhJ",
    "mediaURL": "http://pbs.twimg.com/media/CBAf4gcVEAAR-sD.jpg",
    "mediaURLHttps": "https://pbs.twimg.com/media/CBAf4gcVEAAR-sD.jpg",
    "expandedURL": "http://twitter.com/Ronaldo_7Facts/status/58099421514498048/photo/1",
    "displayURL": "pic.twitter.com/Lr3LXAFIhJ",
    "sizes": {
      "0": {
        "width": 150,
        "height": 150,
        "resize": 101
      },
      "1": {
        "width": 340,
        "height": 340,
        "resize": 100
      },
      "2": {
        "width": 600,
        "height": 600,
        "resize": 100
      },
      "3": {
        "width": 600,
        "height": 600,
        "resize": 100
      }
    }
  }
],
"extendedMediaEntities": [
  {
    "start": 53,
    "end": 75,
    "id": 58099407572684800,
    "url": "http://t.co/Lr3LXAFIhJ",
    "mediaURL": "http://pbs.twimg.com/media/CBAf4gcVEAAR-sD.jpg",
    "mediaURLHttps": "https://pbs.twimg.com/media/CBAf4gcVEAAR-sD.jpg",
    "expandedURL": "http://twitter.com/Ronaldo_7Facts/status/58099421514498048/photo/1",
    "displayURL": "pic.twitter.com/Lr3LXAFIhJ",
    "sizes": {
      "0": {
        "width": 150,
        "height": 150,
        "resize": 101
      },
      "1": {
        "width": 340,
        "height": 340,
        "resize": 100
      },
      "2": {
        "width": 600,
        "height": 600,
        "resize": 100
      },
      "3": {

About Cristiano Ronaldo.

https://abs.twimg.com/images/themes/theme1/bg.png

https://pbs.twimg.com/profile_banners/2317110479/1425445997/web

https://pbs.twimg.com/profile_images/580817065197453313/wCiYGhzO_normal.jpg

https://pbs.twimg.com/profile_images/580817065197453313/wCiYGhzO_bigger.jpg

https://pbs.twimg.com/profile_images/580817065197453313/wCiYGhzO_mini.jpg

https://pbs.twimg.com/profile_images/580817065197453313/wCiYGhzO.jpg