<https://github.com/spoddutur/spark-notes>

**Apache Spark - Deep Dive into Storage Format's**

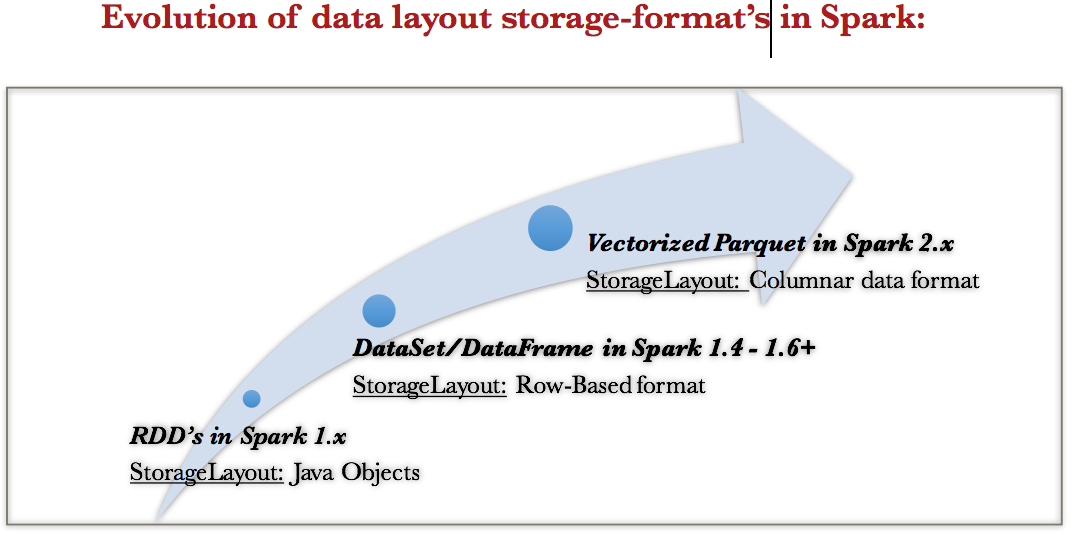
<https://github.com/spoddutur/spark-notes/blob/master/deep_dive_into_storage_formats.md>

Apache Spark has been evolving at a rapid pace, including changes and additions to core APIs. Spark being an in-memory big-data processing system, memory is a critical indispensable resource for it. So, efficient usage of memory becomes very vital to it. Let’s try to find answer to following questions in this article:

* What storage format did Spark use?
* How did storage format evolve over a period of time?

### What storage format did Spark use?

* **Spark 1.0 to 1.3:** It started with RDD’s where data is represented as Java Objects.
* **Spark 1.4 to 1.6:** Deprioritised Java objects. DataSet and DataFrame evolved where data is stored in row-based format.
* **Spark 2.x:** Support for Vectorized Parquet which is columnar in-memory data is added.

[](https://user-images.githubusercontent.com/22542670/26998244-18ed1b40-4d9f-11e7-897d-5d2b032c3001.png)

### How did storage format evolve over a period of time?

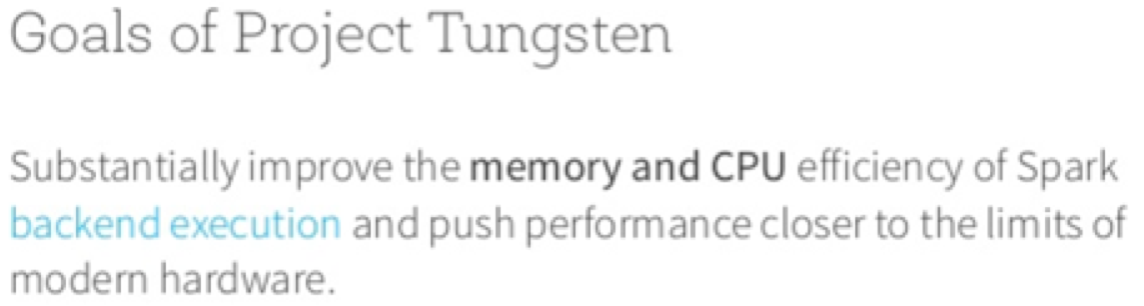
The answer to this question is not only interesting but also lengthy. Let’s dive into the driving factors and the journey of how Storage formats evolved in two parts:

1. Advancement from RDD to Row-based Dataset.
2. From Row-based Dataset to Column-based Parquet.

## Part1: Evolution of Data Storage Format's - Advancement of RDD to Row-based Dataset.

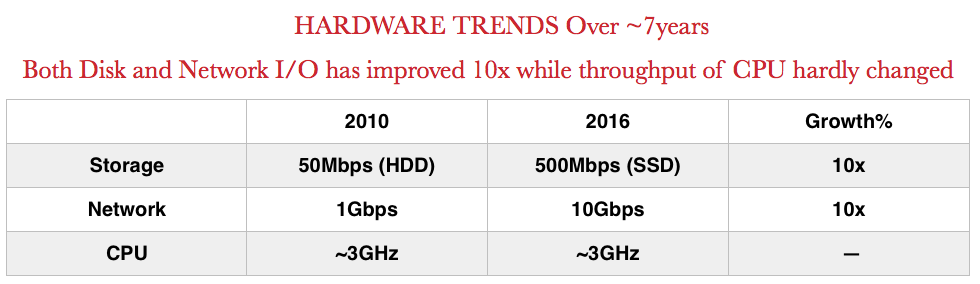
We cannot find the answer for this question, without diving into Project Tungsten. Just go with the flow and you’ll reason out as to why are we looking into Tungsten soon. Project Tungsten has been the largest change to Spark’s execution engine since its inception. This project has led to some fundamental changes in Spark. One such change gave birth to DataSet. Let’s start by looking at the goal of Project Tungsten.

## Objective of Project Tungsten

[](https://user-images.githubusercontent.com/22542670/26983352-183c0c06-4d59-11e7-8fd9-a4d0cfd3c1d1.png)

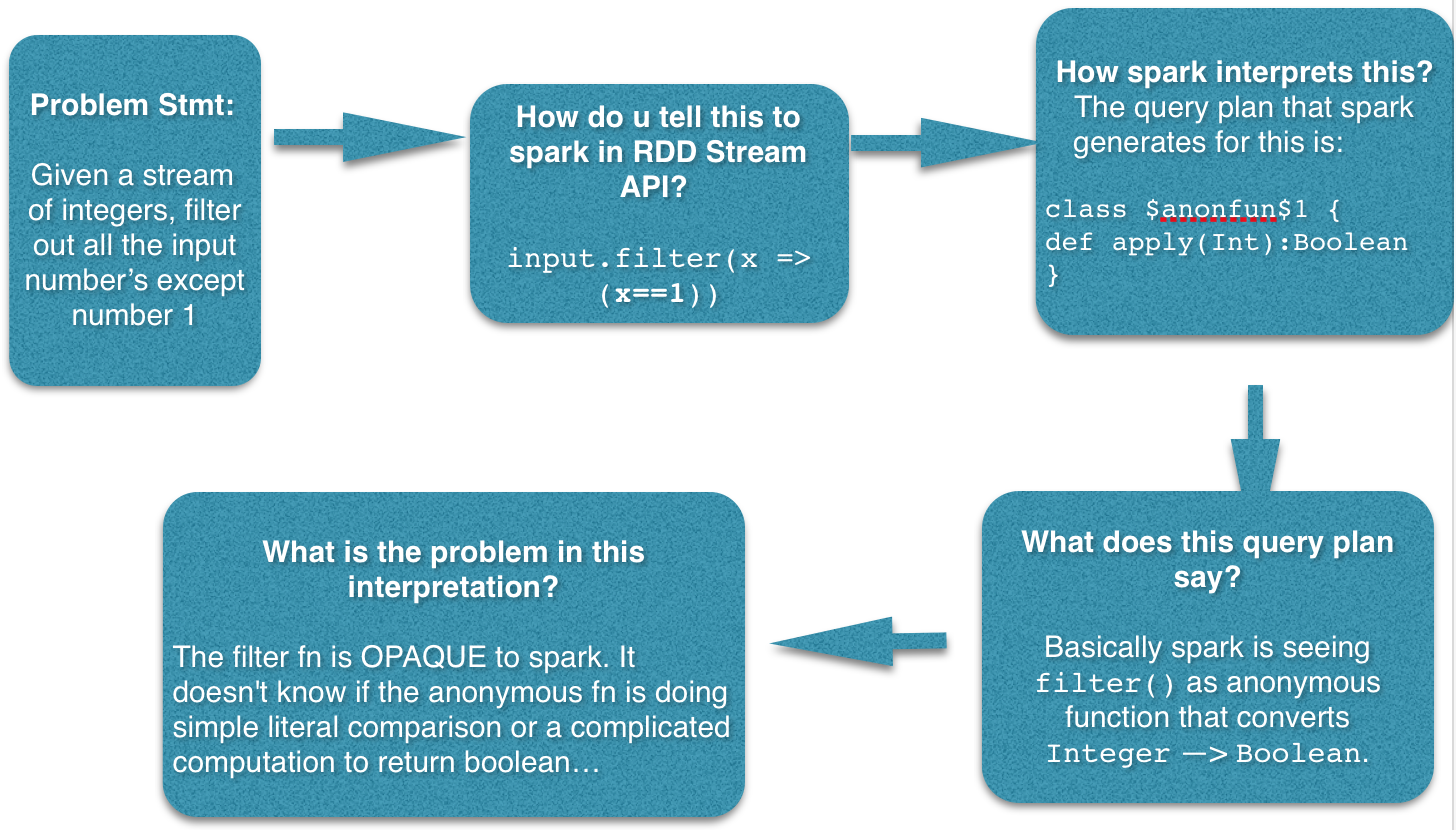
### Why this objective?

The focus on CPU efficiency is motivated by the fact that Spark workloads are increasingly bottlenecked by CPU and memory use rather than IO and network communication. To understand this better:

* Let’s see the hardware trends over the past 7 years. Looking at the graph, its obvious that both DISK I/O and Network I/O have improved 10x faster but CPU remained pretty much same: [](https://user-images.githubusercontent.com/22542670/27210375-b82e0f50-526f-11e7-98b9-37b0cfb00bbb.png)
* Another trend that’s noticed was, in Spark’s shuffle subsystem, serialisation and hashing (which are CPU bound) have been shown to be key bottlenecks, rather than raw network throughput of underlying hardware.

These two trends mean that Spark today is constrained more by CPU efficiency and memory pressure rather than network or disk IO

### Problem1:

First things first..We need to understand the problem with old execution engine before we think of solutions to improve efficiency. To better understand this, let’s take a simple task of filtering a stream of integers and see how spark 1.x used to interpret it. [](https://user-images.githubusercontent.com/22542670/26983465-7fd4819a-4d59-11e7-8f4b-4d5ee63b76de.png)

### What is the problem with this query plan?

As mentioned in the picture above, Spark doesn't know:

* What operation user is trying to perform on data? and
* What is the type of data?

Because of the lack of transparency, there’s barely any scope for spark to optimise the query plan.

### What transparency does spark need?

Spark needs transparency in 2 aspects:

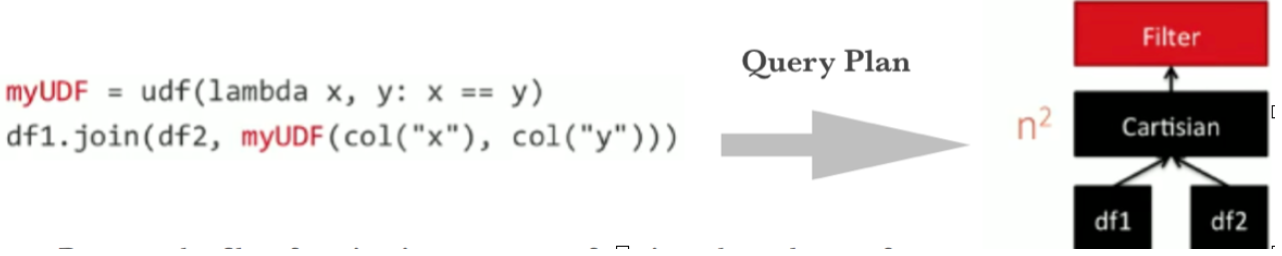
1. **Data Schema:**

* How many and what fields are there in each data record?
* What are the datatypes of each field?

1. **User operation:** Spark should know what kind of operation user is trying to perform on which field of the data record.

### How does this transparency help Spark?

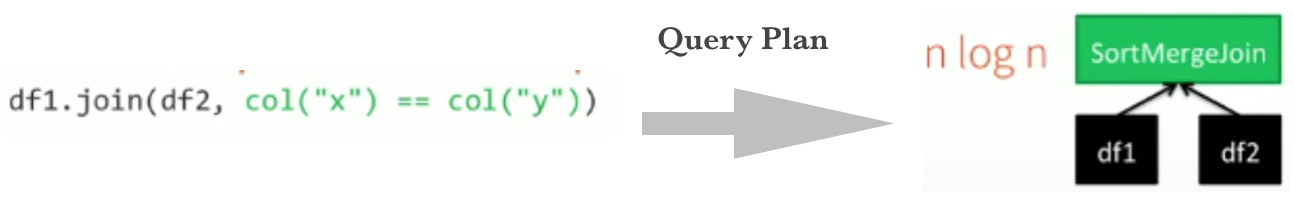
It helps spark improve performance. Its easier to illustrate it with an example. Consider joining two inputs df1 and df2. JoinCondition: column x of df1 = column y of df2.

Following figure depicts the join and its query plan that spark generates: [](https://user-images.githubusercontent.com/22542670/26983475-8bf4d308-4d59-11e7-9a53-ffc48f712fa1.png) Because the join condition is an anonymous function (myUDF), the only way for spark to perform this join is:

1. Perform cartesian product of df1 with df2.
2. Now, apply filter fn (myUDF) on the cartesian result which is an anonymous function that filters with df1[x] == df2[y] condition. Runtime = n^2!!

### How can transparency help spark improve this?

The problem in the above query was that join condition is an anonymous function. If spark is aware of the join condition and data schema [datatype of x and y columns] then spark would:

* First sort df1 by x
* Next sort df2 by y
* SortMerge Join df1 and df2 by df1[x] == df2[y]!! Runtime reduced to nlogn!! [](https://user-images.githubusercontent.com/22542670/26983488-9872b8b6-4d59-11e7-9562-661f7b9e305e.png)

### Action Plan1:

1. **Have user register dataschema** - This will spark transparency on what data it is handling
2. **Make the operations that user want to perform on data transparent to Spark**

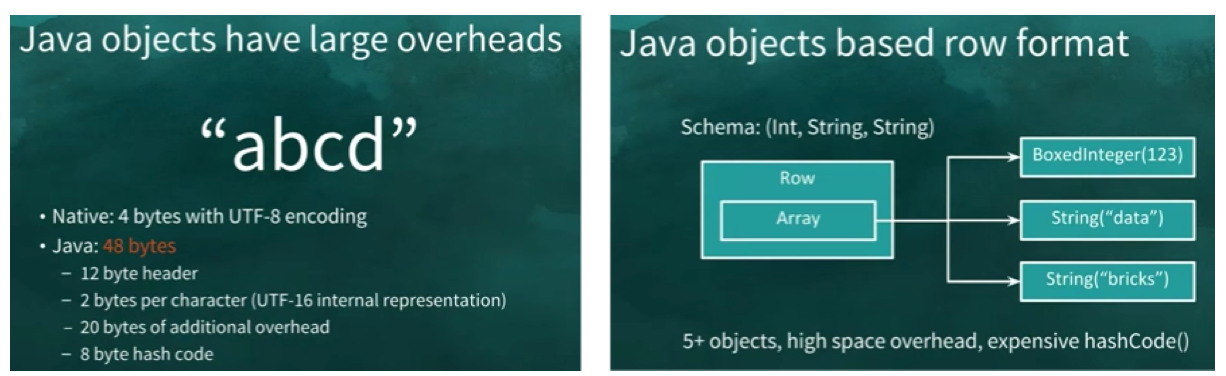
### Problem2:

Spark tries to do everything in-memory. So, the next question is to know if there is a way to reduce memory footprint. We need to understand how is data laid out in memory for this.

### How is data laid out in memory?

With RDD’s data is stored as Java Objects. There’s a whole lot of serialisation, deserialisation, hashing and object creation that happens whenever we want to perform an operation on these java objects’s and during shuffles. Apart from this Java objects have large overheads.

### Should spark stop relying on JavaObjects? Why?

Ans: Java Objects have large overheads Consider a simple string “abcd”. One would think it would take about ~4bytes of memory. But in reality, a java string variable storing the value “abcd” would take 48 bytes. Its breakdown is shown in the picture below. Now, imagine the amount of large overheads a proper JavaObject like a tuple3 of (Int, String, String) shown on the right hand side of the picture below takes. [](https://user-images.githubusercontent.com/22542670/26983508-a7bc2d5c-4d59-11e7-8d51-18ae60cd9ef0.png)

### Action Plan2:

#### Instead of java objects, come up with a data format which is more compact and less overhead

### Action Plan1 + Action Plan2 together:

1. **Have user register dataschema** - This will spark transparency on what data it is handling
2. **Make the operations that user want to perform on data transparent to Spark**
3. Create new data layout which is more compact and less overhead.

#### This paved way to “Dataframes and Datasets”

### What is DataSet/DataFrame?

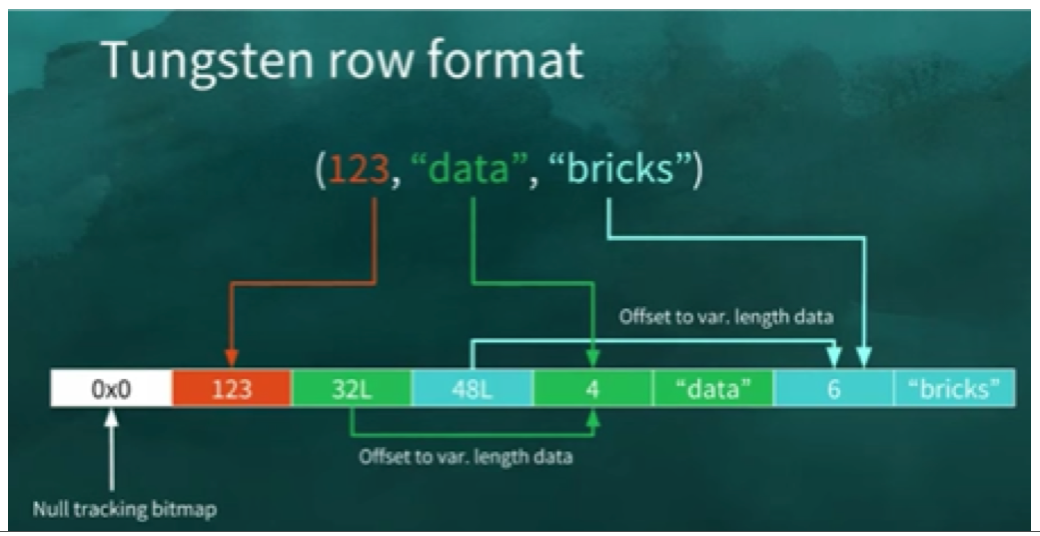
A Dataset is a strongly-typed, immutable collection of objects with 2 important changes that spark introduced as we discussed in our action plan:

* Introduced new Binary Row-Based Format
* Data Schema Registration

**Introduced new Binary Row-Based Format:**

* To avoid the large overheads of a java object, Spark adapted new binary row-based storage as described in the table below:

[](https://user-images.githubusercontent.com/22542670/26983544-be7f6cf2-4d59-11e7-9398-47c5b0959b9d.png)

Following picture illustrates the same with an example. In this example, we took a tuple3 object **(123, “data”, “bricks”) and** and let’s see how its stored in this new row-format.  
[](https://user-images.githubusercontent.com/22542670/26983560-cc595054-4d59-11e7-805e-c3526ca4d38e.png)

* The first field 123 is stored in place as its primitive.
* The next 2 fields data and bricks are strings and are of variable length. So, an offset for these two strings is stored in place [32L and 48L respectively shown in the picture below].
* The data stored in these two offset’s are of format “length + data”. At offset 32L, we store 4 + data and likewise at offset 48L we store 6 + bricks.

### Data Schema Registration

Following example shows how to register data schema: In this example, we're creating "students" Dataset.

case class Student(id: Long, name: String, yearOfJoining: Long, depId: Long)

val students = spark.read.json(“/students.json").as[Student]

Note that \*\*.as[Student]\*\* function call is registering schema of input students data with Spark.

### Let's see how transformations are applied on dataset

Transformations are nothing but simple operations like filter, join, map etc which take a dataset as input and return new transformed dataset. Please find below two transformations examples:

1. Filter students by YearOfJoining > 2015

// syntax: dataset.filter(filter\_condition)

students.filter("yearOfJoining".gt(2015))

You might have noticed that filter condition is not anonymous function anymore. we're explicitly telling spark now to filter using yearOfJoining > 2015 condition

1. Join Student with Department

// syntax: ds1.join(ds2, join\_condition)

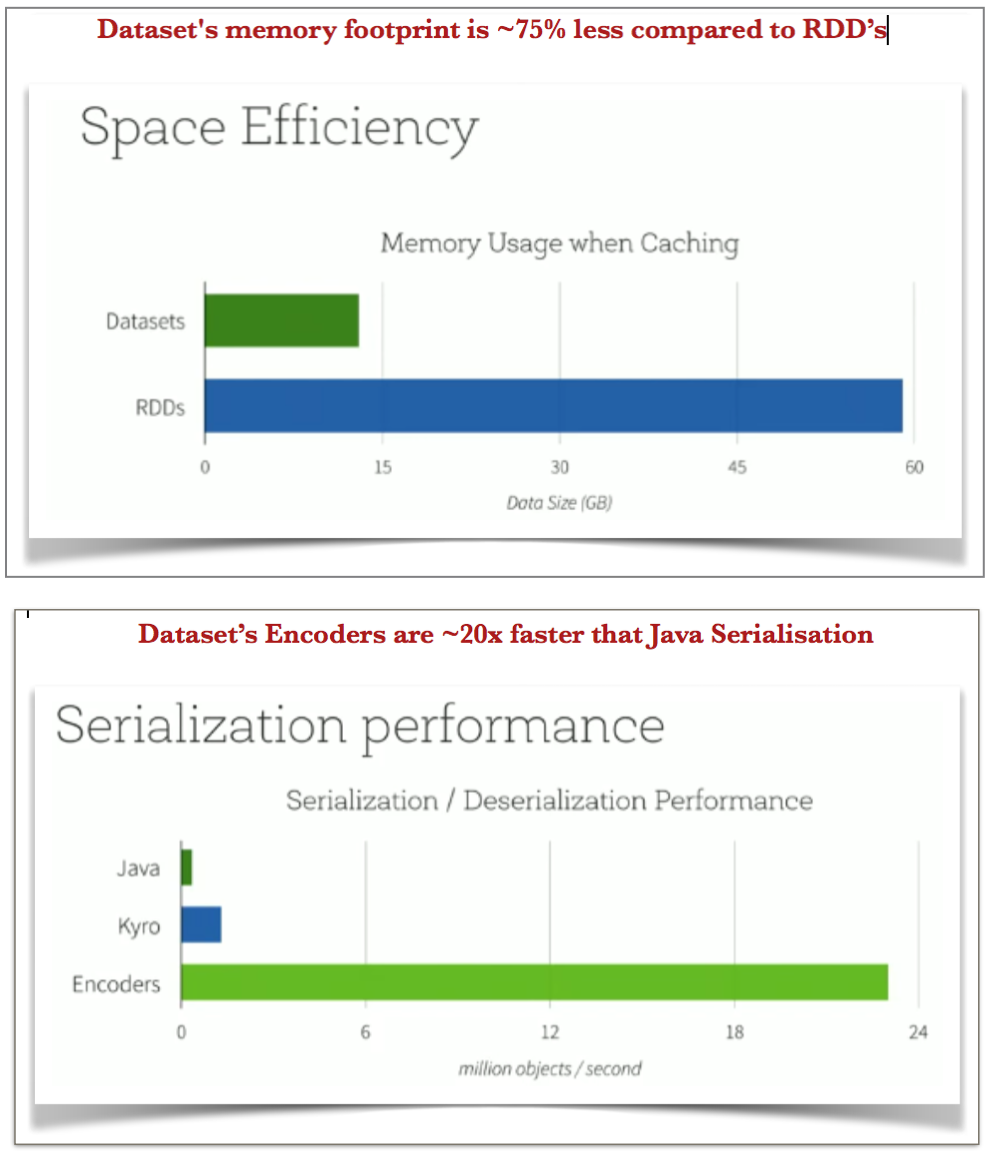
students.join(department, students.col("deptId").equalTo(department.col("id")))

Notice that join condition (students.col("deptId").equalTo(department.col("id"))) is also not anonymous function!! We're explicitly telling spark to join on students[depId] == department[id] condition

**Some more things to note on Dataset's:**

* So, we registered dataschema with simple one liner code: .as[Student].
* DataSet operations are very explicit. In that, what operation is user performing on which column is evident to Spark.
* So, Spark got the transparency it wanted.
* **Who converts DataSet to Tungsten Binary format and vice-versa?**
* Ans: Spark provides Encoder API for DataSet’s which is responsible for converting DataSet to spark internal Tungsten binary format and vice-versa.
* **A less obvious advantage with Encoders:** Encoders eagerly check that your data matches the expected schema, providing helpful error messages before you attempt to incorrectly process TBs of data

### RDD’s of JavaObjects ****(vs)**** Dataset’s

[](https://user-images.githubusercontent.com/22542670/27128201-351d0b84-511b-11e7-8c08-5f0dd0b4085b.png)

### Benefits of Dataset’s

* Compact (less overhead)
* Reduce our memory footprint significantly.
* Spark knows what data is it handling now.
* Spark also knows the operation that user wants to perform on Dataset
* Both the above two listed benefits paved way to one more not so obvious advantage which is:
* Possible in-place transformations for simple one’s without the need to deserialise. (Let’s see how this happens in detail below)

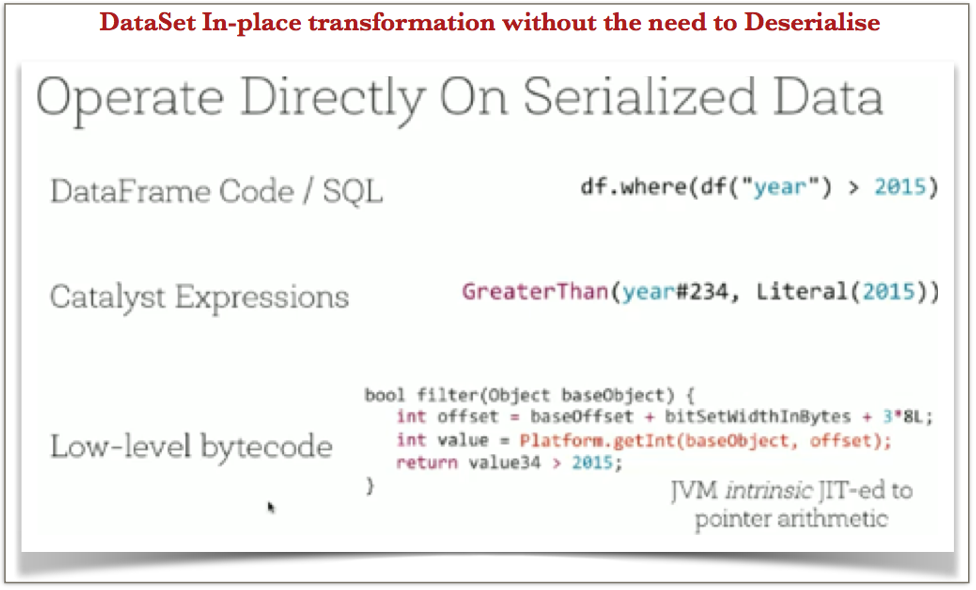
### What does in-place transformation with dataset's mean?

With RDD's, to apply a transformation operation on data (like filter, map, groupBy, count etc), there are 3 steps:

1. Its first deserialized into java object
2. We apply the transformation operation on JavaObject and
3. Finally serialize javaobject back into bytes.

With Dataset's, in-Place Transformation essentially means that, we need not deserialize a dataset to apply a transformation on it. Let's see how it happens next..

### How does in-place transformation happen in DataSet/Dataframe?

Let’s see how the same old filter fn behaves now. Consider the case where we’ve to filter input by year>2015 condition in a dataframe. Note that the filter condition specified via the dataframe code df.where(df(“year” > 2015)) is not an anonymous function. Spark exactly knows which column it needs for this task and that it needs to do greater than comparison. [](https://user-images.githubusercontent.com/22542670/27127611-51e2148c-5119-11e7-93cc-7544a8e6cdf0.png)

The low-level byte code generated for this query looks something like this as shown in the above figure:

// This filter function is returning boolean on 'year > 2015' condition

bool filter(Object baseObject) {

// 1. compute offset from where to fetch the column 'year'

int offset = baseoffset + <..>

// 2. Fetch the value of 'year' column of the given baseObject

// directly without deserialing baseObject

int value = Platform.getInt(baseObject, offset)

// 3. return the boolean

return value > 2015

}

**Interesting things to note here is that:**

* Filter fn is not anonymous to spark
* Object wasn't deserialised to find the value of column ‘year’
* The value of ‘year’ is directly fetched from the offset
* In-place transformation can be done only for simpler operations like this

### This is great!! Let’s see how does in-place transformation helps in speed?

Atleast for the simple use cases..

* We can Directly operate on serialised data => lesser Deserialisation
* Lesser Deserialisation => less Object creation
* Less Object creation => Lesser GC
* Hence Speed!!

### Finally: What is DataSet/DataFrame?

* Strongly typed object mapped with schema
* Stored in tungsten row-format
* At the core of the Dataset API is a new concept called an Encoder
* Encoder is responsible for converting to and from spark internal Tungsten binary format.
* Encoder uses data schema while performing any operations.
* While DataSet is a strongly typed object, DataFrame is DataSet[GenericRowObject]
* Now, that we’ve seen in-depth detail of how and why Dataset emerged and it looks so promising, Why did spark moved to column-based Parquet? Let’s find out why..

## Part2: Evolution of Data Storage Format's - from Row-based Dataset to Column-based Parquet:

As per our discussions so far, Spark 1.x used row-based storage format for Dataset’s. Spark 2.x added support for column-based format. columnar format is basically transpose of row-based storage. All of the integers are packed together, all the strings are together. Following picture illustrates the memory layout of a row-baed vs column-based storage formats. [](https://user-images.githubusercontent.com/22542670/26983627-187e1384-4d5a-11e7-9856-2ae5d20071c6.png)

### Why columnar?

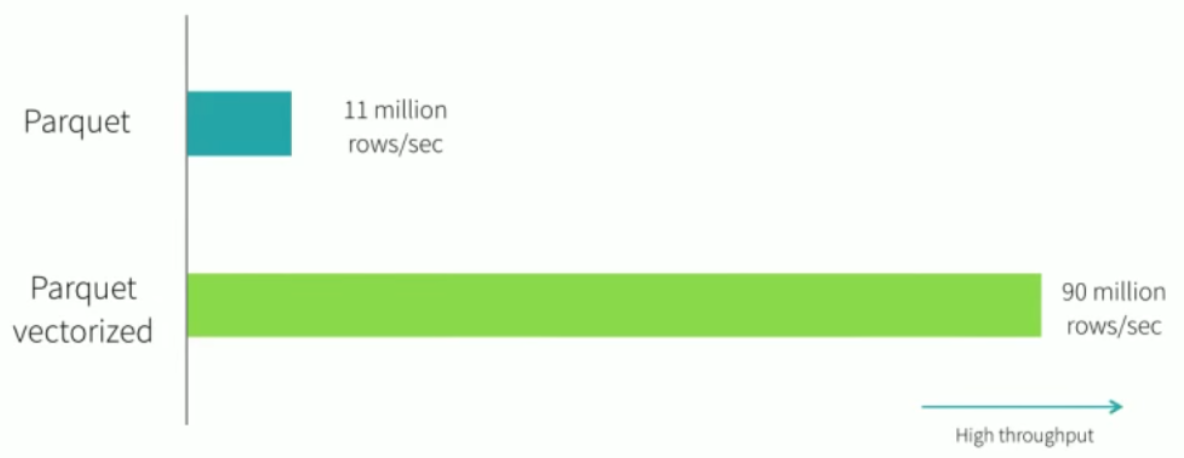
2nd Generation Tungsten Engine tried to use some standard optimisation techniques like loop unrolling, SIMD, prefetching etc which modern compilers and CPUs apply to speed up runtime. As part of this, Spark came up with two new optimization techniques called WholeStageCodeGeneration and Vectorization (Detail of these technologies can be found [here](https://spoddutur.github.io/spark-notes/second_generation_tungsten_engine)). For this spark did 2 things:

1. Tweaked its execution engine to perform vector operations and attained data level parallelism at algorithm level
2. Spark moved from row-based to columnar in-memory data enabling themselves for further SIMD optimisations like data striping for better cache and other advantages listed below.

### What opportunities did columnar storage open up?

1. **Regular data access vs Complicated off-set computation:** Data access is more regular in columnar format. For example, if we have an integer, we always access them 4 bytes apart. That’s very nice for the cpu. With row-based format there’s complicated offset computation to know where am I.
2. **Denser storage:** Because the nature of the data is homogeneous, we can apply better compression techniques according to the data type.
3. **Compatibility and zero serialization:** Columnar format is more compatible because many high performance systems already use columnar like numpy, tensorflow etc. Add on top of it, with spark having them in memory implies zero serialisation and zero copy. For example, most of our spark plan is evaluated in spark and at the end of it we want to call tensor flow and when its done, we want to get back. With spark using columnar in-memory format, that’s compatible with tensorflow. So, its gonna be done without ever having to do serialisation etc. It just works together.
4. **Compatibility with in-memory cache:** Having columnar storage is more compatible for obvious reasons with spark’s in-memory columnar-cache.
5. **More Extensions:** Modern day data-intensive machine learning applications through-put is achieved in industry by running on GPUs. GPUs are more powerful than CPUs for homogeneous data crunching. Having homogenous columnar storage paves way for future off-loading the processing to GPU's and TPU's to avail its advanced hardwares. (Please find performance comparision between GPU & TPU in Appendix section below)

**Performance Benchmarking:**

* Let's benchmark **Spark 1.x Columnar** data (Vs) **Spark 2.x Vectorized Columnar** data.
* For this, Parquet which is the most popular columnar-format for hadoop stack was considered.
* Parquet scan performance in spark 1.6 ran at the rate of 11million/sec.
* Parquet vectorized in spark 2.x ran at about 90 million rows/sec roughly 9x faster.
* Parquet vectored is basically directly scanning the data and materialising it in the vectorized way.
* This is promising and clearly shows that this is right thing to do!! [](https://user-images.githubusercontent.com/22542670/26983657-2e3aaa84-4d5a-11e7-8848-fd84d6ee0612.png)

### CONCLUSION - Journey of Data Layout in Spark

**We’ve seen:**

* Disadvantages with java objects
* Evolution of strongly typed Datasets with schema using the new internal tungsten row-format
* Dataset Benefits: lesser serialisation/deserialisation, less GC, less object creation and possible In-place transformations for simpler cases.
* Spark introduced support for columnar data using a new technique called Vectorization in spark 2.x to better leverage the advancements of Modern CPU’s and hardware like loop-unrolling, SIMD etc.

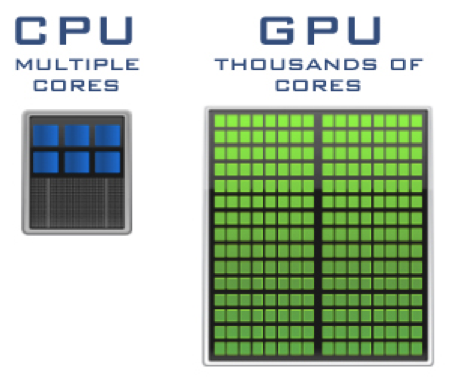
[My HomePage](https://spoddutur.github.io/spark-notes/)

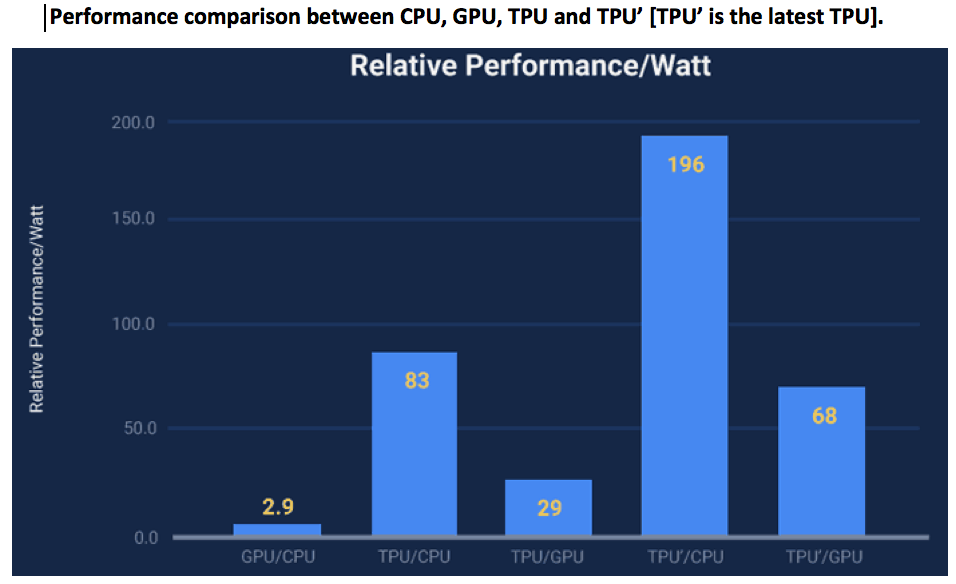
## References

* [Spark Memory Management](https://www.youtube.com/watch?v=dPHrykZL8Cg)
* [Deep Dive into Project Tungsten](https://www.youtube.com/watch?v=5ajs8EIPWGI)

## Appendix

### A quick additional note on GPU & TPU's:

* GPU:
  + Modern day GPUs thrive on SIMD architecture.
  + GPU is tailored for highly parallel operation while CPU executes and is designed for serial execution.
  + GPU have significantly faster and more advanced memory interfaces as there is need to shift around a lot more data than CPUs.
  + GPU are a result of evolution into a highly parallel, multi-threaded cores with independent piplines & stages supported by high memory band width. Modern day cpus are 4-8 cores but GPUs are 1000s of cores. [](https://user-images.githubusercontent.com/22542670/27191344-2037a490-5215-11e7-8f4b-41e1b96a55f6.png)
* TPU:
  + Tensor Processing unit (TPU) is an application specific hardware developed by Google for Machine Learning.
  + Compared to GPU it is designed explicitly for higher volume of reduced precision computation with higher throughput of execution per watt.
  + Hardware has been specifically designed for Google’s Tensor Flow framework.

Following graph depicts the performance/watt comparision between CPU, GPU, TPU and TPU' - latestTPU: [](https://user-images.githubusercontent.com/22542670/27191111-68ac4f6a-5214-11e7-93f6-7dcb12338251.png)