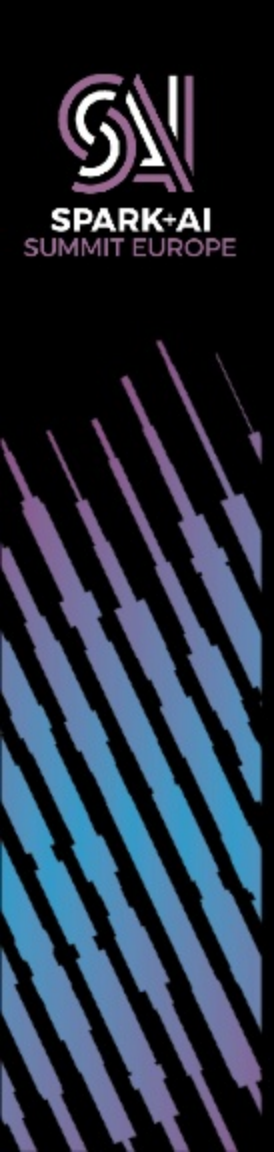


Productionizing a Machine Learning System at a Large Australian Telco

Cameron Joannidis, Simple Machines

#SAISML6

Who Am I?

- I'm Cameron
  - Linkedin - https://www.linkedin.com/in/cameron-joannidis/
  - Twitter - @camjo89

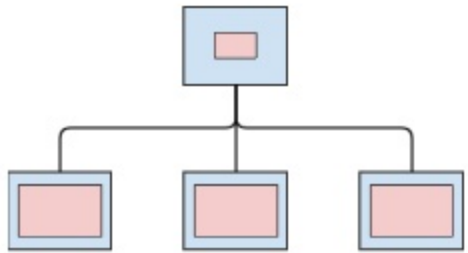
- Principal Machine Learning Engineer at Simple Machines

- Open source contributor - https://github.com/camjo

Simple
Machines.

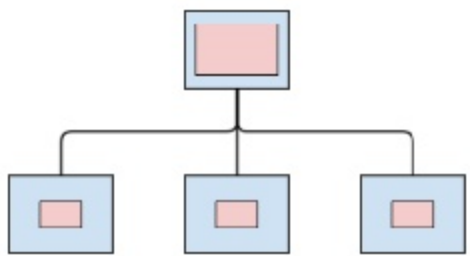

DO WE HAVE ANY ACTIONABLE ANALYTICS FROM OUR BIG DATA IN THE CLOUD?
YES, THE DATA SHOWS THAT MY PRODUCTIVITY PLUNGES WHENEVER YOU LEARN NEW JARGON.
MAYBE IN-MEMORY COMPUTING WILL ACCEL-ERATE YOUR APPLICA-TIONS.
PLUNGE, PLUNGE, PLUNGE.
Dilbert.com DilbertCartoonist@gmail.com
1-9-13 ©2013 Scott Adams, Inc./Dist. by Universal Uclick

Data Science



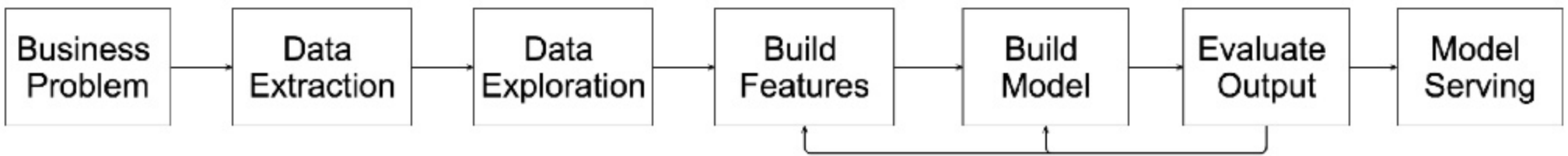
Mostly Distributed

Data Engineering



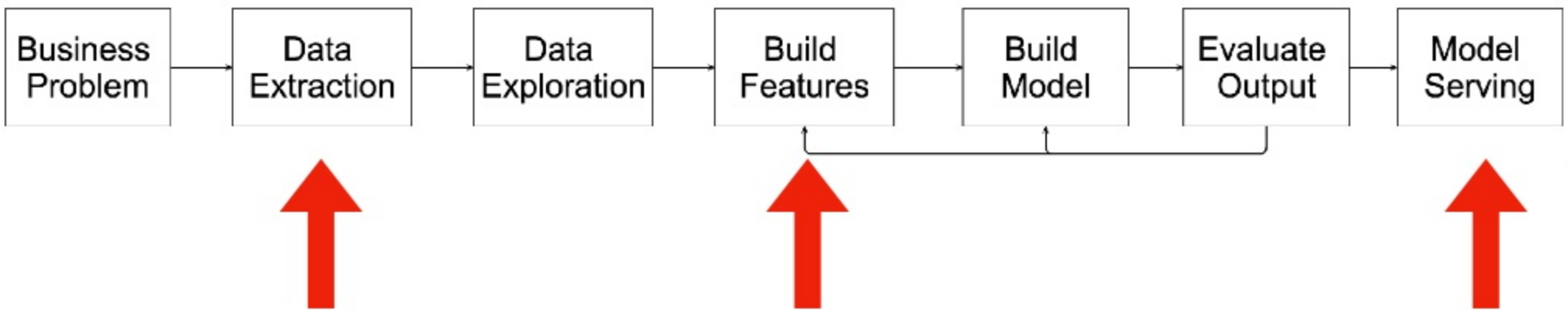
Mostly Centralised

Data Science Workflow

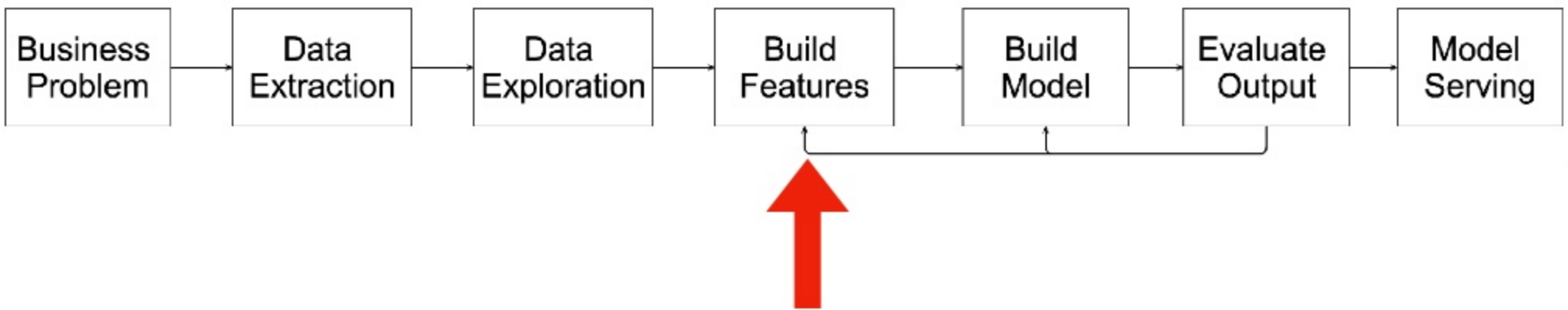


Data Science Workflow

Data Engineering



Feature Engineering

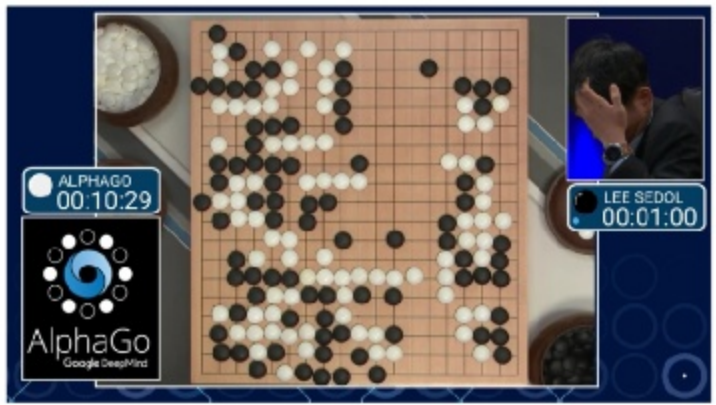


What is Machine Learning?

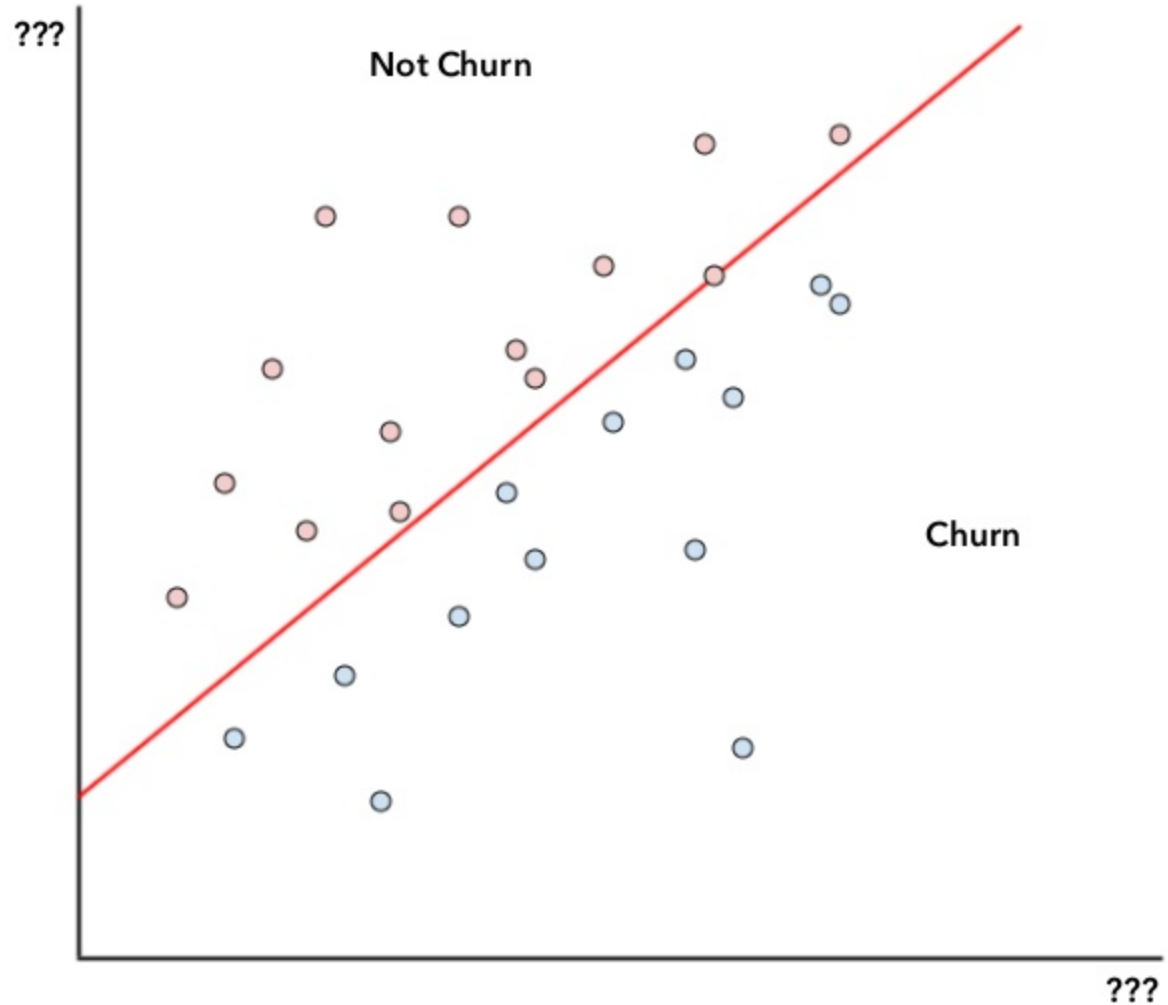
What is Machine Learning?

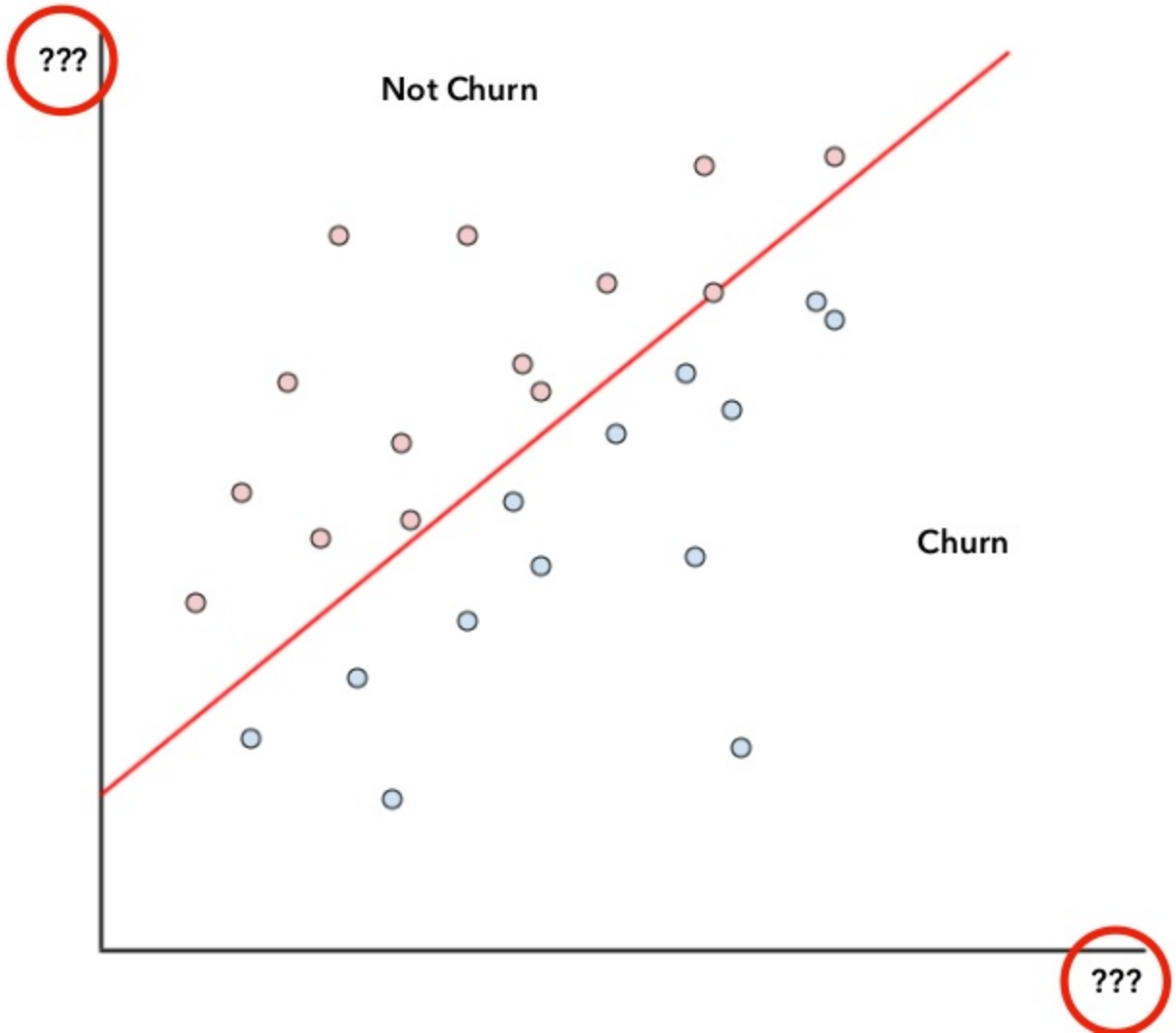
What is Machine Learning?

What is Machine Learning?



What is Machine Learning - Reality...





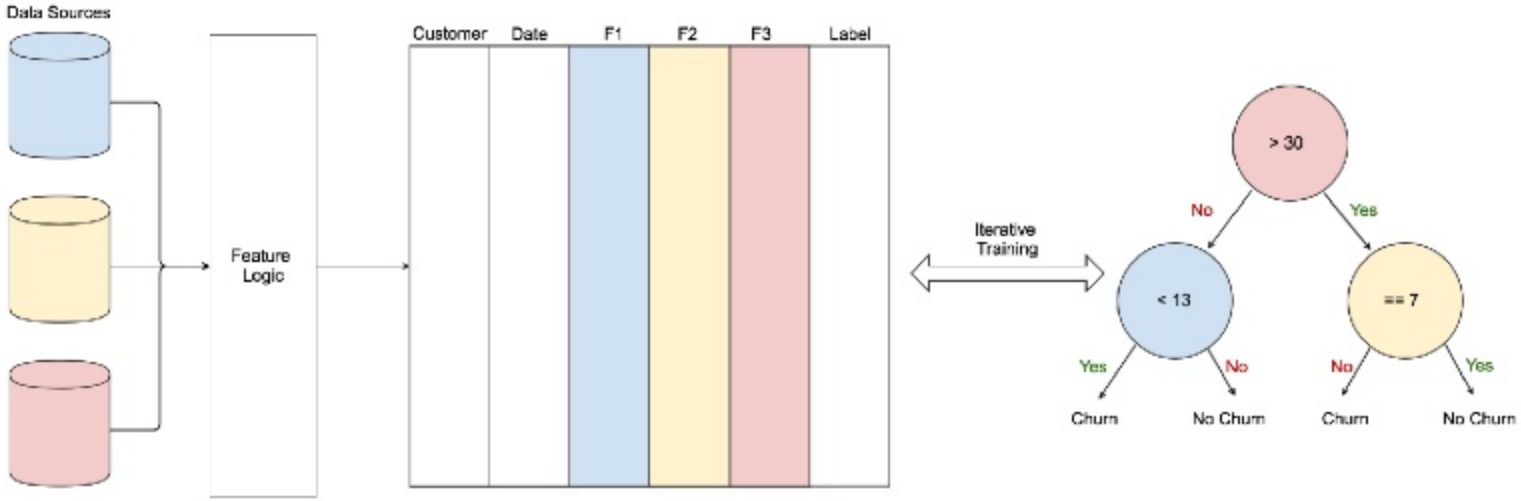
Features

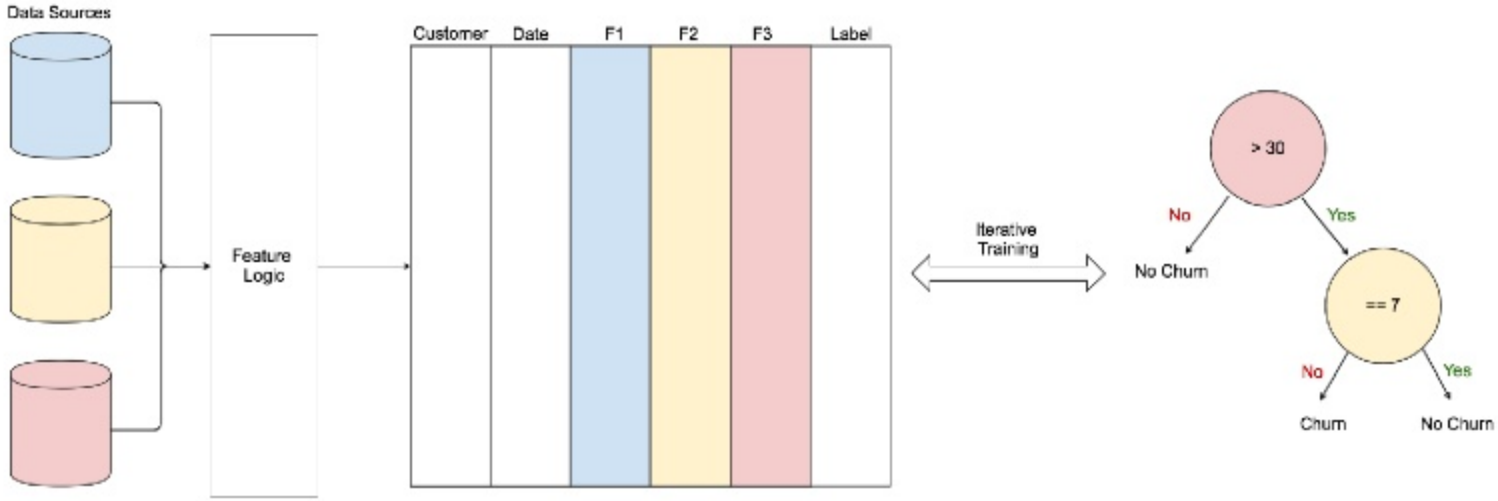
- Since we can't directly measure churn, we need to instead try and predict churn using something we **can** measure.
- These measurements are called features an are typically associated with an entity and an instant in time (e.g. customer, 2018-01-01)
- Can be as simple as some raw data all the way through to a complex transformation across multiple systems to capture some domain knowledge within your business
- Most of the performance of ML algorithms comes from well engineered features

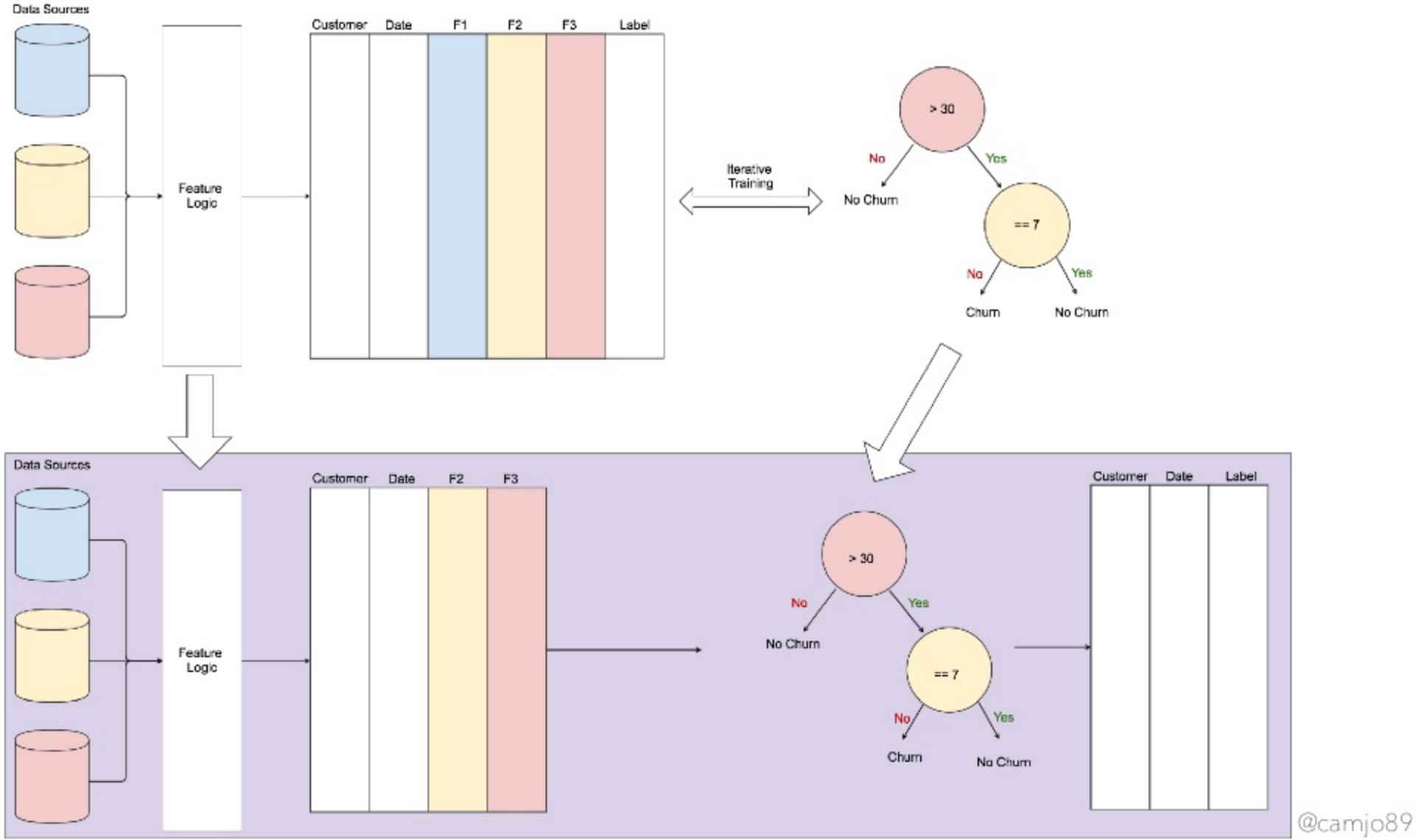
Training vs. Scoring

- Training dataset requires large volumes of historical data and many more features than will likely end up in the finished model.

- Scoring typically uses a subset of the training features and is only calculated for the most recent values of those features.


Data Sources
Feature Logic
Customer
Date
F1
F2
F3
Label
Iterative Training
> 30
No
Yes
< 13
== 7
Yes
No
No
Yes
Churn
No Churn
Churn
No Churn


Data Sources
Feature Logic
Customer
Date
F1
F2
F3
Label
Iterative Training
> 30
No
Yes
No Churn
== 7
No
Yes
Churn
No Churn


Data Sources
Feature Logic
Customer
Date
F1
F2
F3
Label
Iterative Training
> 30
No
Yes
No Churn
== 7
No
Yes
Churn
No Churn
Data Sources
Feature Logic
Customer
Date
F2
F3
> 30
No
Yes
No Churn
== 7
No
Yes
Churn
No Churn
Customer
Date
Label

Feature Store

- A feature store is a data store that holds features for some given scope of the business (e.g. customer/household etc) at a particular time granularity (e.g. daily/weekly/yearly).

EAVT

- **E**ntity **A**ttribute **V**alue **T**imestamp

EAVT

- **E**ntity **A**ttribute **V**alue **T**imestamp

Entity	Attribute	Value	Time
Cathy Smith	Feature 1	33	2018-01-01
Cathy Smith	Feature 1	37	2018-01-02
Jason Poser	Feature 2	164	2018-01-01

EAVT

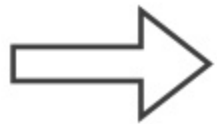
- **E**ntity **A**ttribute **V**alue **T**imestamp

Entity	Attribute	Value	Time
Cathy Smith	Feature 1	33	2018-01-01
Cathy Smith	Feature 1	37	2018-01-02
Jason Poser	Feature 2	164	2018-01-01
Cathy Smith	Feature 3	“yes”	2018-01-01

EAVT

- **E**ntity **A**ttribute **V**alue **T**imestamp

Entity	Attribute	Value	Time
Cathy Smith	Feature 1	33	2018-01-01
Cathy Smith	Feature 1	37	2018-01-02
Jason Poser	Feature 2	164	2018-01-01
Cathy Smith	Feature 3	“yes”	2018-01-01



Customer	Date	Feature 1	Feature 2	Feature 3
Cathy Smith	2018-01-01	33	null	1
Cathy Smith	2018-01-02	37	null	7
Jason Poser	2018-01-01	null	164	13

- Requires expensive pivot to turn into tabular format used in training/scoring*

*Naively on HDFS

Tabular

- Relational database table structure

Tabular

- Relational database table structure

Customer	Date	Feature 1	Feature 2
Cathy Smith	2018-01-01	33	null
Cathy Smith	2018-01-02	37	null
Jason Poser	2018-01-01	null	164

Tabular

- Relational database table structure

Customer	Date	Feature 1	Feature 2
Cathy Smith	2018-01-01	33	null
Cathy Smith	2018-01-02	37	null
Jason Poser	2018-01-01	null	164

+

Customer	Date	Feature 3
Cathy Smith	2018-01-01	1
Cathy Smith	2018-01-02	7
Jason Poser	2018-01-01	13

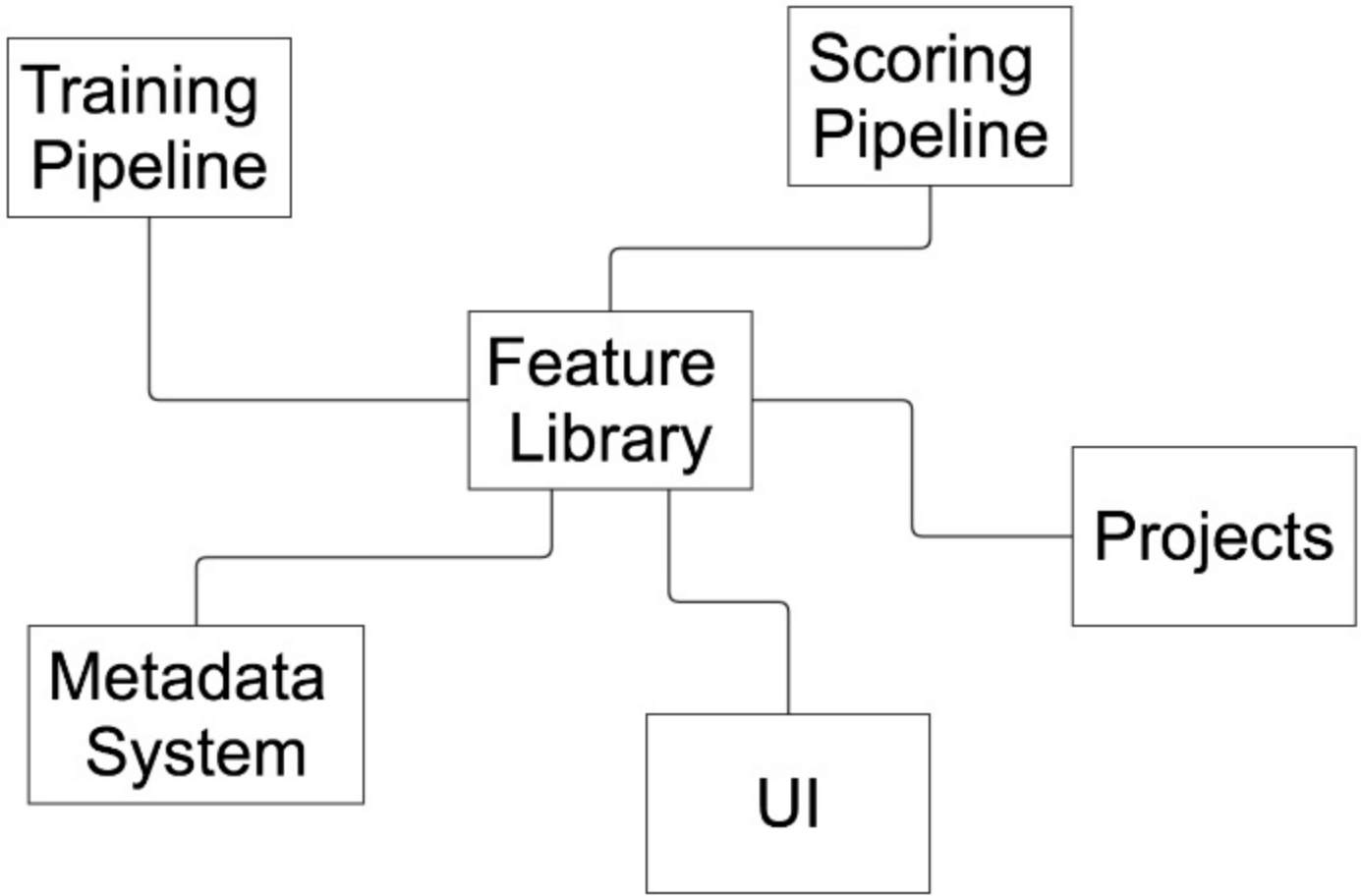
=

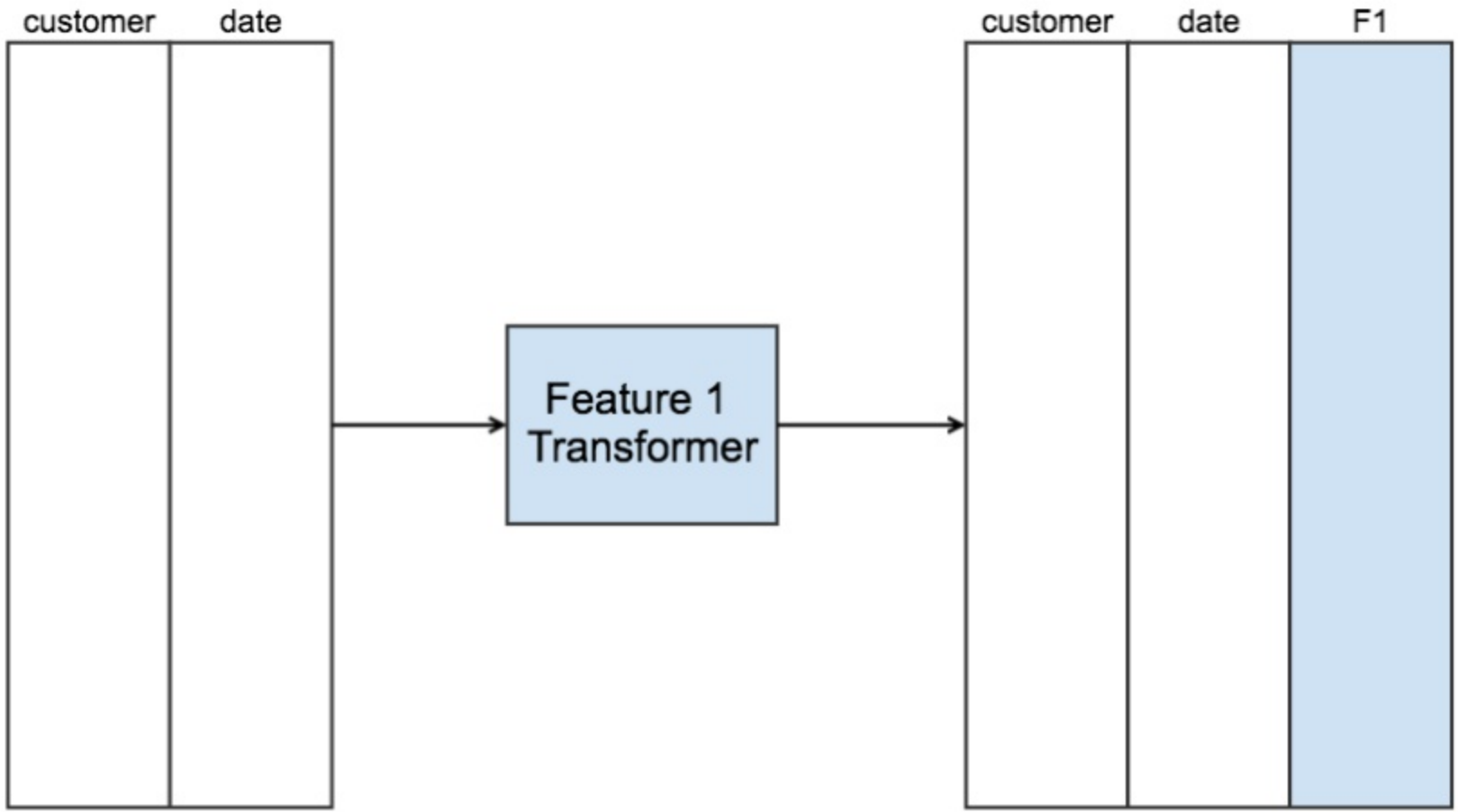
Customer	Date	Feature 1	Feature 2	Feature 3
Cathy Smith	2018-01-01	33	null	1
Cathy Smith	2018-01-02	37	null	7
Jason Poser	2018-01-01	null	164	13

- Adding new feature requires a join

Context

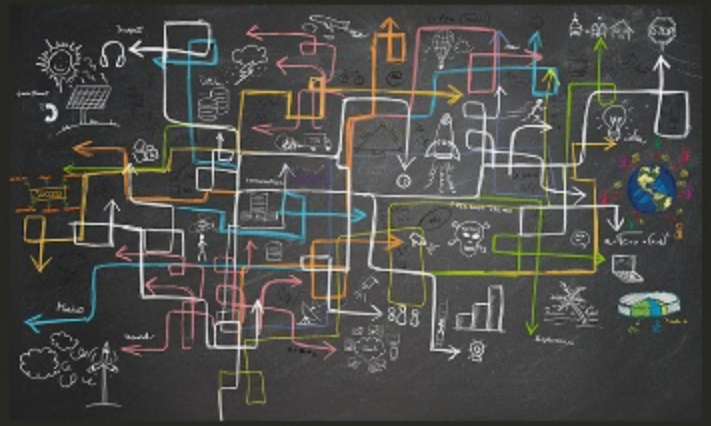
- On premises data centre (often fighting for cluster resources)
- Hortonworks 2.5 (Spark 1.6.3, Hive 1.2.1, HBase 1.1.2)


Training Pipeline
Scoring Pipeline
Feature Library
Projects
Metadata System
UI



```scala
object MySpecialFeature extends TelcoFeature {
    override val featureName = "my_special_feature"
    override val featureType = IntegerType()
    override val dependencies = Seq("my_other_special_feature")
    override val description = "The special feature which calculates the stuff"

    override def generate(dateRange: DateRange): Transformer =
        Feature.create(featureName, featureType, dependencies) { (df: DataFrame) =>
            df.withColumn(featureName, dateDiff(col("my_other_special_feature_column"), col("date")))
        }
}
```

```scala
object MySpecialFeature extends TelcoFeature {
    override val featureName = "my_special_feature"
    override val featureType = IntegerType()
    override val dependencies = Seq("my_other_special_feature")
    override val description = "The special feature which calculates the stuff"

    override def generate(dateRange: DateRange): Transformer =
        Feature.create(featureName, featureType, dependencies) { (df: DataFrame) =>
            df.withColumn(featureName, dateDiff(col("my_other_special_feature_column"), col("date")))
        }
}
```



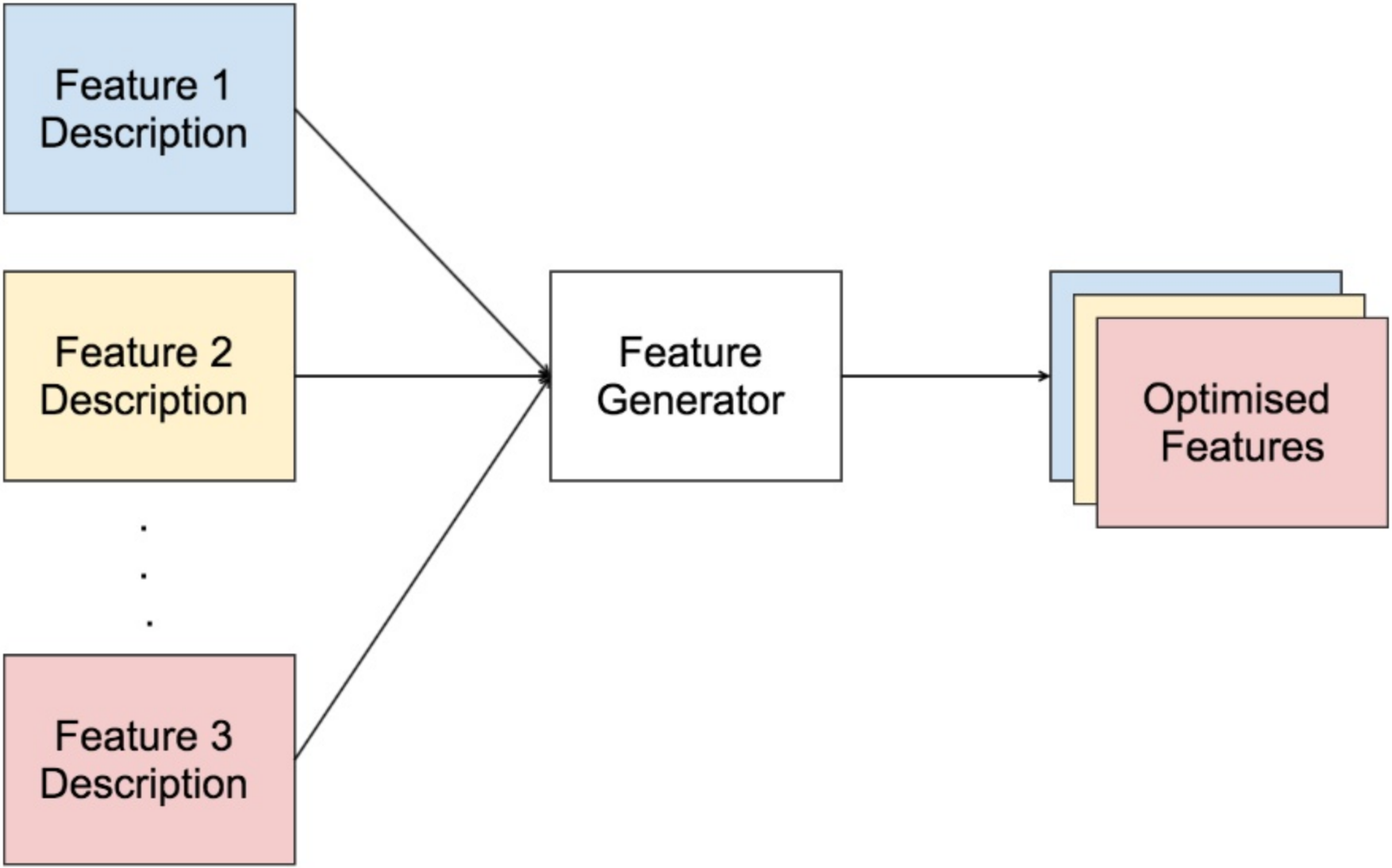
```scala
object MySpecialFeature extends TelcoFeature {
    override val featureName = "my_special_feature"
    override val featureType = IntegerType()
    override val dependencies = Seq("my_other_special_feature")
    override val description = "The special feature which calculates the stuff"

    override def generate(dateRange: DateRange): Transformer =
        Feature.create(featureName, featureType, dependencies) { (df: DataFrame) =>
            df.withColumn(featureName, dateDiff(col("my_other_special_feature_column"), col("date")))
        }
}
```

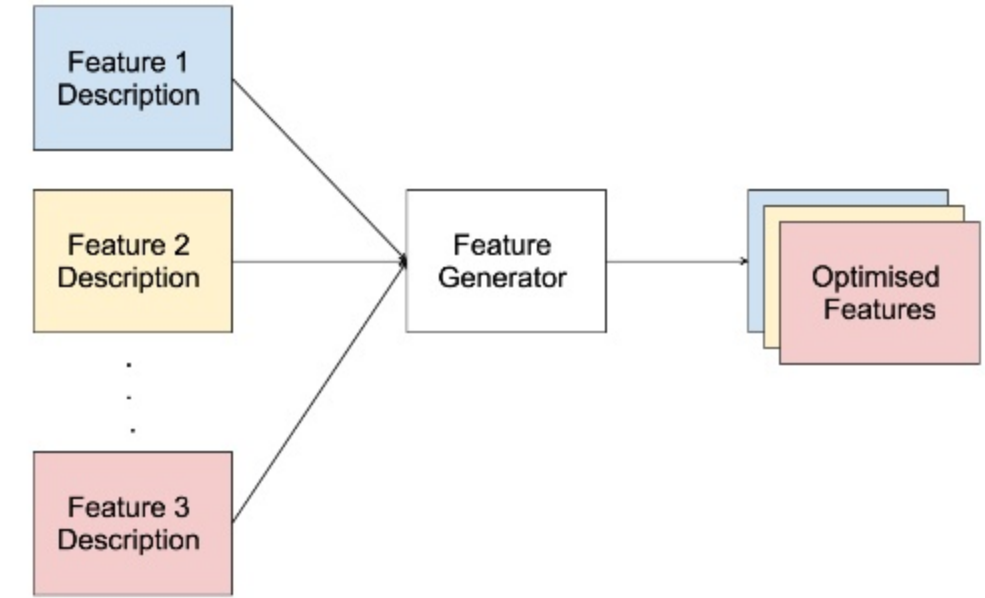
```scala
override def transform(dataset: DataFrame): DataFrame = {
    val sqlC = dataset.sqlContext
    import sqlC.implicits._
    val myDataSource = sqlC.sparkContext
        .parallelize(Seq(("a", 23), ("b", 24), ("c", 36), ("d", 52)))
        .toDF("customer", "stuff")
    myDataSource.registerTempTable("myTable")

    val result = sqlC.sql(
      """
        |SELECT '%c%' as Chapter,
        |SUM(CASE WHEN ticket.status IN ('new','assigned') THEN 1 ELSE 0 END) as `New`,
        |...
        |SUM(CASE WHEN ticket.status='closed' THEN 1 ELSE 0 END) as 'Closed',
        |count(id) AS Total,
        |ticket.id AS _id
        |   FROM engine.ticket
        |INNER JOIN engine.ticket_custom
        |   ON ticket.id = ticket_custom.ticket
        |WHERE ticket_custom.name='chapter'
        |   AND ticket_custom.value LIKE '%c%'
        |   AND type='New material'
        |   AND milestone='1.1.12'
        |   AND component NOT LIKE 'internal_engine'
        |GROUP BY ticket.id
      """.stripMargin)

    dataset.join(result, Seq("customer"), "left_outer")
  }
```

```scala
override def transform(dataset: DataFrame): DataFrame = {
    val sqlC = dataset.sqlContext
    import sqlC.implicits._
    val myDataSource = sqlC.sparkContext
        .parallelize(Seq(("a", 23), ("b", 24), ("c", 36), ("d", 52)))
        .toDF("customer", "stuff")
    myDataSource.registerTempTable("myTable")

    val result = sqlC.sql(
      """
        |SELECT '%c%' as Chapter,
        |SUM(CASE WHEN ticket.status IN ('new','assigned') THEN 1 ELSE 0 END) as `New`,
        |...
        |SUM(CASE WHEN ticket.status='closed' THEN 1 ELSE 0 END) as 'Closed',
        |count(id) AS Total,
        |ticket.id AS _id
        |   FROM engine.ticket
        |INNER JOIN engine.ticket_custom
        |   ON ticket.id = ticket_custom.ticket
        |WHERE ticket_custom.name='chapter'
        |   AND ticket_custom.value LIKE '%c%'
        |   AND type='New material'
        |   AND milestone='1.1.12'
        |   AND component NOT LIKE 'internal_engine'
        |GROUP BY ticket.id
      """.stripMargin)

    dataset.join(result, Seq("customer"), "left_outer")
  }
```

Back to the Drawing Board

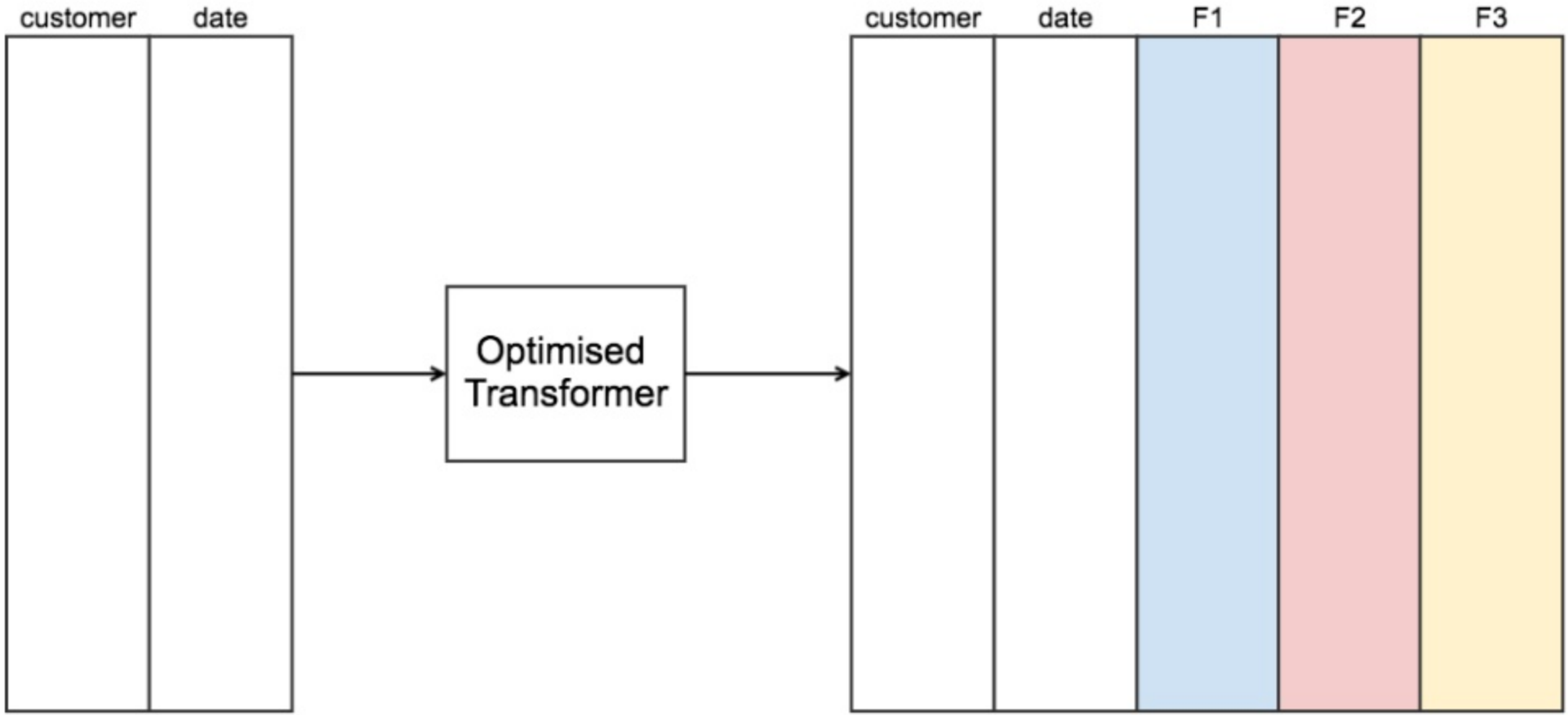
- Too complex to use
- Too many ways that users can make mistakes
- Need a higher level declarative language
- Need to globally optimise the set of features that are being run and utilise domain knowledge to help spark optimise properly without lots of fine tuning
- Needs to support SQL (or some subset) without losing the ability to optimise things ourselves


Feature 1
Description
Feature 2
Description
Feature 3
Description
Feature
Generator
Optimised
Features



Domain Optimisations

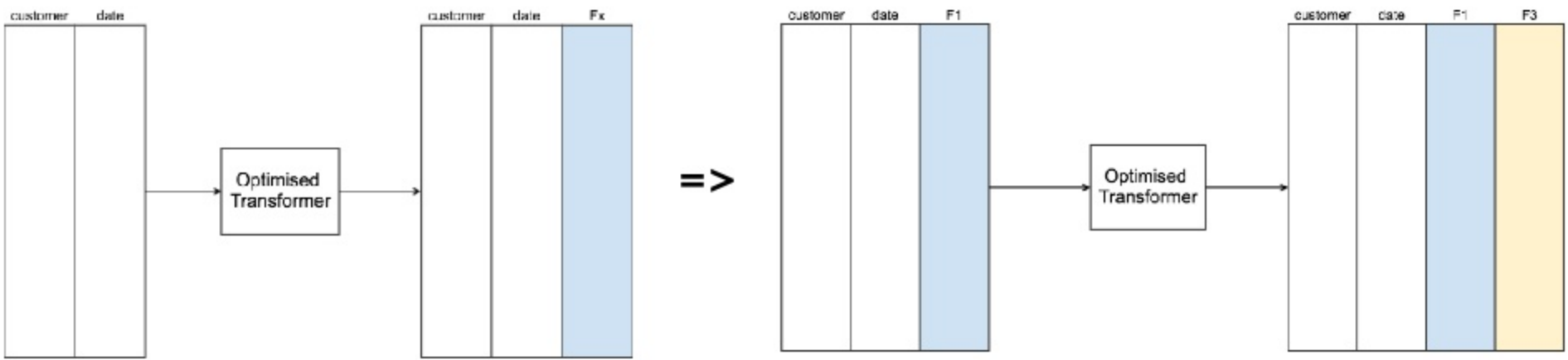
Spark DAG Optimisations


customer
date
Optimised
Transformer
customer
date
F1
F2
F3

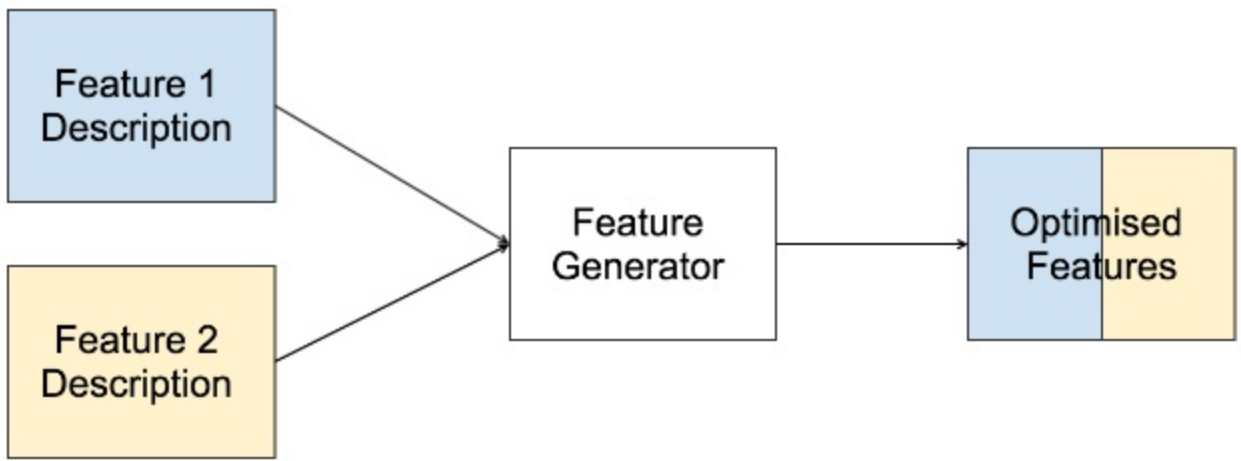
Optimisation

- Most optimisation techniques are about **reducing IO, sharing compute** and **sharing caching** where possible.
  - Expose interfaces to make smart use of hdfs partitioning to avoid reading data
  - Combine multiple features that read from the same table into a single big read rather than one read per feature
  - Cache common aggregates across features to avoid unnecessary re-computation

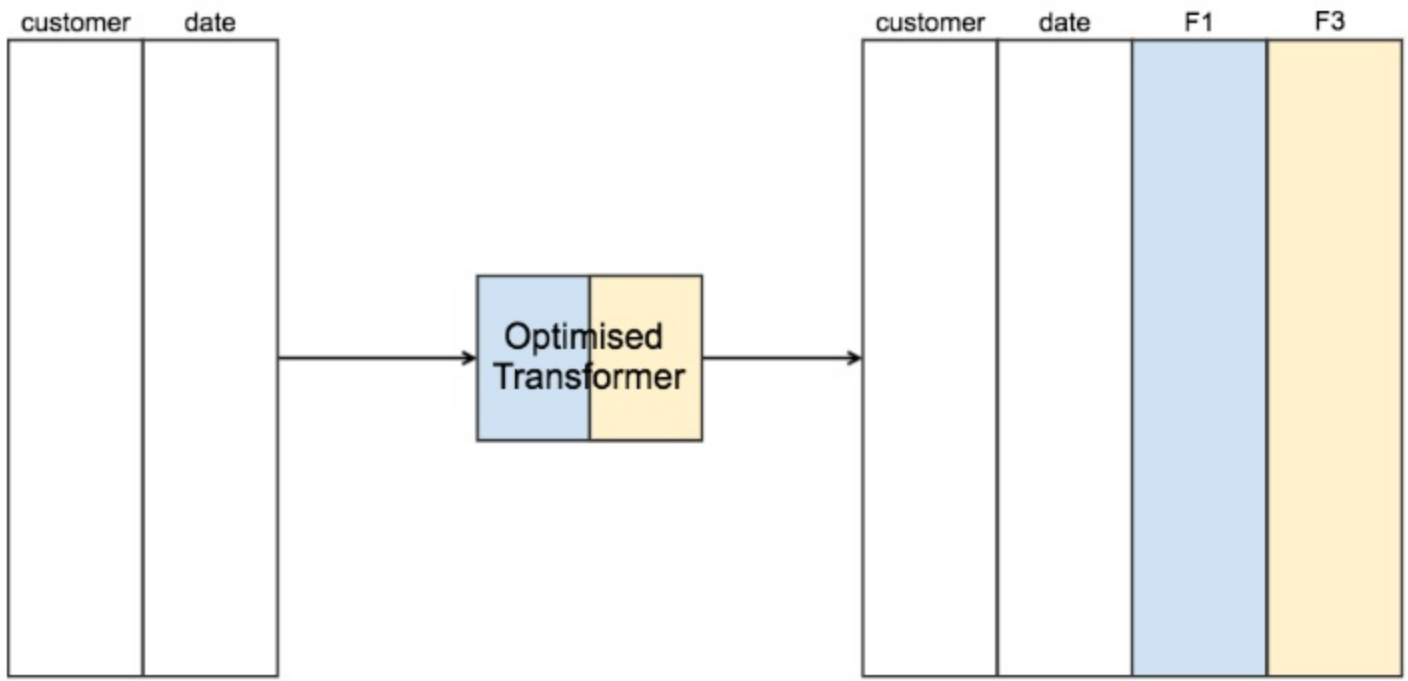
```
select col1 from table;

select col2 from table;
```



```
select col1, col2 from table;
```



```
select col1, col2 from table;
```

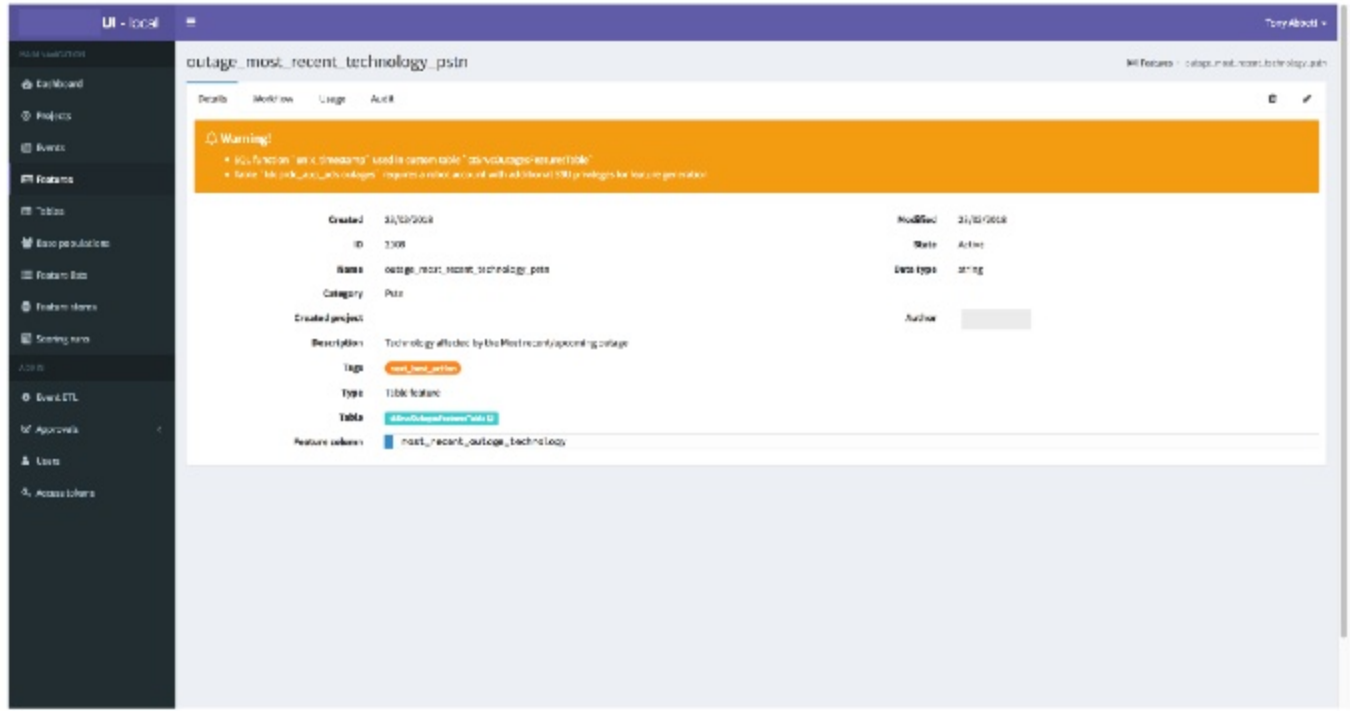


```scala
object MySpecialFeature extends TelcoFeature {
    override val featureName = "my_special_feature"
    override val featureType = LongType
    override val tags = Seq(tag1, tag2)
    override val description = "The number of calls a customer made in the last 3 months"
    override val author = "Joe Bloggs"

    override def generate(dateRange: DateRange): Transformer =
        Feature.create(featureName, featureType, dependencies) { (df: DataFrame) =>
            val sqlC = df.sqlContext
            import sqlC.implicits._
            val telcoData = sqlC.sql("""
                |select num_calls_in_the_last_3_months, customer_id, date
                |from telco_customer_table
                |where date between(${dateRange.start}, ${dateRange.end})
                """.stripMargin)
            df.join(telcoData, Seq("customer_id", "date"), "left")
        }
}
```

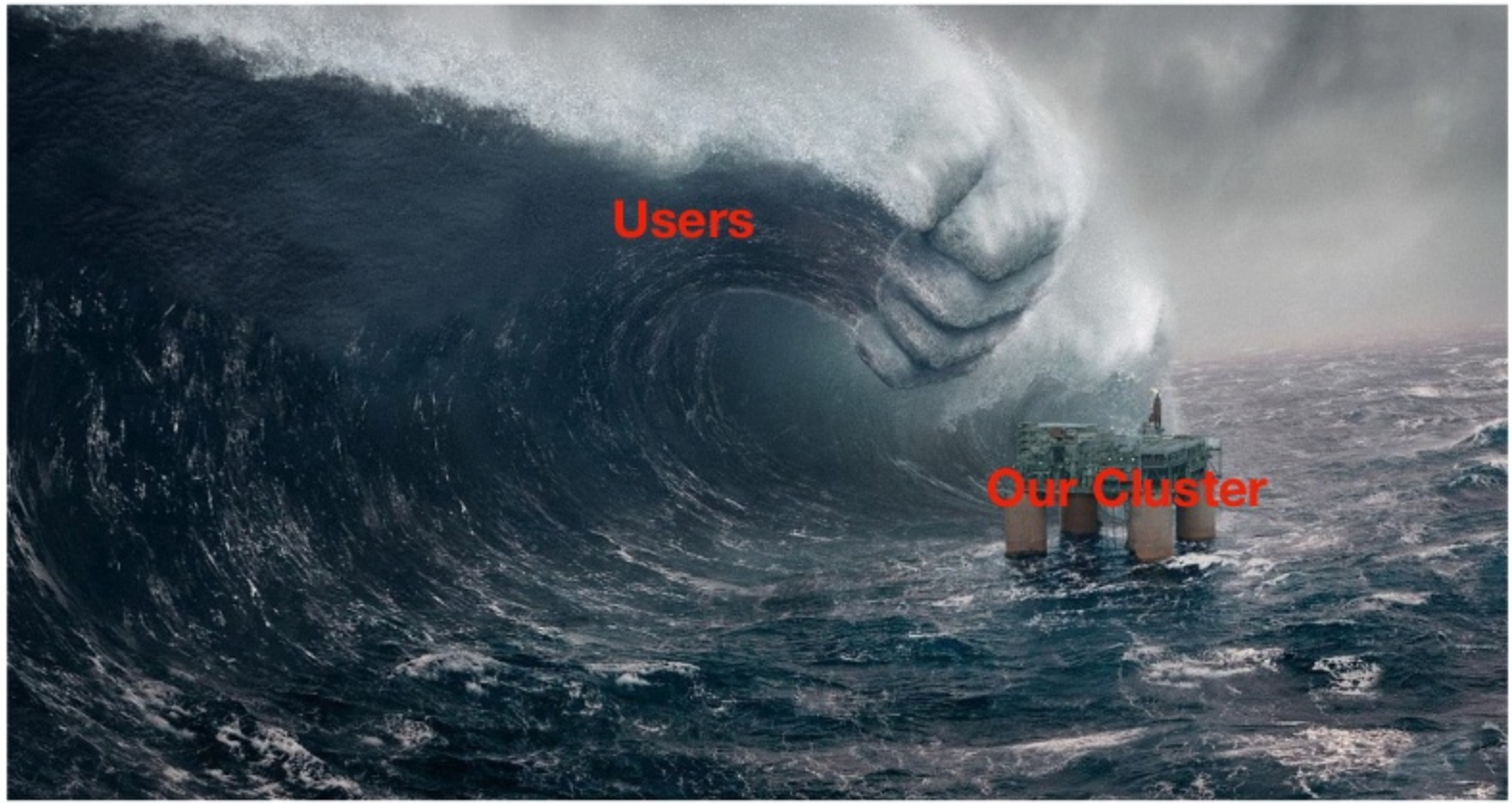
```
val mySpecialFeature = ColumnFeature {
    col("num_calls_in_the_last_3_months"),
    TelcoCustomerTable,
    FeatureMetadata {
        name = "my_special_feature",
        `type` = LongType,
        tags = Seq(tag1, tag2),
        description = "The number of calls a customer made in the last 3 months",
        author = "Joe Bloggs"
    }
```

```
val numberOfViewsInLastNDays = (windows: List[Long]) => WindowedEventFeature.count(
  windowInDays = windows,
  eventName = "landing_page_views",
  featureNamePrefix = "landing_page_views_last",
  description = "The count of landing page views over the last ${windows} days",
  author = "Robert'); DROP TABLE Customers;--"
)
```

UI



- Features
  - Creation
  - Discovery
  - Metadata
  - Management
  - Promotion between environments (Dev/UAT/Prod)
- Projects
  - Define training + scoring feature sets


Users
Our Cluster


BACK IN MY DAY
WE GENERATED TRILLIONS OF FEATURES WITHOUT BREAKING A SWEAT
imgflip.com


Y'ALL GOT ANY MORE OF THEM
EXECUTORS?
imgflip.com

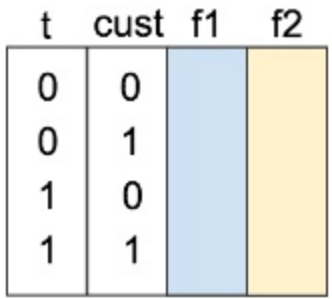
Cost

- Questions to ask
  - Does the feature store get a budget of its own?

Cost

- Questions to ask
  - Does the feature store get a budget of its own?
  - Do the projects pay for usage?

The Null Problem



The Null Problem

t	cust	f1	f2
0	0		
0	1		
1	0		
1	1		

t	cust	f1	f3
3	1		
3	2		
4	1		
4	2		

The Null Problem

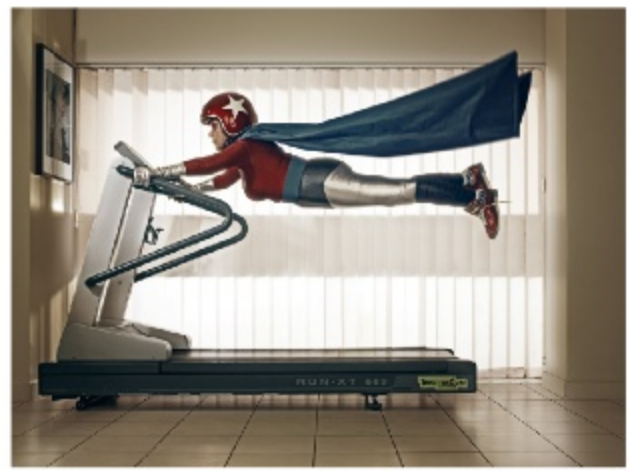
t	cust	f1	f2
0	0		
0	1		
1	0		
1	1		

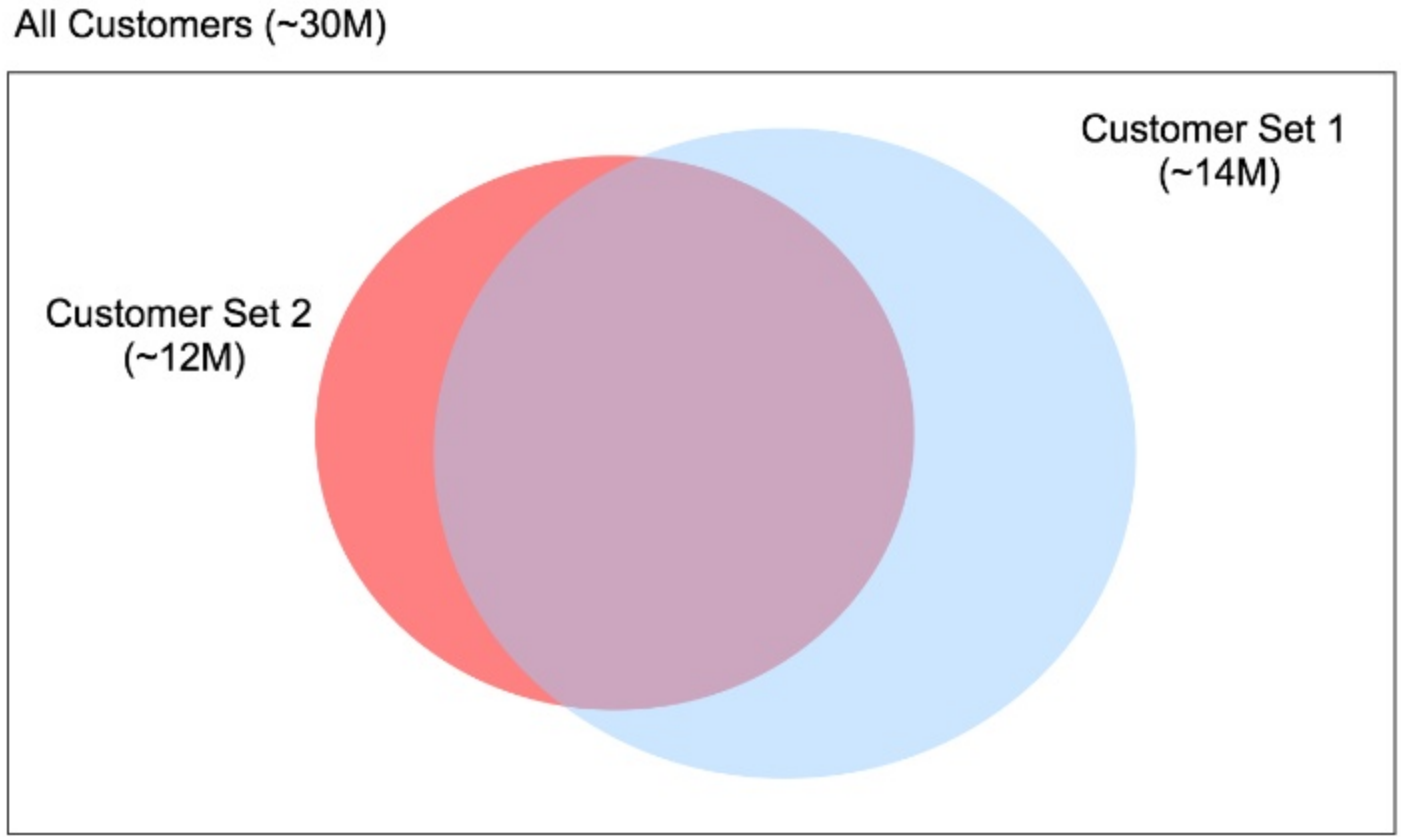
t	cust	f1	f3
3	1		
3	2		
4	1		
4	2		

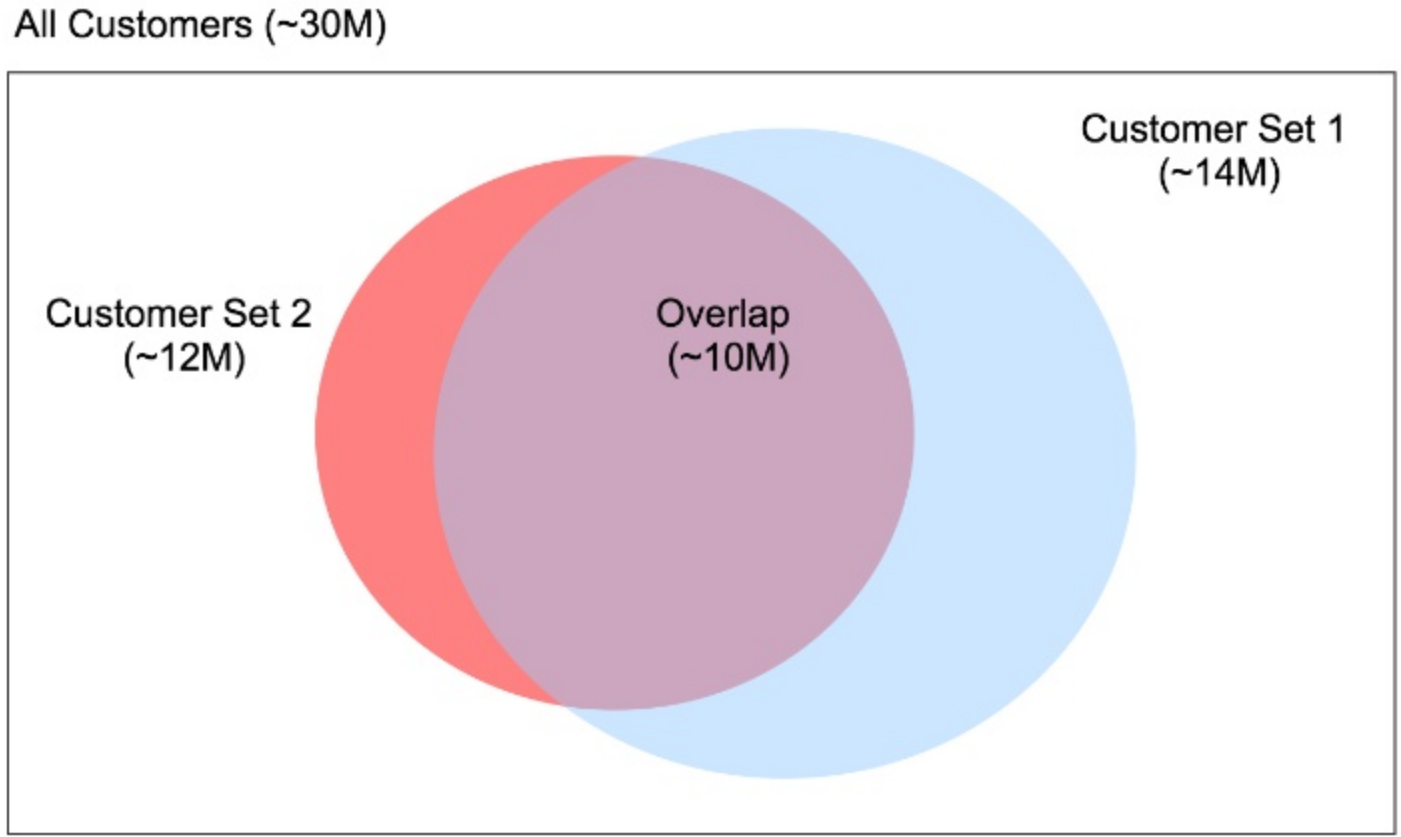
t	cust	f1	f2	f3
0	0			
0	1			
0	2			
1	0			
1	1			
1	2			
2	0			
2	1			
2	2			
3	0			
3	1			
3	2			
4	0			
4	1			
4	2			

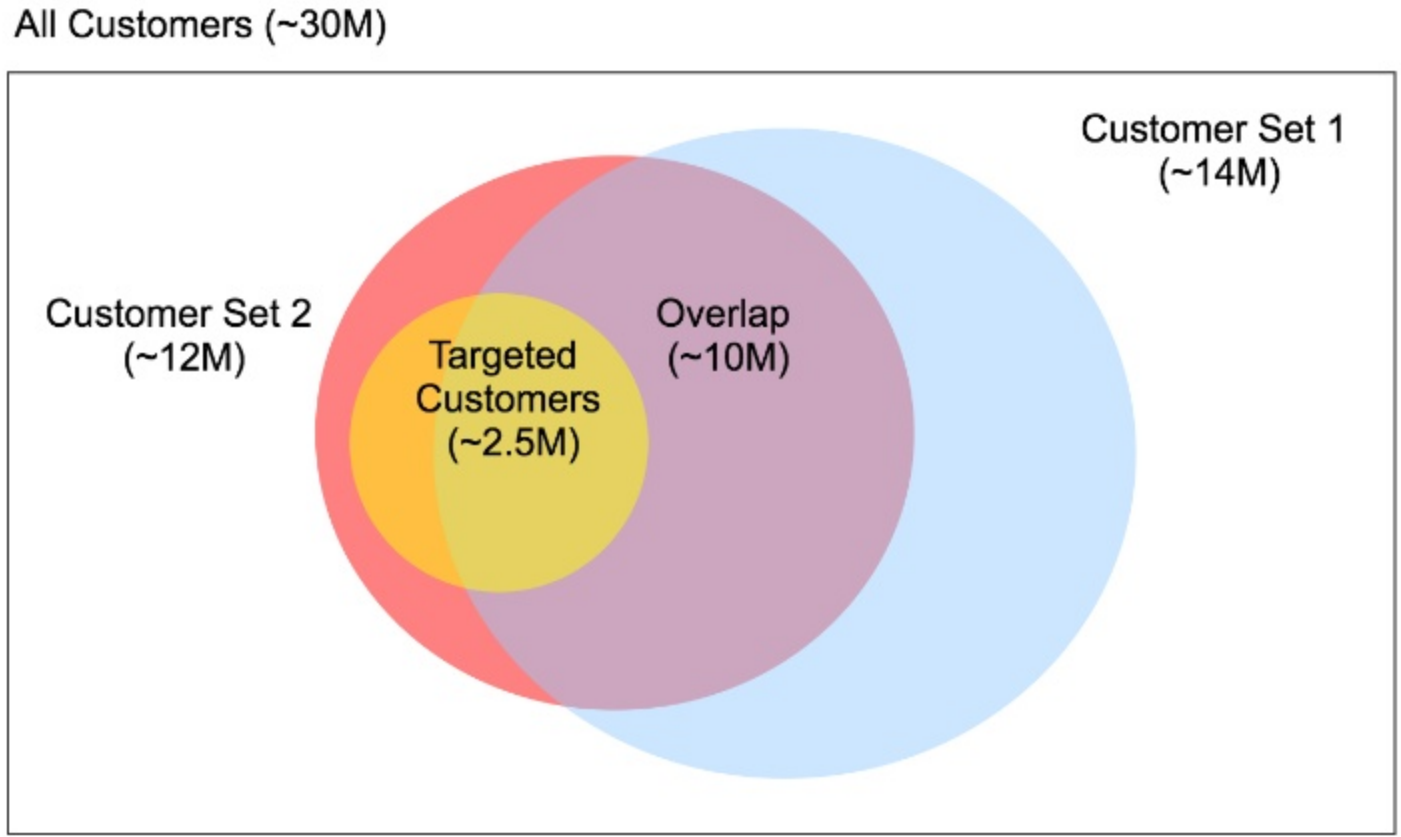
Optimisation

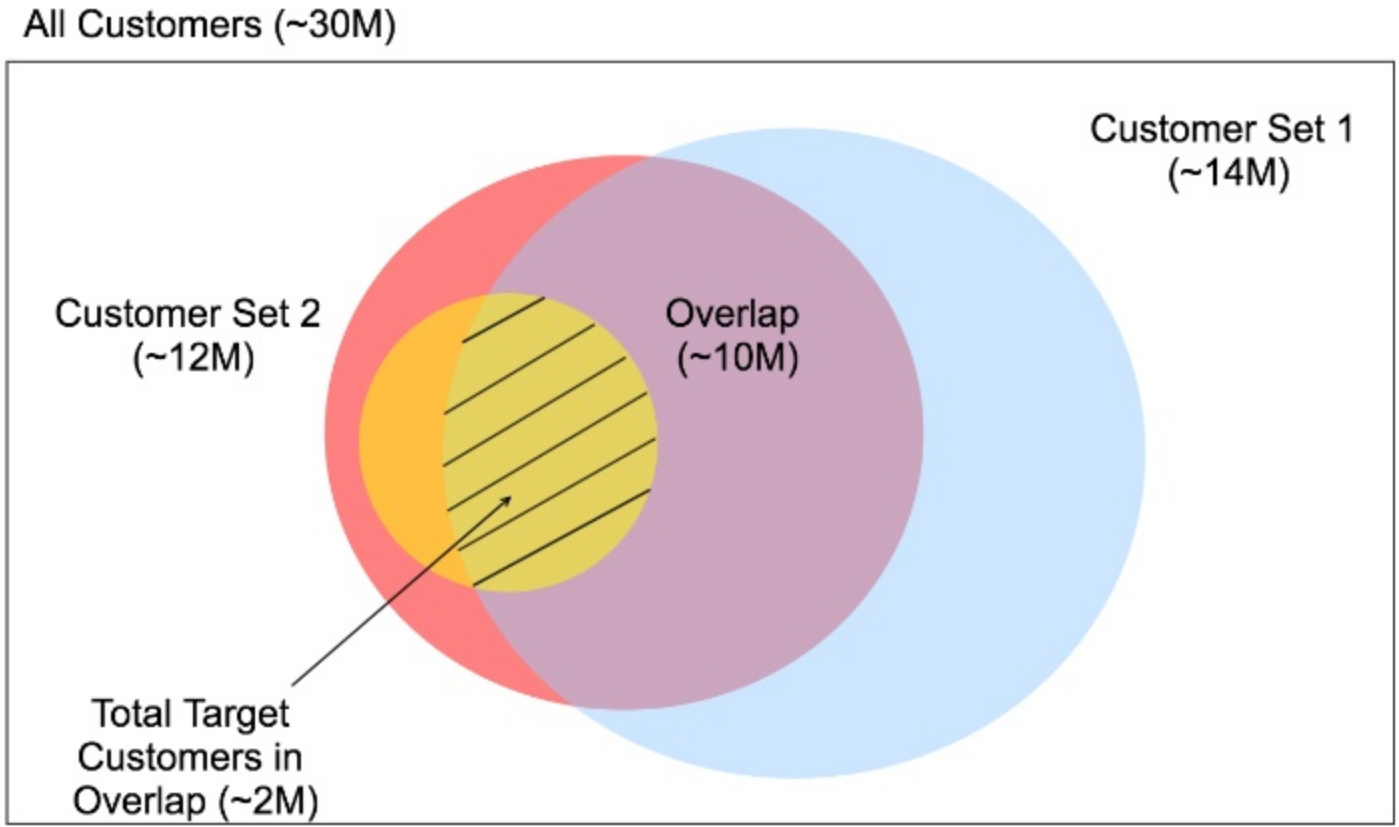
- Investigate the actual usage patterns of the feature table
- Spark DAG optimisation
- General Spark Performance Tuning
- Spark Version

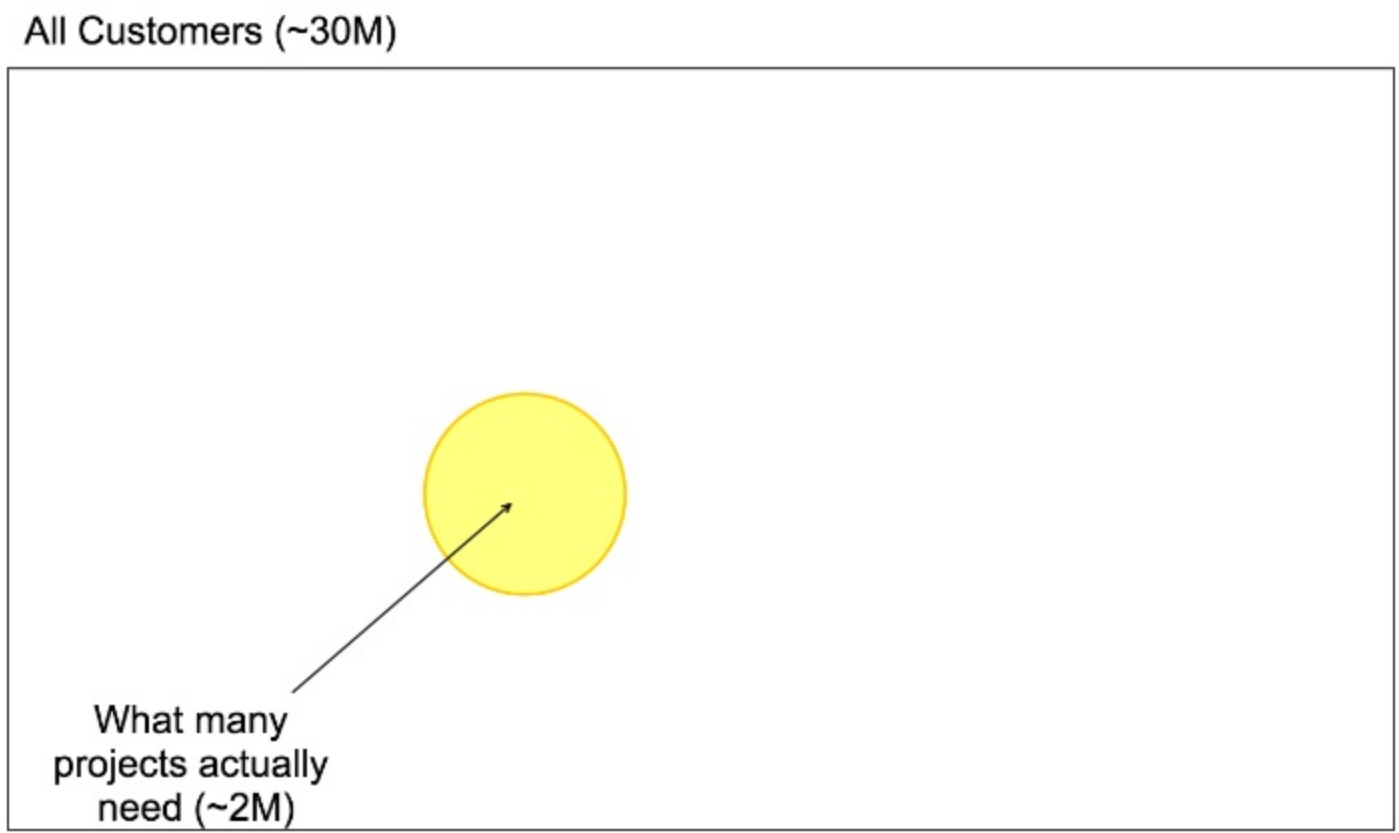





All Customers (~30M)
Customer Set 1
(~14M)
Customer Set 2
(~12M)
Overlap
(~10M)


All Customers (~30M)
Customer Set 1
(~14M)
Customer Set 2
(~12M)
Overlap
(~10M)
Targeted
Customers
(~2.5M)


All Customers (~30M)
Customer Set 1
(~14M)
Customer Set 2
(~12M)
Overlap
(~10M)
Total Target
Customers in
Overlap (~2M)



Back of the Envelope

- Full Table:
  - 30M users * 365 days * 2 years * 1000 features ~= 22 trillion data points

Back of the Envelope

- Full Table:
  - 30M users * 365 days * 2 years * 1000 features ~= 22 trillion data points
- Per Project:
  - 2M users * 365 days * 2 years * 300 features ~= 0.5 billion data points

Existing headaches

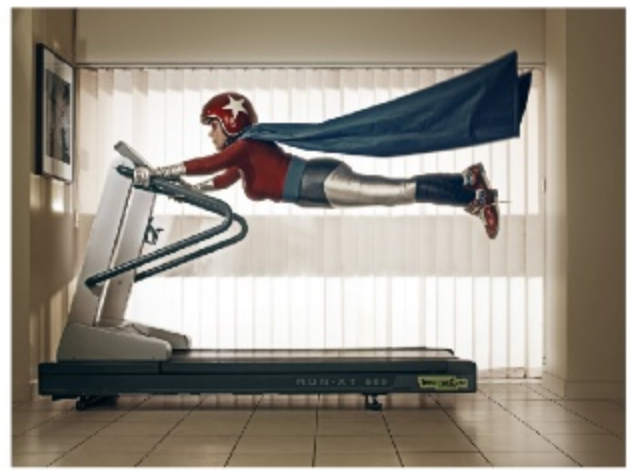
- Generating all features for all customers over the full time range is an enormous data challenge.
- You are doing way more work than is necessary for a lot of use cases.
- You are always increasing your compute burden as you add new features/ history.
- You need to join new features onto the table and keep it highly available (which typically means a copy of the table on hdfs)
- Users generally need to do a repartition to optimise their table for use anyway

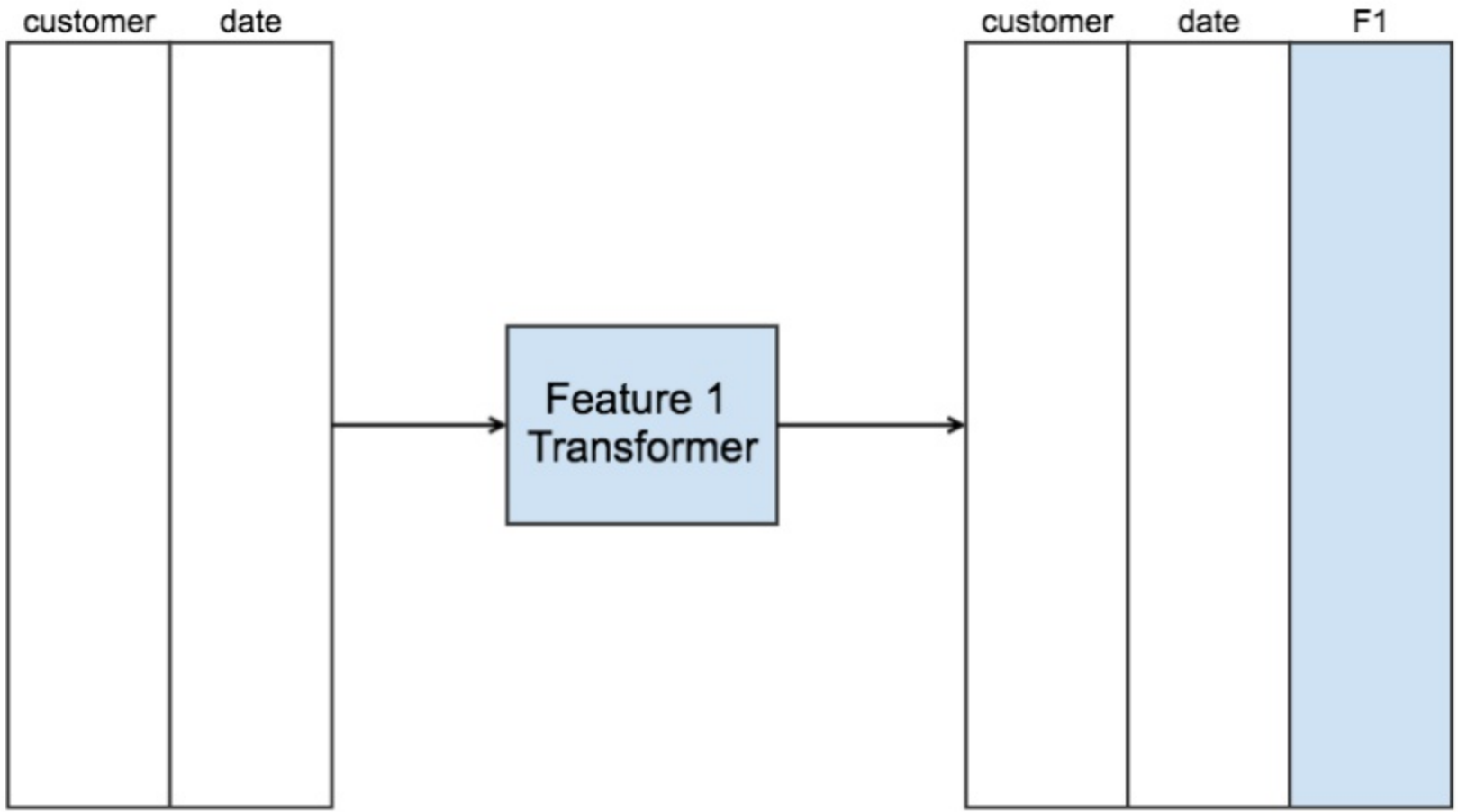
Crazy Idea?

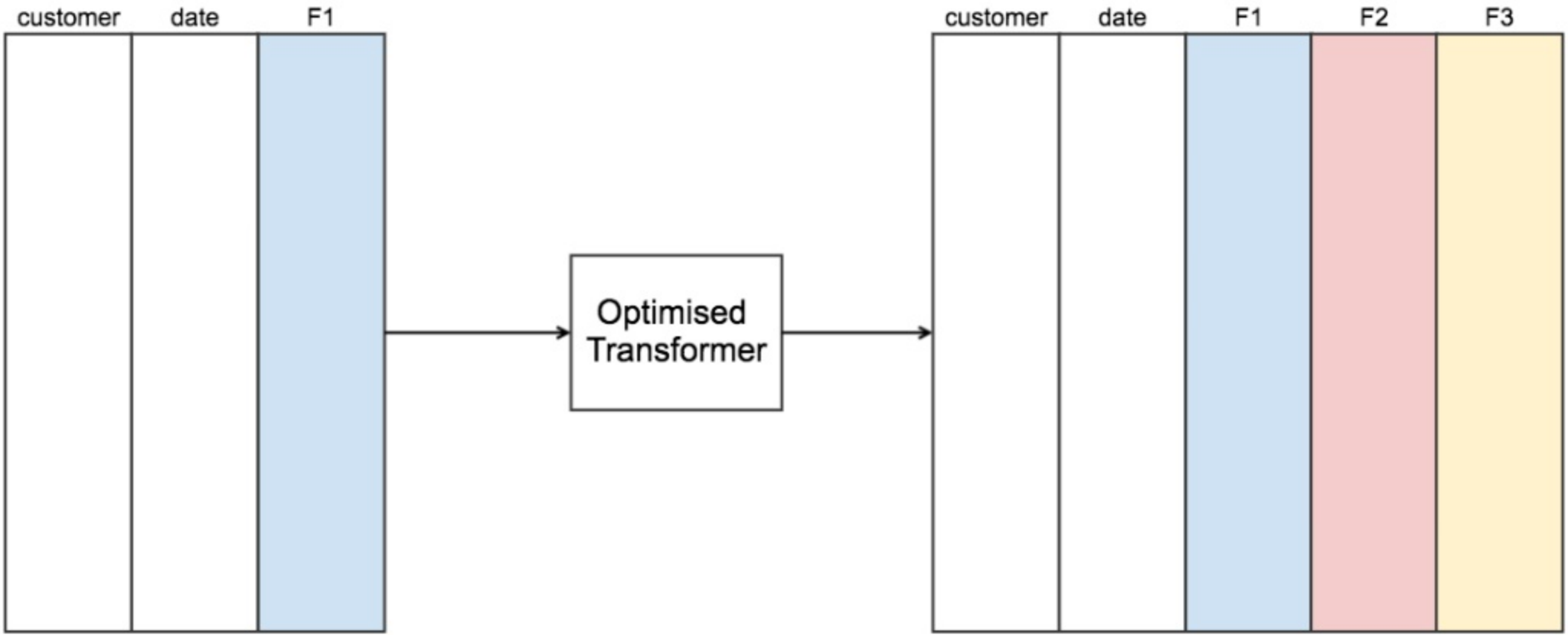
- **Generate training data sets on demand**
  - Removes an enormous engineering cost of building and maintaining a monstrous table that is mostly useless
  - Allows teams to get started on the modelling much faster since we were only generating the data they actually needed
  - Training data set is clean and efficiently partitioned on delivery

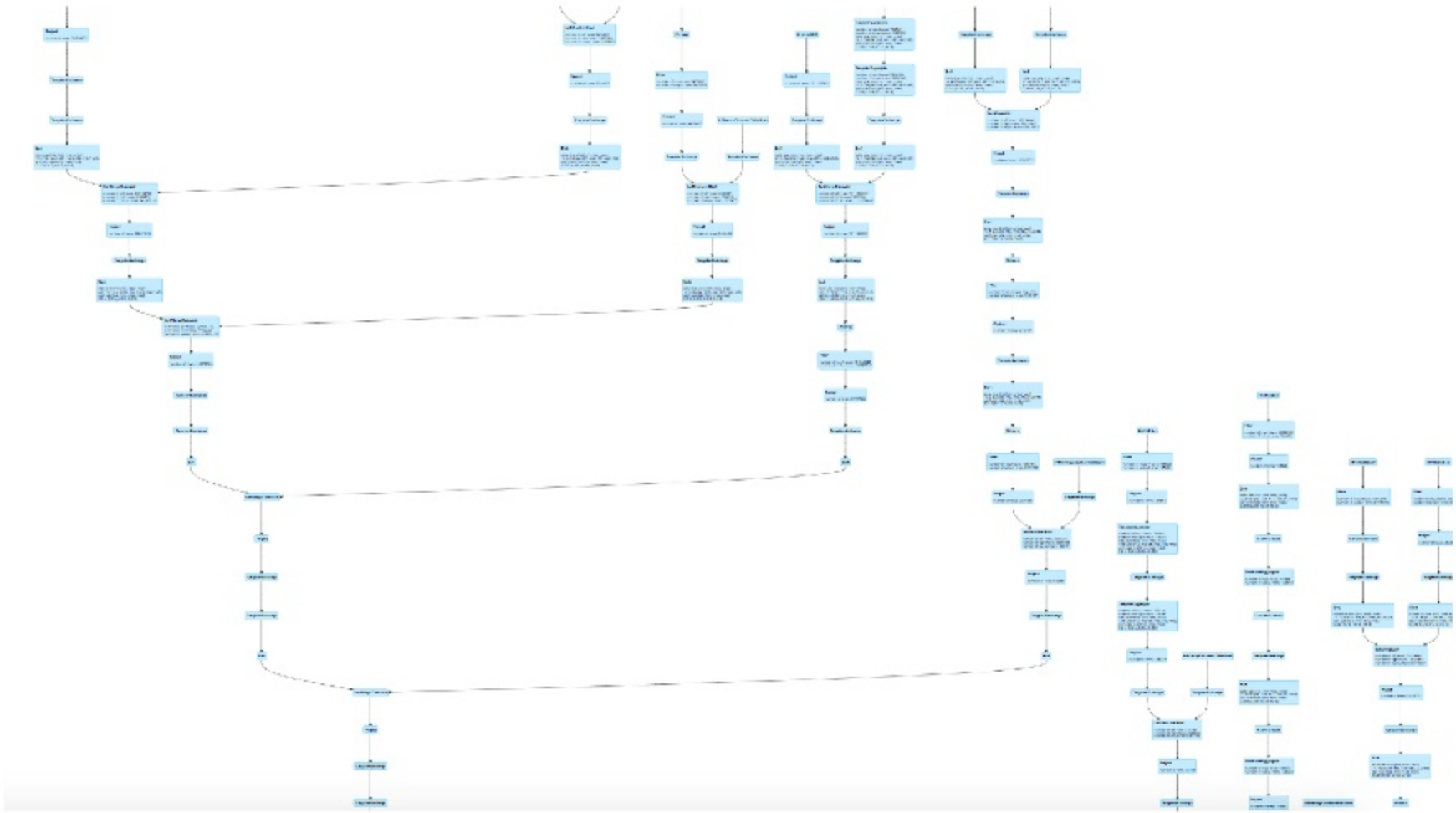
Optimisation

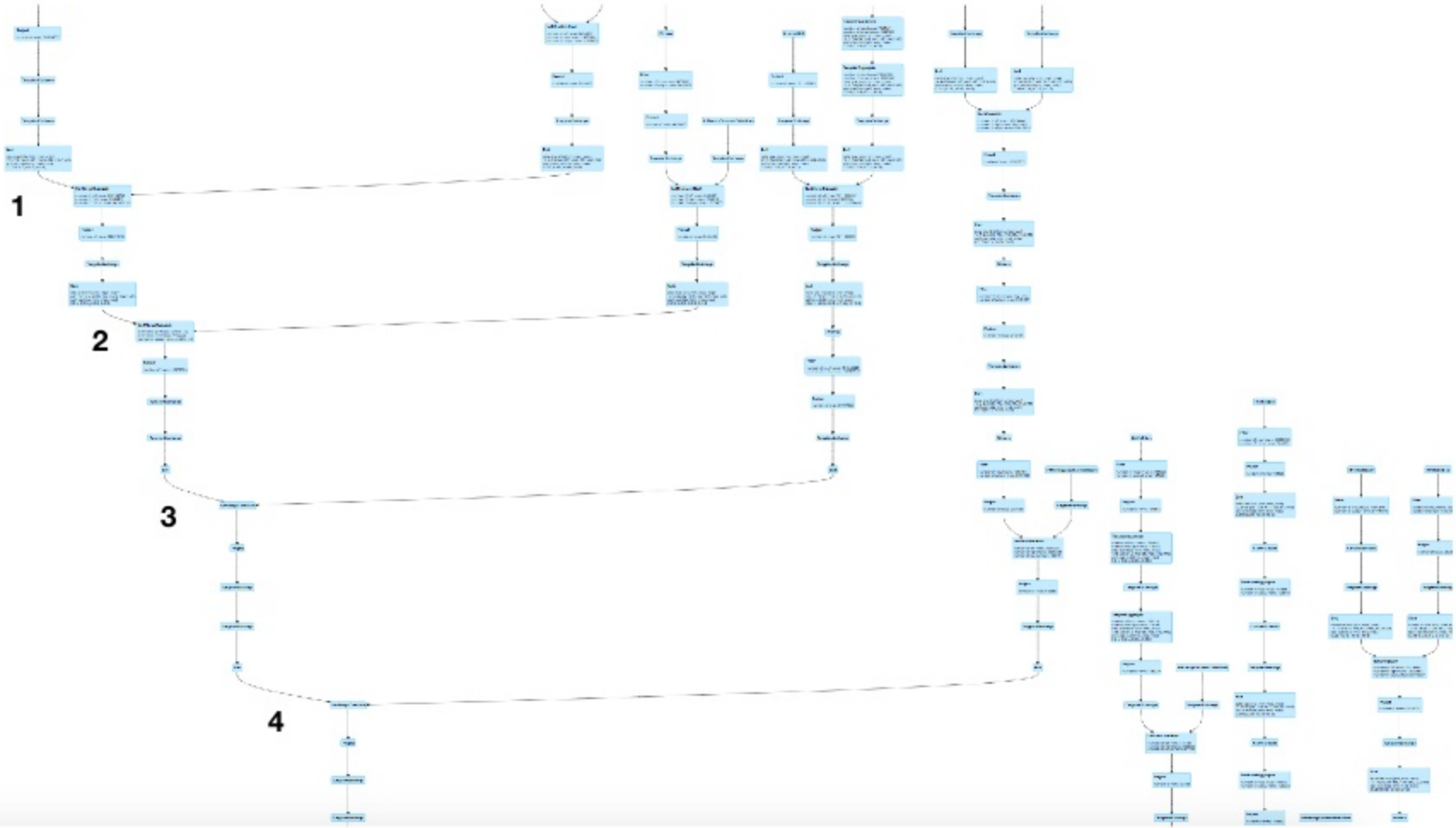
- Investigate the actual usage patterns of the feature table
- Spark DAG optimisation
- General Spark Performance Tuning
- Spark Version

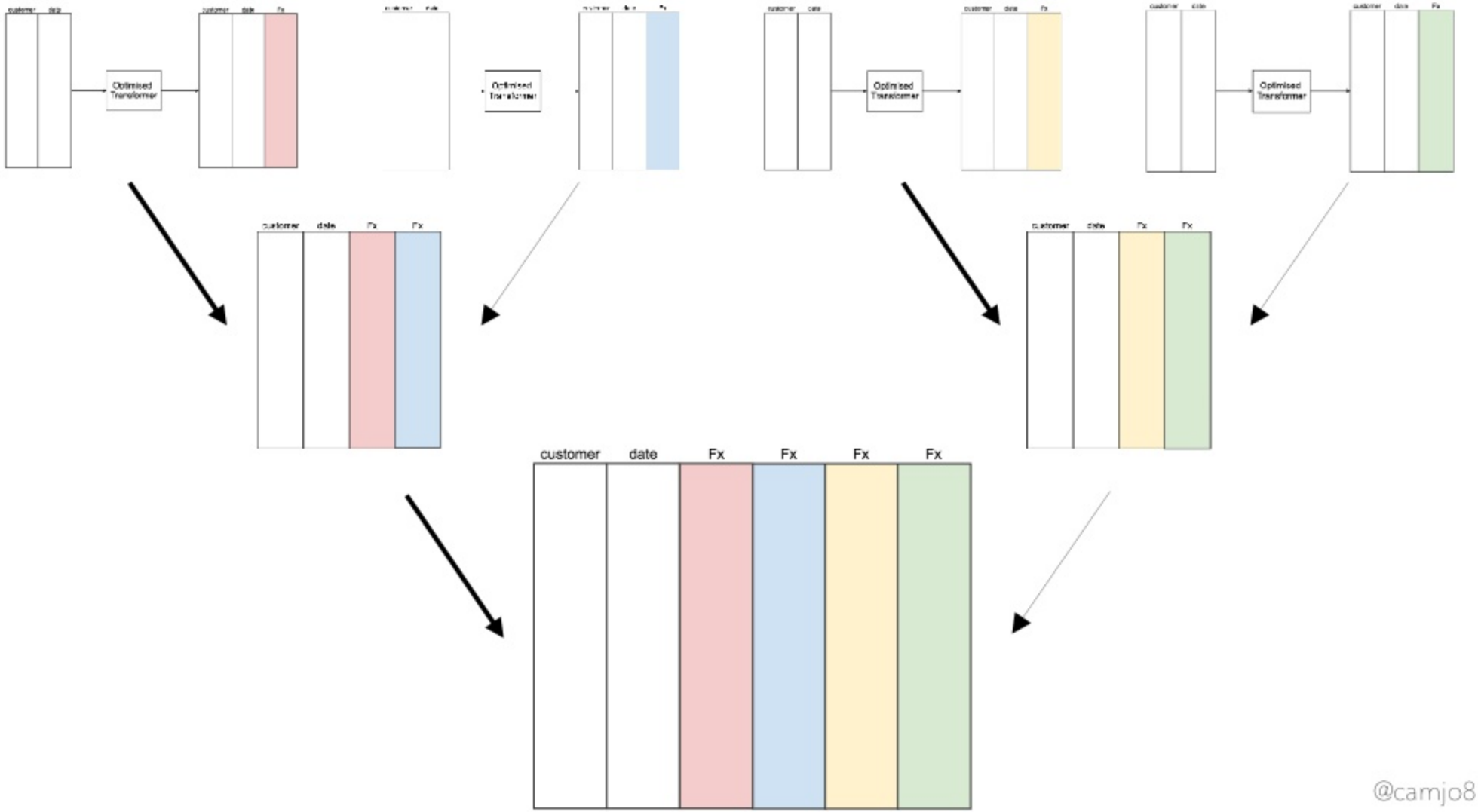


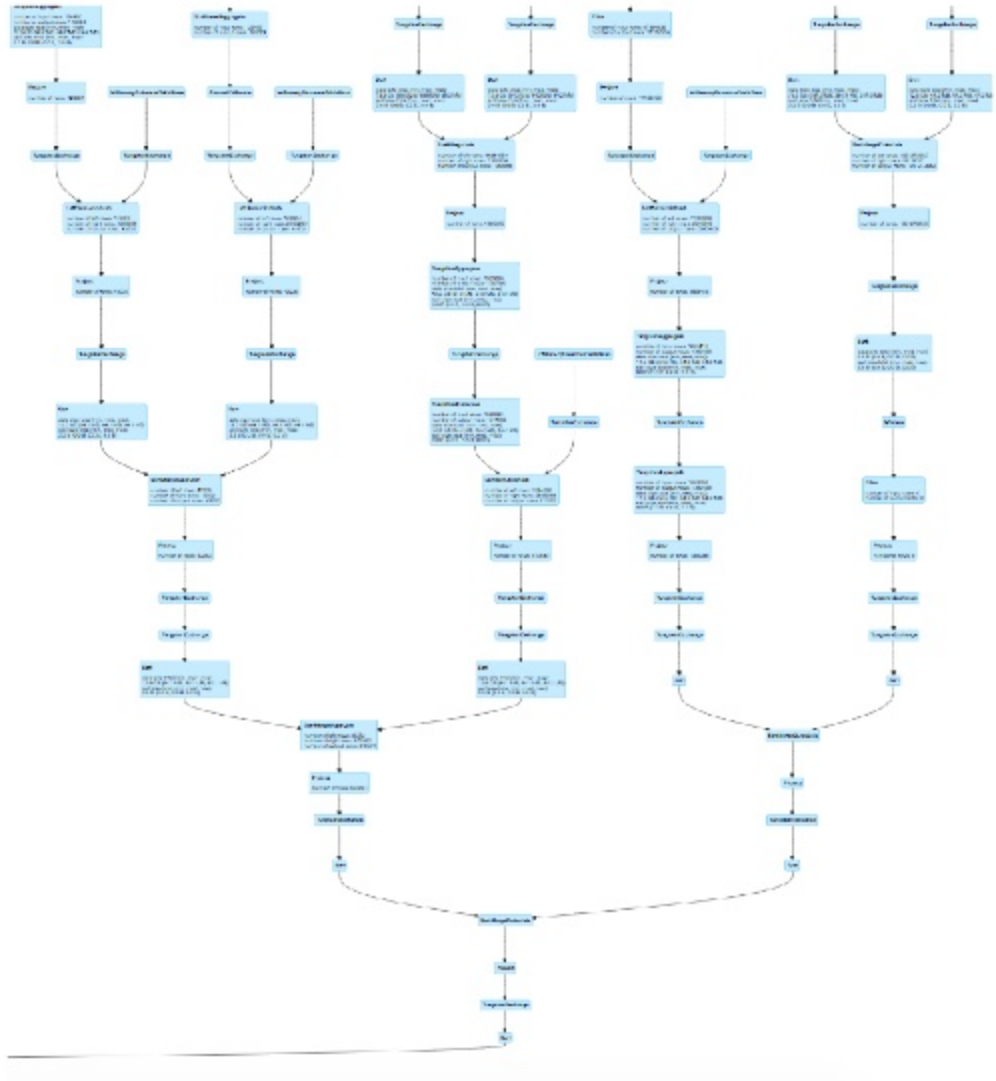


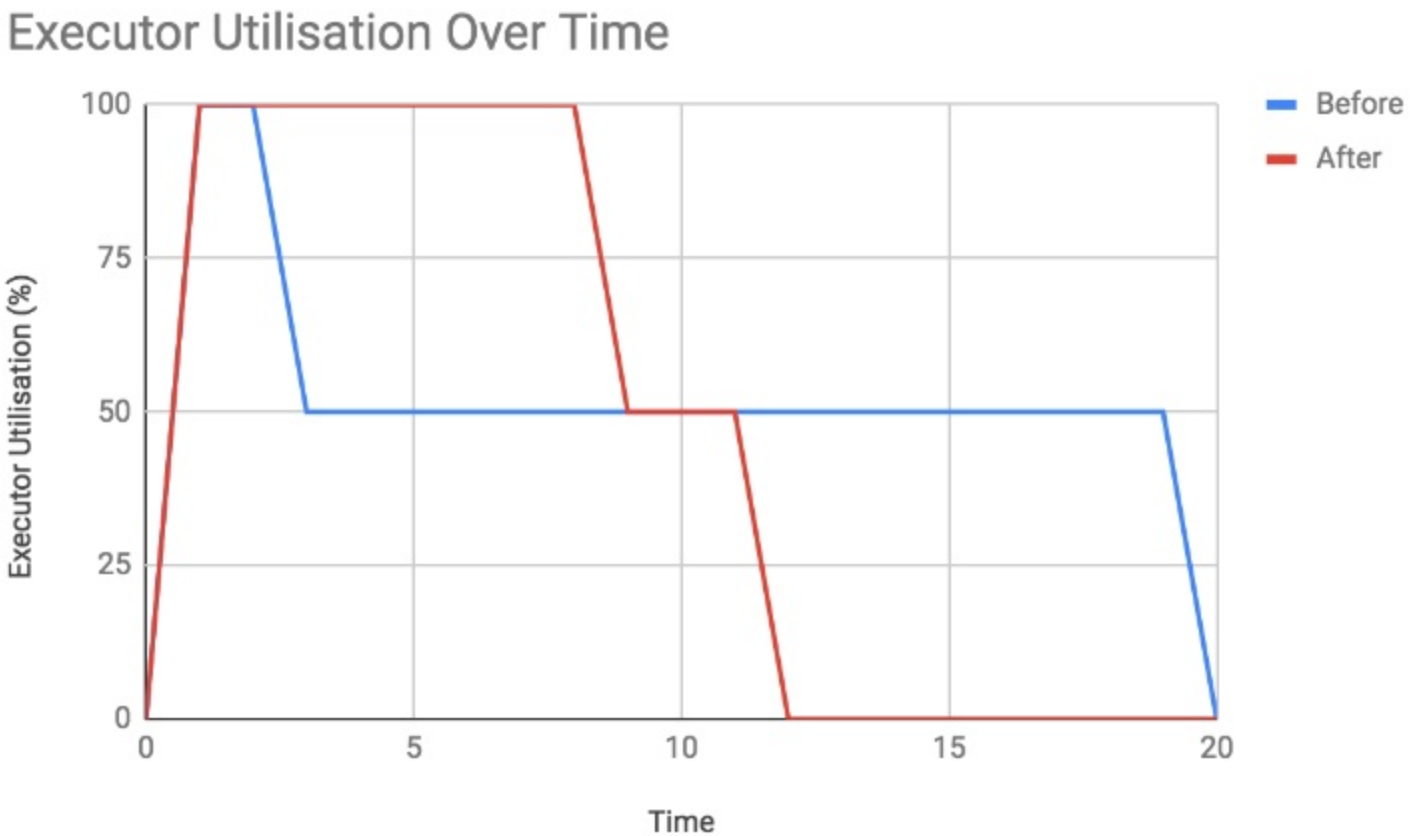

customer
date
F1
Optimised Transformer
customer
date
F1
F2
F3





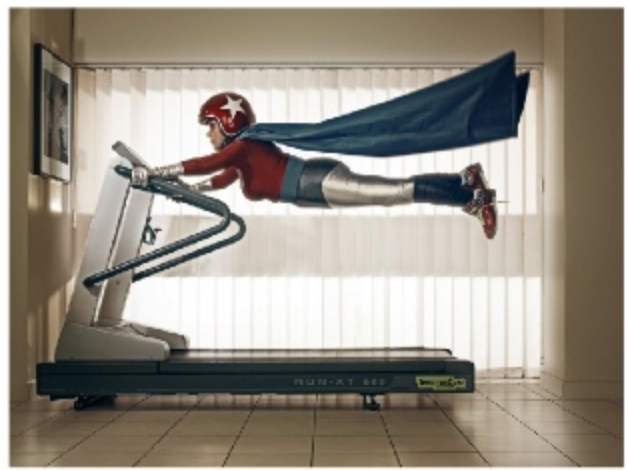

Optimised Transformer
Optimised Transformer
Optimised Transformer
Optimised Transformer
customer
date
Fx
Fx
Fx
Fx
Fx
Fx

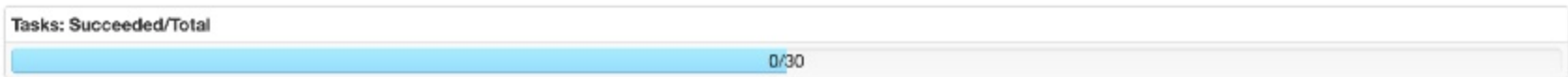


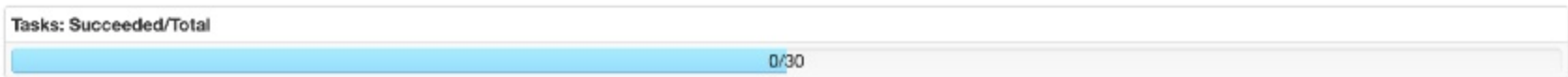

Executor Utilisation Over Time
Executor Utilisation (%)
100
75
50
25
0
0
5
10
15
20
Time
Before
After

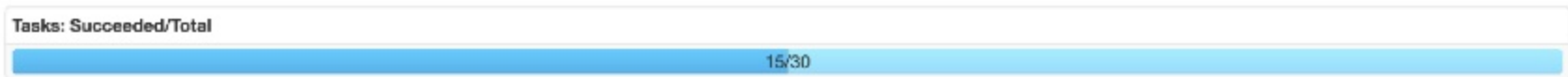
Optimisation

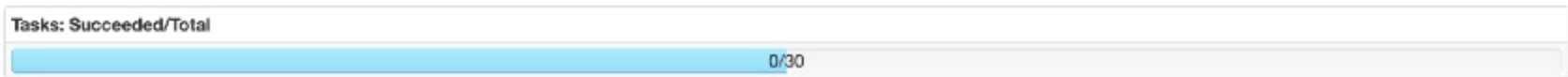
- Investigate the actual usage patterns of the feature table
- Spark DAG optimisation
- General Spark Performance Tuning
- Spark Version

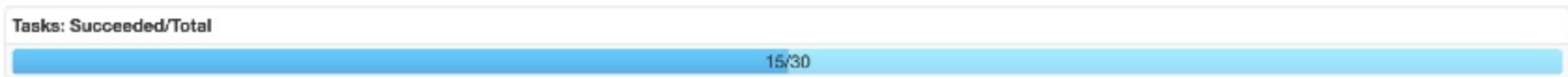




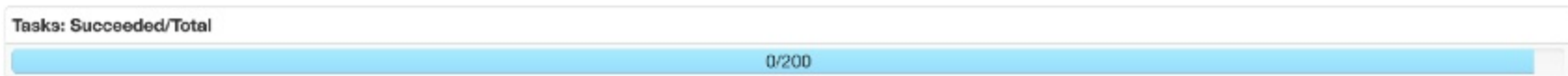


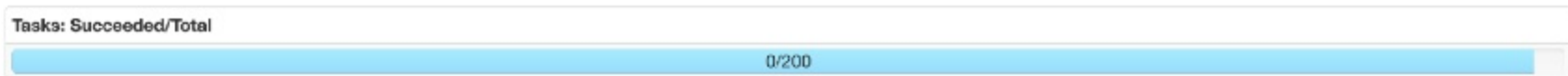


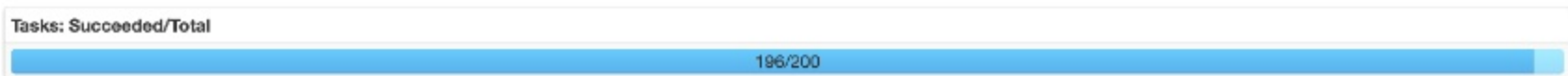


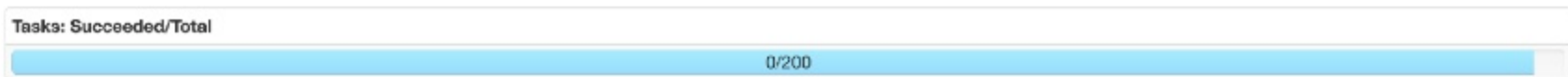


total time = 2T







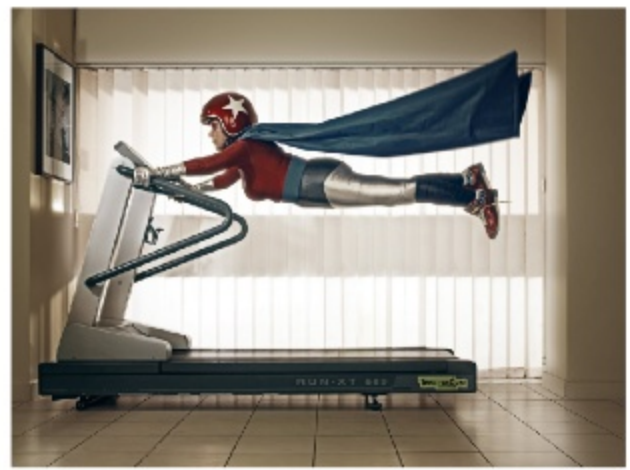


T

total time = 2T

Optimisation

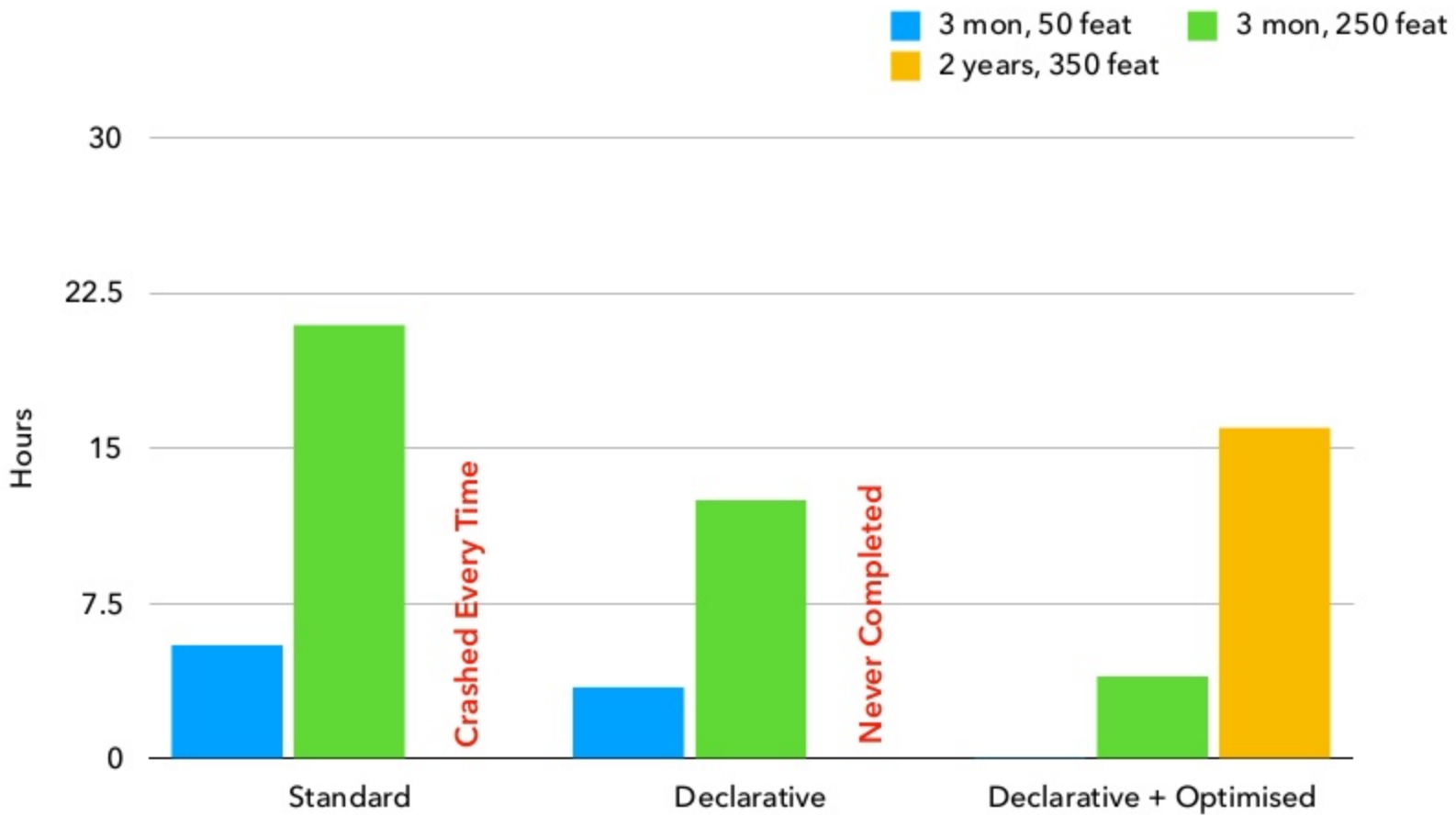
- Investigate the actual usage patterns of the feature table
- Spark DAG optimisation
- General Spark Performance Tuning
- Spark Version



Spark Migration

- 1.6 => 2.2
- Reasons to Migrate:
  - 1.6 is now several years old
  - Performance is astronomically better in 2.2 - we observed up to 17x speedups for certain workloads (e.g. windowed features)
  - Stability and logging is far better in 2.2. Spark 1.6 had a lot of painful bugs that we had to work around

Relative Performance Gains



Takeaways

- Naïve Centralisation is hard and expensive. Work out your usage patterns before you go too far in.
- Work out how to manage resource usage and cost early on.
- Constrain the surface area of the optimisation problem you solve in order to scale. (80/20 rule)
- Features should be independently declarable but globally optimised.
- The more declarative the interaction, the more you can do for your users transparently.

Thank You
Questions?

@camjo89

Simple
Machines.

@simplemach