Data-cubing made-simple with Spark, Algebird and HBase

Vidmantas Zemleris

Vinted

goo.gl/DbGR0h

Agenda

- Intro
  - Analytics at Vinted
  - What is Data-cubing?
  - Why did we build it?
- Architecture
  - Preliminaries
  - Metric computation
  - Storage & Serving metrics
  - Optimisations
- Conclusions

Analytics at Vinted

- P2P marketplace for lifestyle & clothing

  10M members 2M active monthly 8 countries ~10TB of data

- Data-driven company
- SQL solutions too slow or inflexible for analytics
- Ended up using Spark for ETL
- Developed little own OLAP engine:
  - better understand user needs
  - made complex reporting possible
  - automatic insight discovery, and much more?

OLAP Reporting

- explore metrics by all combinations of dimensions
- similar to pivot-tables Excel

Dimensions

Dimension 1

Portal

Dimension 2

Session Platform

Dimension 3

-

Dimension 4

-

Chart	Period	2015-09-01	2015-09-02	2015-09-03	2015-09-04	2015-09-05
	AU Total Sessions	602,862.31	1,856,668.52	1,166,953.67	242,860.61	88,093.16
	AU Count	460,301.83	162,735.6	11,190.6	100,130.92	82,854.61
	CZ	32,426.26	45,211.87	29,240.01	11,478.0	4,613.57
	Android	21,568.72	12,835.75	3,730.01	4,973.94	2,624.7
	Desktop	48.49	22,428.87	7,133.23	18,854.01	4,964.23
	iOS	7,541.05	6,207.62	6,366.07	2,387.5	530.39
	DE	107,812.37	166,960.05	149,859.85	271,118.32	150,097.2
	DE Babies	34,057.15	3,727.76	14,375.7	57,697.22	18,636.06
	FR	7,250.21	26,130.13	23,278.11	44,711.69	2,046.55
	LT	20,010.86	2,491.53	20,296.13	33,774.04	17,052.7
	LT Babies	5,182.3	15,559.41	7,155.7	19,632.81	13,666.98

P.S. numbers are randomised

What is Data-cubing?

(pre)compute metrics by all* dimension combinations:

portal	platform	product_features	guid
LT	iphone	["A", "B", "C"]	1
DE	android	["A"]	1001

Dimensions		Metrics
portal	**product_feature**	**unique_visitors**
any	*any*	2
any	A	2
LT	B	1
DE	A	1

Equivalent in SQL:

```
SELECT COUNT(DISTINCT guid) AS unique_visitors,
       portal
FROM sessions
GROUP BY portal

UNION ALL

SELECT COUNT(DISTINCT guid) AS unique_visitors,
       EXPLODE(product_features) AS product_feature,
       portal, product_feature
FROM sessions
GROUP BY portal, product_feature

UNION ALL

...
```

Problem definition

Given

- clean fact tables with:
  - 10-20+ Dimensions (things to filter on):

    country=It age='18..25' devices_used=Array("iphone", "android")
  - Measures (things to aggregate over)

    price=10.23 user_id=123 uuid=abc1def
  - Metrics definitions

    unique seller count

    unique members who made a transaction

Requirements

- query metrics at any* viewpoint within seconds
- be simple & integrate well with Hadoop & Spark
- support multivalued dimensions

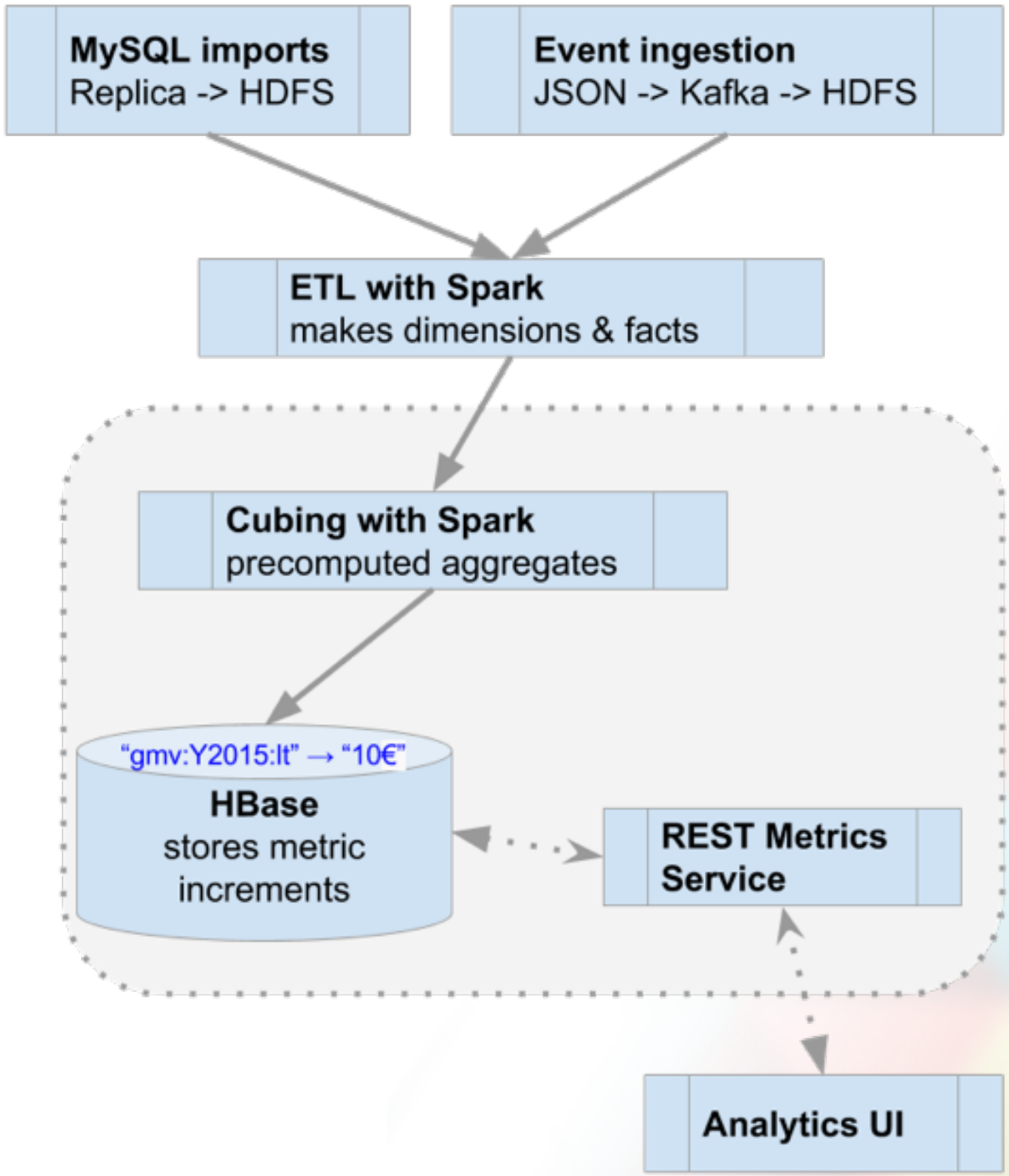
Existing solutions

- Apache Kylin (by Ebay)
  - pros: commodity infrastructure (Hadoop map-reduce)
  - cons: complex, spark not ready yet
- Druid (used by Ebay, as much as Kylin)
  - pros:
    - scalable for high dimension count
    - batch+stream (lambda architecture)
    - spacial indexing
  - cons:
    - complex
    - custom cluster services, reads JSON only*
    - missing exact count-distinct
- also Linked-in's Cubert, Mondrian ROLAP, etc

Why a custom cubing solution?

- Less complexity
- Easier integration with existing ETL
- Use shared/regular Hadoop Infrastructure
  - Spark is faster than map-reduce
- Missing features:
  - multivalued dimensions
  - exact count-distinct

Architecture overview



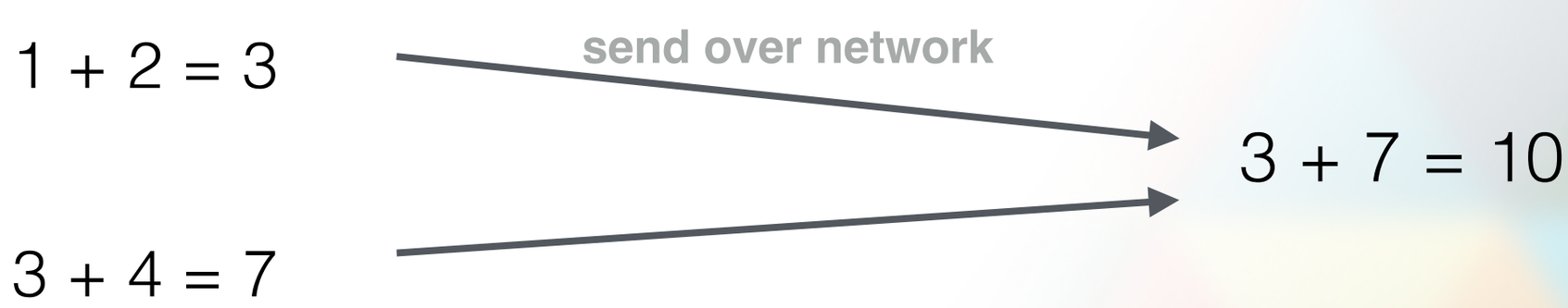
Cube definitions

```
class SessionsCube(hiveCtx: HiveContext) extends Cube {
  val maxGroupingSize: Int = 3

  val dimensionNames = Set("portal", "gender", "platform")

  override val multiValuedDimensionNames = Set("product_features")

  val metrics = MapAggregator(
    metric(name = “Session length sum",
           aggregator = sumAggregator(_.getAs[Int](“session_length"))),

    metric(id = “Unique visitors",
           aggregator = countDistinctAggregator(_.getAs[Int](“guid")))
  )

  def facts = SessionEnrichedFact.read(hiveCtx)
}
```

portal	gender	platform	product_features	guid	session_length
LT	M	iphone	["A", "B", "C"]	1	100
DE	F	android	["A"]	100	500

Adding ALL the things

- Monoid - adder, we use it for aggregations
- ordering does not matter - commutativity
  - (1 + 2) + 3 = 6
  - 1+ (2 + 3) = 6
- aggregations optimised by adding partial sums




```
class MinMonoid {
  def zero = Int.MaxValue
  def plus(l: Int, r: Int) = Math.min(l, r)
}

List(3, 4, 2).reduce(MinMonoid.plus) // 2
```

Adding ALL the things with Monoid aggregators

- can ADD complex things
  - top-K values
  - std-dev
  - exact count-distinct
  - approximate count-distinct, e.g. 5KB for 0.5% error

```
class TopKMonoid(k: Int) {
  def zero = List.empty

  def plus(l: List, r: List) =
    (l + r).sorted.take(k)
}
```

Adders in real world: exact count distinct

Problem

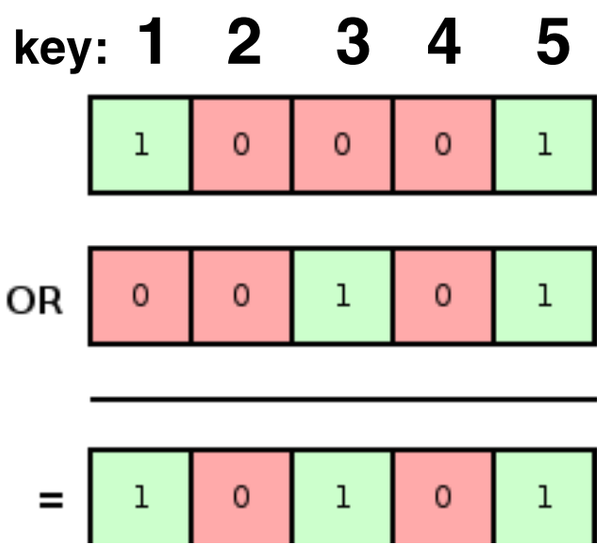
- accurate counts are often important
- naive grouping & counting is inefficient

Solution

- use a Monoid with compact bit-sets (*RoaringBitMap*)
- set a bit “on” for present values
- small memory footprint - compress zeros

Adding bit sets:

```
BitsetMonoid.plus(BitSet(1, 5),
                  BitSet(3, 5)) ===  BitSet(1, 3, 5)
```



Twitter's Algebird

- A rich library of monoid-based aggregators
  - min, sum, std-dev, quantiles, …
  - approx. count-distinct (HLL)
- Abstraction layer which hides complexity
- Composable - multiple aggregations in one pass

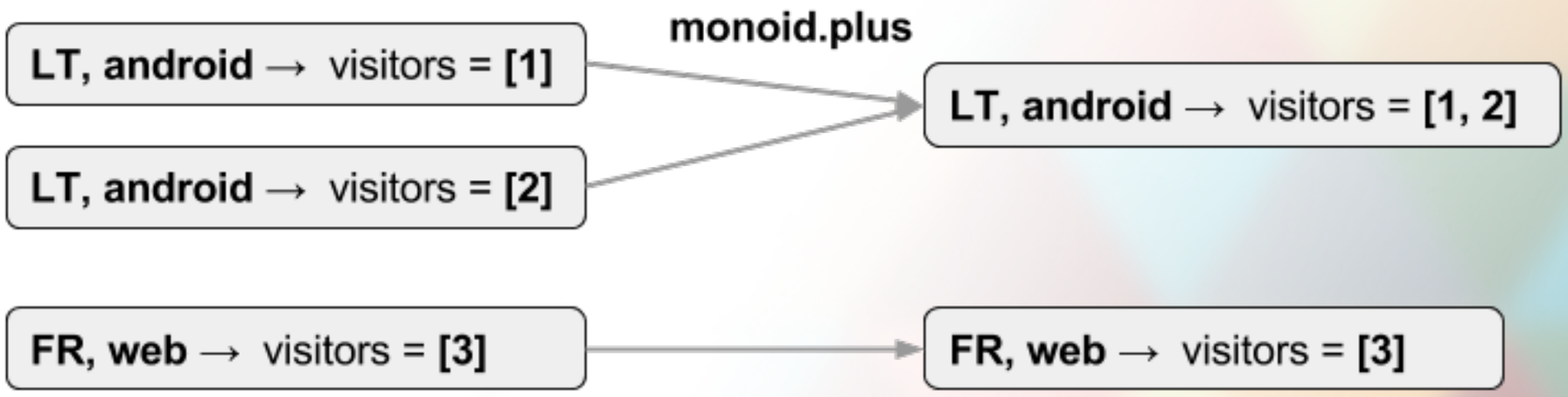
Cube definitions (again)

```
class SessionsCube(hiveCtx: HiveContext) extends Cube {
  val maxGroupingSize: Int = 3

  val dimensionNames = Set("portal", "gender", "platform")

  override val multiValuedDimensionNames = Set("product_features")

  val metrics = MapAggregator(
    metric(name = “Session length sum",
           aggregator = sumAggregator(_.getAs[Int](“session_length"))),

    metric(id = “Unique visitors",
           aggregator = countDistinctAggregator(_.getAs[Int](“guid")))
  )

  def facts = SessionEnrichedFact.read(hiveCtx)
}
```

portal	gender	platform	product_features	guid	session_length
LT	M	iphone	["A", "B", "C"]	1	100
DE	F	android	["A"]	100	500

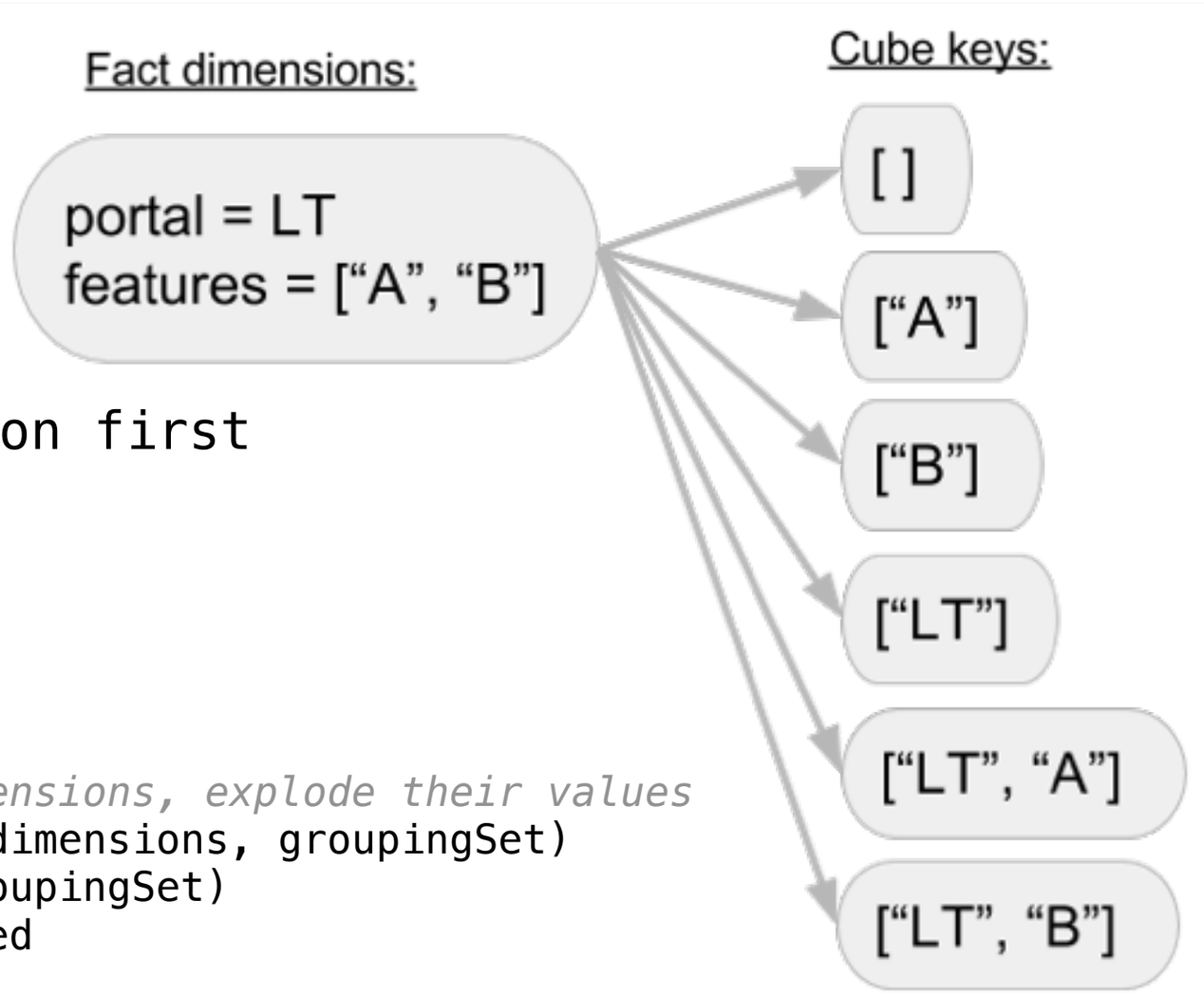
Naive Cubing algorithm (Step 1/3)

```
// #1. pre-aggregate the input: dimensions -> additiveMetrics
input.reduceByKey(monoid.plus)
```



Naive Cubing algorithm (Step 2/3)

```
// #1. pre-aggregate the input: dimensions -> additiveMetrics
input.reduceByKey(monoid.plus)

// #2. explode rows per each combination of dimensions
.flatMapKeys(cubify)
.reduceByKey(monoid.plus)
```

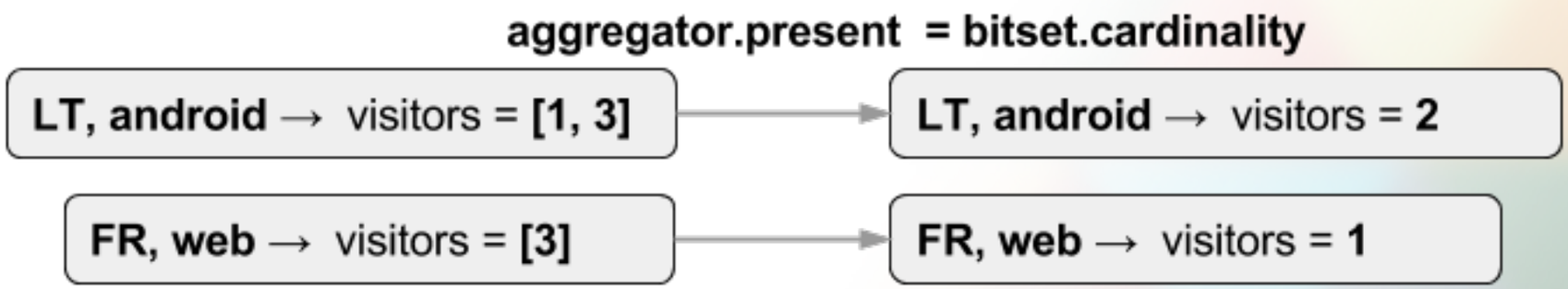


P.S.reduceByKey does local-aggregation first

```
def cubify(dimensions) = {
  for {
    groupingSet <- groupingSets
    // if groupingSet contains multivalued dimensions, explode their values
    explodedMultivalued <- explodeMultivalued(dimensions, groupingSet)
    dimensionValues = dimensions.filterKeys(groupingSet)
  } yield dimensionValues ++ explodedMultivalued
}
```

Naive Cubing algorithm (Step 3/3)

```
 // #1. pre-aggregate the input (dimensions, additiveMetrics)
input.reduceByKey(monoid.plus)

// #2. explode rows per each combination of dimensions
.flatMapKeys(cubify)
.reduceByKey(monoid.plus)

// #3. transform the metric values for end-use
.mapValues(aggr.present)
```



Writing metrics to HBase

- distributed key-value store
- fast scanning by sequential key
- HBase key:
  - dimensionsHash:version:metricName:period:dimensionValues

    e.g. **“da7c31ac:v1:gmv:Y2013:LT:android”**
- store metrics as string values: **“12.0Eur”**

Serving the metrics to end-user

- Analytics UI queries REST service:
  - REST service scans HBase
    - start_key=**“da7c31ac:v1:gmv:Y2013”**
    - end_key=**“da7c31ac:v1:gmv:Y2016”**
  - decodes HBase records
  - applies simple transformations
  - returns JSON
- Show pivot table and pretty Graphs

Obtaining reasonable performance

- pre-aggregate input
  - trivial with monoids
- limit the grouping-sets *2^n combinations*
  - look at 3-4 out of 17 dimensions *n!/(n-k)!*
  - split dimensions in groups *$2^{10} + 2^{10} + 2^{10} < 2^{30}$*
- do increments by time
  - old metrics are quite immutable
  - derived metrics (e.g. accumulation) in serving layer

Conclusions

- A Naïve cubing algorithm is fine
  - for querying moderate # of dimensions in seconds
  - scales with input size (dimension count limited so far)
  - pre-aggregation and limiting groupings is key to performance
- If not enough, hybrid approach needed
- Monoids - efficient abstraction for complex aggregations
  - makes pre-aggregation easy
  - offers even better scalability via hybrid offline/online aggregation (store serialised monadic value)

Extra: more on related tools

- Kylin
  - saves Monoids into HBase
  - pre-compute predefined grouping-sets
  - other views on-the-fly from existing partial aggregates
- Druid
  - pre-aggregates by time
  - creates inverted-index and computes on-the-fly
  - allows filtering by arbitrary dimensions
  - tolerates large # of dimensions

Resources

- http://druid.io/
- http://kylin.incubator.apache.org/
- https://github.com/linkedin/Cubert
- https://github.com/twitter/algebird
- http://www.infoq.com/presentations/abstract-algebra-analytics

About Me

- Data-warehouse developer at Vinted

  - Clojure, Scala, Spark, Hadoop, …

vidma

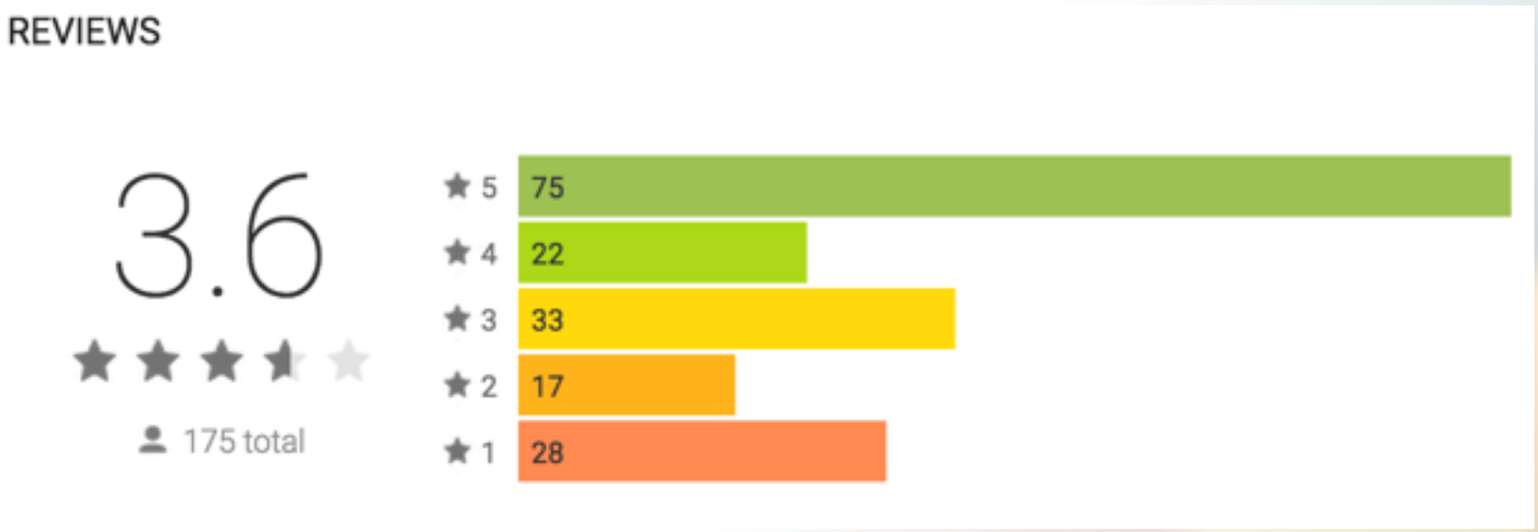
Advertisement: Help disabled speak

- You like open-source?
- You code Android/Java?
- JOIN-IN and help the disabled to speak :)
- an app for children with speech disabilities that forms sentences from a list of pictograms clicked
  - natural-language-generation & TTS inside
- started as semester project @ EPFL

  https://github.com/vidma/aac-speech-android

much beta 

, but people already like it:



Thank you!

https://goo.gl/DbGR0h