
celtra

How we evolved data pipeline at Celtra and what we learned along the way

whoami

Grega Kespret
Director of Engineering,
Analytics @ celtra
San Francisco

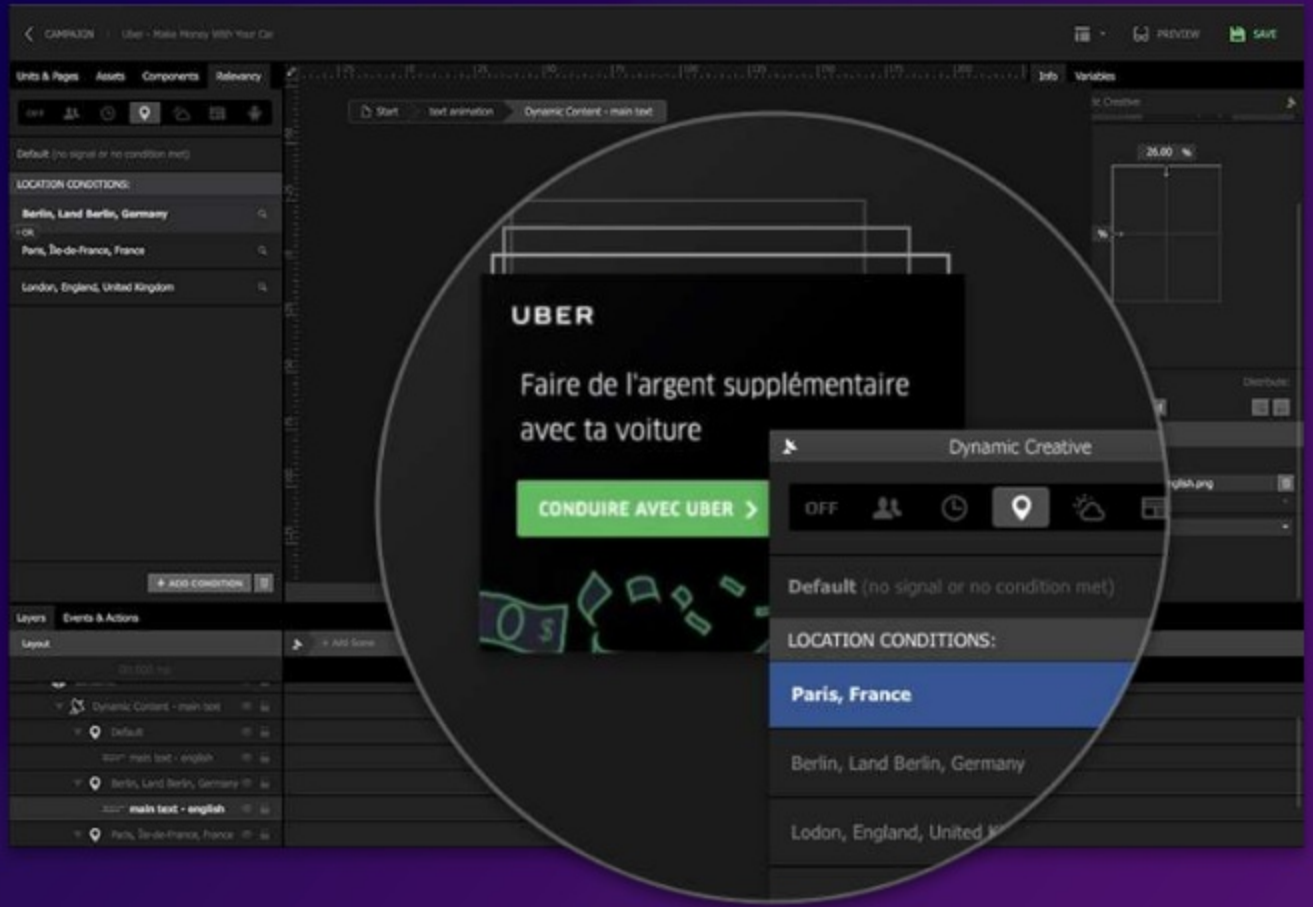
@gregakespret
github.com/gregakespret
slideshare.net/gregak


Data Pipeline

Big Data Problems vs. Big Data Problems

This is **not** going to be a talk about Data Science

Creative Management Platform



Celtra in Numbers

5000 Brands

70,000 Campaigns

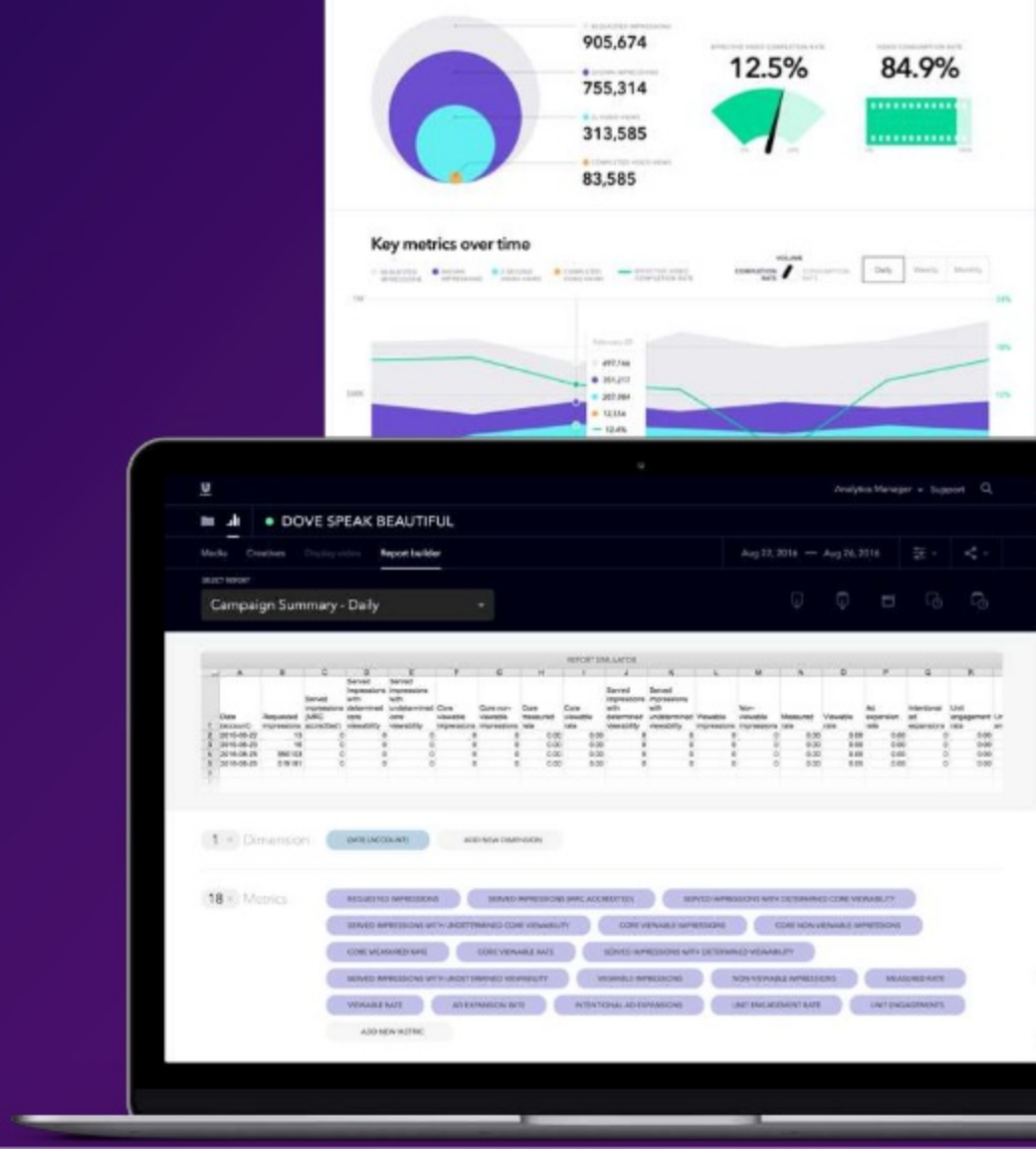
700,000 Ads Built

6G Analytics Events / Day

1TB Compressed Data / Day

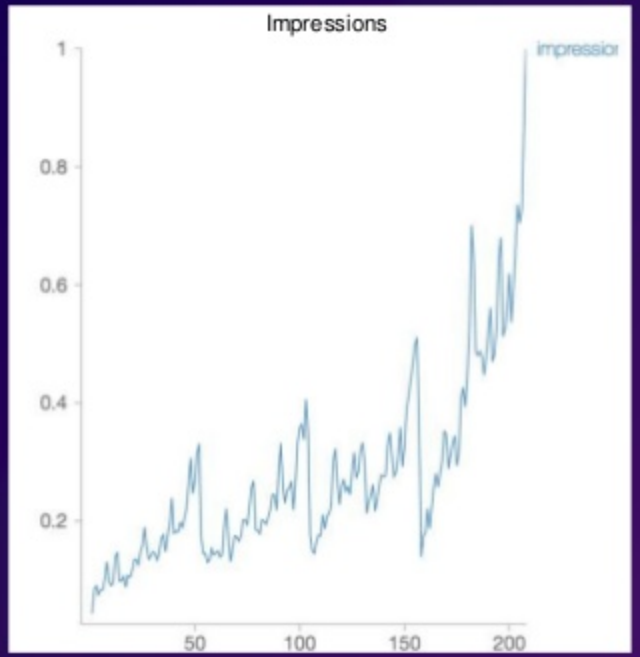
300 Metrics

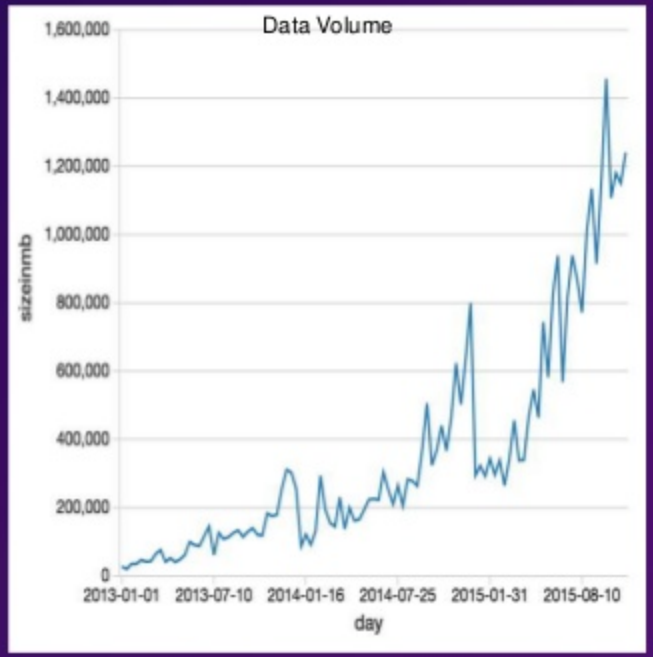
200 Dimensions

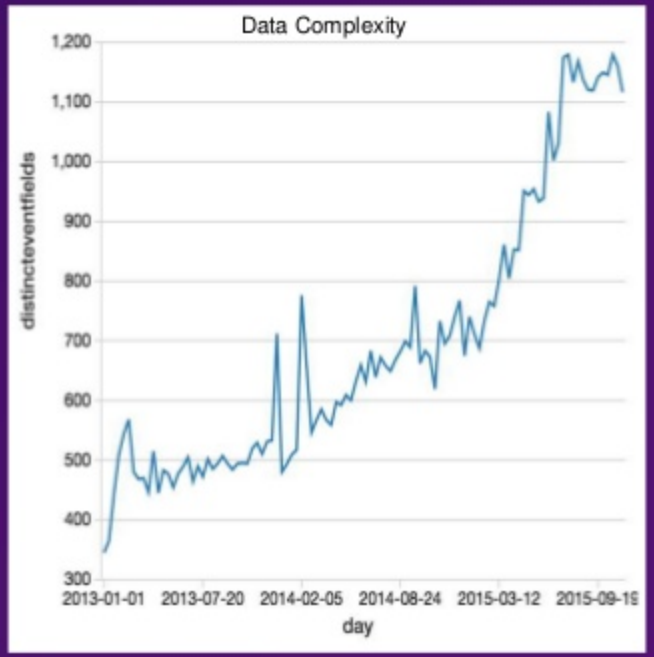


Growth

The analytics platform at Celtra has experienced tremendous growth over the past few years in terms of size, complexity, number of users, and variety of use cases.



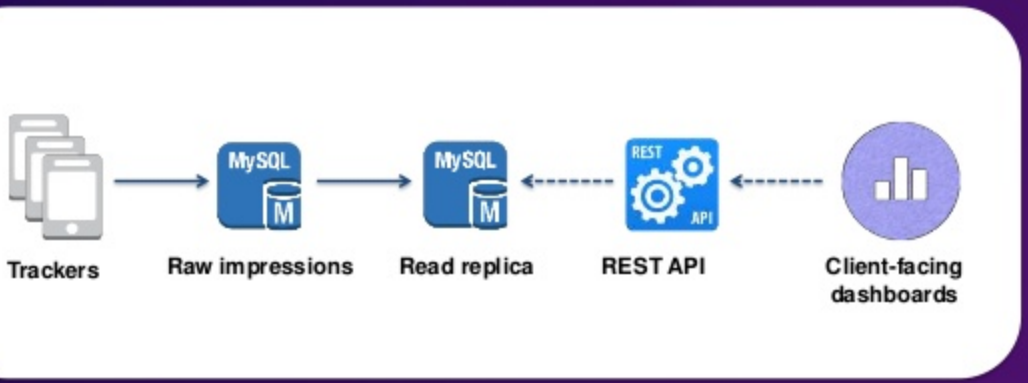





WHAT DO WE WANT
ALL THE DATA
WHEN DO WE WANT IT
NOW
WHY DO YOU WANT IT NOW
BECAUSE

MySQL for everything

- INSERT INTO impressions in track.php => 1 row per impression
- SELECT creativeId, placementId, sdk, platform, COUNT(*) sessions
  FROM impressions
  WHERE campaignId = ...
  GROUP BY creativeId, placementId, sdk, platform
- It was obvious this wouldn't scale, pretty much an anti-pattern



REST API

GET /api/analytics?

metrics=sessions,creativeLoads,sessionsWithInteraction

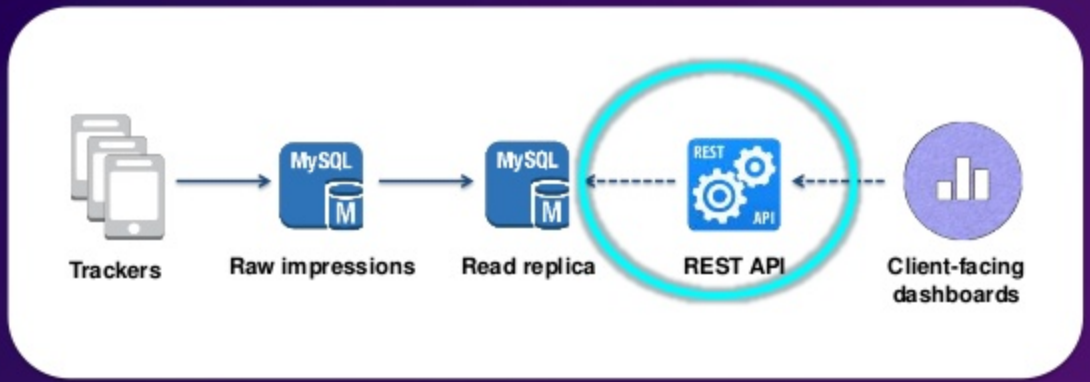
&dimensions=campaignId,campaignName

&filters.accountId=d4950f2c

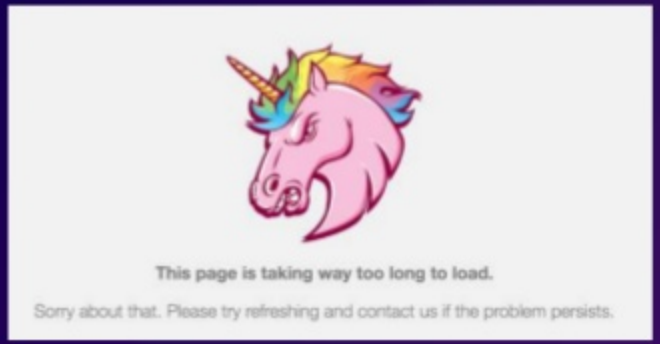
&filters.utcDate.gte=2018-01-01

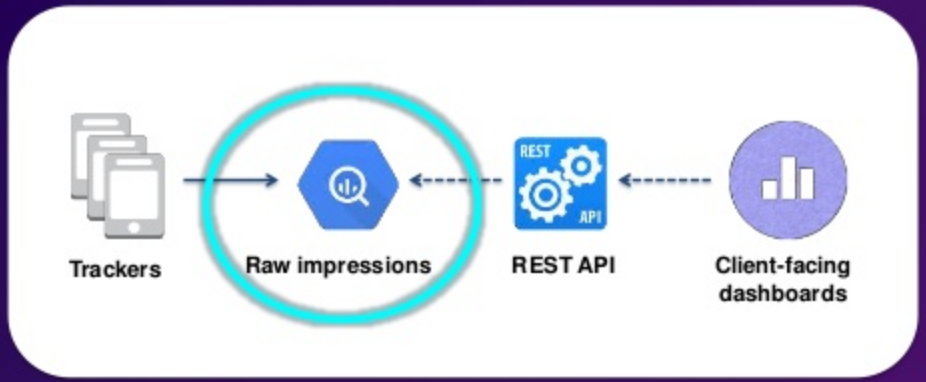
&filters.utcDate.lt=2018-04-01

&sort=-sessions&limit=25&format=json



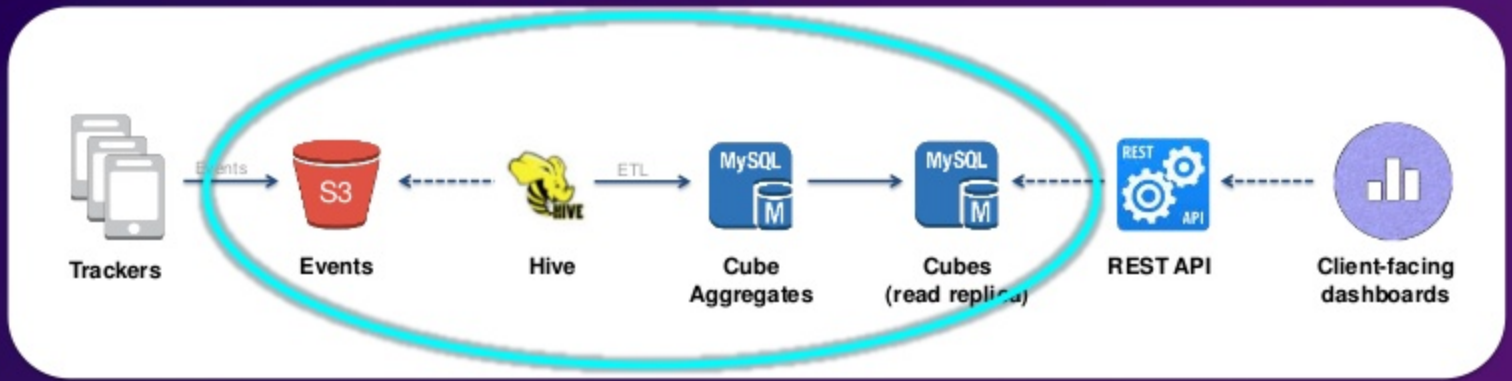
BigQuery (private beta)





Hive + Events + Cube aggregates

- Separated write and read paths
- Write events to S3, read and process with Hive, store to MySQL, asynchronously
- Tried & tested, conservative solution
- Conceptually almost 1:1 to BigQuery (almost the same SQL)



Event data

- Immutable, append-only set of raw data
- Store on S3

  s3://celtra-mab/events/2017-09-22/13-15/3fdc5604/part0000.events.gz
- Point in time facts about what happened
- Bread and butter of our analytics data
- JSON, one event per line

  - server & client timestamp

  - crc32 to detect invalid content

  - index to detect missing events

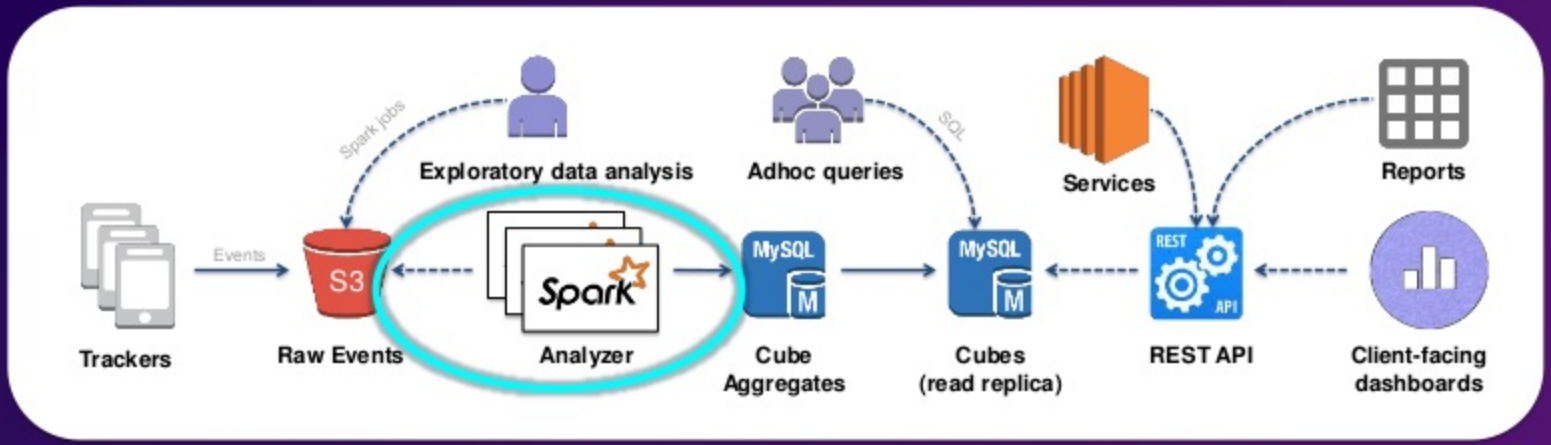
  - instantiation to deduplicate events

  - accountId to partition the data

  - name, implicitly defines the schema (very sparse)

Spark + Events + Cube aggregates

- Replaced Hive with Spark (the new kid on the block)
- Spark 0.5 in production
- **sessionization**



Sessionization

== Combining discrete events into sessions

- Complex relationships between events
- Patterns more interesting than sums and counts of events
- Easier to troubleshoot/debug with context
- Able to check for/enforce causality (if X happened, Y must also have happened)
- De-duplication possible (no skewed rates because of outliers)
- Later events reveal information about earlier arriving events (e.g. session duration, attribution, etc.)

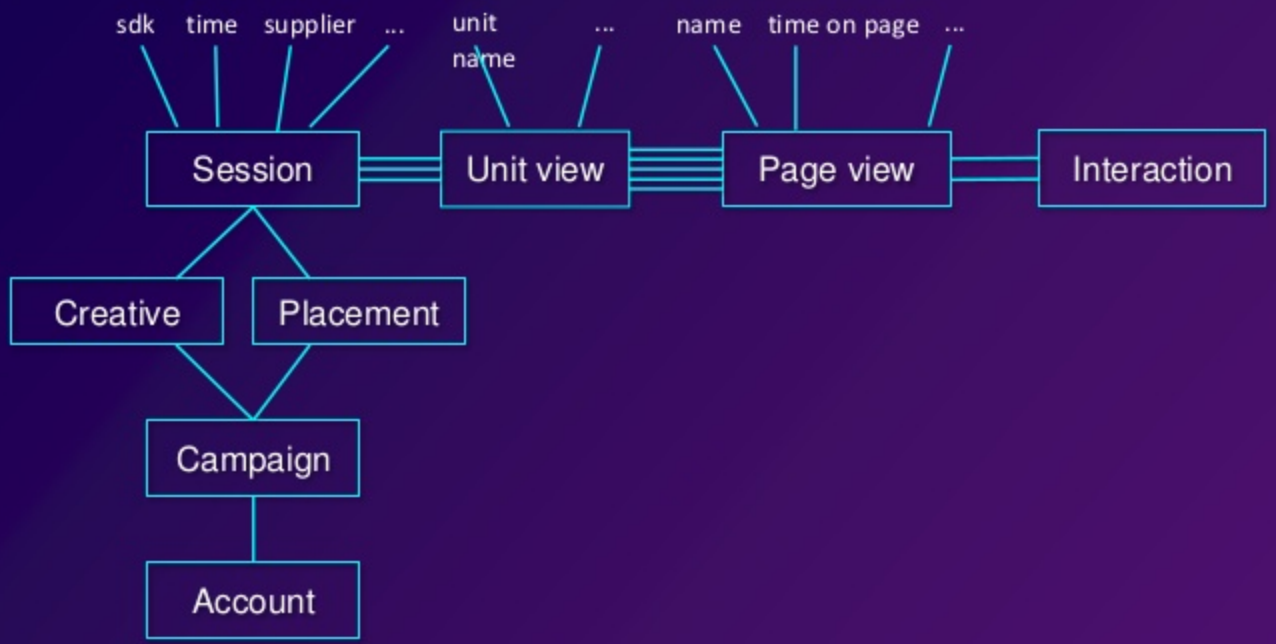
Spark

- Expressive computation layer: Provides nice functional abstractions over distributed collections
- Get full expressive power of Scala for ETL
- Complex ETL: de-duplicate, sessionize, clean, validate, emit facts
- Shuffle needed for sessionization
- Seamless integration with S3
- Speed of innovation

```
def analyze(events: RDD[Event]): RDD[Fact] = {
  events
    .keyBy(_.sessionId)
    .groupByKey(groupTasks)
    .values
    .flatMap(generateFacts)
}
```

(we don't really use groupByKey any more)

Simplified Data Model



Denormalize vs. normalize

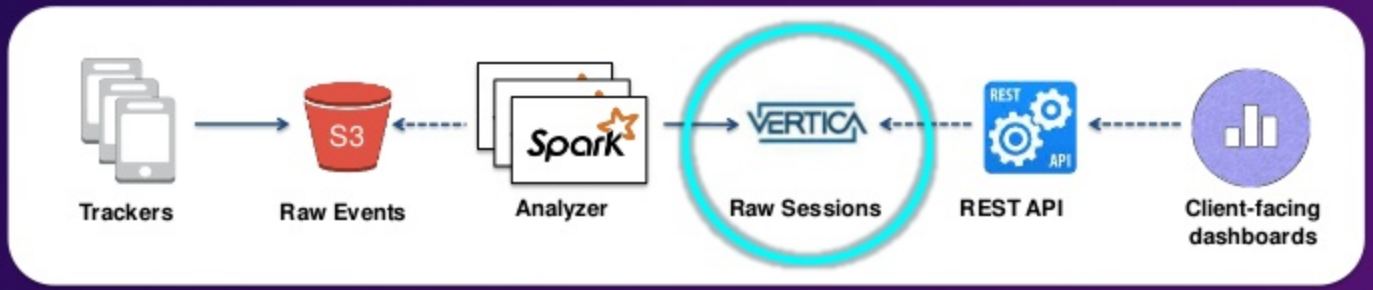
1. Denormalize
   - ✓ Speed
     (aggregations without joins)
   - ✗ Expensive storage
2. Normalize
   - ✗ Speed (joins)
   - ✓ Cheap storage

Vertica

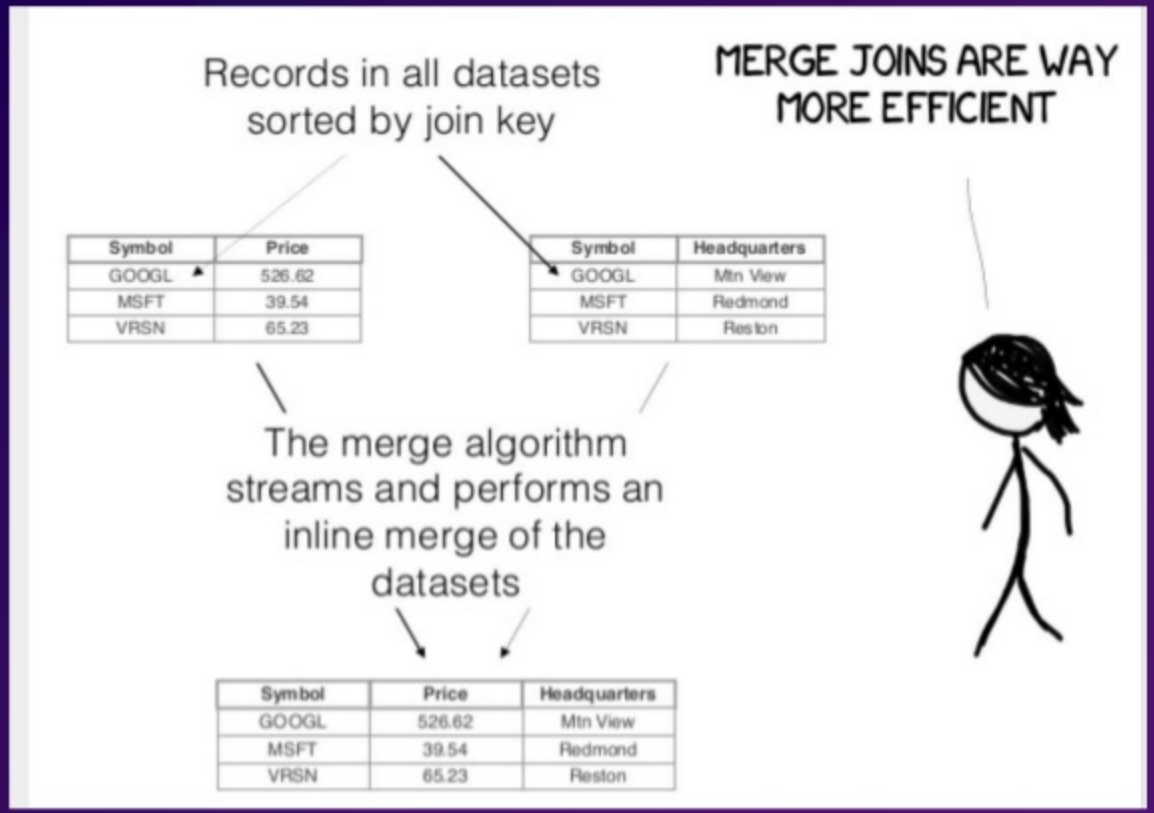
- Idea: store sessions (unaggregated data after sessionization) instead of cubes
- Columnar MPP database
- Normalized logical schema, denormalized physical schema (projections)
- Use pre-join projections to move the join step into the load

8 VPC nodes

Description
Processor - Dual Intel Xeon E5-2620
RAM - 128GB DDR3 ECC
Hard Drive - 2TB SATA
Second Hard Drive - 2TB SATA
Uplink Port Speed - 1000mbps
Bandwidth - 5000GB
Operating System - Debian 6.0 (squeeze) ...
IP Addresses - 8 IPs (5 Usable)
Backup (NAS) - 5gb NAS
KVM Over IP - External Dedicated KVM ove...
Third Hard Drive - 2TB SATA
Fourth Hard Drive - 2TB SATA
Bandwidth Protection - Bandwidth/IP Pooling
Support Level - Standard Support



Hash Joins vs. Merge Joins

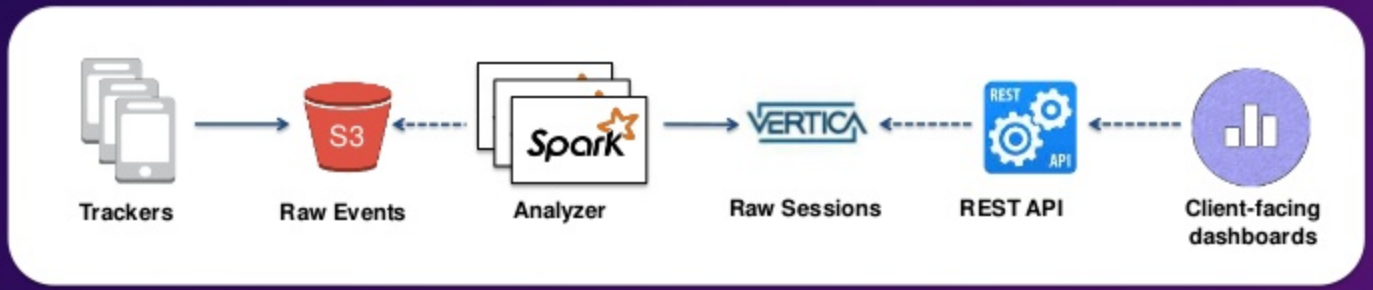


Vertica

- Always use MERGE JOINS (as opposed to HASH JOINS)
- Great performance and fast POC => start with implementation
- Pre-production load testing => a lot of problems:

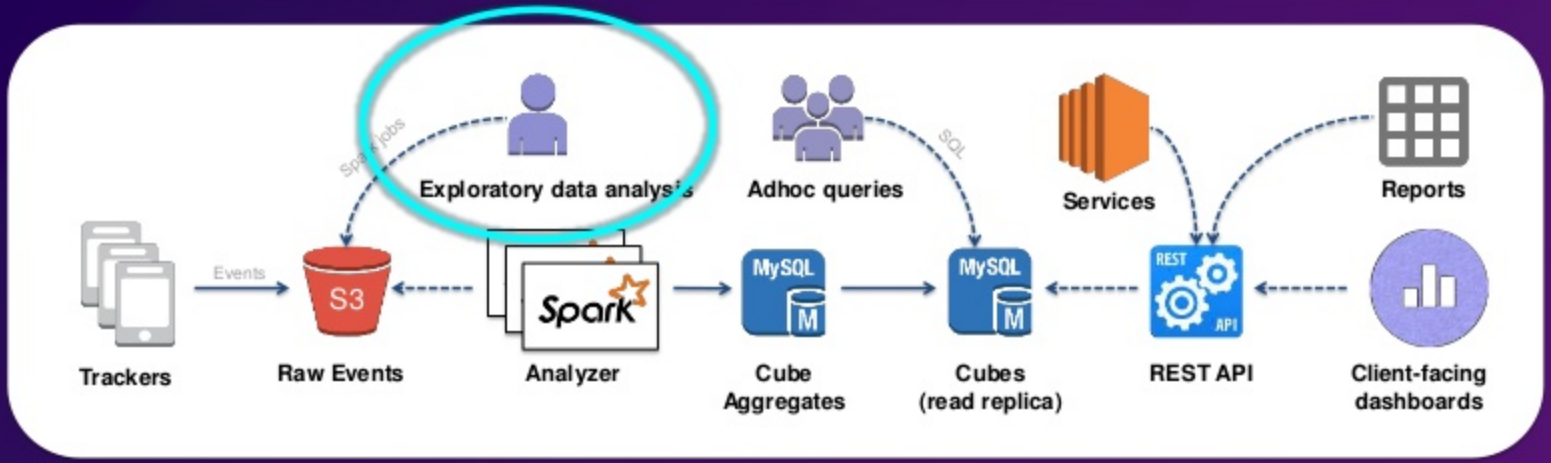
  - Inserts do no scale (VER-25968)
  - Merge-join is single-threaded (VER-27781)
  - Non-transitivity of pre-join projections
  - Remove column problem
  - Cannot rebuild data
  - ...

- Decision to abandon the project



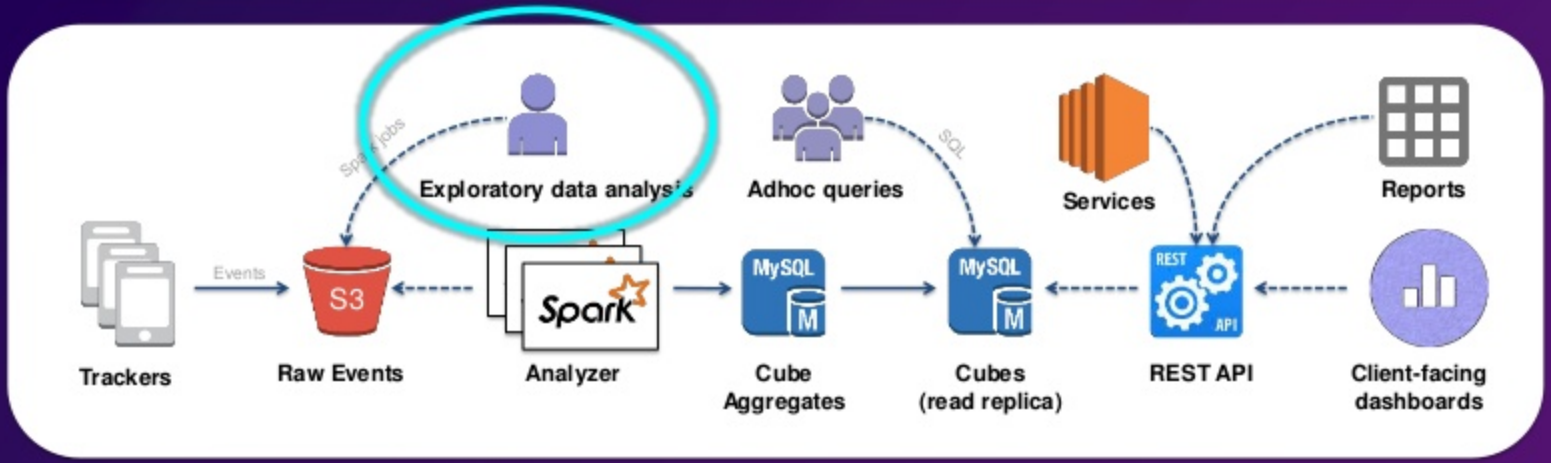
Difficult to Analyze the Data Collected

- Complex setup and configuration required
- Analyses not reproducible and repeatable
- No collaboration
- Moving data between different stages in troubleshooting/analysis lifecycle (e.g. Scala for aggregations, R for viz)
- Heterogeneity of the various components (Spark in production, something else for exploratory data analysis)
- Analytics team (3 people) bottleneck



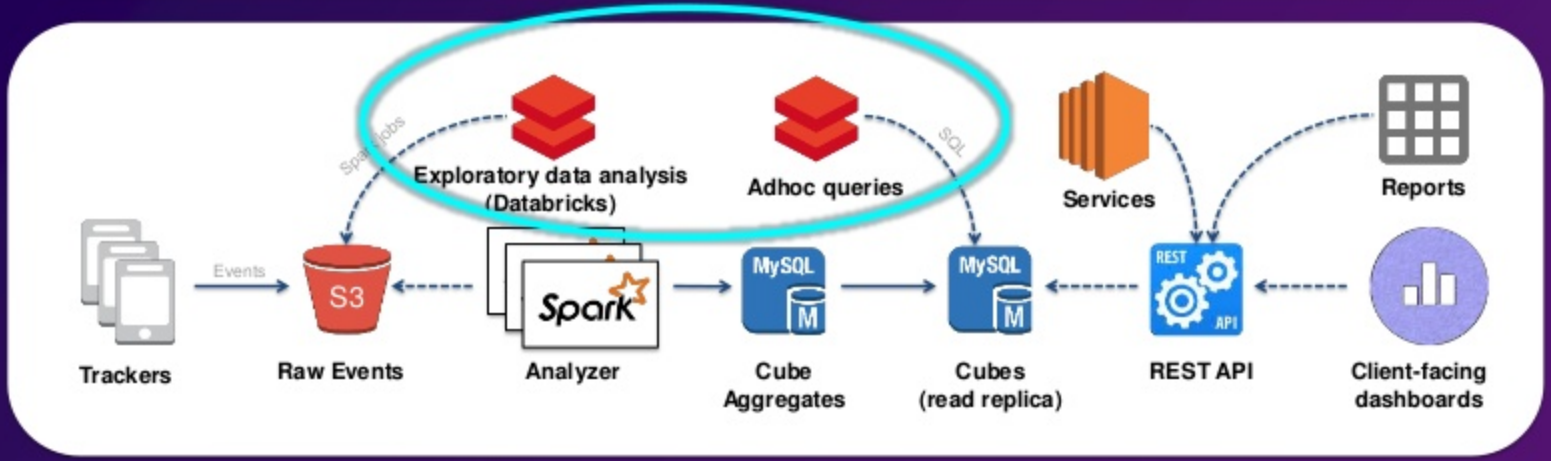
Difficult to Analyze the Data Collected

- Needed flexibility, not provided by precomputed aggregates (unique counting, order statistics, outliers, etc.)
- Needed answers to questions that existing data model did not support
- Visualizations
- For example:
  - Analyzing effects of placement position on engagement rates
  - Troubleshooting 95th percentile of ad loading time performance



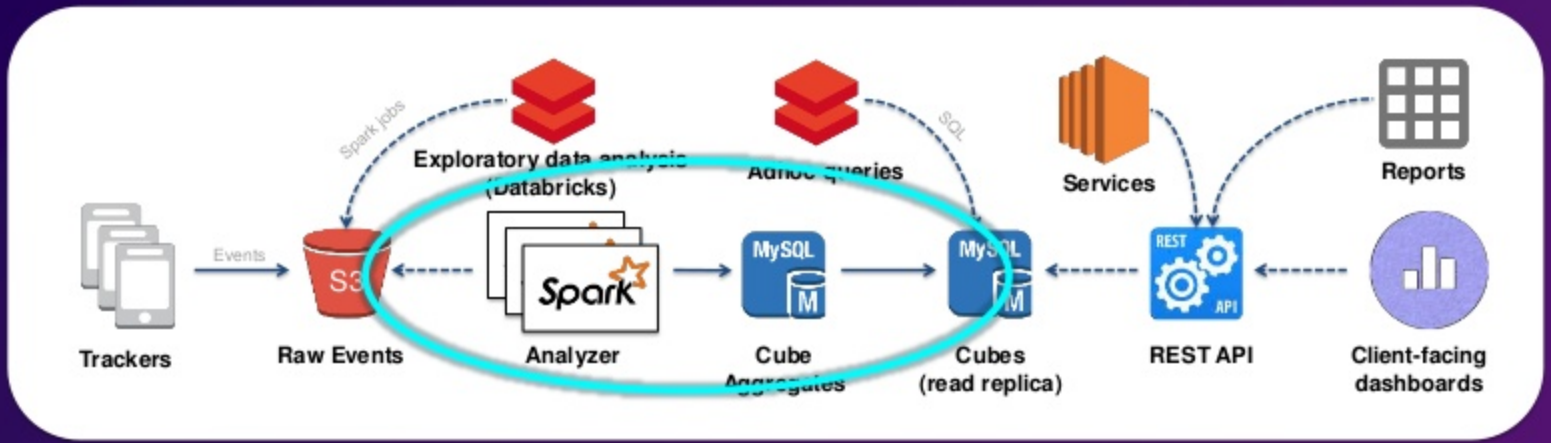
Difficult to Analyze the Data Collected

- Needed flexibility, not provided by precomputed aggregates (unique counting, order statistics, outliers, etc.)
- Needed answers to questions that existing data model did not support
- Visualizations
- For example:
  - Analyzing effects of placement position on engagement rates
  - Troubleshooting 95th percentile of ad loading time performance



Some of the problems

- ✗ Complex ETL repeated in adhoc queries (slow, error-prone)
- ✗ Slow to make schema changes in cubes (e.g. adding / removing metric)
- ✗ Keep cubes small (no geo, hourly, external dimensions)
- ✗ Recompute cube from events to add new breakdowns to existing metric (slow, not exactly deterministic)



Idea: Split ETL, Materialize Sessions

Part 1: Complex

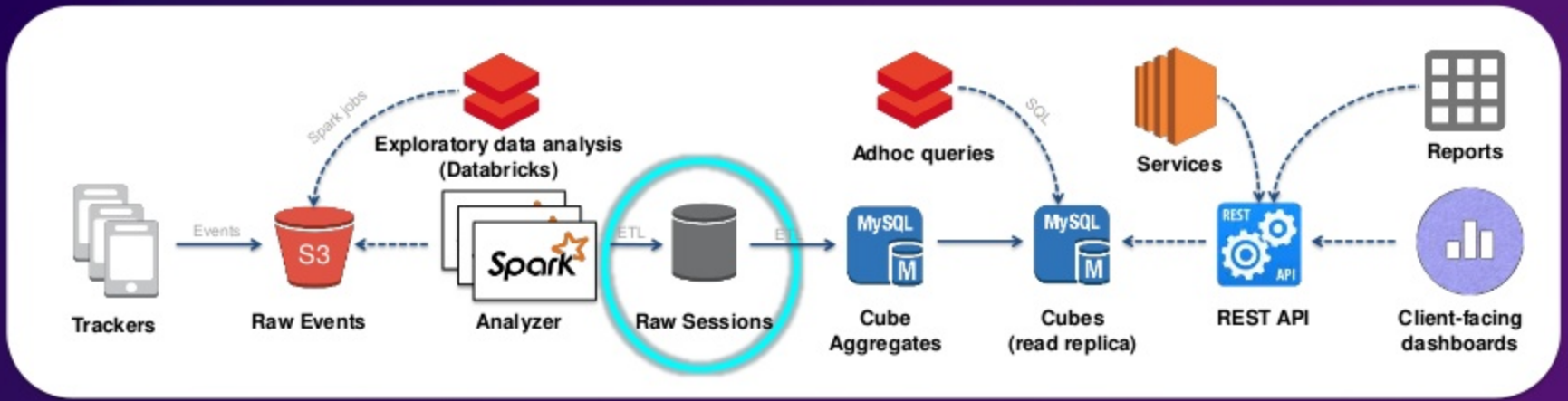
```
def computeSessions(events: RDD[Event]): RDD[Session] = {
  events
    .keyBy(_.sessionId)
    .groupByKey(groupTasks)
    .values
    .flatMap(computeSession)
}
```

deduplication, sessionization, cleaning, validation, external dependencies

Part 2: Simple

```
def computeFacts(sessions: RDD[Session]): RDD[Fact] = {
  sessions.flatMap(generateFacts)
}
```

aggregating across different dimensions



Data Warehouse to store sessions

celtra

Requirements

- Fully managed service
- Columnar storage format
- Support for complex nested structures
- Schema evolution possible
- Data rewrites possible
- Scale compute resources separately from storage

Nice-to-Haves

- Transactions
- Partitioning
- Skipping
- Access control
- Appropriate for OLAP use case

	Snowflake	BigQuery	Spark + HCatalog + Parquet + S3
First-class nested objects (schema)	✖[1]	✔	✔
Schema evolution			
• Adding columns	instant	instant	instant
• Adding columns with default	instant	rewrite	rewrite
• Removing columns	instant	rewrite	instant[2]
• Complex transformations	rewrite	rewrite	rewrite
Managed service	✔	✔	✖
Elasticity of compute separately from storage	✔	✔	✔
Atomicity	Database	Table	File
Partitioning	✖	✖	✔
Skipping	✔	✖	✔
S3 integration (to/from)	✔	✖	✔
JDBC/ODBC connectivity	✔	✖[3]	✔[4]
Access control for storage	✔	✔	✔
Access control for compute	✔	✖	✖
Fast for full table scans	✖	✔	✖
Fast for OLAP	✔	✖	✖

[1] schema-on-write for top-level columns and schema-on-read for nested (VARIANT) column

[2] just specify schema without column when reading

[3] there exist some 3rd party drivers, but are outdated and not supported by Google

[4] through Thrift JDBC/ODBC server

Final Contenders for New Data Warehouse

Spark + HCatalog + Parquet + S3

- Too many small files problem => file stitching
- No consistency guarantees over set of files on S3 => secondary index | convention
- Liked one layer vs. separate Query layer (Spark), Metadata layer (HCatalog), Storage format layer (Parquet), Data layer (S3)

We really wanted a database-like abstraction with transactions, not a file format!

Why We Wanted a Managed Service

celtra

Operational tasks with Vertica:

- Replace failed node
- Refresh projection
- Restart database with one node down
- Remove dead node from DNS
- Ensure enough (at least 2x) disk space available for rewrites
- Backup data
- Archive data

We did not want to deal with these tasks

Support for complex nested structures

1. Denormalize
   - ✓ Speed
     (aggregations without joins)
   - ✗ Expensive storage
2. Normalize
   - ✗ Speed (joins)
   - ✓ Cheap storage
3. ~~Normalized logical schema, denormalized physical schema (Vertica use case)~~
   - ✓ Speed (move the join step into the load)
   - ✓ Cheap storage
4. Nested objects: pre-group the data on each grain
   - ✓ Speed (a "join" between parent and child is essentially free)
   - ✓ Cheap storage

Pre-group the data on each grain


```
{
  "id": "s1523120401x263215af605420x35562961",
  "creativeId": "f21d6f4f",
  "actualDeviceType": "Phone",
  "loaded": true,
  "rendered": true,
  "platform": "IOS",
  ...
  "unitShows": [
    {
      "unit": { ... },
      "unitVariantShows": [
        {
          "unitVariant": { ... },
          "screenShows": [
            {
              "screen": { ... },
            },
            {
              "screen": {... },
            }
          ],
          "hasInteraction": false
        }
      ],
      "screenDepth": "2",
      "hasInteraction": false
    }
  ]
}
```

Flat vs. Nested Queries

celtra

Find top 10 pages on creative units with most interactions on average

Flat + Normalized

```
SELECT
  creativeId,
  us.name unitName,
  ss.name pageName,
  AVG(COUNT(*)) avgInteractions
FROM
  sessions s
JOIN unitShows us ON us.id = s.id
JOIN screenShows ss ON ss.usid = us.id
JOIN interactions i ON i.ssid = ss.id
GROUP BY 1, 2, 3
ORDER BY avgInteractions DESC LIMIT 10
```

Joins
Requires unique ID at every grain

Nested

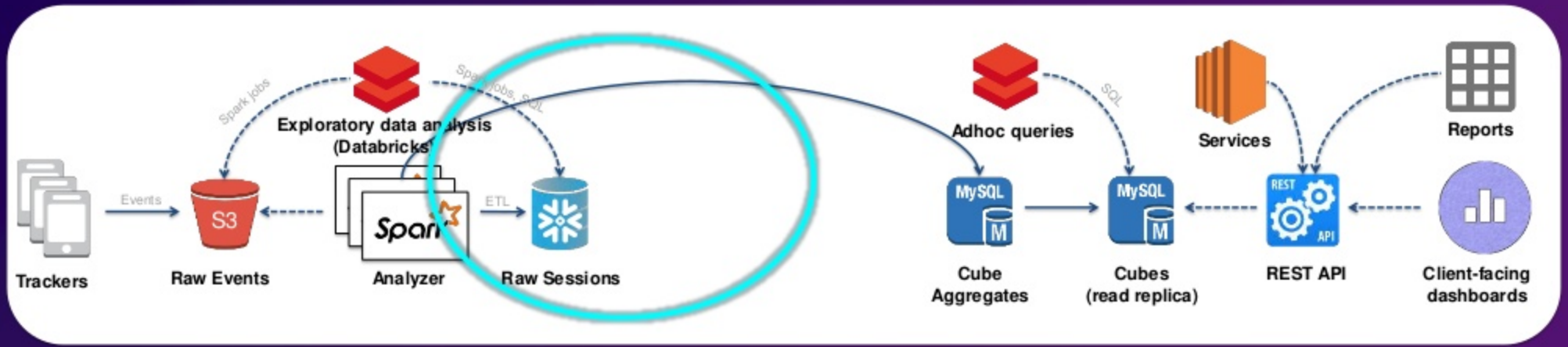
```
SELECT
  creativeId,
  unitShows.value:name unitName,
  screenShows.value:name pageName,
  AVG(ARRAY_SIZE(screenShows.value:interactions)) avgInteractions
FROM
  sessions,
  LATERAL FLATTEN(json:unitShows) unitShows,
  LATERAL FLATTEN(unitShows.value:screenShows) screenShows
GROUP BY 1, 2, 3
ORDER BY avgInteractions DESC LIMIT 10
```

Distributed join turned into local join

Sessions: Path to production

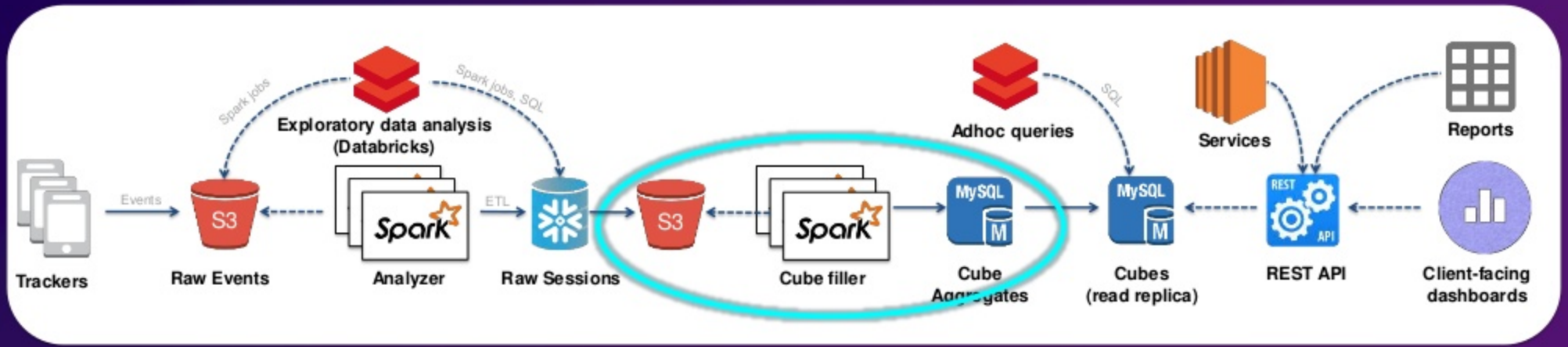
1. Backfilling
   - Materialized historical sessions of last 2 years and inserted into Snowflake
   - Verify counts for each day versus cube aggregates, investigate discrepancies & fix

2. "Soft deploy", period of mirrored writes
   - Add SnowflakeSink to Analyzer (./bin/analyzer --sinks mysql,snowflake)
   - Sink to Cubes and Snowflake in one run (sequentially)
   - Investigate potential problems of snowflake sink in production, compare the data, etc.



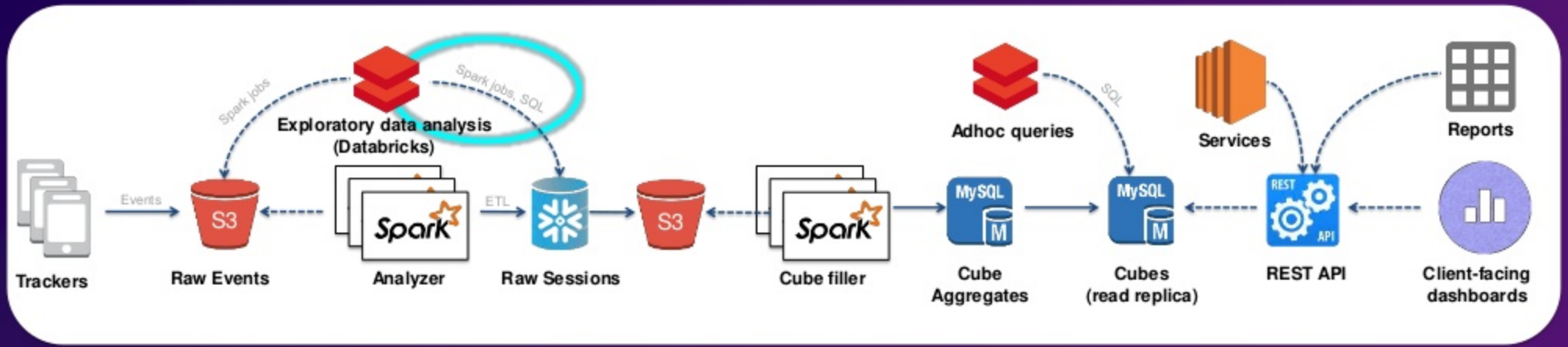
Sessions in production

3. Split Analyzer into Event analyzer and Cube filler
- Add Cubefiller
- Switch pipeline to Analyzer -> Sessions, Cubefiller -> Cubes
- Refactor and clean up analyzer



Sessions in production

- ✓ Data processed once & consumed many times (from sessions)
- ✗ Slow to make schema changes in cubes (e.g. adding / removing metric)
- ✗ Keep cubes small (no geo, hourly, external dimensions)
- ✗ Recompute cube from events to add new breakdowns to existing metric (slow, not exactly deterministic)



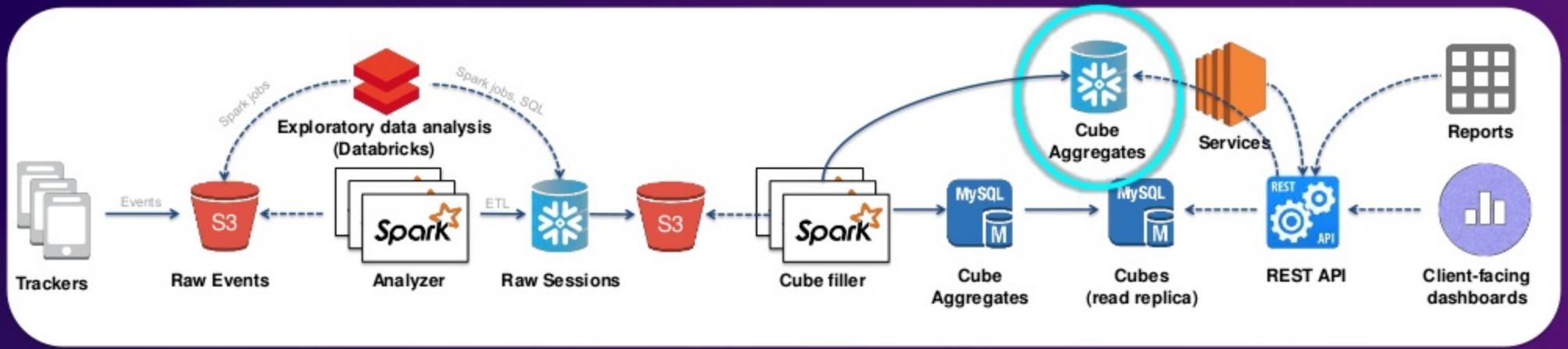
Move Cubes to Snowflake

1. Backfilling & Testing

- Export Cube aggregates data from MySQL and import into Snowflake
- Test performance of single queries, parallel queries for different use cases: Dashboard, Reporting BI through Report generator, ...

2. "Soft deploy", period of mirrored writes

- Add SnowflakeSink to Cubefiller (./bin/cube-filler --sinks mysql,snowflake)
- Sink to MySQL Cubes and Snowflake Cubes in one run
- Investigate potential problems of snowflake sink in production, compare the data, etc.



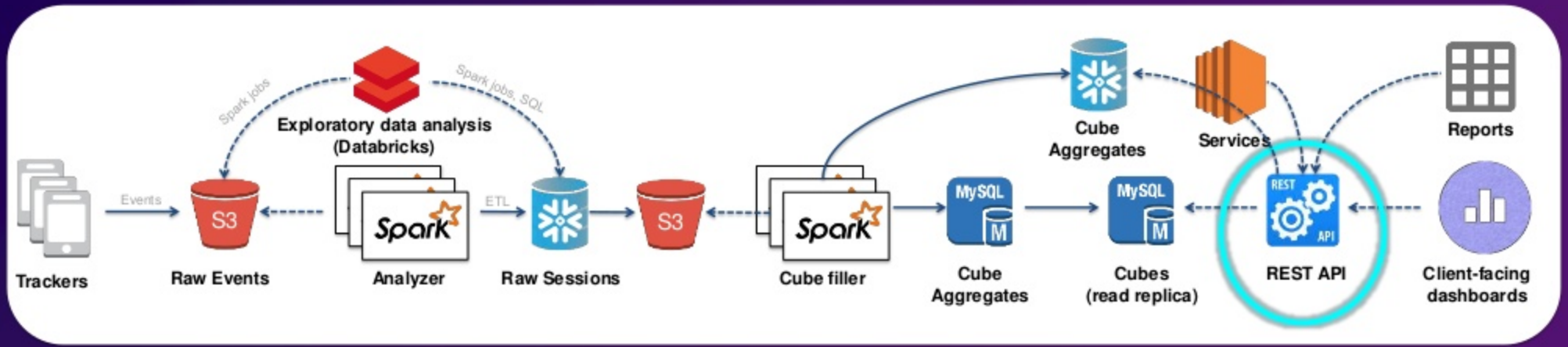
Move Cubes to Snowflake

3. REST API *can* read from Snowflake Cube aggregates (SnowflakeAnalyzer)

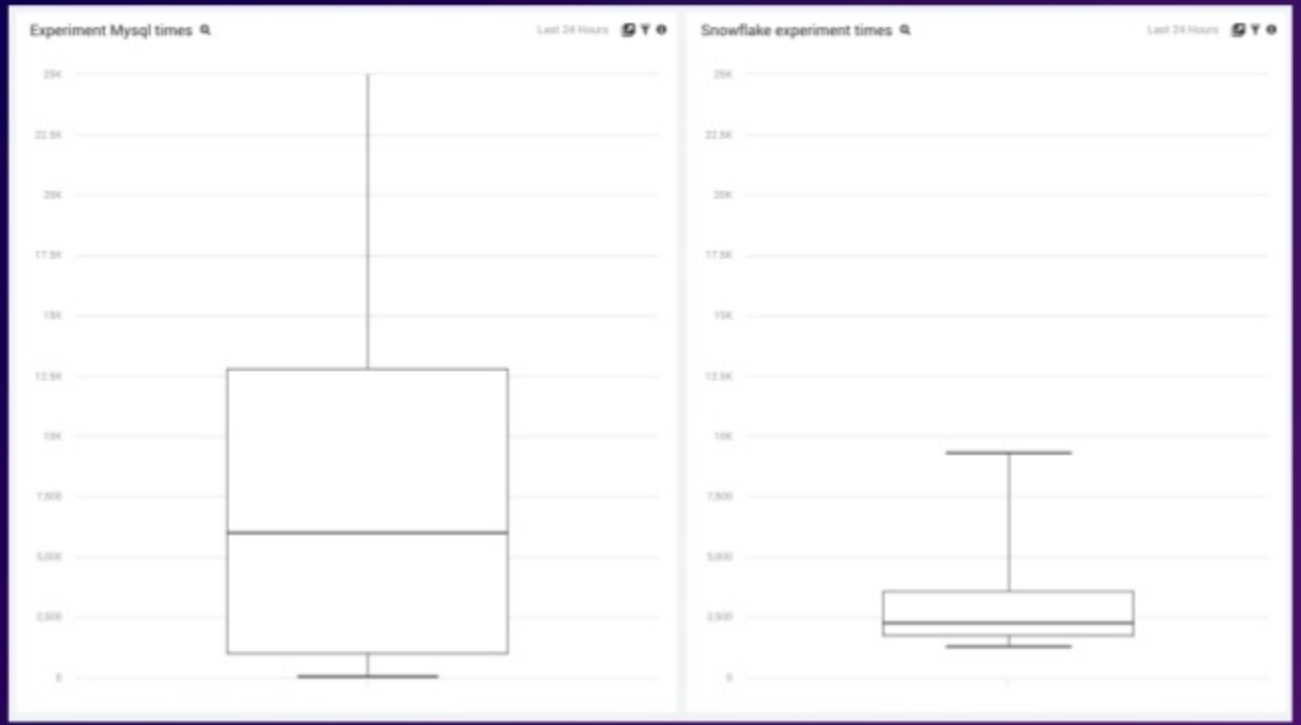
4. Validating SnowflakeAnalyzer results on 1% of requests.

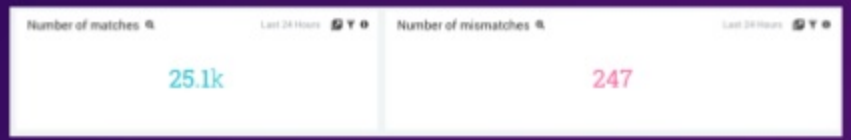
```
/**
 * Proxies requests to a main analyzer and a percent of requests to the
secondary analyzer and compares responses. When responses do not match,
it logs the context needed for troubleshooting.
 */
class ExperimentAnalyzer extends Analyzer
```

5. Gradually switch REST API to get data from Snowflake instead of MySQL
   - Start with 10% of requests, increase to 100%
   - This allows us to see how Snowflake performs under load



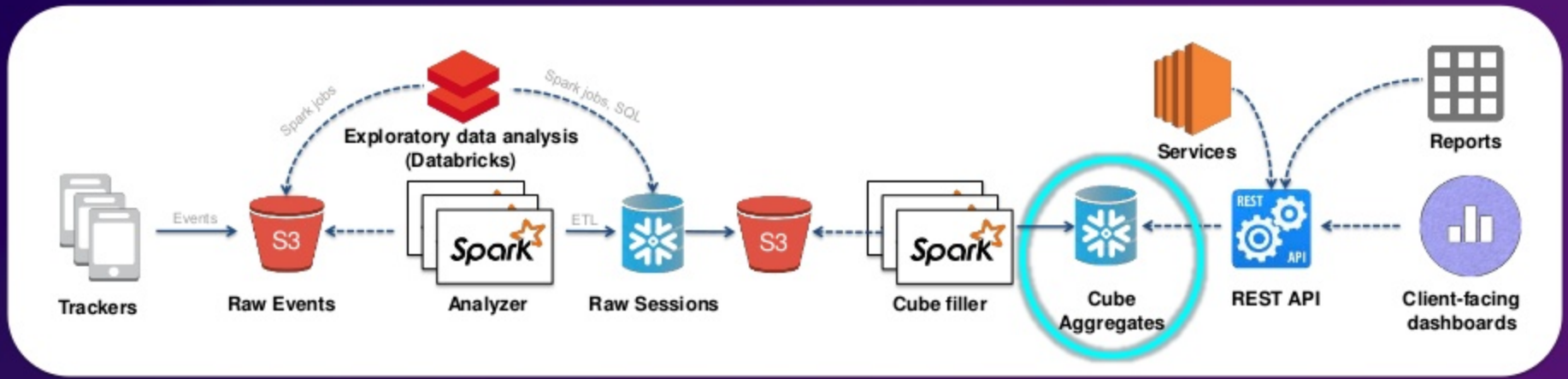
Check matches and response times





Cubes aggregates in Snowflake

- ✓ Data processed once & consumed many times (from sessions)
- ✓ Fast schema changes in cubes (e.g. adding / removing metric)
- ✓ Can support geographical, hourly, external dimensions
- ✗ Recompute cube from sessions to add new breakdowns to existing metric (slow, but deterministic)



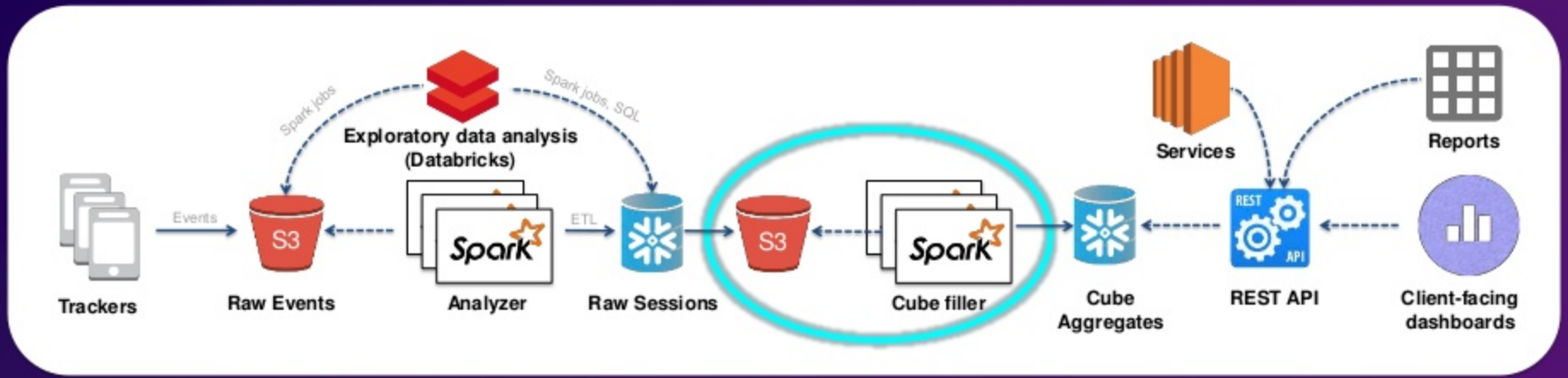
Compute cube aggregates with SQL

- Move computation to data, not data to computation
- Instead of exporting sessions to Spark cluster only to compute aggregates, do it in database with SELECT … FROM sessions GROUP BY ...

Part 2: Simple

```
def computeFacts(sessions: RDD[Session]): RDD[Fact] = {
  sessions.flatMap(generateFacts)
}
```

aggregating across different dimensions

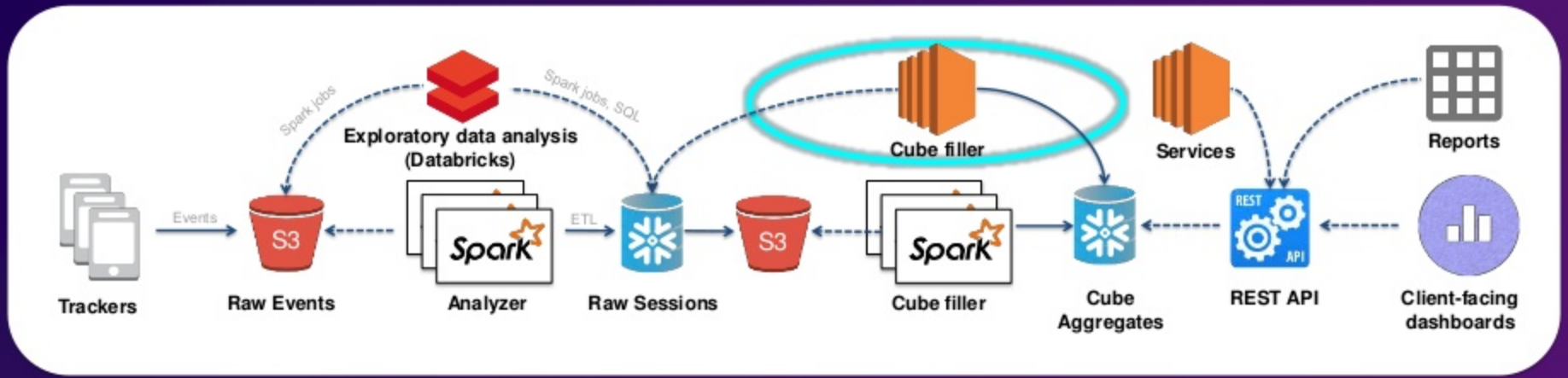


Compute cube aggregates with SQL

1. Period of double writes

./bin/cube-filler --logic sql,cubefiller --sink snowflakecubes

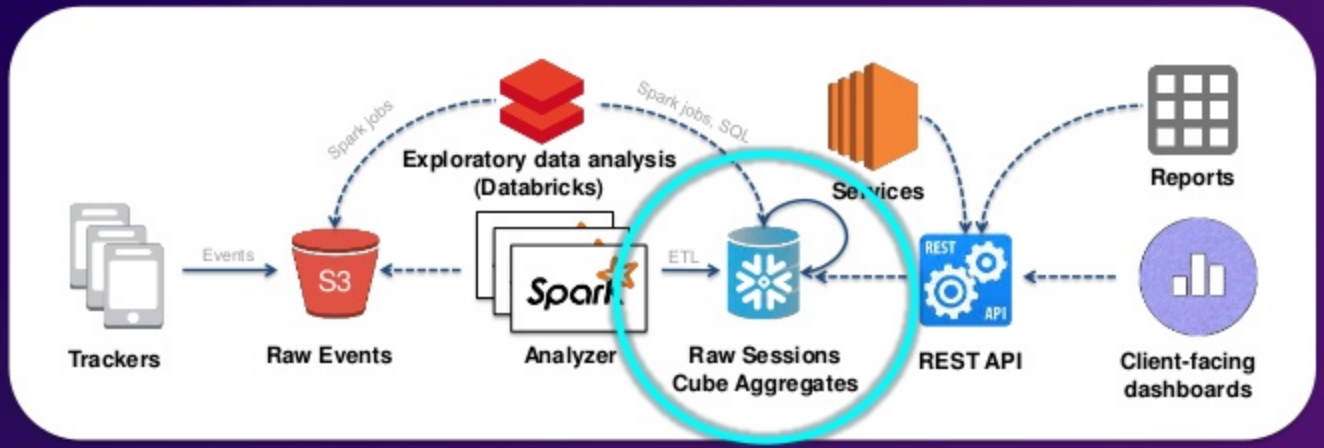
2. Compare differences, fix problems



Compute cube aggregates with SQL

3. Remove Spark completely from cubefiller

- REST API can read directly from Raw sessions OR from Cube Aggregates, generating the same response



How we handle schema evolution

- Session schema
  Known, well defined (by Session Scala model) and enforced
- Latest Session model
  Authoritative source for sessions schema
- Historical sessions conform to the latest Session model
  Can de-serialize any historical session
- Readers should ignore fields not in Session model
  We do not guarantee to preserve this data
- Computing metrics, dimensions from Session model is time-invariant
  Computed 1 year ago or today, numbers must be the same

Schema Evolution

Change in Session model	Top level / scalar column	Nested / VARIANT column
Rename field	ALTER TABLE tbl RENAME COLUMN col1 TO col2;	data rewrite (!)
Remove field	ALTER TABLE tbl DROP COLUMN col;	batch together in next rewrite
Add field, no historical values	ALTER TABLE tbl ADD COLUMN col type;	no change necessary

Also considered views for VARIANT schema evolution

- For complex scenarios have to use **Javascript UDF** => lose benefits of columnar access
- Not good for practical use

Data Rewrites

- They are sometimes *necessary*
- We have the *ability* to do data rewrites
- Rewrite must maintain sort order fast access (UPDATE breaks it!)
- Did rewrite of 150TB of (compressed) data in December
  - Complex and time consuming, so we fully automate them
  - Costly, so we batch multiple changes together
- Javascript UDFs are our default approach for rewrites of data in VARIANT

Inline Rewrites with Javascript UDFs

- Expressive power of Javascript (vs. SQL)
- Run on the whole VARIANT record
- (Almost) constant performance
- More readable and understandable
- For changing a single field, OBJECT_INSERT/OBJECT_DELETE are preferred

```
CREATE OR REPLACE FUNCTION transform("json" variant)
RETURNS VARIANT
LANGUAGE JAVASCRIPT
AS '
  // modify json
  return json;
';


SELECT transform(json) FROM sessions;
```

Stuff you have seen today...

- There are many ways to develop a data pipeline, none of them are perfect
- Make sure that load/scale testing is a required part of your go-to-production plan
- Rollups make things cheaper, but at a great expense later
- Good abstractions survive a long time (e.g. Analytics API)
- Evolve pipeline modularly
- Maintain data consistency before/after

celtra

Thank You.

P.S. We're hiring