Flexible Tables

Where SQL meets semi-structured data

James Fraumeni and Ben Vandiver

Systems Software Engineer, Distributed Infrastructure, HP Vertica

April 23rd, 2014

Who are we?

And if we do all that, how’re we so small?

We’re a start-up driving the biggest Silicon Valley tech company

Academic roots

- Founded in 2005 by Mike Stonebraker (MIT)
- “C-Store: A Column Oriented DBMS” (VLDB 2005)
- “The Vertica Analytic Database: C-Store 7 Years Later” (VLDB 2012)

Acquired by Hewlett-Packard in 2011

- HP was having trouble; we were busy being awesome so they left us alone
  - They gave us lots of sweet hardware, though
- HP’s now recovering; they’re (very actively) asking us how we do it

Small group of highly competent engineers

- About 40 core developers today
- Small teams; big challenges; lots of freedom

What do we enable?

What’s the big deal?

Our tools power the world’s big analytics shops

What if you knew, and could query in seconds…

- Every stock trade ever?
- Every phone call ever?
- Every tweet?
- Every gene in your genome?

Clusters of hundreds of machines, with tens of petabytes of data.

You can’t do that with MySQL.

Whose users want sub-second response times on real queries.

You can’t do that with Hadoop.

Vertica users

```
SELECT HEAT_MAP_IMAGE(lat, lon)
  OVER (PARTITION BY country)
  FROM tweets
  WHERE SENTIMENT(text) < -1;
```

Yep, we do that.

What is Vertica?

“One cluster to rule them all…”

Speed

+

Scale

+

Simplicity

“MPP OLAP RDBMS”

- SQL Database for Real-time Analytics
- Runs on commodity x86_64 hardware
- MPP Columnar Architecture – scales to PBs!
- Easy to setup and use
- Extensive Ecosystem of analytic tools

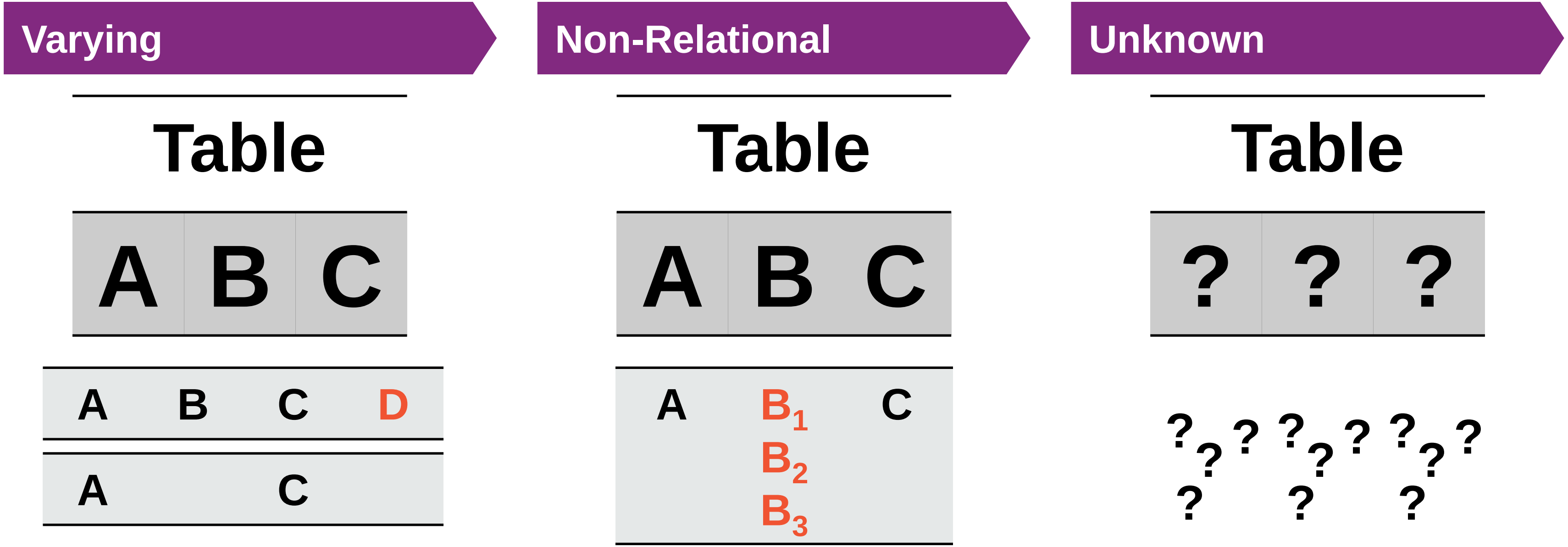
High Level Goals for Flexible Tables

- SQL Databases must
  - Ingest semi-structured data easily
  - Query it naturally
- Queries should be input-format agnostic (*i.e.*, vanilla SQL against relational tables)
- Vertica shows good results along this path

What is Semi-Structured Data?

A very “RDBMS” point of view

Data that has structure, but that structure is:



Varying

Table

A B C

A B C D

A C

Non-Relational

Table

A B C

A B_1 B_2 B_3 C

Unknown

Table

? ? ?

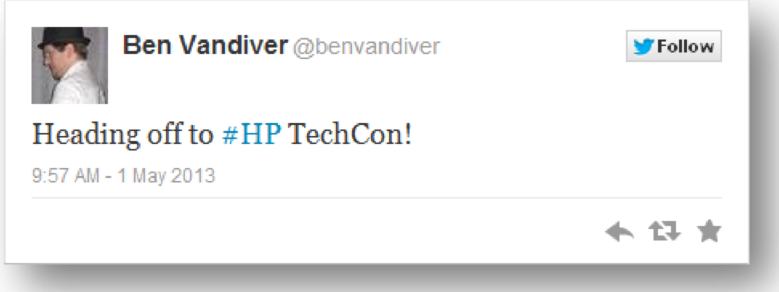
? ? ? ? ? ?
? ? ?
? ? ?

A Semi-Structured Data Example:

Are German tweets longer than English tweets?

Tweets: Raw Data

“Tweet child of mine”



Ben Vandiver @benvandiver

Heading off to #HP TechCon!

9:57 AM - 1 May 2013

Twitter Public Streaming API

https://stream.twitter.com/1.1/statuses/sample.json

Contains tweets:

```
{"filter_level":"medium","contributors":null,"text":"Heading off to #HP TechCon!",
"geo":null,"retweeted":false,"in_reply_to_screen_name":null,"truncated":false,"lang":"en","entities":{"symbols":[],"urls":[],"hasht
ags":[{"text":"HP","indices":[15,18]}],"user_mentions":[]},"in_reply_to_status_id_str":null,"id":329595489773309952,"source":"web",
"in_reply_to_user_id_str":null,"favorited":false,"in_reply_to_status_id":null,"retweet_count":0,"created_at":"Wed May 01 13:57:43
+0000 2013", "in_reply_to_user_id":null,"favorite_count":0,"id_str":"329595489773309952“,"place":null,
"user":{"location":"Cambridge,
MA","default_profile":true,"statuses_count":1,"profile_background_tile":false,"lang":"en","profile_link_color":"0084B4","id":148363
627,"following":null,"favourites_count":0,"protected":false,"profile_text_color":"333333","description":"A database
guy.","verified":false,"contributors_enabled":false,"profile_sidebar_border_color":"C0DEED","name":"Ben
Vandiver","profile_background_color":"C0DEED","created_at":"Wed May 26 14:22:29 +0000
2010","default_profile_image":false,"followers_count":12,"profile_image_url_https":"https://si0.twimg.com/profile_images/3596493897
/a4056576ec72b18a6087213f46080eeb_normal.jpeg","geo_enabled":false,"profile_background_image_url":"http://a0.twimg.com/images/theme
s/theme1/bg.png","profile_background_image_url_https":"https://si0.twimg.com/images/themes/theme1/bg.png","follow_request_sent":nul
l,"url":null,"utc_offset":null,"time_zone":null,"notifications":null,"profile_use_background_image":true,"friends_count":4,"profile
_sidebar_fill_color":"DDEEF6","screen_name":"benvandiver","id_str":"148363627","profile_image_url":"http://a0.twimg.com/profile_ima
ges/3596493897/a4056576ec72b18a6087213f46080eeb_normal.jpeg","listed_count":0,"is_translator":false},"coordinates":null}
```

and delete markers:

```
{"delete":{"status":{"user_id":178172685,"id":257897002996727808,"user_id_str":"178172
685","id_str":"257897002996727808"}}}
```

Using a Relational Database

“Old skool”

Define Schema

```
CREATE TABLE tweets(
  text varchar(~~140~~) (2000)
  “user.lang” varchar(10)
  …
);
```

Query

```
SELECT “user.lang”,
       AVG(LENGTH(text))
  FROM tweets
 GROUP BY “user.lang”;
```

Load Data

- Save tweets to disk
- Parse the JSON with Python to pull out the desired output columns into delimited format
- COPY tweets(text, “user.lang”) FROM ‘/path/to/parsed.json’

Pros and Cons

+ Easy to query

- Tricky to pick schema

- Awkward to load

Hadoop / Pig

“Going whole hog NoSQL”

Define Schema

- No need

Query

- Write some Pig:

```
register hdfs://…/user/benchmark/piggybank.jar
tweets = load 'hdfs://tweets.json' using JsonLoader() as (m: Map[]);
twt_r1 = foreach tweets generate m#'text', m#'user';
twt_r2 = foreach twt_r1 generate LENGTH($0), $1#’lang’ as m;
twt_r3 = group twt _r2 by $1
avg_lengths = foreach twt_r3 generate $1as lang, AVG($0) as text_len;
avg_lengths_ordered = order avg_lengths by text_len desc;
dump avg_lengths_ordered;
```

Load Data

- Save tweets to disk
- “cp” directly into HDFS, as with a file copy

Pros and Cons

+ No schema definition

+ Easy to load

- Trickier to query

- Degraded performance

Alternative SQL approach: JSON in DB

“Passing the buck”

Define Schema

```
CREATE TABLE tweets(
  contents VARCHAR(4096)
);
```

Load Data

- Save tweets to disk
- Break on JSON record boundaries
- COPY tweets(contents) FROM ‘/path/to/parsed.json’

Query

```
SELECT json_value(contents,’user.lang’),
 AVG(LENGTH(json_value(contents,’text’))
FROM tweets
GROUP BY json_value(contents,’user.lang’);
```

Pros and Cons

+ Minimal schema definition

+ Simpler to load

- Query exposes JSON storage

Database Principles

“Here I stand…”

SQL is declarative:

Specify the what, database determines how

Agnostic of storage:

Same SQL, if DB is row-store, column-store, compressed, loaded fixed-width or delimited, …

New approach: Flexible Tables

“Look Ma, no columns!”

Define Schema

```
CREATE FLEX TABLE tweets();
```

Load Data

- Save tweets to disk
- COPY tweets() FROM ‘/path/to/raw.json’

Query

```
SELECT “user.lang”,
       AVG(LENGTH(text))
  FROM tweets
 GROUP BY “user.lang”;
```

Pros and Cons

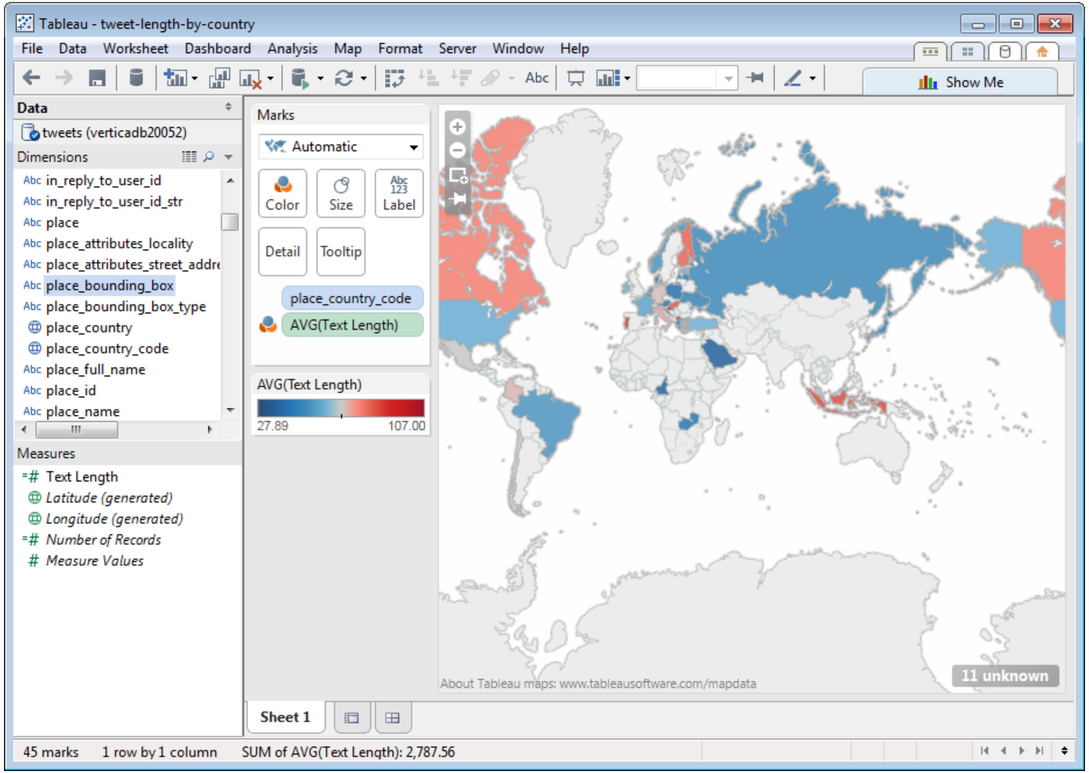
+ No schema definition

+ Simple to load

+ Easy to query

Tools Integration

"SQL makes a world of difference"



45 marks 1 row by 1 column SUM of AVG(Text Length): 2,787.56

Why Semi-Structured?

“Enabling ELT in addition to ETL”

- Just toss the data in (data lake)
  - Not worth putting the effort into discovering/defining schema when you’re not sure you’ll ever use the data
- Postpone the decision of which columns and types are “important” in the source data
- Allow for schema discovery while leveraging (a subset of) the benefits of the Vertica System rather than forcing it to be known at load-time
- Allow for incremental transform into the final, desired form while keeping the table usable
- Smooth out version/implementation differences in otherwise well-structured data
- Retain a SQL interface and its tools support, employee trainings, etc.

Implementation

“The devil’s in the details… and there are lots of details”

Components:

- **Supporting FLEX tables**
- **Performance optimization**
- **Validation of approach**

Objectives:

- **Simple to use**
- **Vanilla SQL**
- **Format agnostic storage**
- **Minimal added database complexity**

Flexible Tables

“I’d like my table *medium rare*”

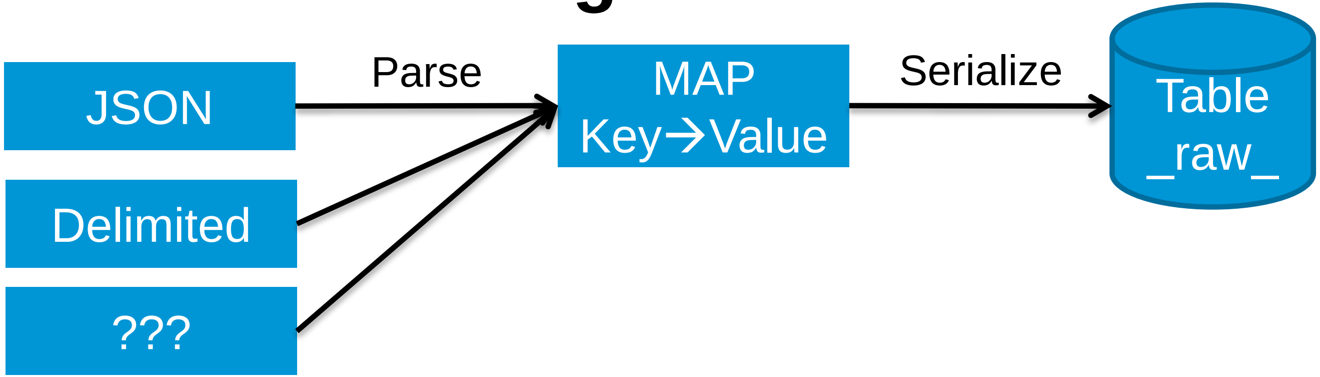
Store data in large binary column:

```
CREATE FLEX TABLE tweets();
```

is

```
CREATE TABLE tweets (
  __raw__ long varbinary
);
```

Flexible Tables: Parsing Semi-structured Data



JSON:

```
{
  “id”: 1,
  “name”: “ben”
}
```

Becomes a record:

{ “id” → “1”, “name” → “ben” }

Delimited with header row:

```
id,name
1,ben
2,jen
```

Becomes 2 records:

{ “id” → “1”, “name” → “ben” }

{ “id” → “2”, “name” → “jen” }

Extensible with User Defined Parsers

```
Avro, Protocol buffers,
Thrift, …
```

Fun Fact: Our JSON and Delimited parsers are actually user defined parsers

Flexible Tables: Supporting Vanilla SQL

“Column if you got ‘em!”

Automatic Query Rewrite

```
SELECT text FROM tweets;
```

becomes:

```
SELECT MapLookup(__raw__,’text’)
  FROM tweets;
```

Implementation

- **Vertica SQL-parser-level change**
- **Translates any column reference**
- **maplookup is null if key is missing**

Flexible Tables: SELECT * FROM tweets

“Wish upon a falling *”

- What it probably means:

Return all the data as a relational table

- Bad choices:
  - Return strings of JSON
    - Must track provenance (could be delimited, avro, …)
    - Useless to SQL tools
  - Return rows with all keys as columns
    - Requires full scan to compute columns
    - Could result in millions of columns
    - Loading data could invalidate consumer assumptions about schema

Flexible Tables: Column Metadata

"A view from the top"

Solution: build a view

- Compute set of available keys and strore in a keys table: compute_flextable_keys()
- Generate view from keys: build_flextable_view() / compute_flextable_keys_and_build_view()
- SQL Tools understand views
- Regenerate keys & view to expose new schema to applications
- Can still access base flex table

Example:

Table people has rows:

{ name → ben, hair → blond }

{ name → jen, eyes → green }

```
SELECT * FROM people_view:
name | hair  | eyes
Ben  | blond |
Jen  |       | green
```

Handling Nested Structure: Exploding

Give me the text of all posts tagged “big data”

```
SELECT text
    FROM posts
    WHERE 'big data' IN tags;


SELECT * FROM (SELECT
    text, MapItems(tags)
    OVER(PARTITION BY text, tags)
    FROM posts) innerPosts
  WHERE values = 'big data';
```

```
                  text                       | keys |  values
---------------------------------------------+------+----------
 Giving a talk on Flexible Tables at Brandeis. | 3    | big data
(1 row)
```

```
{
   "postID": 52737,
   "posterID": 134028,
   "text": "Giving a talk on
Flexible Tables at Brandeis.",

   "tags":
     ["flex", "vertica", "hp", "big
data"],

   "replyPostsIDs":
     [52740, 52756, 52757, 52810]
}
```

Handling Nested Structure (Maps & Lists)

“… may none of them be missed!”

Maps get flattened

```
{
  “a”: 5,
  “b”: {
         “c”: 4,
         “d”: 7
       }
}
```

becomes

{ a → 5, “b.c” → 4, “b.d” → 7 }

Lists become sub-maps

```
{
  “a”: [ “red”, “green”]
}
```

becomes

{ a → { 0 → “red”, 1 → “green” } }

Performance

- Likely in the “Big Data” arena – performance at scale matters
- Load is wonderfully parallel – each record is self-contained
- Query performance:
  - Structure hidden inside __raw__ column
  - SQL does not distinguish between **real** columns and **virtual** columns
  - A column-store database is crucial

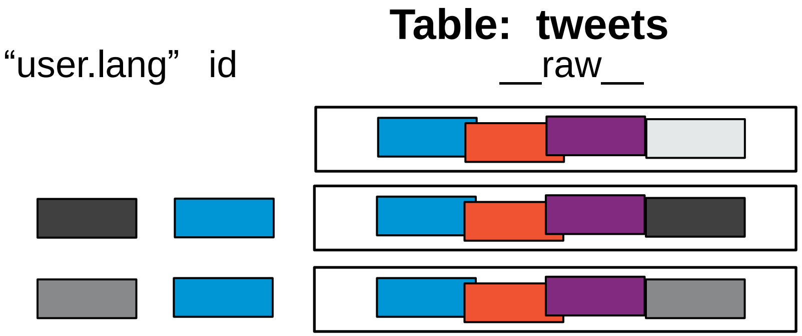
Performance: Column-Store Basics

“My precious… Column! Column!”

- Store each column’s data separately, often sorted and compressed
- Benefits:
  - Pay I/O cost only for columns referenced
  - Late materialize columns after predicates or even joins
  - Easy to add columns to a table

Performance: Flex → Relational

“Optimization via PowerPoint”



```
CREATE TABLE tweets (
  __raw__ long varbinary,
  “user.lang” varchar(10),
  id int
)
ORDER BY “user.lang”;
```

Push-Button Process:

1. **Column Materialization**
   DB tracks used columns
2. **Database Designer**
   Optimizes physical layout

Performance: Hybrid Flex Tables

“Best of both worlds”

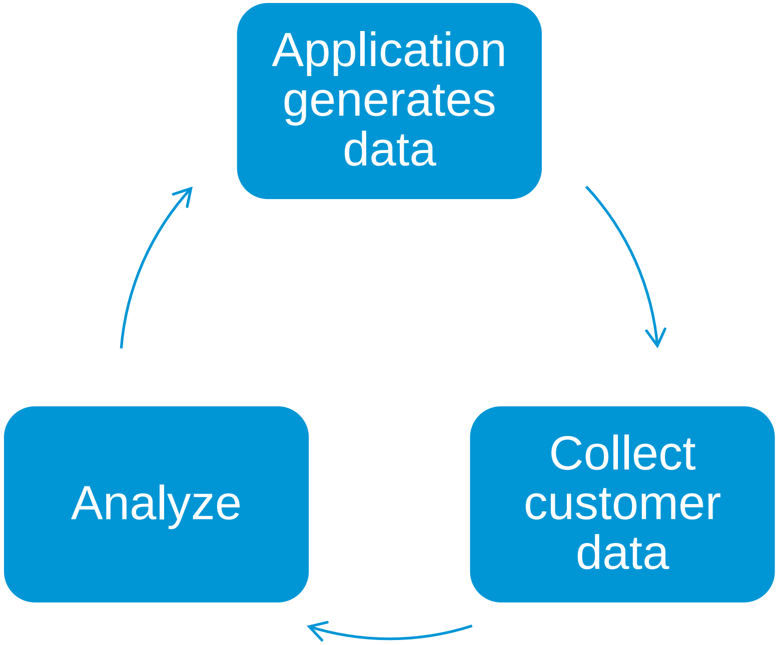
- Supports partial description
  - Some parts of data fixed & known
  - Improved performance
- Re-uses performance mechanisms
- SQL Query is identical

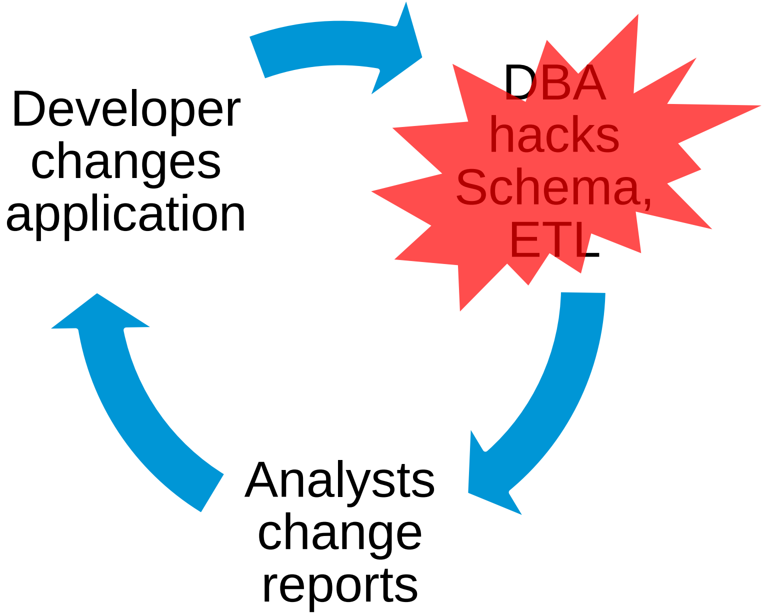
Partial Description

```
CREATE FLEX TABLE tweets(
  time timestamp,
  id int,
  “user.lang” varchar(20)
)
ORDER BY “user.lang”,time
SEGMENTED BY hash(id) ALL NODES;
```

Validation: Customers

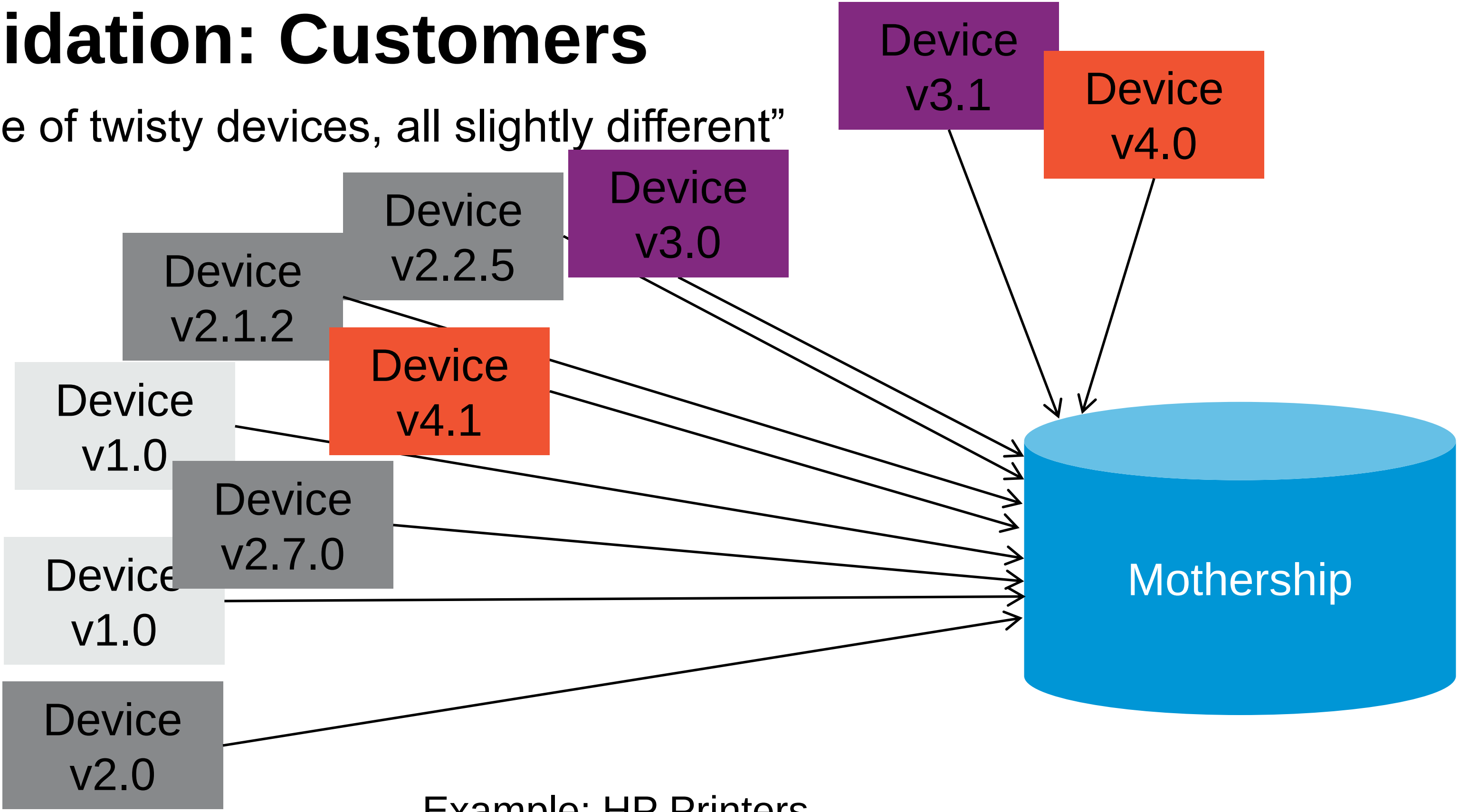
"Finding pain points"





Validation: Customers

“Maze of twisty devices, all slightly different”



Example: HP Printers

Schema evolves, cannot change installed devices

Validation: Results

“Dogfooding”

Usability

Case study: Vertica internal log tables

- Standard base columns (time,txnid,…)
- Variety of payload columns
- Columns vary between versions
- Stored in delimited format with headers

Requirement

Simple load process which works regardless of Vertica version

Example: dc_transaction_ends table

```
            time              |   transaction_id   | is_ddl | rows_written
------------------------------+-------------------+--------+-------------
 2013-07-20 15:51:29.065174-04 | 45035996275768002 | f      |            0
 2013-07-20 15:51:29.119749-04 | 45035996275768020 | t      |            0
 2013-07-20 15:51:29.182637-04 | 45035996275768021 | t      |            0
 2013-07-20 15:51:39.811272-04 | 45035996275768024 | f      |            0
 2013-07-20 15:51:51.213977-04 | 45035996275768025 | f      |            0
 2013-07-20 15:51:51.221161-04 | 45035996275768026 | t      |            0
 2013-07-20 15:51:52.984709-04 | 45035996275768027 | t      |            0
 2013-07-20 15:51:52.995551-04 | 45035996275768028 | t      |            0
 2013-07-20 15:51:54.7341-04   | 45035996275768029 | f      |           94
 2013-07-20 15:52:29.468858-04 | 45035996275768032 | t      |            0
```

Validation: Results

“Hybrid table FTW!”

Usage

For each log file:

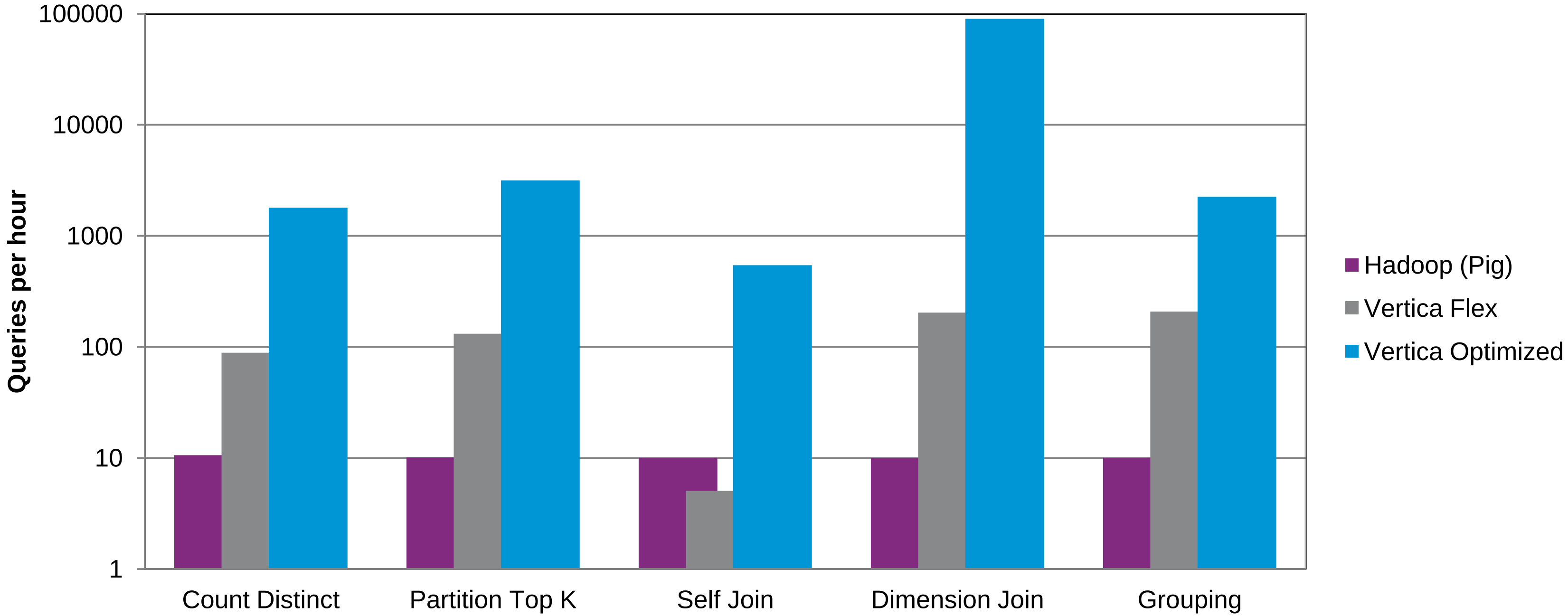
- Create hybrid flex table
- Load file into table

No need for performance optimization:
Important columns already exist

```
CREATE FLEX TABLE dc_table (
    time timestamp,
    node_name varchar(100),
    session_id varchar(100),
    transaction_id int,
    statement_id int
 )
ORDER BY
transaction_id,statement_id,node_name,time
SEGMENTED BY
hash(transaction_id,statement_id) ALL NODES;
```

Validation: Performance of Vertica Flex vs Pig

“I’ll huff and I’ll puff…”



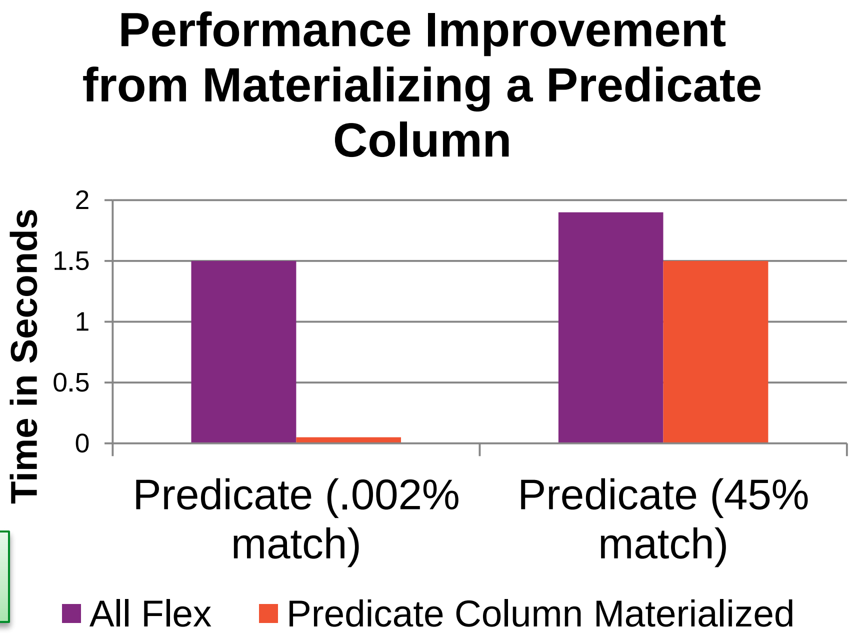
Tweets: 100 hrs of samples, 38GB raw, 17 million rows – **Cluster**: 4 Nodes, 100GB RAM, 8 cores

Validation: Effectiveness of Hybrid Tables

SELECT text FROM tweets
WHERE “user.lang” = ______

- All flex:
  - scan all rows
- Materialize “user.lang”:
  - scan “user.lang” first
  - scan __raw__ for matching

Cost to Store Column: 194 bytes!


Performance Improvement from Materializing a Predicate Column

Conclusions

“What next?”

Current State

- **Support load, exploration, and productization of semi-structured data**
- **Provides vanilla SQL experience key to ecosystem integration**
- **Push button performance optimization with orders of magnitude impact**
- **Shipped in Vertica 7.0 (Dec 2013)**

Future Work

- **Support additional formats**
- **Better algorithms**
  - View Generation
  - Column Materialization
- **Performance tuning**
  - Storage compression
  - SQL Optimizer statistics

We’re hiring!

In case you hadn’t guessed.

Full-time and internship positions available!

We want intelligent, creative developers who aren’t afraid to learn new stuff.

- Everything you just saw came out of some engineer’s head.
- Are you someone like that? Want to work with people like that?

We’ll help you find the right place at Vertica.

- Interview with (and get to know) your teammates
- We want people who know they want us

http://www.vertica.com/about/careers/ | http://www.vertica.com/blog/ | http://my.vertica.com/ | https://vertica.hpwsportal.com/ | resumes@vertica.com

Artwork courtesy of openclipart.org

Thank you! Questions?

Try it yourself with Vertica Community Edition:
http://www.vertica.com/community
Full featured; first 2 terabytes are free!
(1TB Flex, 1TB columnar)

Flex QuickStart Guide at:
http://my.vertica.com/docs/7.0.x/PDF/HP_Vertica_7.0.x_FlextablesQuickstart.pdf

Thanks to the whole Flex Crew:

Ben Vandiver, Shalu Tiwari, Kanti Mann,

Tina Hsu, Adam Seering, Derrick Rice,
Jason Slaunwhite, Sue Francis,

Lyric Doshi, Matt Fuller

Map Functions

Scalar Functions

- MapLookup(map,key) → value
  or *null* if map does not contain key
- MapContainsKey(map,key) → boolean
- MapSize(map) → integer

Transform Functions (multi-row output)

- MapKeys(map) → keys, one per row
- MapValues(map) → values, one per row

Aggregate Functions

- MapAggregate(keys,values) → map

…

Maps can stand in for arrays: use keys 0,1,2,…

Examples:

View the contents of a Flex map:

SELECT MapToString(__raw__) FROM tweets;

Find all unique keys in a map column:

SELECT DINSTINCT key FROM

(SELECT MapKeys(__raw__) OVER() FROM tweets) a;

Handling Nested Structure: Exploding

How do I query nested objects?

```
SELECT MapItems(tags)
    OVER(PARTITION AUTO)
    FROM posts;
```

```
 keys |  values
------+----------
 0    | flex
 1    | vertica
 2    | hp
 3    | big data
(4 rows)
```

```
SELECT text, MapItems(tags)
    OVER(PARTITION BY text, tags)
    FROM posts;
```

```
                    text                    | keys |  values
--------------------------------------------+------+----------
 Giving a talk on Flexible Tables at Brandeis. | 0    | flex
 Giving a talk on Flexible Tables at Brandeis. | 1    | vertica
 Giving a talk on Flexible Tables at Brandeis. | 2    | hp
 Giving a talk on Flexible Tables at Brandeis. | 3    | big data
(4 rows)
```

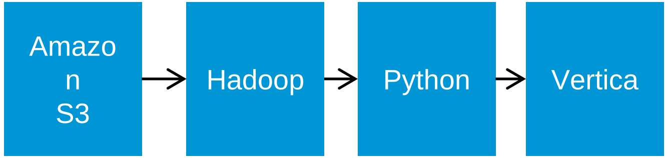
```
{
  "postID": 52737,
  "posterID": 134028,
  "text": "Giving a talk on
Flexible Tables at Brandeis.",

  "tags":
    ["flex", "vertica", "hp", "big
data"],

  "replyPostsIDs":
    [52740, 52756, 52757, 52810]
}
```

Customer Stories

“Of woe and tears”

Gaming Company Scenario

- Event data from games
- Load process:



Amazo n S3 → Hadoop → Python → Vertica

- Repeat for every event type
- Events change: make new tables (tbl1 → tbl2)
- Want to co-locate all events for a game
- All events have 8 known fields

Not just gaming:

- **Advertising**
- **Medical**
- **Web analytics**
- **…**

Common Elements

Customer defines data to be collected

Customer loads & analyzes data in Vertica

Adjusts collection over time

Related Work

- Many customers do ETL to work around
- Postgres, Oracle, and other support JSON – but not in SQL
- Clustrix gets close, but keys don’t look like columns
  http://sergei.clustrix.com/2011/02/clustrix-as-document-store-blending-sql.html
- XML databases – xpath queries aren’t SQL