Building a semantic/metrics layer using Apache Calcite

Julian Hyde (Calcite, Google)

Community over Code • Halifax, Nova Scotia • 2023-10-09

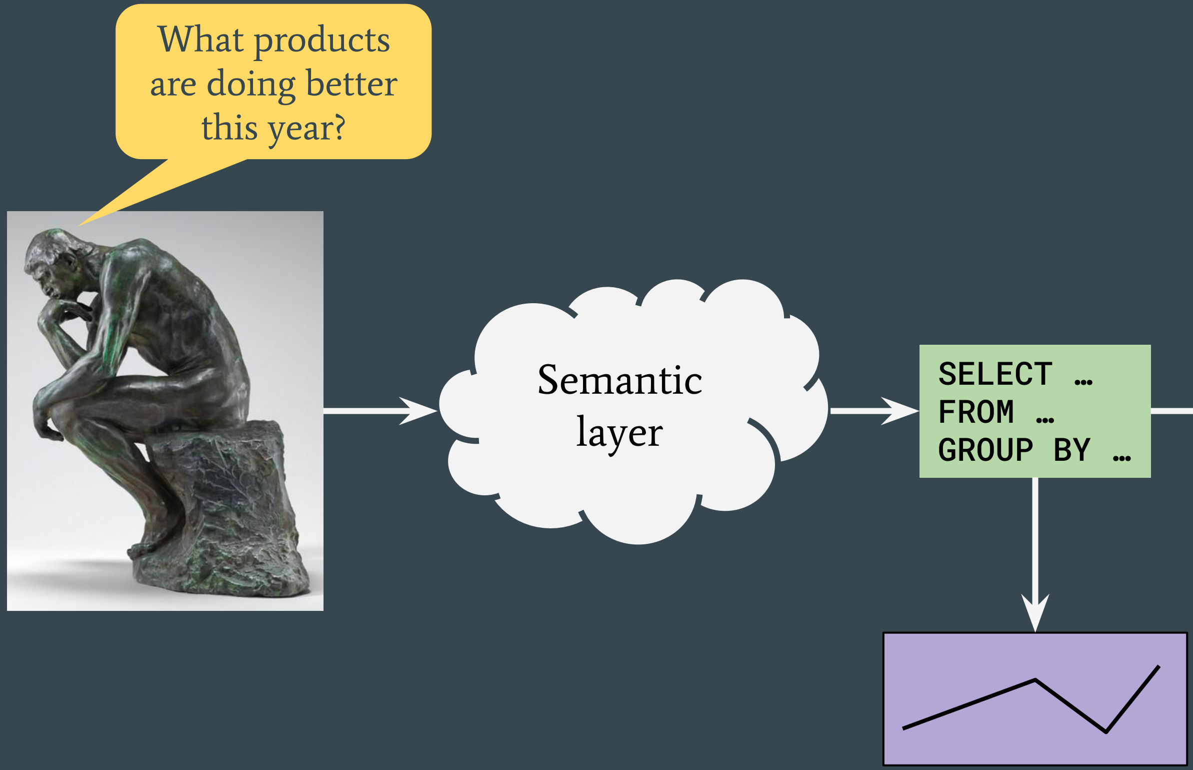
Abstract

A semantic layer, also known as a metrics layer, lies between business users and the database, and lets those users compose queries in the concepts that they understand. It also governs access to the data, manages data transformations, and can tune the database by defining materializations.

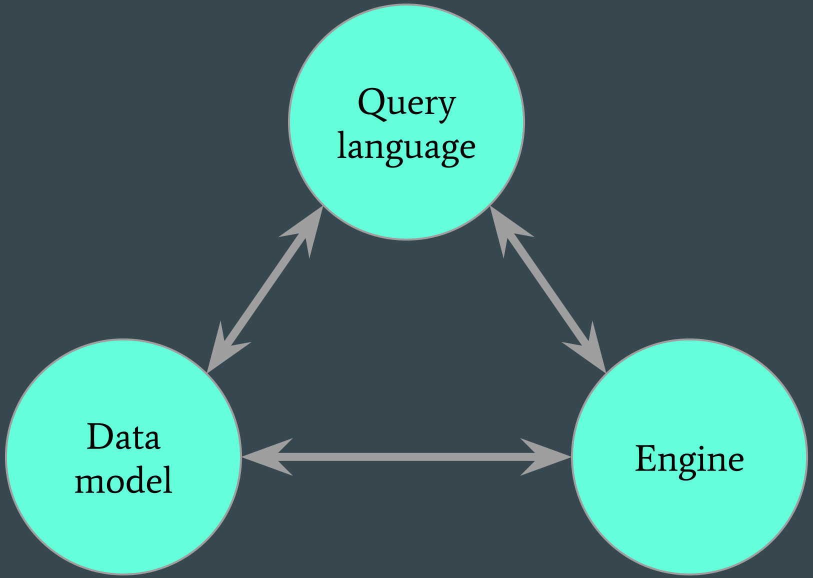
Like many new ideas, the semantic layer is a distillation and evolution of many old ideas, such as query languages, multidimensional OLAP, and query federation.

In this talk, we describe the features we are adding to Calcite to define business views, query measures, and optimize performance.

Julian Hyde is the original developer of Apache Calcite, an open source framework for building data management systems, and Morel, a functional query language. Previously he created Mondrian, an analytics engine, and SQLstream, an engine for continuous queries. He is a staff engineer at Google, where he works on Looker and BigQuery.


What products are doing better this year?
Semantic layer
SELECT …
FROM …
GROUP BY …
Database

Data system = Model + Query + Engine



Agenda

1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

1. **Relational model vs dimensional model**
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

SQL vs BI

BI tools implement their own languages on top of SQL. Why not SQL?

Possible reasons:

- Semantic Model
- Control presentation / visualization
- Governance
- Pre-join tables
- Define reusable calculations
- Ask complex questions in a concise way

Processing BI in SQL

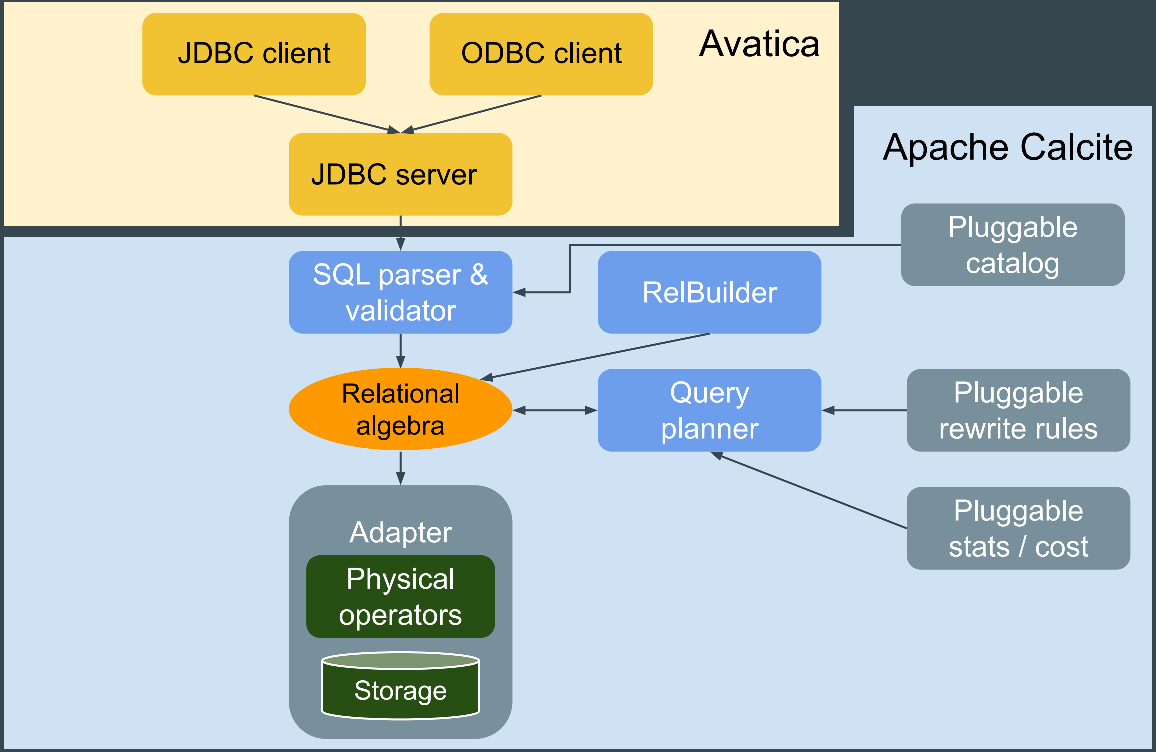
Why we should do it

- Move processing, not data
- Cloud SQL scale
- Remove data lag
- SQL is open

Why it's hard

- Different paradigm
- More complex data model
- Can't break SQL

Apache Calcite



Core – Operator expressions (relational algebra) and planner (based on Cascades)

External – Data storage, algorithms and catalog

Optional – SQL parser, JDBC & ODBC drivers

Extensible – Planner rewrite rules, statistics, cost model, algebra, UDFs

Used by

HIVE

beam

APACHE STORM™

NGDATA

APACHE DRILL

HerdDB

Apache Apex™

druid

Apache Kylin

APACHE PHOENIX

Apache Solr

Alibaba Group

Tencent

Apache Flink

samza

CASCADING

Qubole

ANT FINANCIAL

dremio

TBDS

omni sci

SQL Gremlin

UBER

Connects to

JDBC

APACHE Spark

splunk>

CSV

cassandra

{JSON}

elasticsearch

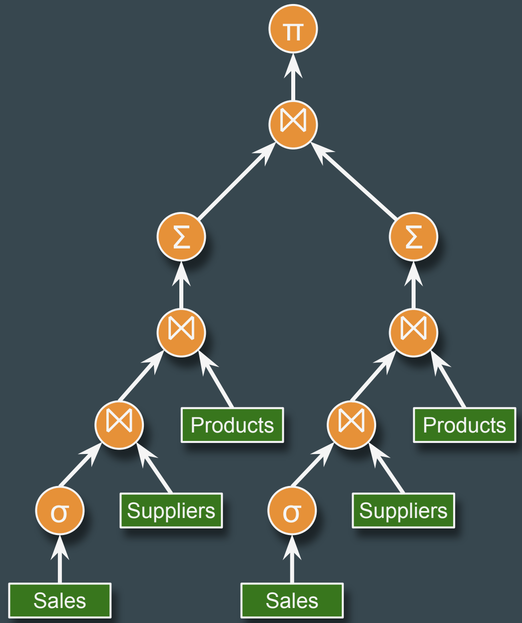
mongoDB

APACHE GEODE

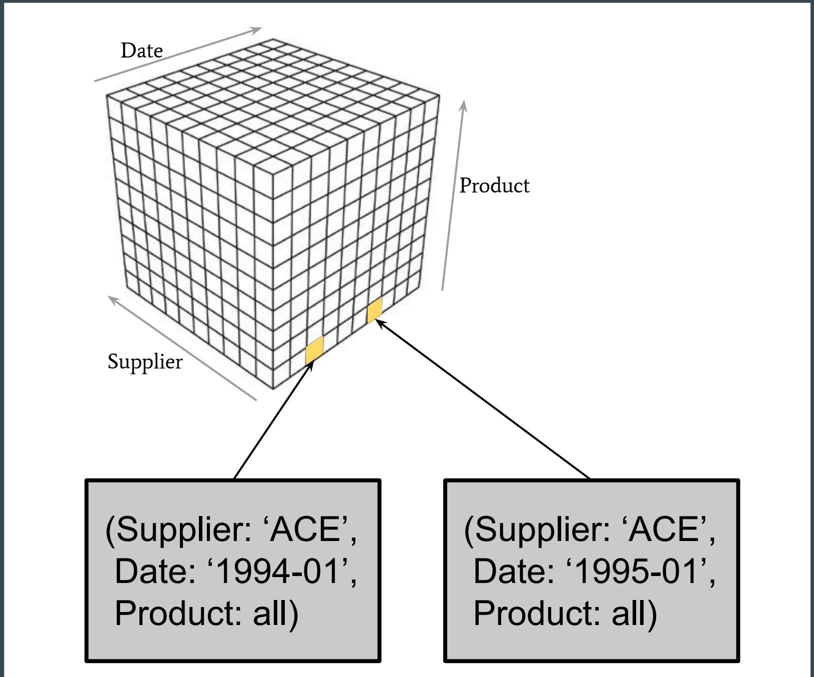
Pasta machine vs Pizza delivery

Bottom-up vs Top-down query

Relational algebra (bottom-up)



Multidimensional (top-down)



1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

1. Relational model vs dimensional model
2. **Adding measures to SQL**
3. Machine-learning patterns
4. Semantic layer

Some multidimensional queries

- Give the total sales for each product in each quarter of 1995. (Note that quarter is a function of date).
- For supplier "Ace" and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.
- For each product give its market share in its category today minus its market share in its category in October 1994.
- Select top 5 suppliers for each product category for last year, based on total sales.
- For each product category, select total sales this month of the product that had highest sales in that category last month.
- Select suppliers that currently sell the highest selling product of last month.
- Select suppliers for which the total sale of every product increased in each of last 5 years.
- Select suppliers for which the total sale of every product category increased in each of last 5 years.

From [Agrawal1997]. Assumes a database with dimensions {supplier, date, product} and measure {sales}.)

Some multidimensional queries

- Give the total sales for each product in each quarter of 1995. (Note that quarter is a function of date).
- **For supplier "Ace" and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.**
- For each product give its market share in its category today minus its market share in its category in October 1994.
- Select top 5 suppliers for each product category for last year, based on total sales.
- For each product category, select total sales this month of the product that had highest sales in that category last month.
- Select suppliers that currently sell the highest selling product of last month.
- Select suppliers for which the total sale of every product increased in each of last 5 years
- Select suppliers for which the total sale of every product category increased in each of last 5 years.

From [Agrawal1997]. Assumes a database with dimensions {supplier, date, product} and measure {sales}.)

Query:

- For supplier "Ace" and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.

SQL

```
SELECT p.prodId,
  s95.sales,
  (s95.sales - s94.sales) / s95.sales
FROM (
  SELECT p.prodId, SUM(s.sales) AS sales
  FROM Sales AS s
    JOIN Suppliers AS u USING (suppId)
    JOIN Products AS p USING (prodId)
  WHERE u.name = 'ACE'
  AND FLOOR(s.date TO MONTH) = '1995-01-01'
  GROUP BY p.prodId) AS s95
LEFT JOIN (
  SELECT p.prodId, SUM(s.sales) AS sales
  FROM Sales AS s
    JOIN Suppliers AS u USING (suppId)
    JOIN Products AS p USING (prodId)
  WHERE u.name = 'ACE'
  AND FLOOR(s.date TO MONTH) = '1994-01-01'
  GROUP BY p.prodId) AS s94
USING (prodId)
```

MDX

```
WITH MEMBER [Measures].[Sales Last Year] =
    ([Measures].[Sales],
     ParallelPeriod([Date], 1, [Date].[Year]))
  MEMBER [Measures].[Sales Growth] =
    ([Measures].[Sales]
        - [Measures].[Sales Last Year])
      / [Measures].[Sales Last Year]
SELECT [Measures].[Sales Growth] ON COLUMNS,
  [Product].Members ON ROWS
FROM [Sales]
WHERE [Supplier].[ACE]
```

Query:

- For supplier “Ace” and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.

SQL

```
SELECT p.prodId,
  s95.sales,
  (s95.sales - s94.sales) / s95.sales
FROM (
  SELECT p.prodId, SUM(s.sales) AS sales
  FROM Sales AS s
    JOIN Suppliers AS u USING (suppId)
    JOIN Products AS p USING (prodId)
  WHERE u.name = ‘ACE’
  AND FLOOR(s.date TO MONTH) = ‘1995-01-01’
  GROUP BY p.prodId) AS s95
LEFT JOIN (
  SELECT p.prodId, SUM(s.sales) AS sales
  FROM Sales AS s
    JOIN Suppliers AS u USING (suppId)
    JOIN Products AS p USING (prodId)
  WHERE u.name = ‘ACE’
  AND FLOOR(s.date TO MONTH) = ‘1994-01-01’
  GROUP BY p.prodId) AS s94
USING (prodId)
```

SQL with measures

```
SELECT p.prodId,
   SUM(s.sales) AS MEASURE sumSales,
   sumSales AT (SET FLOOR(s.date TO MONTH)
                     = ‘1994-01-01’)
     AS MEASURE sumSalesLastYear
FROM Sales AS s
  JOIN Suppliers AS u USING (suppId)
  JOIN Products AS p USING (prodId))
WHERE u.name = ‘ACE’
AND FLOOR(s.date TO MONTH) = ‘1995-01-01’
GROUP BY p.prodId
```

Self-joins, correlated subqueries, window aggregates, measures

Window aggregate functions were introduced to save on self-joins.

Some DBs rewrite scalar subqueries and self-joins to window aggregates [Zuzarte2003].

Window aggregates are more concise, easier to optimize, and often more efficient.

However, window aggregates can only see data that is from the same table, and is allowed by the `WHERE` clause. Measures overcome that limitation.

```
SELECT *
FROM Employees AS e
WHERE sal > (
  SELECT AVG(sal)
  FROM Employees
  WHERE deptno = e.deptno)
```

```
SELECT *
FROM Employees AS e
WHERE sal > AVG(sal)
 OVER (PARTITION BY deptno)
```

A measure is… ?

… a column with an aggregate function.

```
SUM(sales)
```

A measure is… ?

… a column with an aggregate function.

```
SUM(sales)
```

… a column that, when used as an expression, knows how to aggregate itself.

```
(SUM(sales) - SUM(cost))
   / SUM(sales)
```

A measure is… ?

… a column with an aggregate function.

```
SUM(sales)
```

… a column that, when used as an expression, knows how to aggregate itself.

```
(SUM(sales) - SUM(cost))
   / SUM(sales)
```

… a column that, when used as expression, can evaluate itself in any context.

```
(SELECT SUM(forecastSales)
  FROM SalesForecast AS s
  WHERE predicate(s))

ExchService$ClosingRate(
  'USD', 'EUR', sales.date)
```

A measure is...

... a column with an aggregate function.

```
SUM(sales)
```

... a column that, when used as an expression, knows how to aggregate itself.

```
(SUM(sales) - SUM(cost))
    / SUM(sales)
```

... a column that, when used as expression, can evaluate itself in any context.

```
(SELECT SUM(forecastSales)
  FROM SalesForecast AS s
  WHERE predicate(s))
```

Its value depends on, and only on, the predicate placed on its dimensions.

```
ExchService$ClosingRate(
  'USD', 'EUR', sales.date)
```

Table model

Tables are SQL's fundamental model.

The model is closed – queries consume and produce tables.

Tables are opaque – you can't deduce the type, structure or private data of a table.

```
SELECT MOD(deptno, 2) = 0 AS evenDeptno, avgSal2
FROM
    Employees3

WHERE deptno < 30
```

Table model

Tables are SQL's fundamental model.

The model is closed – queries consume and produce tables.

Tables are opaque – you can't deduce the type, structure or private data of a table.

```
SELECT MOD(deptno, 2) = 0 AS evenDeptno, avgSal2
FROM
    SELECT deptno, AVG(avgSal) AS avgSal2
    FROM
        SELECT deptno, job,
          AVG(sal) AS avgSal
        FROM Employees
        GROUP BY deptno, job
    GROUP BY deptno
WHERE deptno < 30
```

Table model with measures

We propose to allow any table and query to have measure columns.

The model is closed – queries consume and produce tables-with-measures.

Tables-with-measures are semi-opaque – you can't deduce the type, structure or private data, but you can evaluate the measure in any context that can be expressed as a predicate on the measure's dimensions.

```
SELECT e.deptno, e.job, d.dname, e.avgSal / e.deptAvgSal
FROM  AnalyticEmployees2
  AS e
JOIN Departments AS d USING (deptno)
WHERE d.dname <> 'MARKETING'
GROUP BY deptno, job
```

Table model with measures

We propose to allow any table and query to have measure columns.

The model is closed – queries consume and produce tables-with-measures.

Tables-with-measures are semi-opaque – you can't deduce the type, structure or private data, but you can evaluate the measure in any context that can be expressed as a predicate on the measure's dimensions.

```
SELECT e.deptno, e.job, d.dname, e.avgSal / e.deptAvgSal
FROM
       SELECT *,
         avgSal AS MEASURE avgSal,
         avgSal AT (ALL deptno) AS MEASURE deptAvgSal
       FROM
              SELECT *,
                AVG(sal) AS MEASURE avgSal
              FROM Employees
  AS e
JOIN Departments AS d USING (deptno)
WHERE d.dname <> 'MARKETING'
GROUP BY deptno, job
```

Syntax

expression `AS MEASURE` – defines a measure in the `SELECT` clause

`AGGREGATE(`*measure*`)` – evaluates a measure in a `GROUP BY` query

expression `AT` (*contextModifier…*) – evaluates expression in a modified context

contextModifier ::=
  `ALL`
 | `ALL` *dimension* [, *dimension…*]
 | `ALL EXCEPT` *dimension* [, *dimension…*]
 | `SET` *dimension* = [`CURRENT`] *expression*
 | `VISIBLE`

aggFunction(*aggFunction*(*expression*) `PER` *dimension*) – multi-level aggregation

Plan of attack

1. Add measures to the table model, and allow queries to use them
    - Measures are defined only via the Table API
2. Define measures using SQL expressions (`AS MEASURE`)
    - You can still define them using the Table API
3. Context-sensitive expressions (`AT`)

Semantics

0. We have a measure *M*, value type *V*, in a table *T*.

1. System defines a row type *R* with the non-measure columns.

2. System defines an auxiliary function for *M*. (Function is typically a scalar subquery that references the measure's underlying table.)

```
CREATE VIEW AnalyticEmployees AS
  SELECT *, AVG(sal) AS MEASURE avgSal
  FROM Employees
```

```
CREATE TYPE R AS
  ROW (deptno: INTEGER, job: VARCHAR)
```

```
CREATE FUNCTION computeAvgSal(
    rowPredicate: FUNCTION<R, BOOLEAN>) =
  (SELECT AVG(e.sal)
    FROM Employees AS e
    WHERE APPLY(rowPredicate, e))
```

Semantics (continued)

3. We have a query that uses *M*.

```
SELECT deptno,
  avgSal
   / avgSal AT (ALL deptno)
FROM AnalyticEmployees AS e
GROUP BY deptno
```

4. Substitute measure references with calls to the auxiliary function with the appropriate predicate

```
SELECT deptno,
  computeAvgSal(r ->□(r.deptno = e.deptno))
    / computeAvgSal(r □ TRUE))
FROM AnalyticEmployees AS e
GROUP BY deptno
```

5. Planner inlines `computeAvgSal` and scalar subqueries

```
SELECT deptno, AVG(sal) / MIN(avgSal)
FROM (
  SELECT deptno, sal,
    AVG(sal) OVER () AS avgSal
  FROM Employees)
GROUP BY deptno
```

Calculating at the right grain

Example	Formula	Grain
Computing the revenue from units and unit price	`units * pricePerUnit AS revenue`	Row
Sum of revenue (additive)	`SUM(revenue) AS MEASURE sumRevenue`	Top
Profit margin (non-additive)	`(SUM(revenue) - SUM(cost)) / SUM(revenue) AS MEASURE profitMargin`	Top
Inventory (semi-additive)	`SUM(LAST_VALUE(unitsInStock) PER inventoryDate) AS MEASURE sumInventory`	Intermediate
Daily average (weighted average)	`AVG(sumRevenue PER orderDate) AS MEASURE dailyAvgRevenue`	Intermediate

Subtotals & visible

```
SELECT deptno, job,
  SUM(sal), sumSal
FROM (
  SELECT *,
    SUM(sal) AS MEASURE sumSal
  FROM Employees)
WHERE job <> 'ANALYST'
GROUP BY ROLLUP(deptno, job)
ORDER BY 1,2
```

deptno	job	SUM(sal)	sumSal
10	CLERK	1,300	1,300
10	MANAGER	2,450	2,450
10	PRESIDENT	5,000	5,000
10		8,750	8,750
20	CLERK	1,900	1,900
20	MANAGER	2,975	2,975
20		4,875	10,875
30	CLERK	950	950
30	MANAGER	2,850	2,850
30	SALES	5,600	5,600
30		9,400	9,400
		20,750	29,025

Measures by default sum ALL rows;
Aggregate functions sum only VISIBLE rows

Visible

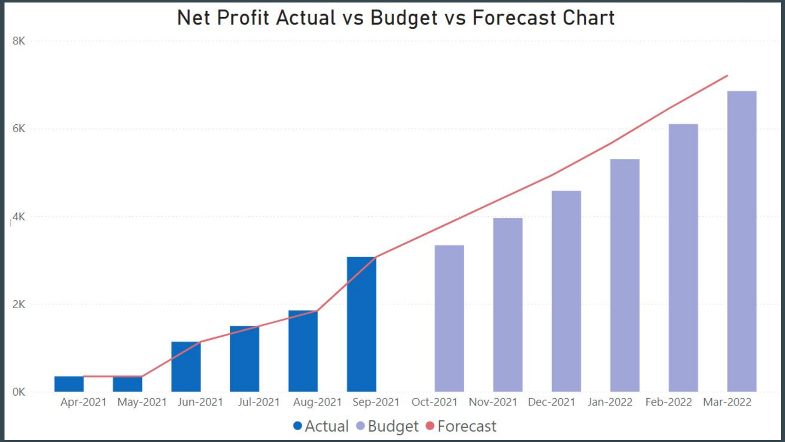
Expression	Example	Which rows?
Aggregate function	`SUM(sal)`	Visible only
Measure	`sumSal`	All
AGGREGATE applied to measure	`AGGREGATE(sumSal)`	Visible only
Measure with VISIBLE	`sumSal AT (VISIBLE)`	Visible only
Measure with ALL	`sumSal AT (ALL)`	All

1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

1. Relational model vs dimensional model
2. Adding measures to SQL
3. **Machine-learning patterns**
4. Semantic layer

Forecasting

A forecast is simply a measure whose value at some point in the future is determined, in some manner, by a calculation on past data.




```
SELECT year(order_date), product, revenue,
    forecast_revenue
FROM Orders
WHERE year(order_date) BETWEEN 2018 AND 2022
GROUP BY 1, 2
```

Forecasting: implementation

Problems

1. Predictive model under the forecast (such as ARIMA or linear regression) is probably too expensive to re-compute for every query

2. We want to evaluate forecast for regions for which there is not (yet) any data

Solutions

1. Amortize the cost of running the model using (some kind of) materialized view

2. Add a SQL `EXTEND` operation to implicitly generate data

```
SELECT year(order_date), product, revenue,
forecast_revenue
FROM Orders EXTEND (order_date)
WHERE year(order_date) BETWEEN 2021 AND 2025
GROUP BY 1, 2
```

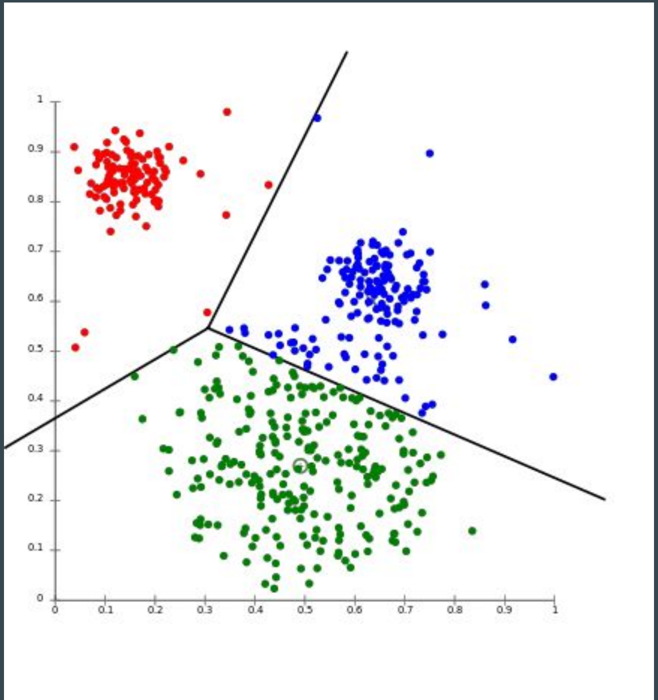
Clustering

```
SELECT id, firstName, lastName, firstPurchaseDate,
    latitude, longitude, revenue, region
FROM Customers;
```

that points that are in th[...]e cluster are, by some me[...]stant from points in other clusters.

`region` is a measure (based on the centroid of a cluster)

```
CREATE VIEW Customers AS
SELECT *,
    KMEANS(3, ROW(latitude, longitude)) AS MEASURE
region,
    (SELECT SUM(revenue)
     FROM Orders AS o
     WHERE o.customerId = c.id) AS MEASURE revenue
FROM BaseCustomers AS c;
```



Clustering: fixing the baseline

The measure is a little too dynamic. Fix the baseline, so that cluster centroids don’t change from one query to the next:

```
SELECT id, firstName, lastName, firstPurchaseDate,
    latitude, longitude,
    region AT (ALL
               SET YEAR(firstPurchaseDate) = 2020)
FROM Customers;
```

Clustering: amortizing the cost

To amortize the cost of the algorithm, create a materialized view:

```
CREATE MATERIALIZED VIEW CustomersMV AS
SELECT *,
    region AT (ALL
               SET YEAR(firstPurchaseDate) = 2020) AS region2020
FROM Customers;
```

Classification

Classification predicts the value of a variable given the values of other variables and a model trained on similar data.

For example, does a particular household own a dog?

Whether they have a dog may depend on household income, education level, location of the household, purchasing history of the household.

```
SELECT last_name, zipcode, probability_that_household_has_dog,
    expected_dog_count
FROM Customers
WHERE state = ‘AZ’
```

Classification: training & running

Pseudo-function CLASSIFY:

```
FUNCTION classify(isTraining, actualValue, features)
```

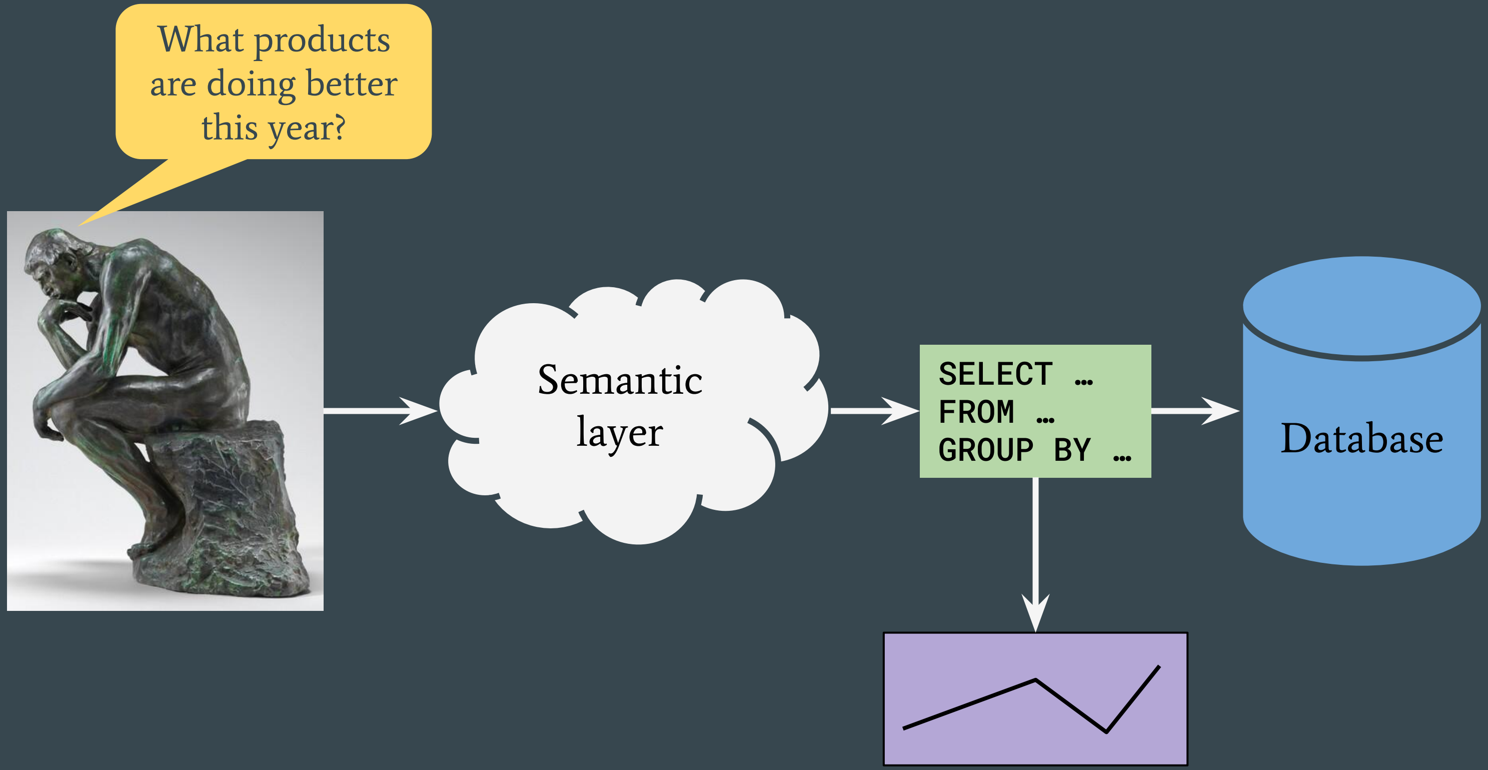
A SQL view can both train the algorithm (given the correct result) and execute it (generating the result from features):

We assume that **has_dog** has the correct value for customers who purchased on 2023-05-01

```
SELECT last_name, zipcode,
    CLASSIFY(firstPurchaseDate = ‘2023-05-01’,
        has_dog,
        ROW (zipcode, state, income_level, education_level))
      AS probability_that_household_has_dog,
    expected_dog_count
FROM Customers
GROUP BY state
```

1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. Semantic layer

1. Relational model vs dimensional model
2. Adding measures to SQL
3. Machine-learning patterns
4. **Semantic layer**


What products are doing better this year?
Semantic layer
SELECT …
FROM …
GROUP BY …
Database

Natural language query

Example query:

> “Show me the top 5 products in each state where revenue declined since last year”

“Revenue” is a measure.

“Declined since last year” asks whether

```
revenue - revenue AT (SET year = CURRENT year - 1)
```

is negative.

“Products in each state” establishes the filter context.

Semantic model for natural-language query

Analyza: Exploring Data with Con

Kedar Dhamdhere
Google Research
Mountain View, USA
<firstname>@google.com

Kevin S. McCurley
Google Research
Mountain View, USA
<lastname>@google.com

Mukund Sundararajan
Google Research
Mountain View, USA
mukunds@google.com

Qiqi Yan
Google Resea
Mountain View
qiqiyan@googl

ABSTRACT

We describe Analyza, a system that helps lay users explore data. Analyza has been used within two large real world systems. The first is a question-and-answer feature in a spreadsheet product. The second provides convenient access to a revenue/inventory database for a large sales force. Both user bases consist of users who do not necessarily have coding skills, demonstrating Analyza's ability to democratize access to data.

We discuss the key design decisions in implementing this system. For instance, how to mix structured and natural language modalities, how to use conversation to disambiguate and simplify querying, how to rely on the "semantics" of the data to compensate for the lack of syntactic structure, and how to efficiently curate the data.

Author Keywords

Exploratory data analysis; Natural language

ACM Classification Keywords

H.5.2. User interfaces: Natural language

We should note that th
cision making, includi
scribed by complex hig
ten require looking at
problem is no longer j
to uncover patterns, bu
data set that provides th
that this demands a hig
queries may be quickly
this paper is to describe

There have been sever
of exploring data [12,
tween these is that it is
including discovery, in
marization and visualiz
complete characterizati
beyond the scope of thi

Data discovery The us
of data exist, and wh
on the data. As organizations continue to gather more and
more data, this problem is increasing in importance [16].

Data inquiry The user should be able to examine different

Metadata store

This holds three types of metadata. The first type of metadata is the set of intent words, such as "top", "compare", "list" etc., which helps us disambiguate the user's question. The second type of metadata is *schema* information, similar to a SQL database schema, with additional information about the type of the column (e.g. is it a metric, dimension, etc), data formats (e.g. should the number be formatted as a dollar amount), date range defaults, etc. The third type of metadata is a *knowledge base* about entities in the data (e.g., "fr"), i.e., the columns they belong to (e.g., "country"), the lexicon for this entity (e.g., "france"). We also derive an additional lexicon for entities in our knowledge base by joining with a much larger knowledge graph [4, 34].

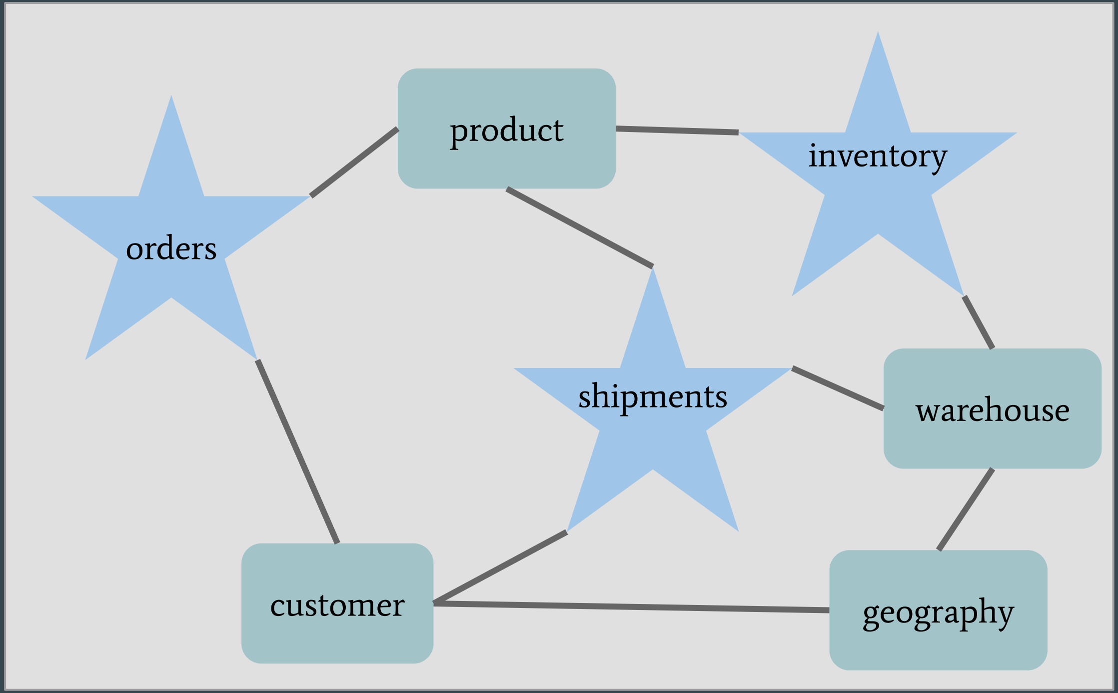
Extended semantic model

"Show me regions where customers ordered low-inventory products last year"

Data model is a graph that connects business views:

- **Business views** – tables, possibly based on joins, with measures, and display hints
- **Domains** – shared attributes
- **Entities** – shared dimensions
- **Metrics** – shared measures
- Ontology/synonyms

Do we need a new query language?



Summary

Measures in SQL allow…

- concise queries without self-joins
- top-down evaluation
- reusable calculations
- natural-language query

…and don't break SQL

A semantic model is table with measures, accessed via analytic SQL..

A extended semantic model links such tables into a knowledge graph.

Resources

Papers

- "Modeling multidimensional databases" (Agrawal, Gupta, and Sarawagi, 1997)
- "WinMagic: Subquery Elimination Using Window Aggregation" (Zuzarte, Pirahash, Ma, Cheng, Liu, and Wong, 2003)
- "Analyza: Exploring Data with Conversation" (Dhamdhere, McCurley, Nahmias, Sundararajan, Yan, 2017)

Issues

- [CALCITE-4488] WITHIN DISTINCT clause for aggregate functions (experimental)
- [CALCITE-4496] Measure columns ("SELECT … AS MEASURE")
- [CALCITE-5105] Add MEASURE type and AGGREGATE aggregate function
- [CALCITE-5155] Custom time frames
- [CALCITE-xxxx] PER operator
- [CALCITE-5692] Add AT operator, for context-sensitive expressions
- [CALCITE-5951] PRECEDES function, for period-to-date calculations

Thank you!
Any questions?

@julianhyde
@ApacheCalcite
https://calcite.apache.org