Yieldigo

amazon web services

How retailer determine price of products?

1.28 €

1.40 €

Cost plus method or **Price follower method**

But is it the price that shoppers are willing to pay and price that ensures accomplishing current company targets?

Who we are?

AI price optimization for retail

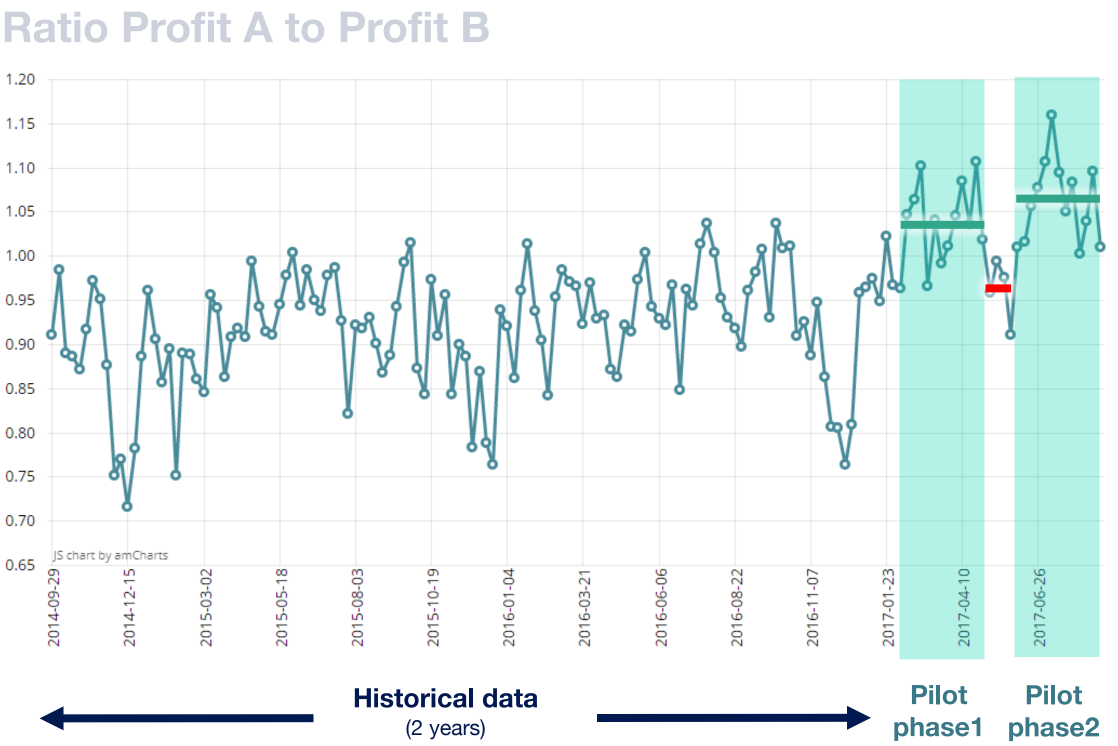
Yieldigo is a SaaS solution that helps retailers to set optimal prices based on their shoppers behavior by using advanced mathematical algorithms.

5 – 13%	**3 – 8%**	**2**
Profit increase	Revenue increase	Months to meet results
while preserving Price Index, Revenue and Sales trends	while preserving given minimal Margin rate	

Price optimization process

1. Yieldigo algorithm
2. Training on client data
3. Mapping customers behavior
4. Setting client’s strategy
5. Regular store repricing
6. Machine Learning

Yieldigo brings measurable improvements



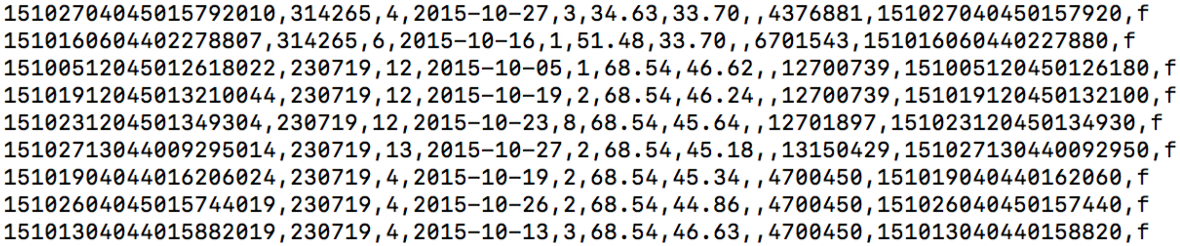
A/B testing on two groups of stores.
Group A prices are managed by Yieldigo.

● **Tested assortment** (Sweats)

Use-case:
Client data analysis
before pilot

Problems

- Milions of transactions need to be analyzed before pilot starts.
- Data is not in database yet.



```
15102704045015792010,314265,4,2015-10-27,3,34.63,33.70,,4376881,151027040450157920,f
```

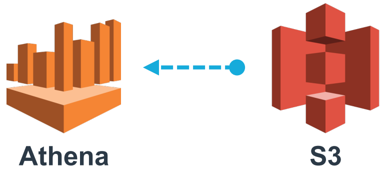
Service overview

- **Athena** – serverless, Presto SQL engine, allow using columnar formats, data partitioning (Apache Hive)
- **Apache Parquet** – columnar format, more than 80% compression, needs data conversion (EMR)

By using Parquet format can be query more than 10 times faster with up to 99% cost savings.

Table definition

```
CREATE EXTERNAL TABLE transactions (
        id bigint,
        article_id int,
        site_id int,
        transaction_date date,
        quantity double,
        price double,
        supplier_price double,
        promo_type_id int,
        client_id int,
        basket_id bigint,
        is_irregular_price bool
)
STORED AS PARQUET
LOCATION 's3://yieldigo-transactions/sample/parquet/'
tblproperties ("parquet.compress"="SNAPPY");
```



Use-case:
Aggregating views from data warehouse

Problems

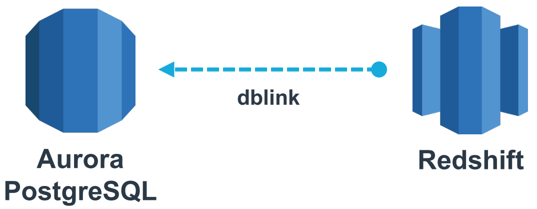
- Relational database is not able to process big data analysis quickly.
- We need to have many data aggregation views refreshed every day.

Service overview

- **Redshift** – high-performance, petabyte-scale data warehouse service (OLAP), columnar architecture, massively parallel processing (MPP), shared-nothing architecture
- **Aurora PostgreSQL** – transactional (OLTP), row-based architecture

Database connection

```
CREATE EXTENSION postgres_fdw;
CREATE EXTENSION dblink;
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host 'yieldigo.[redacted].redshift.amazonaws.com', port '5439', dbname 'yieldigo', sslmode 'require');
CREATE USER MAPPING FOR yieldigo
        SERVER foreign_server
        OPTIONS (user 'yieldigo', password '???');
```



Aurora PostgreSQL

dblink

Redshift

Creating views from Redshift

```
CREATE OR REPLACE VIEW site_sales AS
SELECT *
FROM dblink ('foreign_server',$REDSHIFT$
    SELECT site_id, transaction_date,
        ROUND(SUM((supplier_price) * quantity), 2) costs,
        ROUND(SUM(quantity), 2) quantity,
        ROUND(SUM(quantity * Price), 2) revenue,
        ROUND(SUM(CASE WHEN promo_type_id IS NULL THEN (supplier_price) * quantity ELSE 0 END), 2) regular_costs,
        ROUND(SUM(CASE WHEN promo_type_id IS NULL THEN quantity ELSE 0 END), 2) regular_quantity,
        ROUND(SUM(CASE WHEN promo_type_id IS NULL THEN quantity * price ELSE 0 END), 2) regular_revenue
    FROM transactions tr
    GROUP BY site_id, date
$REDSHIFT$) AS t1 (site_id int, transaction_date date,
        costs decimal, quantity decimal, revenue decimal,
        regular_costs decimal, regular_quantity decimal,
        regular_revenue decimal
);
```

Use-case:
Computing of optimal prices

Problems

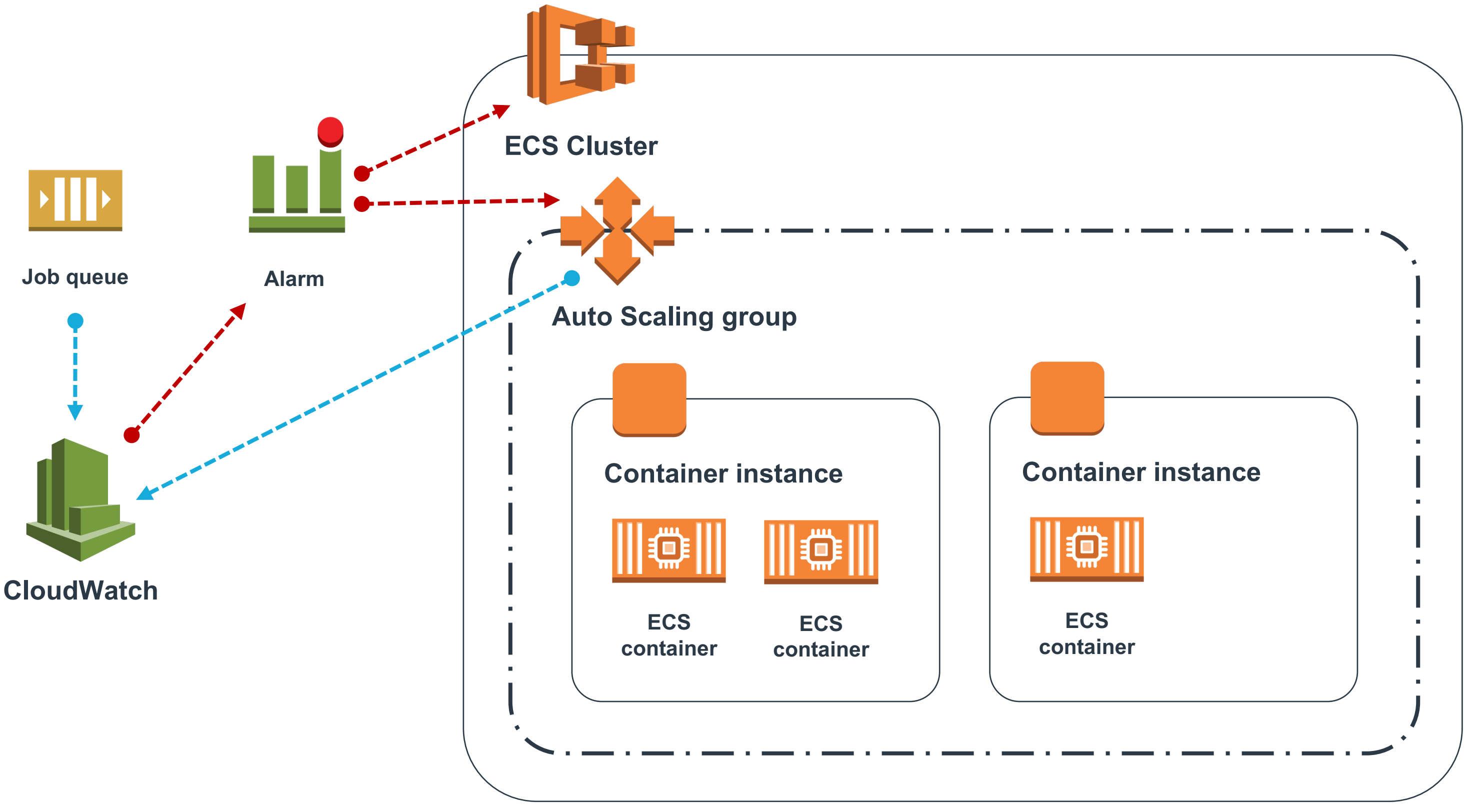
- Thousands of optimization tasks must be computed in short time to delivery results to clients fast.
- Infrastructure must be cost effective and allways ready for big workloads (lot of clients, different run scenarios).

Service overview

- **Elastic Container Service** – highly scalable, fast, Docker container management service, EC2 and Fargate launch type
- **Elastic Container Register** – Docker container registry

Fargate launch type allows to run containers without the need to provision and manage the backend infrastructure.

Architecture overview



Thank you!

Feel free to contact me at

jiri@yieldigo.com

Yieldigo