Programming Project Databases
Database diagram Team 6

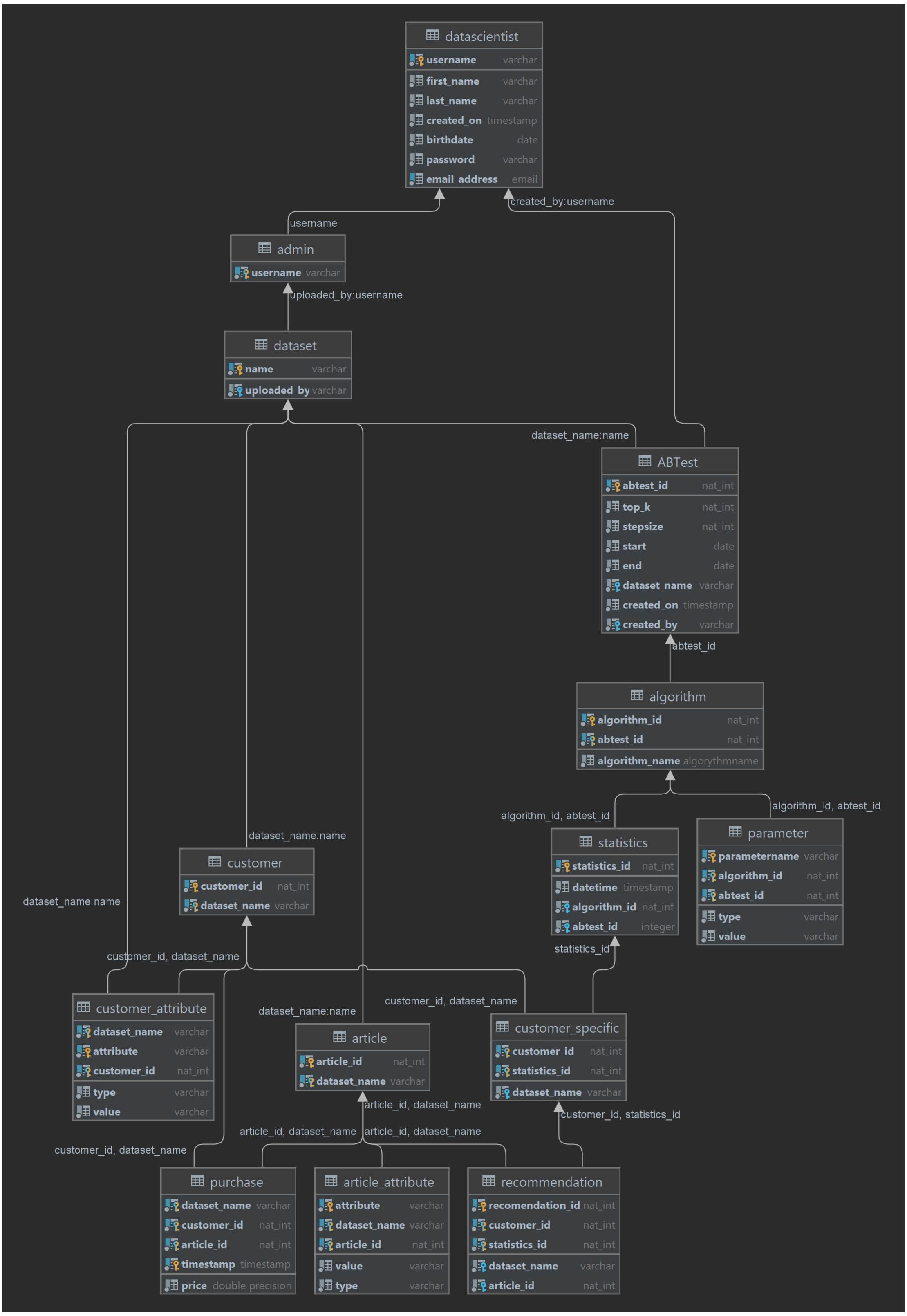
Jorden Van Handenhoven Mohammed Shakleya
Said Yandarbiev Sam Roggeman William Steklein

May 29, 2022

Contents

1 Introduction

To make our database diagram we used datagrip, we also used datagrip to make a diagraph from our sql-input. Our full diagram below. The included images show both the structures and connections of the tables in questions. The indexes that are present are also showed.



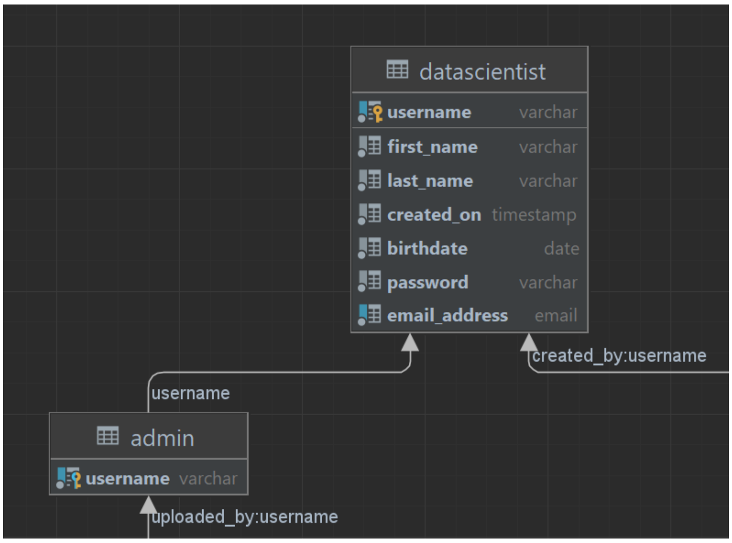
2 Users

2.1 datascientist

First we have the datascientist which all have a unique username (PK). They can log in onto the site and create ABTests, view their history and look at the datasets. Every user has a first and last name, creation date of the account, birth-date and password and unique email-address.

2.2 Admin

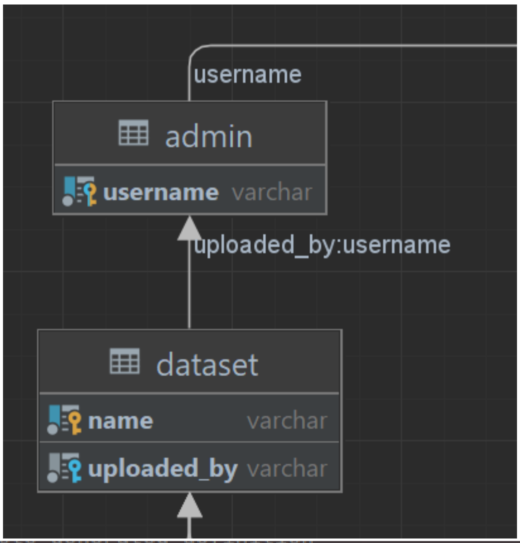
An admin has a 'isa' relationship with a datascientist. An admin s the only one who can upload datasets of purchases, customers and articles. Admin is connected through a foreign key to datascientist, which is also its only field and its primary key.



```sql
create table datascientist
(
    username         varchar                 not null
        primary key,
    datascientist_id serial
        unique,
    first_name       varchar                 not null,
    last_name        varchar                 not null,
    created_on       timestamp default now() not null,
    birthdate        date                    not null,
    password         varchar                 not null,
    email_address    email                   not null
        unique
);

create index idx_user_username_id
    on datascientist (username);

create table admin
(
    username varchar not null
        primary key
        references datascientist
            on update cascade on delete cascade
);
```

3 Dataset

A dataset has a dataset-name as primary key. It is connected to an admin through a foreign key "uploaded-by" as only admins can upload such a set.



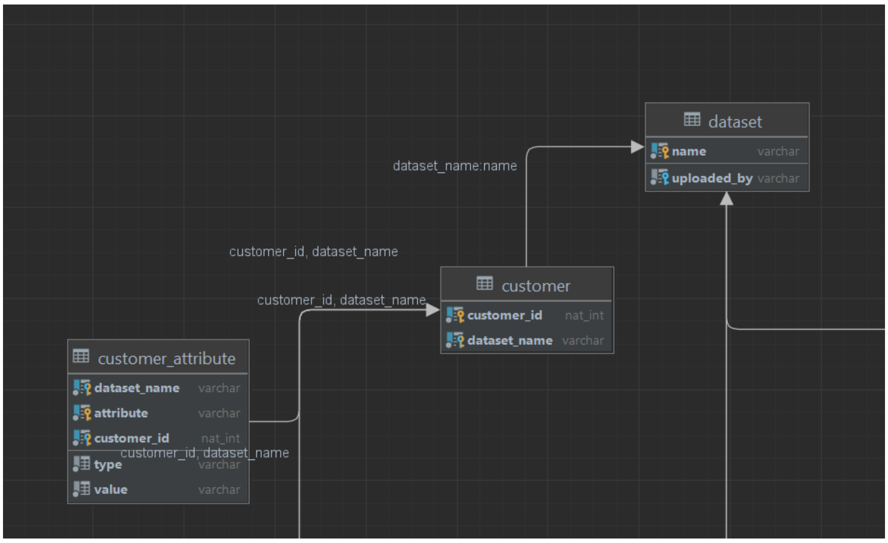

```
create table dataset
(
    name        varchar not null
        primary key,
    uploaded_by varchar not null
        references admin
            on update cascade on delete cascade
);

alter table dataset
    owner to app;

create index idx_dataset_uploader
    on dataset (uploaded_by);
```

3.1 Customers

A customer has unique_customer_id as primary key, this is unique through the whole database, aside of that it also has a secondary key namely a composed of a dataset-name (foreign key to dataset) and a customer-id to differentiate between customers in a given dataset.

3.1.1 Customer-Attribute

Customer Attribute is where we store the actual (dynamic) data of a customer. It is a weak entity of customer and therefore contains a foreign key to a customer (customer-id, dataset-name). On top of this foreign key, the "attribute" is added which is the name of the field/attribute. So the final primary/composite key is (customer-id, dataset-name, attribute). It has a type and a value so it can be used as intended.



```
create table customer
(
    unique_customer_id serial
        primary key,
    customer_id        nat_int not null,
    dataset_name       varchar not null
        references dataset
            on update cascade on delete cascade,
    unique (customer_id, dataset_name)
);

alter table customer
    owner to app;

create index idx_customer_dataset_name
    on customer (dataset_name);

create index idx_customer_unique_id
    on customer (unique_customer_id);

create index idx_customer_dataset_id
    on customer (customer_id, dataset_name);
```

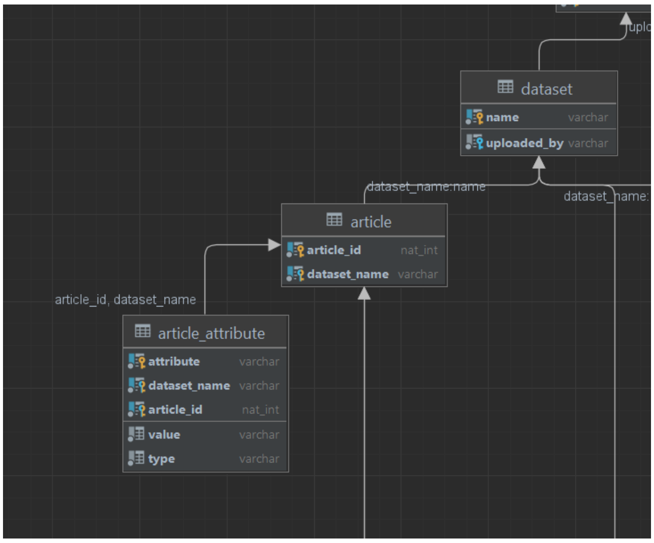
```
create table customer_attribute
(
    type            varchar not null,
    attribute_name  varchar not null,
    attribute_value varchar not null,
    customer_id     nat_int not null,
    dataset_name    varchar not null,
    primary key (attribute_name, customer_id, dataset_name),
    foreign key (customer_id, dataset_name) references customer (customer_id, dataset_name)
        on update cascade on delete cascade
);

alter table customer_attribute
    owner to app;

create index idx_customer_attribute_on_unq_customer_id
    on customer_attribute (customer_id, dataset_name);
```

3.2 Articles

An Article has a unique article id throughout the dataset as primary key. It also has a secondary key namely a composed of a dataset-name (foreign key to dataset) and an article-id to differentiate between articles in a given dataset = PK[dataset-name, article-id].

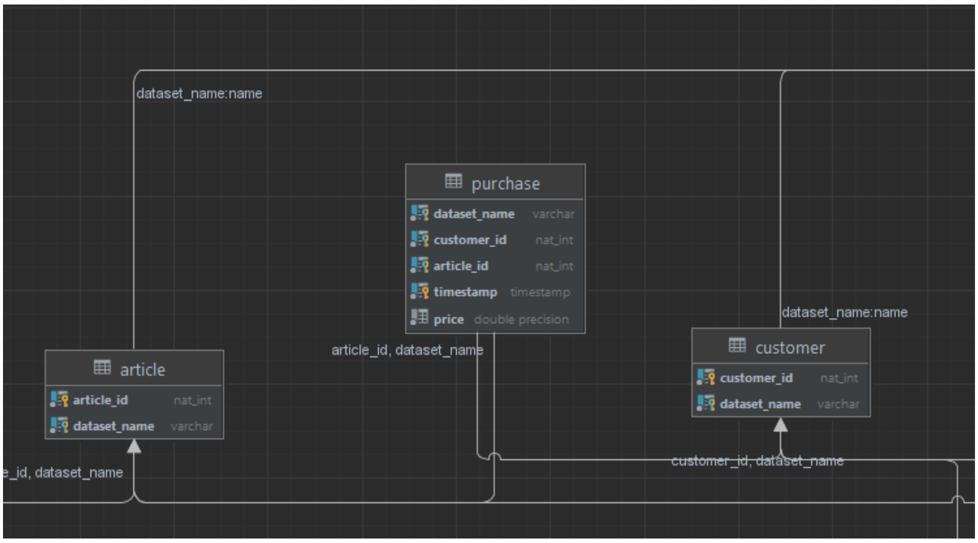
3.2.1 Article-Attribute

Article Attribute is where we store the actual (dynamic) data of an article. It is a weak entity of article and therefore contains a foreign key to an article (article-id, dataset-name). On top of this foreign key, the "attribute" is added which is the name of the field/attribute. So the final primary/composite key is (article-id, dataset-name, attribute). It has a type and a value so it can be used as intended.



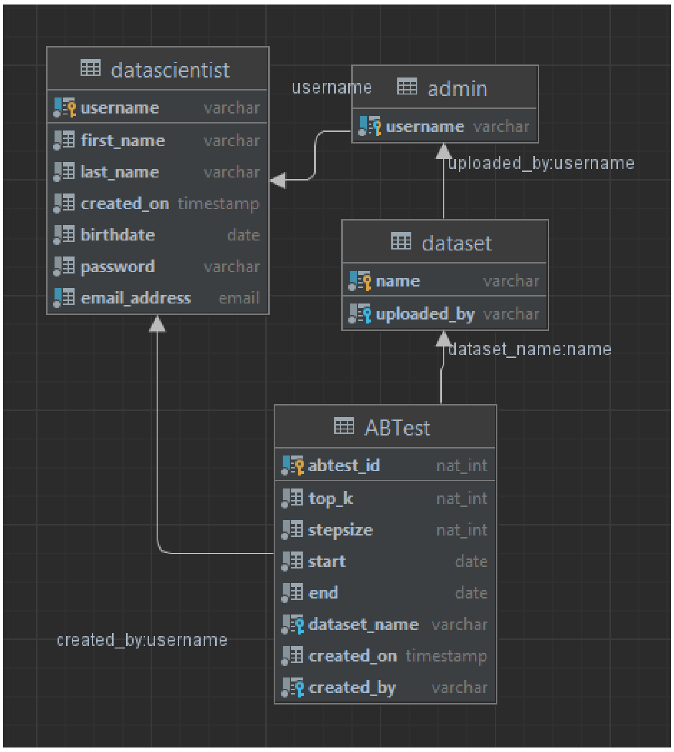
3.3 Purchases

A purchase is defined by a person with "customer-id" buying an article with "article-id" belonging to the same dataset on "timestamp" for the price "price". Therefore the primary key of a purchase is (article-id,customer-id, dataset-name, timestamp). The price at the time of purchase is kept in an extra field. It references to customer(customer-id, dataset-name) and article(article-id, dataset-name), with the same dataset-name in both references.



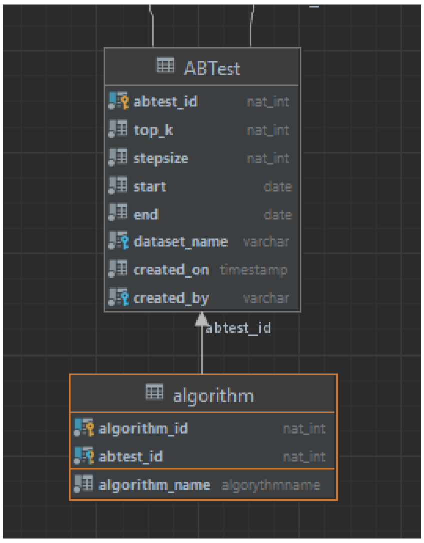
4 ABTest

An abtest has a unique id [PK], It contains the test-wide parameters namely top-k, stepsize,start,end and dataset-name which references dataset(dataset-name). Its creator is kept in "uploaded-by" which references a datascientist. And its creation time is also kept which defaults to now().



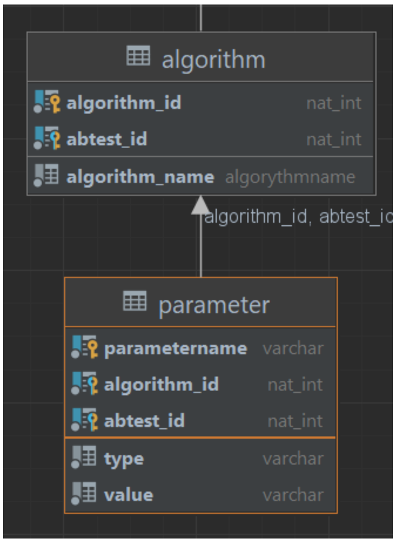
4.1 Algorithm

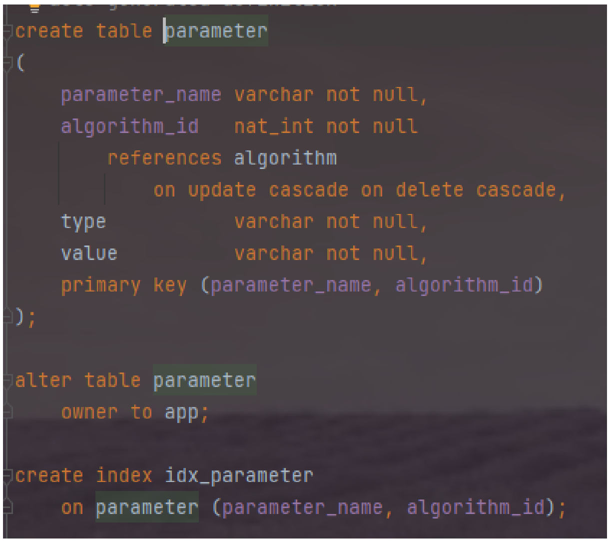
An algorithm itself has an id as primary key and belongs to an abtest(abtest-id). It also contains the type of the algorithm (this is so that we can identify which exact algorithm we have to execute). We can make multiple algorithms and link them to the same ABTest.



4.1.1 Parameter

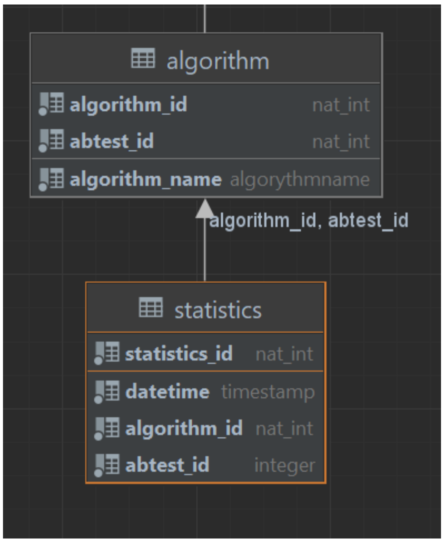
Parameter contains the name and value of a parameter that belongs to the algorithm. A parameter is a weak entity of an algorithm. The name of the parameter is unique in a given algorithm and thus the PK(parametername,algorithm-id,ABTest-id) where (algorithm-id,ABTest-id) references an algorithm.





4.2 Statistics

Statistics has a primary key in the form of "statistics-id" but a secondary key (abtest-id, algorithm-id, datetime) is also present. Statistics holds the data that is fetched on a given date. One entity of statistics is kept per stepsize.




```
create table statistics
(
    statistics_id serial
        primary key,
    date_of       date    not null,
    algorithm_id  nat_int not null
        references algorithm
            on update cascade on delete cascade,
    unique (algorithm_id, date_of)
);

alter table statistics
    owner to app;

create index idx_statistics_abtest_algorithm_date
    on statistics (algorithm_id, date_of);

create index idx_statistics_on_date_of
    on statistics (date_of);

create index idx_statistics_algorithm_id
    on statistics (algorithm_id);
```

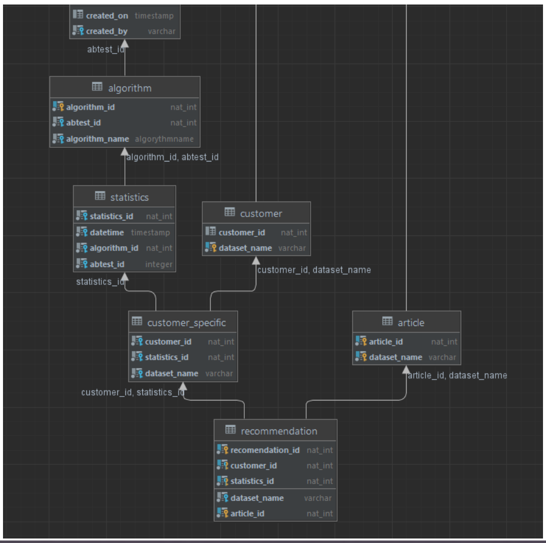
4.2.1 Dynamic-Stepsize-Variable

Dynamic-Stepsize-Variable allows for storing variables on every stepsize, therefore it has a foreign key to statistics. The PK = [statistics,parameter-name]. Aside of that it has the value that we want to store in parameter-value.

```
create table dynamic_stepsize_var
(
    statistics_id   integer not null
        references statistics
            on update cascade on delete cascade,
    parameter_name  varchar not null,
    parameter_value varchar,
    primary key (statistics_id, parameter_name)
);

alter table dynamic_stepsize_var
    owner to app;

create index idx_dynamic_stepsie_var
    on dynamic_stepsize_var (statistics_id, parameter_name);
```

4.2.2 Customer-Specific-Statistics

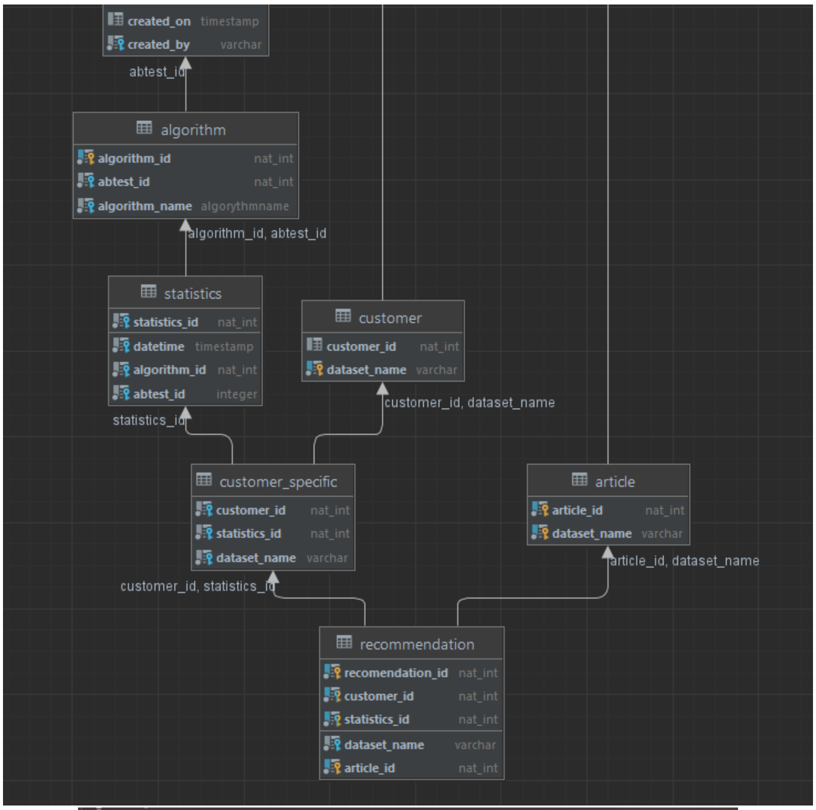
Here we keep an entry with customer specific data for every customer that has been active (bought something) on the date present in statistics. Customer-specific is a weak entity with of statistics with PK(statistics-id,unique=customer-id). It basically links statistics with customer without copying the fields of statistics mindlessly.




```sql
create table customer_attribute
(
    type            varchar not null,
    attribute_name  varchar not null,
    attribute_value varchar not null,
    customer_id     nat_int not null,
    dataset_name    varchar not null,
    primary key (attribute_name, customer_id, dataset_name),
    foreign key (customer_id, dataset_name) references customer (customer_id, dataset_name)
        on update cascade on delete cascade
);

alter table customer_attribute
    owner to app;

create index idx_customer_attribute_on_unq_customer_id
    on customer_attribute (customer_id, dataset_name);
```

4.2.3 recommendation

A recommendation entity is kept for every recommendation, this table links the customer-specific statistics with the recommendation-id Which goes up to k-1 in a top-k scenario. It is a weak entity of customer-specific. Its PK = [recommendation-id, unique-customer-id, statistics-id]. It has a foreign key to its stronger entity customer-specific and unique-article-id. It can be traced back to the "recommendation-id"th recommendation of customer "unique-customer-id" at (next are linked via statistics-id) "datetime" from algorithm "algorithm-id" in abtest "abtest-id". And even further to dataset and parameters if needed.




```sql
create table recommendation
(
    recommendation_id  nat_int not null,
    unique_customer_id nat_int not null,
    statistics_id      nat_int not null,
    unique_article_id  nat_int not null
        references article
            on update cascade on delete cascade,
    primary key (recommendation_id, unique_customer_id, statistics_id),
    foreign key (unique_customer_id, statistics_id) references customer_specific_statistics
        on update cascade on delete cascade
);

alter table recommendation
    owner to app;

create index idx_recommendation_on_unq_article_id
    on recommendation (unique_article_id);

create index idx_recommendation
    on recommendation (unique_customer_id, recommendation_id);

create index idx_recommendation_on_unq_customer_id
    on recommendation (unique_customer_id);

create index idx_recommendation_on_statistics_id
    on recommendation ((statistics_id::integer));
```

5 Metrics

Datetime is available in the table of statistics and all dependencies so it is accessible and can be used to chose a window size.

5.1 Purchases over time

Can be done using a simple query where we count the purchases made within a single day for every day in the ABTest, we query for dataset_name, start_start and end_date.
SELECT bought_on,COUNT(unique_article_id)
FROM purchase NATURAL JOIN article
WHERE bought_on between 'start' and 'end' and dataset_name = 'dataset_name'
group by bought_on;

5.2 Unique Active Users Over Time

Can be done using a simple query very similar to Purchases over time:

```
SELECT bought_on,COUNT(DISTINCT(unique_customer_id))
FROM purchase NATURAL JOIN customer
WHERE bought_on between '{start}' and '{end}' and
dataset_name = '{dataset_name}'
group by bought_on;
```

5.3 Click Through Rate (CTR)

We know all the purchases that have been made and all the top-k list so we can figure out the click through rate. We calculate the ctr per day beforehand and store it into dynamic_stepsize_variable. Now we can fetch it over time with the following query:

```
SELECT date_of, algorithm_id,parameter_value, algorithm_name
FROM statistics
NATURAL JOIN dynamic_stepsize_var
NATURAL JOIN named_algorithm
NATURAL JOIN ab_test
WHERE abtest_id = {abtest_id}
AND parameter_name = 'CTR'
ORDER BY date_of;
```

5.4 Attributions

We can take an intersection of purchases and recommendations, we can see divide the count of purchases with this intersection over a period of 7 or 30 days.

Firstly we find the intersection, and count the times it was recommended. We filter on distinct articles as two recommendations on the same article in the past days only results attribution point to the algorithm.algorithm_id on bought_on for customer recommendation.unique_customer_id, the interval can be entered dynamically in this query (7 days in this case). We would only have to run this once per ABTest.

```sql
create materialized view attr as
select algorithm.algorithm_id,
       bought_on,
       recommendation.unique_customer_id,
       count(distinct (article.unique_article_id))              as attributions,
       sum(price) / count(distinct (article.unique_article_id)) as revenue_per_attr
from (select abtest_id, start_date, end_date from ab_test where abtest_id = 18) ab_test
         join algorithm on algorithm.abtest_id = ab_test.abtest_id
         join statistics
              on algorithm.algorithm_id = statistics.algorithm_id
         join recommendation on statistics.statistics_id = recommendation.statistics_id
         join customer on recommendation.unique_customer_id = customer.unique_customer_id
         join article on article.unique_article_id = recommendation.unique_article_id
         join purchase on
            customer.dataset_name = purchase.dataset_name and
            customer.customer_id = purchase.customer_id and
            article.article_id = purchase.article_id and
            article.dataset_name = purchase.dataset_name and date_of between bought_on - interval '7 days' and bought_on
where bought_on between start_date and end_date
group by algorithm.algorithm_id, bought_on, recommendation.unique_customer_id;

create index attr_by_date on attr (bought_on);
```

query

```sql
select *
from attr
order by attributions desc;
```

Output | Result 82 | ppdb_database.public.attr

1-500 of 501+ | Tx: Auto | DDL

	algorithm_id	bought_on	unique_customer_id	attributions	revenue_per_attr
1	23	2020-01-02	550311	9	0.025971751412429345
2	22	2020-01-09	67979	5	0.027976271186440617
3	21	2020-01-09	67979	5	0.02797627118644062
4	22	2020-01-03	423474	4	0.015889830508474527
5	23	2020-01-10	313812	4	0.008033898305084676
6	23	2020-01-08	371527	4	0.0317542372881355
7	22	2020-01-03	820494	4	0.011423728813559277
8	22	2020-01-08	300524	4	0.018627118644067776
9	23	2020-01-02	297789	4	0.015237288135593174
10	21	2020-01-03	350161	3	0.031056497175141168

Example of the query-result

The query is encapsulated in a materialized view for two reasons. Firstly we can index on it and use it to calculate our average attributions rate for one user over a period or for all users over the time (thus per day). Secondly we can also speed up all of the following queries by pre calculating the most intensive part.

5.5 Attribution Rate (AR@D)

Next the sum of the attribution count for a customer over a period can be taken and divided by the number of purchases the customer has made in the same period, this is the attribution rate.

```
select algorithm_id,
       unique_customer_id,
       sum(attributions) / (select count(*)
                            from purchase b
                                     natural join customer c
                            where b.bought_on between '2020-01-01' and '2020-01-09'
                              and c.unique_customer_id = attr.unique_customer_id) ATR
from attr
where bought_on between '2020-01-01' and '2020-01-9'
group by algorithm_id, unique_customer_id
order by atr desc;
```

query

5.6 Average Revenue Per User (ARPU@D)

At last, total revenue attributed to the algorithm for the user over an interval can be easily calculated.

```sql
select algorithm_id,
       unique_customer_id,
       sum(attributions * revenue_per_attr) ARPU
from attr
where bought_on between '2020-01-01' and '2020-01-9'
group by algorithm_id, unique_customer_id
;
```

query