Online aggregate tables

A method for implementing big data analysis in PostgreSQL using real time pre-calculations

FABIAN BERGMARK
Online aggregate tables: A method for implementing big data analysis in PostgreSQL using real time pre-calculations

Fabian Bergmark
fbergmar@kth.se

Master’s Thesis in Computer Science
Written at CSC, KTH
Supervisor: Johan Hoffman
Examiner: Johan Hästad
Abstract

In modern user-centric applications, data gathering and analysis is often of vital importance. Current trends in data management software show that traditional relational databases fail to keep up with the growing data sets. Outsourcing data analysis often means data is locked in with a particular service, making transitions between analysis systems nearly impossible. This thesis implements and evaluates a data analysis framework implemented completely within a relational database. The framework provides a structure for implementations of online algorithms of analytical methods to store precomputed results. The result is an even resource utilization with predictable performance that does not decrease over time. The system keeps all raw data gathered to allow for future exportation. A full implementation of the framework is tested based on the current analysis requirements of the company Shortcut Labs, and performance measurements show no problem with managing data sets of over a billion data points.

Sammanfattning

Realtidsaggregerade tabeller

En metod för analys av stora datamängder i PostgreSQL med hjälp av realtidsuppdaterade förberäkningar

# Contents

1 Introduction ............................ 1
   1.1 This thesis ........................... 1

2 Theory ................................ 2
   2.1 Managing large data sets in a RDBMS ..................... 2
      2.1.1 Indexes .......................... 2
      2.1.2 Materialized views .................. 2
      2.1.3 Data mapping ....................... 3
      2.1.4 Archiving .......................... 3
   2.2 Big data analysis and visualization .................... 3
      2.2.1 Single dimensional analysis ............... 4
      2.2.2 Two dimensional analysis ................. 4
      2.2.3 Cohort analysis ...................... 4

3 Study ................................ 4
   3.1 Data .................................. 5
      3.1.1 Resource utilization ................... 5
   3.2 Test cases ................................ 7
      3.2.1 Test case 1 ........................... 7
      3.2.2 Test case 2 ........................... 12
      3.2.3 Analysis ............................. 14

4 Idea ................................ 14
   4.1 Query analysis ........................... 15
      4.1.1 Single dimensional analysis ............... 16
4.1.2 Two-dimensional analysis ........................................... 16
4.1.3 Cohort analysis ......................................................... 16
4.2 Query implementation ..................................................... 17
  4.2.1 Top lists .............................................................. 17
  4.2.2 Frequency graphs ..................................................... 17
  4.2.3 Generalization ......................................................... 18

5 Prototype ................................................................. 18
  5.1 Building blocks ......................................................... 18
    5.1.1 Fact table .......................................................... 18
    5.1.2 Query set ........................................................... 19
    5.1.3 Aggregate set ....................................................... 19
    5.1.4 Online aggregate tables .......................................... 19
    5.1.5 Presentation views ............................................... 19
    5.1.6 Implementation process .......................................... 19
  5.2 Implementation ........................................................ 20
    5.2.1 Fact table .......................................................... 20
    5.2.2 Query set ........................................................... 20
    5.2.3 Aggregate set ....................................................... 20
    5.2.4 Online aggregate table .......................................... 21
    5.2.5 Presentation view ............................................... 21
    5.2.6 Results .............................................................. 21
  5.3 Considerations ........................................................ 22
    5.3.1 Performance ........................................................ 22
    5.3.2 Out of order data ................................................ 22
    5.3.3 New aggregates .................................................... 22
5.3.4 Duplicate events .................................................. 23

6 Multidimensional analysis ........................................... 23

6.1 Single-dimensional analysis ....................................... 24
  6.1.1 Implementation ................................................. 24

6.2 Three-dimensional analysis ....................................... 24
  6.2.1 User cohort ..................................................... 24
  6.2.2 Implementation ................................................. 25
  6.2.3 Button cohort .................................................... 25
  6.2.4 Implementation ................................................. 25

6.3 Data format generation ........................................... 26
  6.3.1 CSV exportation ............................................... 26
  6.3.2 Line graph exportation ........................................ 26

7 Evaluation ................................................................. 27

7.1 Features ............................................................ 27
  7.1.1 Access to raw data to enable migration to another frame- 27
      work .................................................................
  7.1.2 Support of analysis methods common in big data framework 27
  7.1.3 Support of generation of graphs and cohort charts to present 27
      analytical results ................................................
  7.1.4 Support of new analytical queries to be added to the sys- 28
      tem, and applying them retroactively to old data .......... 28

7.2 Performance .......................................................... 28

7.3 Deliverable ............................................................ 30
  7.3.1 Queries .......................................................... 30
  7.3.2 Events ........................................................... 31
**Keywords**

**RDBMS** (Relational Database Management System) is a database management system that models data in a relational manner.

**Transactional Query** In a relational database system, a transactional query modifies the dataset by adding or removing rows.

**OLTP** (Online Transaction Processing) here referring to the properties of a transactional database system, meaning high throughput, availability and concurrency.

**OLAP** (Online Analytical Processing) here refers to the properties of a transactional database, meaning the ability to swiftly answer analytical queries.

**ACID** (Atomicity, Consistency, Integrity and Durability) refers to four common properties of transactional databases that ensure high reliability of the data managed.

**NoSQL** (Not only SQL) is an emerging set of databases that offers a different storage model, functionality and/or query language than traditional RDBMSs. They often offer higher scalability at the expense of lacking true ACID properties.

**NewSQL** is a set of databases that tries to provide the same scalability, availability and performance of NoSQL databases while still granting ACID properties.

**Aggregation** is a database operation where multiple data points are combined under a certain aggregation function to produce a single value.

**Cohort Analysis** is a behavioral analysis that divides the analyzed group into subgroups, or cohorts. The behavior of these groups are then compared to find patterns. A common use case is to compare behaviour patterns between different age groups.

**Cohort Charts** are a way to present a cohort analysis data set.
1 Introduction

Data analysis is the process of inspecting and transforming data to provide useful information. It is today used for both business and scientific applications using different techniques and for different purposes. The art of maximizing the knowledge gained from gathered data has become an industry, and during the last few years, this field has received increased attention in research. Systems with huge user bases such as social media networks gather so much data that traditional analysis tools have become inadequate[1]. These huge data sets, called big data, require new tools and new ways of thinking about data analysis[4].

1.1 This thesis

Research shows that implementing analytical methods directly in a relational database have several benefits compared to exporting data for external analysis[3]. The purpose of this thesis is to investigate whether it is possible to implement a subset of features offered by modern big data analysis frameworks in a relational database. The beneficiaries would be companies that currently are using a relational database for data analysis and are transitioning into the big data realm. This thesis presents a general framework for time series data analysis within a relational database. The framework should have support for the following type of queries and features:

- Access to raw data to enable migration to another framework
- Support analysis methods common in big data frameworks
- Support generation of graphs and cohort charts to present analytical results
- Support new analytical queries to be added to the system, and applying them retroactively to old data
2 Theory

This section explains methodologies offered by RDBMSs to cope with large data sets, as well as common practices deployed by data warehouse implementations to provide insight into which difficulties that need to be addressed.

2.1 Managing large data sets in a RDBMS

Relational databases have been a popular storage system in data warehouses, and was up until recently considered the leading technology [2]. This has influenced the development of these systems to better manage the increasing data sets. This chapter will go through the features offered by modern RDBMSs that enable management of large data sets.

2.1.1 Indexes

Relational databases generally store data in a block-oriented layout. A block is a segment of secondary memory whose size usually corresponds to the block size of the secondary storage unit. The rows stored in the data blocks form an unsorted data set. Allowing the set to be unsorted allows for quick removals and additions without the need of reordering. However, queries against an unsorted data set often require all rows to be read from disk and inspected. To increase performance, databases have a feature called indexes. An index is usually implemented similarly to a binary tree where each node corresponds to an index key value and holds an array of row pointers to rows with that index key value. When the database query engine needs to look up data based on the index key, it can utilize the index to do a binary search instead of reading all rows from disk.

2.1.2 Materialized views

Real time OLAP databases are very resource intensive. As new data is continuously coming in, analytical queries need to be re-run against the whole data set to calculate updated results. Sometimes it is reasonable to drop the real time property to gain performance. Materialized views are a way to save the result of a query, and are also known as snapshots. A view is a feature of RDMSs that allows the result of a query to be treated as a normal table. Each time the view table is queried, the database transparently executes the view query and uses the result instead of reading data from disk as with a normal table. Some RDBMs allows the view to be cached to avoid re-running the view query to often, and such views are called materialized views. The snapshot is cached as a normal
table and can be used in queries instead of querying the underlying base tables to achieve speedup[6]. Some databases even support indexes on materialized views, allowing for even better performance. In data warehouses where complex analytical queries are common, materialized views are today commonplace[7]. The materialized view can be re-cached on request to reflect new data. The task of selecting which views that should be materialized is called view selection, and is a difficult problem[8]. Changes in the database layout or new queries might require a new view selection evaluation.

2.1.3 Data mapping

Another important factor in query performance is the database schema, or the database structure[2]. When storing data in a relational database, a mapping is created between the data and the table structure. Many approaches of data mapping strive to achieve a normalized database, meaning low redundancy and few update anomalies. While normalized databases ensure high data consistency and usability, they usually require a high number of tables and relations, which in turn leads to queries having a large number of joins, which impacts query performance negatively. To address this, data warehouses usually implement a partially normalized schema called star schema. The star schema consists of one fact table and several dimension tables. The fact table has relations to the dimension tables, while the dimension tables has no relations.

2.1.4 Archiving

Even indexes and materialized views have their limits and as the database grows in size, the performance may decrease. Indexes may become too large and rebuilding the materialized views may take several hours. Data warehouses often solve this by archiving data. The database may thus only contain data for the last 30 days. Archived data can be analyzed on command in a secondary database.

2.2 Big data analysis and visualization

Big data frameworks differs from general databases in that they provide tools for data analysis and visualization. As big data frameworks are a specialized solutions, each provide a fixed set of visualization formats combined with an interface to apply them to an analysis of the data set. While the set of visualization formats differs between big data frameworks, the analysis methods are often limited to three categories described below.
2.2.1 Single dimensional analysis

Single dimensional analysis aims to provide overview by representing some key metric as a single value result. An example is the number of page visits during the last hour for a website or the total number of users of an application. While a single value provides limited insight, they are often good indicators of when more in-depth analysis is needed. Single dimensional values are visualized by simply stating the value.

2.2.2 Two dimensional analysis

It is often interesting to visualize how a value behaves with respect to some factor, such as time. The factor can either be time or some category related to the data set. Examples are how sales number differs between different product categories, or how the number of page visits to a website changes during the hours of the day. Different visualization methods can be used depending on the factor the value is analyzed in relation to. If the factor is time, a line graph might be used. Other visualization methods include bar charts circle diagrams.

2.2.3 Cohort analysis

A cohort is defined as a group of subjects where all members share a defining property such as gender, place of birth or age. A cohort analysis aims to analyze differences between cohorts within a data set. Each cohort is analyzed independently and all results are then combined to form the cohort analysis result. An examples is how user retention for an application differs between different age groups. The analysis applied to each cohort can either be a single dimensional analysis in which case the result can be visualized with a bar chart, or a two dimensional analysis in which case the result can be visualized with a graph with multiple lines.

3 Study

This section contains an example of storing time series data in a relational database to show the difficulties in managing large data sets using only standard constructs.
3.1 Data

The data points in the time series data follow the format:

data_column_1 | data_column_2 ...

where the number of columns and their type depend on which event they represent. This has a very natural data mapping where each event type is stored in a separate table with a column for each event column.

3.1.1 Resource utilization

Database performance is heavily influenced by resource utilization. Understanding how different database techniques influence resource utilization is paramount for this thesis. When analyzing resource utilization in a data warehouse, it is important to differentiate between the transactional and analytical queries. The transactional queries insert new data into the database at regular intervals and mostly perform disk writes, but also utilize the CPU as any index present needs to be updated. The analytical queries are performed upon requests by the users, and the resource utilization depends heavily on the types of queries. Besides from improving performance, it is also important to analyze resource utilization to avoid resource starvation. If the transactional queries experience resource starvation it might not be possible to insert data, eventually causing a system failure and lost data. To get insights in the resource utilization, a two week measurement was performed during which only transactional queries was executed. These measurements can later be used with measurements of analytical queries to evaluate the likelyhood of resource starvation.

![CPU utilization](image1.png)

Figure 1: CPU utilization
During the entire two weeks, an average of 2-5 million events were gathered each day. Figure 1, 2 and 3 displays measurements of the three metrics that was most influenced by an increased transactional workload. These metrics should therefore be carefully measured as resource exhaustion could lead to lost data. Two insights can be gathered from these metrics. Firstly, the transactional workload is very even, which is expected as events are gathered by users in many different time zones. Secondly, the required resources to manage the transactional queries are by no means negligible. However, if it was not for the analytical requirements, this database would be able to cope with much larger data flows.
3.2 Test cases

The data gathered during the first two weeks of measurement of transactional queries was considered a good candidate for analyzing analytical queries. A separate database with the same specifications was created and the entire data set was transferred. To analyze resource utilization during execution of analytical queries, three analytical queries were selected. These queries represent requests from both the management team and the development team. The fact table used contains events representing connection events for a Bluetooth device. The data contained includes which user the event occurred for, a device identifier and a timestamp. The test data set contained 77 million rows totalling 24 GB of data.

The first query determines the set of unique device identifiers which has at least one connection event. The second query counts the number of connection events per device. The third query finds all the devices a specific user has connected to.

The three queries were analyzed in two test cases. The first test case uses no indexes to increase query performance and can be seen as a reference measurement. Every other technique should improve on this result.

Query 1

Query:

\[
\text{SELECT DISTINCT data_device_id FROM button_connected_events}
\]

Query planner:

HashAggregate (cost=4153928.85..4154022.43 rows=9358 width=4) (actual time=408148.667..408159.337 rows=36269 loops=1)
Group Key: data_device_id
-> Seq Scan on button_connected_events (cost=0.00..3960671.28 rows=77455105 width=4)

3.2.1 Test case 1

This test case uses no indexes to increase query performance and can be seen as a reference measurement. Every other technique should improve on this result.

Query 1

Query:

\[
\text{SELECT DISTINCT data_device_id FROM button_connected_events}
\]

Query planner:

HashAggregate (cost=4153928.85..4154022.43 rows=9358 width=4) (actual time=408148.667..408159.337 rows=36269 loops=1)
Group Key: data_device_id
-> Seq Scan on button_connected_events (cost=0.00..3960671.28 rows=77455105 width=4)
Measurements:

From the figures we see that the query maximizes the read throughput at 60 MB/s while CPU usage remains relatively low. This is expected as the database without any index needs to read the entire data set from disk to find all device identifiers.
Query 2

Query:

```sql
SELECT data_device_id, COUNT(*) FROM button_connected_events group by data_device_id
```

Query planner:

```sql
HashAggregate (cost=4347946.04..4348039.62 rows=9358 width=4) (actual time=408166.920..408179.983 rows=36279 loops=1)
Group Key: data_device_id
-> Seq Scan on button_connected_events (cost=0.00..3961363.36 rows=77455105 width=4)
   (actual time=1.018..378867.272 rows=77455105 loops=1)
Planning time: 0.085 ms
Execution time: 408186.044 ms
```

Measurements:

![Figure 6: Read throughput (MB/Second)](image)

Figure 6: Read throughput (MB/Second)
The measurements are very similar to the first query, and again read throughput can be seen as the limiting resource.

**Query 3**

Query:

```sql
SELECT DISTINCT data_device_id FROM button_connected_events WHERE user_id=2848
```

Query planner:

```
HashAggregate (cost=4155236.35..4155236.37 rows=2 width=4) (actual time=408175.095..408175.096 rows=1 loops=1)
  Group Key: data_device_id
  ->  Seq Scan on button_connected_events (cost=0.00..4155212.44 rows=9563 width=4) (actual time=95348.011..408175.077 rows=18 loops=1)
    Filter: (user_id = 2848)
      Rows Removed by Filter: 77493078
Planning time: 0.079 ms
Execution time: 408175.123 ms
```
The measurements are very similar to the first and third query, and again read throughput can be seen as the limiting resource. The similarities between these queries come from the fact that they all require the entire dataset to be examined.
3.2.2 Test case 2

This test case adds several indexes to the table to increase query performance if possible.

**Query 1** To fully utilize an index when finding all distinct values for a column, a special form of index scan is used instead of the normal SELECT DISTINCT. This is often orders of magnitude faster for columns with a small number of distinct values compared to the number of rows. An index was created on the device identifier column.

Query:

```sql
WITH RECURSIVE t AS (
    (SELECT data_device_id FROM button_connected_events ORDER BY data_device_id LIMIT 1) UNION ALL
    SELECT (SELECT data_device_id FROM button_connected_events WHERE data_device_id > t.data_device_id ORDER BY data_device_id LIMIT 1)
    FROM t
    WHERE t.data_device_id IS NOT NULL
) SELECT data_device_id FROM t WHERE data_device_id IS NOT NULL;
```

Query planner:

```
| CTE Scan on t (cost=133.85..135.87 rows=100 width=4) (actual time=0.021..402.953 rows=36390 loops=1) |
| Filter: (data_device_id IS NOT NULL) |
| Rows Removed by Filter: 1 |
| CTE t |
| -> Recursive Union (cost=0.57..133.85 rows=101 width=4) (actual time=0.019..383.373 rows=36391 loops=1) |
| -> Limit (cost=0.57..1.28 rows=1 width=4) (actual time=0.018..0.018 rows=1 loops=1) |
| -> Index Only Scan using connected_device on button_connected_events button_connected_events_1 (cost=0.57..55481821.43 rows=77558086 width=4) (actual time=0.017..0.017 rows=1 loops=1) |
| Heap Fetches: 0 |
| -> WorkTable Scan on t t_1 (cost=0.00..13.06 rows=10 width=4) (actual time=0.010..0.010 rows=1 loops=36391) |
| Filter: (data_device_id IS NOT NULL) |
| Rows Removed by Filter: 0 |
| SubPlan 1 |
| -> Limit (cost=0.57..1.29 rows=1 width=4) (actual time=0.009..0.009 rows=1 loops=36390) |
| -> Index Only Scan using connected_device on button_connected_events |
```
The execution time was too short to perform measurements of resource utilization. The EXPLAIN ANALYZE output reveals that the query only uses the index without reading from disk, resulting in a 1000 times faster execution time.

Query 2

While it's possible to write a query that only uses an index, I have been unable to find a combination of index and query that outperforms using no indexes.

Query 3

Query:

```
SELECT DISTINCT data_device_id FROM button_connected_events WHERE user_id=2848
```

Query planner:

```
HashAggregate (cost=37494.30..37494.32 rows=2 width=4) (actual time=0.037..0.038 rows=1 loops=1)
  Group Key: data_device_id
  ->  Index Scan using idx_button_connected_events_user_id on button_connected_events (cost=0.57..37468.17 rows=10453 width=4) (actual time=0.018..0.026 rows=18 loops=1)
      Index Cond: (user_id = 2848)
Planning time: 0.094 ms
Execution time: 0.073 ms
```

The execution time was too short to perform measurements of resource utilization. The EXPLAIN ANALYZE output reveals that the query only uses the index without reading from disk, resulting in a 5000000 times faster execution time.
3.2.3 Analysis

Two important observations can be made by analyzing the performance results of the two test cases, shown in the graphs in figures 4-9. It’s clear from the first test case that without any indexes, disk throughput becomes the bottleneck, which is expected as 24 GB of data needs to be read from disk. The other observation is the all-or-nothing property of indexes. Both the first and the third query were able to avoid disk reads altogether, using only the in-memory index. In these cases, the performance difference is substantial with a speedup of magnitude 3 and 6 respectively. However, the first query had to be written using a special index scan. This is not ideal as it might not integrate well with database libraries and object modeling frameworks. The second query couldn’t benefit from an index at all.

4 Idea

Implementing analytical features like those commonly offered by big data analysis frameworks in a relational database requires those features to be implemented as relational database queries. In data warehousing, queries are often divided into two categories. Transactional queries modifies the data set itself, for example by adding or removing rows. Analytical queries computes an analytical result by using the data in the data set. Analytical queries differ from transactional queries in that they almost always include an aggregate function[9]. In databases, an aggregate function is a function that combines several rows to produce a result. An example of an aggregate function is $COUNT(*)$, which simply counts the number of rows and produces that as an integer, or $MAX(expression)$ which evaluates the expression for each row and produces the maximum value. Simply put, the query engine computes the result for an aggregate function by first computing the rows the query requests, and then applies any aggregate functions to that set. Analytical query performance is thus affected by both the amount of time required to read the involved rows, as well as the computational time required to compute the aggregate functions to produce the result. The former factor can be improved by using the techniques presented in the theory section. Most commonly, indexes can be used to reduce the amount of data read from disk. As shown in the previous chapter, indexes has the potential to increase the performance of some queries substantially, potentially making even very large data sets tractable. A naive implementation could be to translate all high-level analytical questions to queries, and then try to add indexes until those queries becomes fast. However, as stated in the previous chapter, there exists queries for which relevant indexes are hard to find. For example, it is hard to create indexes to speed up queries involving the $COUNT(*)$ aggregate function. Such queries would either have to be rewritten, or would require some changes to be made to the data set to better facilitate
that specific query. One underlying problem with the naive solution is that
queries are recomputed on every execution against a potentially huge data set.
If there exists an online algorithm for a query, this re-computation is unneces-
sary. Generally, an online algorithm can compute its result by consuming one
input at a time in a serial fashion. More exact, an online algorithm for a function
$f$ can, given the result $r$ of the function applied to the set $S$ and another value
$x$, compute the result of $f(S \cup \{x\})$ while only knowing $r$ and $x$, i.e. without
knowing the already consumed elements in $S$. If there exists an online algorithm
for a function $f$, that function is said to be online computable. To give an ex-
ample, an online algorithm for the $MAX(*)$ aggregate function compares each
new input to the previously stored maximum $r$, and updates $r$ if the new input
is strictly larger. If there exists online algorithms for for the common analysis
methods of big data, and if these algorithms can be implemented in a relational
database, this would alleviate the need to compute queries against a large data
set. Instead, query results are stored and kept updated by an online algorithm,
ready to be presented.

4.1 Query analysis

As stated, analytical queries combine multiple rows into a result. Relational
databases offer a set of aggregation functions and a grouping operator to con-
trol how the rows are combined. While all of the standard aggregation functions
offered by postgreSQL are online computable, some queries require functionality
outside of what the built-in aggregation functions offer. Even if standard
aggregate functions are used, they might be used in intricate ways. An impor-
tant observation to make is that a query that uses an aggregate function that
is online computable is not necessarily itself online computable. To illustrate
this, imagine a set $U$ of user interactions, and a query that calculates the most
active nationality. This query most likely uses the $COUNT(*)$ aggregate func-
tion to count the number of interactions for each nationality and selects the
name of the nationality with the highest interaction count. Given the result
for $U$ and a new element $n$, there is not enough information to compute the
result for $U \cup \{n\}$ as the query results only contains the name of the nationality.
To make an query online computable it is often necessary to include additional
information in the stored result. The additional information is only used by the
online algorithm and not included in the query result. This analytical process
of creating online algorithms for queries requires understanding of which data
the query really uses, and how this data relates to the entries in the data set.
As analytical queries might be arbitrarily complex, no general method exists to
ensure that all queries can be successfully analyzed and implemented. However,
many of the big data analytical methods have query analogies for which such
methods exist.
4.1.1 Single dimensional analysis

A query that computes a single value metric is a function that projects a data set onto a single value in some domain. As stated, such functions usually can not be computed online as the single value contains too little information. However, such queries usually follow a common pattern that computes the result in two steps. The first step queries the data set to produce a set of result candidates combined with a value. For example, if the data is a set of erroneous events and the single value metric is the most frequent error code, the candidate set would be the the set of distinct error codes combined with how many times they each have occurred. The second step uses the candidate set together with an ordering function to produce the result. For the same example, the ordering function would select the error with the most occurrences. The idea is to ensure that the query that produces the candidate set is computable online by storing additional data in the candidate set.

4.1.2 Two-dimensional analysis

A query that computes a two dimensional analysis is a function that projects a data set onto a set of two dimensional points. Like single value metric queries, two-dimensional queries also usually follow a pattern. First the interval and discrete points of the domain are determined. Then the value in the co-domain is calculated for each selected point in the domain. The result of these two steps is a set of two dimensional values. The idea here is to introduce a second step in the computation to make the two dimensional computation online computable. The first step still generates the set of $X$ and $Y$ values, but with the added ability to store additional information in each value. This additional information must ensure that it is possible to calculate the $X$ and $Y$ values for new data points, possibly updating a previously calculated $Y$-value. The second step transforms the result from the first step to a two dimensional result by removing the additional information.

4.1.3 Cohort analysis

A cohort analysis can be either two or three dimensional depending on the dimensionality of the analysis applied to each cohort. A two dimensional cohort can be analyzed using two dimensional analysis as described above. Three dimensional cohort analysis however requires a separate analysis. A query that computes a three dimensional cohort analysis is a function that projects a data set onto a set of three dimensional points. A cohort analysis computation usually begins by dividing the data set into the different subsets representing each cohort. Each subset is then analyzed independently and the result from each subset is then combined to produce the final result. As the computation for
each cohort is a two dimensional analysis, the query analysis described above can be used. This can be used to simplify the query analysis. Given that the two dimensional analysis is online computable using the method described above, there is a trivial way to make the cohort analysis online computable by simply storing the result for the two dimensional analysis of each cohort. When a new data entry is inserted, it is categorized into its cohort and applied to the stored result for that cohort.

4.2 Query implementation

To better understand how the analytical process described above translates to practical problems, it is helpful to see the process applied to real analytical methods. Each of the following examples makes as few assumptions about the underlying data set as possible.

4.2.1 Top lists

Top lists fall under the category of single value metrics, where the single value is a list of fixed length. The elements of the top list are values from one of the columns in the data set such as user names or book titles, and the list is sorted using an ordering function applied to the whole or a subset of all columns. Following the query analysis applied to single value metrics as outlined above, a candidate set needs to be defined. For top lists, the candidate set is the set of unique values for the top list element column, each paired with a discriminatory value. The discriminatory value must fulfill two properties. First and foremost, it needs to contain enough information to allow the candidate set to be sorted to produce the top list. Secondly it must be computable online. For any top list that orders by frequency, the discriminatory value can be the number of occurrences which fulfills both requirements.

4.2.2 Frequency graphs

Frequency graphs are commonly used to visualize a metric over time by plotting the number of occurrences in each time interval, usually daily, weekly or monthly. The graph consists of two dimensional points where the X-value is from one of the column in the data set, and the Y-value is a calculated by a function applied to the whole or a subset of the columns. In some cases, the X-values are not values taken directly from a column the data set, but might be rounded values or derived values. If the X-axis represents time flow, the X-points might be rounded of to a weekly resolution. Following the query analysis applied to two dimensional graphs, the additional information needed to
make the graph-computation computable online needs to be defined. A graph that simply counts the number of occurrences during a time interval is already computable online and needs no additional information, as the information contained in the X and Y value suffices. A more complex task is to plot the number of unique occurrences during a time interval, for example the number of unique visitors to a page each day. This requires an additional column which contains information about which unique values that have already been counted, which for the given example might be a list of user identifiers.

4.2.3 Generalization

For some analytical queries it might be difficult to create an online computable result. This may either be due to complex mathematics or complexities that require a lot of additional information to be stored. For such queries there might be a more general result that is easier to compute online from which the sought result can be computed. This has the added benefit that multiple queries might use the same more general result. For the example given above with unique page visitors, a developer might realize that list operations have limited support in relational databases. Instead, the number of page visits each user made each day could be used. From that result, the final result is found by counting the number of users with a page visit count larger than zero each day. This could also be used to calculate the total number of page visits during a day, as well as to identify the most active users.

5 Prototype

This section outlines the steps and considerations that were taken during implementation of the system.

The idea presented above outlines a solution for computing queries using online algorithms. What’s left is to implement this in a relational database and analyze its performance.

5.1 Building blocks

5.1.1 Fact table

The fact tables are the tables storing the raw event data. Each event type is stored in a separate table. The fact tables have a very simple structure with one table column for every column in the event type. As data is stored unaltered in
the fact tables, it can later be exported to another system.

5.1.2 Query set

The query set is the set of analytical queries that the system currently implements. The set size is not static and can change when new queries need to be supported. These analytical queries implement the high-level analytical questions requested by the users of the system.

5.1.3 Aggregate set

The result of the query analysis applied to the query set is a set of online algorithms. Together, these online algorithms computes a set of results from which all of the queries in the query set can be computed. Each online algorithm is thus an aggregate that computes an important aspect of the data set. Together they form the aggregate set.

5.1.4 Online aggregate tables

Each online aggregate table holds online results for one aggregate. Each online aggregate table has an accompanying database function which implements the online algorithm for that aggregate.

5.1.5 Presentation views

The presentation views use the online aggregate tables to present the result to each query in the query set. Each presentation view implements a query that computes its result from the online aggregate tables and presents it as a table using a database view. These views thus hold online results for all queries in the query set.

5.1.6 Implementation process

The above building blocks aim to fulfill two of the three requirements, namely providing access to raw data and achieving good performance. The last requirement concerns extendability, meaning that new high-level analytical queries and event categories can be added to the system. Using the above building blocks, this is an easy process:
1. Identify which aggregates the query uses. When it’s possible, existing aggregates should be used.

2. Create online aggregate tables for any new aggregates with accompanying aggregate functions.

3. Create a presentation view that implements the initial query by using the online aggregate tables.

However, to apply an analysis retroactively to old data, some additional constraints needs to be fulfilled. This is explained in detail in section 5.3.

5.2 Implementation

The descriptions above do not include implementation details. However, implementing each of the building blocks is straightforward using standard PostgreSQL constructs. An implementation of an analytical query is presented below. The high-level goal of this query is to visualize product growth over time based on the number of purchased items of a small Bluetooth button. The analytical goal of the query is to plot the total number of Bluetooth devices that has been connected over time with a daily resolution.

5.2.1 Fact table

In this case, there is only one fact table containing the connection events for the Bluetooth device mentioned earlier.

5.2.2 Query set

The set contains the single query that computes the cumulative sum of the number of devices that has been connected for each day from the launch of the product until the current date.

5.2.3 Aggregate set

Each Bluetooth device is likely to have many connection events, but the query is only interested in the first event chronologically. A good aggregate is thus the time of the first connected event for each button.
5.2.4 Online aggregate table

The online aggregate table needs to hold the data that represents the aggregate. For the sole aggregate in this case, the online aggregate table has two data columns, the first containing the device identifier, and the second containing the time of the first connection event for that button.

The accompanying function needs to ensure that the time stored for each device is the smallest value of all stored connection events in the fact table. This is as straightforward as, for each inserted row check if the device has a previous entry in the aggregate table. If there is no entry, the time is inserted as a new row in the aggregate table. If there is an existing entry, its time column is compared to the time column of the new row and if the new value is smaller, the aggregate entry is updated. By adding an index on the device identifier column in the aggregate table, the check for an existing entry becomes very fast.

5.2.5 Presentation view

The above online aggregate table stores the time of the first connected event for each button. To plot the total number of connected buttons over time a query that, for each day counts the total number of buttons that was connected before and including that day. Such queries can be implemented very efficiently in PostgreSQL. The presentation view thus has two columns, the first containing a date, and the second containing the number of buttons that was connected before and including that date.

5.2.6 Results

With a fact table size of 155 million events, the presentation view query executes in less than 100 ms. To compare this timing, an optimized query against the fact table that utilized an index to compute the same result as the presentation view required two minutes to complete, about 1000 times slower. It is important to analyze how the performance will degrade over time. Both the aggregate function and the presentation view query only the online aggregation table. The online aggregate table has one row for each device that has been connected at least once, and is thus limited by the number of sold devices. This number is predictable and is unlikely to become large enough to cause performance problems.
5.3 Considerations

This section outlines the considerations that were made to ensure that the framework followed specification.

5.3.1 Performance

The potential performance improvement offered by the presented framework is the ability to use precomputed results to answer analytical queries. However, some limitations need to be imposed to ensure good and predictable performance. The performance impact of this solution depends on the performance of the online algorithms that updates the online aggregate tables, and the performance of presentation views that computes the final results using the online aggregate tables. The performance of both of these is determined by the queries’ complexity, and the size of the online aggregate tables. If the size of an online aggregate table is proportional to the size of its underlying fact table, its performance is likely to worsen over time and the performance improvement is lost. The size of an online aggregate table should therefore not depend on the size of the fact table. Moreover, its size should be predictable.

5.3.2 Out of order data

As data is gathered from clients with unstable connectivity and limited storage capabilities, it is hard to design a protocol to ensure in-order delivery of data. Instead, events can be received in any order, and might even be delayed several days. This mostly affects the aggregate functions, and particularly those whose aggregates include time. Specifically, an aggregation function can never assume that the first received event is the first event chronologically. The aggregate function in the implementation above follows this rule.

5.3.3 New aggregates

A core criteria was that the system should support adding new analytical queries to accommodate for new research requirements. These queries should be applied to all available data, not only data received after its addition. The system described above only applies the aggregate function to data when rows are inserted in the fact table. To solve this, one solution is to let each aggregate keep track of which rows in the fact table that have been applied to the aggregate function. This information is stored in a separate track table for each aggregate. The track table has a single column which holds identifiers to fact table rows. A unique constraint ensures that no data is aggregated more than once.
In the original implementation, a trigger on the fact table automatically aggregated new data on insertion. With a track table, only rows present in the track table should be included in the aggregate. The original trigger is therefore moved to the track table, so that when identifiers are added to the track table, the corresponding row in the fact table is aggregated. A new trigger is added to the fact table to automatically add rows to the track table when inserted in the fact table. In combination, the two triggers ensures that new data is aggregated, and that the track table is updated.

When a new aggregate is added to the system, the track function will automatically add all new events to the track table for that aggregate and consequently also to the online aggregate table. The problem of aggregating existing events is solved by simply inserting the earlier events into the track table after which it is processed by the aggregate function.

5.3.4 Duplicate events

The protocol between the clients and the storage server is designed to retransmit events if the upload fails. Unstable network conditions might, however, result in situations where the server successfully receives the events but the acknowledgement is lost. In this case, the client will retransmit these events resulting in duplicates. To detect duplicates, an universally unique identifier (UUID) field is added to every event type. However, adding a uniqueness constraint in the database is not viable. As the storage server uses bulk insertion to increase speed, a duplicate event would cause the entire insertion to fail. To solve this, an insertion rule on every event table can be used to discard duplicate events silently.

It’s important to notice that this violates the rule that insertions shouldn’t query an event table. However, an index on the UUID column will enable an index scan. For the connection event table, this query executes in 0.05 ms.

6 Multidimensional analysis

In this section the concepts described and implemented in the previous chapter are used to implement support for multidimensional analysis.

The framework presented above provides a recipe for how queries involving aggregates against the fact tables can be implemented using online aggregate tables to achieve large performance improvements. The previous section implemented two-dimensional analysis which plotted a variable over time. What is left is to see how analysis methods of other dimensions such as single value
metrics and cohort charts can be implemented, and how exportation to different data formats can be done directly in the database.

6.1 Single-dimensional analysis

A single-dimensional value is represented by a view with only one row and one column and often represent some key metric in the data. The online aggregate table holding its data can therefore also be very dense. While dense online aggregate tables save disk space, they often contain too little information to be reusable by other queries. It’s therefore preferential to reuse an online aggregate table from a higher dimensional query.

6.1.1 Implementation

To calculate the average number of clicks per second the last week is as simple as summarizing the number of click events and divide by the number of seconds in a week. Here, an aggregation that summarises the number of clicks per week would suffice to implement the query. However, such an aggregation would unlikely be reusable for other queries. Instead, an aggregate that summarises the number of clicks per button per day was used. The presentation view holds a single value which is computed as the sum of the number of clicks each day for all buttons the last week divided by the number of seconds in a week.

6.2 Three-dimensional analysis

While two-dimensional graphs are a powerful visualization form, they are not good at showing difference within a data set. A common method when analyzing user data is to depict differences between different user groups. The user group is divided by some factor such as gender, nationality or sign up date and then compared. Cohort charts can be used to effectively visualize such data sets. An important metric when analyzing user behaviour is churn rate. Churn rate measures the percentage rate in which users are leaving a service. To measure churn rate, a criterion has to be created that separates active users from inactive users.

6.2.1 User cohort

The first cohort separates the user group by the month they first started using the product. For each subgroup, the percentage of users still using the product under each of the following month were plotted. This allows analysis of whether
new customers are more satisfied with the product and thus more likely to keep using it.

6.2.2 Implementation

Creating aggregates for a three-dimensional query poses the opposite problem to creating aggregates for single dimensional queries. To create an aggregation that is generic enough to support several three-dimensional queries would require a sparse aggregation, leading to large online aggregation tables. Instead, one should try to use several more dense aggregates. To implement the user cohort, information about when each user first started using a product as well as usage information each month is needed. Two aggregates store the information, the first aggregate storing the date each user started using the product, and the second aggregate storing the number of product interactions for each user each month. The presentation view divides all users into groups based on the month of the date stored in the first aggregate, and presents the churn rate for each group based on information in the second aggregate.

6.2.3 Button cohort

The second cohort separates all sold buttons by the month they were first used. For each subgroup, the percentage of buttons still in use under each of the following months were plotted. The reason to analyze both users and buttons is that most users bought several buttons at once. By comparing the two charts, it’s possible to detect whether users stopped using some of their buttons implying they bought too many, or if they stopped using all their buttons, implying they are not satisfied with the product.

6.2.4 Implementation

To implement the button cohort, information about when each button were first used as well as if it was clicked during each of the following months is needed. This is easiest to implement as two separate aggregates. The first aggregate stores the date each button was first used, while the second stores usage information for each day for each button. The presentation view divides all buttons into groups based on the month of the date stored in the first aggregate, and presents the churn rate for each group based on information in the second aggregate.
6.3 Data format generation

Presentation tools often require data to be formatted in one of potentially several supported formats. In the framework presented so far, there is no unification of the structure for the presentation views for different queries. Generating formatted output for the different queries thus needs separate implementations for each presentation view. If data exportation is implemented in the middleware server, each new analysis query would require additions to both the database as well as accompanying formatting code in the middleware. A better option is to implement data formatting directly in the database. Formatted presentation views extend the concept of presentation views by following a specific table and data format. As different formatted presentation views for the same query differ only in the way data is formatted, it’s often possible to reduce code duplication by sharing a common presentation view. The presentation requirements in this project required three different presentation formats.

6.3.1 CSV exportation

A CSV formatted presentation view was developed to allow easy importation of aggregation data into spreadsheet software such as Microsoft Excel. A CSV formatted presentation view consists of a single column named csv that contains a valid CSV formatted string. The exportation is often trivial as a CSV document is very similar to a database table. A typical query simply concatenates all the columns of a row with interspersing commas, and concatenates all rows with interspersing line breaks.

6.3.2 Line graph exportation

A data presentation tool was developed alongside the database implementation. The presentation tool uses the concept of formatted presentation views to allow data presentations to be generated in the database. The tool allows presentation of multiple two-dimensional aggregations as line graphs. The formatted presentation view contains one or more rows with two columns. The first column is a label for the line and the second contains an array of two-dimensional points. An endpoint in the middleware converts the graph formatted presentation views to JSON objects that the presentation tool plots.
7 Evaluation

This section evaluates the presented work with respect to the feature and performance requirements.

7.1 Features

The problem statement for this thesis contained four specific feature requirements. The delivered implementation succeeds to fulfill all these requirements.

7.1.1 Access to raw data to enable migration to another framework

All event data is kept in the event tables in a loss-less format. This data can later be exported using a database exportation tool.

7.1.2 Support of analysis methods common in big data framework

The analysis methods common in big data frameworks can be split up in three different categories; one, two and three dimensional analysis. As shown in the query analysis section, each of these usually conforms to patterns that enables them to be online computable. The implementation process described provides clearly defined components and limitations that enables the analytical queries to be implemented in the framework and at the same time ensuring predictable performance.

7.1.3 Support of generation of graphs and cohort charts to present analytical results

The ability to display analytical results using different visual representations is enabled by the online aggregate tables. In section 4.1 and 4.2, it was shown how query analysis can be applied to create online aggregate tables that support graph and cohort presentations. The two step process in where the online aggregate tables hold online analytical results, and presentation views transform these results into different visual representations greatly aids in the implementation of the presentation views. As the online aggregate tables refine the time series data into a set often magnitudes smaller, the presentation views avoids the performance problems of querying a large data set.
7.1.4 Support of new analytical queries to be added to the system, and applying them retroactively to old data

Each new analytical query requires a query analysis to be performed as described in section 4.1 and implemented by following the steps in section 5.1.6. The ability to retroactively apply an analysis is granted by the order-independence requirement made on the aggregates together with the bookkeeping provided by track tables. When a new aggregate is added to the system, it will automatically be applied to all later events. By using the track table, it is possible to determine which events in the fact table that has not been aggregated, and selectively aggregate these. Because the aggregates are order-agnostic, the result is guaranteed to be correct even if the events are aggregated out-of-order.

7.2 Performance

The study conducted about existing performance improvement techniques made it clear that large tables easily introduce bottlenecks in the resource utilization. While some queries may be tractable using such techniques, the study revealed one case were this was not the case. Relying on only such techniques would thus probably lead to an uneven resource utilization. Online aggregate tables achieve an even resource utilization by performing cheap calculation on every insert. The requirements made on the aggregates ensure that overall system performance will not degrade unpredictably when the event tables grow. To compare resource utilization patterns, a measurement like the one presented in the study chapter was made. Metrics for a two week measurement of the transactional workload is presented below.

![CPU utilization](image.png)

Figure 10: CPU utilization
Compared to the first measurement, all three measurements show a higher resource utilization. This is expected as every transactional query now might cause an aggregate to be updated. However, this increase is not significant enough to cause resource utilization problems. A measurement was also performed during which analytical queries were executed at one minute interval was also executed. However, there was no significant difference compared to the above results.
7.3 Deliverable

Besides this report, a complete system was implemented and delivered, supporting 13 queries, 25 event tables, 10 aggregations and 25 presentation views. The system follows the implementation detailed in the prototype chapter with the additional modifications mentioned in section 7.3. The data represents user interactions with a wireless button. The button can be connected to a smart phone and configured in an app to perform one or more actions when clicked. Actions include making a phone call, controlling smart home appliances and navigating music playback. A brief description of the implemented system is given below. Most event tables are not yet used by any query and are only used by the development team to assist in solving customer issues, and are not included below.

7.3.1 Queries

The following queries were implemented:

1. The total number of button clicks.
2. The total number of actions triggered.
3. The average number of clicks per second during the last week.
4. The most executed action the last week.
5. The number of button clicks per day.
6. The number of button clicks per week.
7. The number of actions executed per day.
8. The number of actions executed per week.
9. The number of buttons that have an assigned configuration and is connected to a phone per day.
10. The number of buttons that have been clicked at least once per day.
11. The number of users that has triggered at least one action per day.
12. The cumulative number of buttons that has been connected to a phone per day.
13. The churn rate for buttons based on cohorts identified by the week of the first button click.
7.3.2 Events

To support the implementation of the above queries, the following events were tracked by the smart phone app.

1. Action execution events containing information about which user and type of action that was triggered.
2. Button click events containing information about which user and button that was clicked.
3. Button connected events containing information about which button that was connected.
4. Button configuration events containing information which actions a button was configured to trigger each day.

7.3.3 Query analysis

Before performing an individual analysis of each of the above queries, it is often beneficial to analyse the query set to find similarities. By applying the concept of generalization as discussed in section 6.2.3, several queries might be able to use the same aggregate. Experience shows that this heavily reduces the amount of work required to implement queries. In the system delivered, the 13 queries only required 4 aggregates to implement. When analysing the query set above the following similarities can be seen:

Queries 1, 3, 5, 6, 10 and 13 all use the number of clicks for different buttons. Query 1, 3, 5 and 5 use summarizations of the number of clicks over different time intervals, while query 10 and 13 only considers if a button was pressed during each day or week. All these results can be calculated from an aggregate which summarizes the number of clicks each day per button.

Query 2, 4, 7, 8 and 11 all use the number of action executions for different action types and users. Query 2, 4, 7 and 8 uses summarizations of the number of executions for each action type over different time intervals, while query 11 only considers how many users that executed an action during each day. All these results can be calculated from an aggregate which summarizes the number of executions for each user for each action type each day.

Query 9 considers both connection events from a button and the buttons configuration. Whether a button has been connected during a day is similar to query 10 and 11 and can be calculated from an aggregation which summarizes the number of connection events for a particular button per day. However, determining if a button has an assigned configuration is not straightforward. While
it is possible to add an event for when the button configuration is updated, such events should be avoided. The reason for this is they are very sensitive to partial data loss as one missed update event means that the configuration information is wrong until the next update event. Instead, a snapshot of all button configurations stored in a separate database is performed each day and stored in the statistics database. The data contains information about which action types a button was configured with each day. This can be used in combination with the connection aggregate to compute the result.

Query 12 only considers the first connection event for each button. While this can be derived using the connection aggregate for query 9, a secondary aggregate was created to increase performance. This aggregate stores the information about the first connection event for each button.

### 7.3.4 Aggregate set and Online aggregate tables

The above query analysis results in four aggregates with accompanying online aggregate tables. All aggregates have simple online algorithms that follows a similar pattern.

1. Total number of clicks for each button each day.
2. Total number of executions for each action type for each user each day.
3. Whether or not a button was connected at least once during each day.
4. The time of the first connection event for each button.

The first aggregate is stored in an table with three columns, containing a date, a button identifier and an integer representing the total number of clicks that occurred for that date and button. The online algorithm works by querying the online aggregate table for an entry with the same date and button identifier as the new event. If there is an existing entry, it is updated with an incremented number of clicks. Else a new entry is inserted with the date and button identifier of the event and a click number number of one.

The second aggregate is stored in a table with four columns, containing a date, a string representing the action type, a user identifier and an integer representing the number of executions. The online algorithm works by querying the online aggregate table for an entry with the same date, action type and user identifier as the new event. If there is an existing entry, it is updated with an incremented number of executions. Else a new entry is inserted with the same date, action type and user identifier as the new event, and number of executions of one.

The third aggregate is stored in a table with two columns, containing a date and a button identifier. An entry represents that a particular button has been
connected during that date. The online algorithm works by querying the online aggregate table for an existing entry with the same date and button identifier. If one is found, nothing is updated, else a new entry is inserted with the date and button identifier on the new event.

The fourth aggregate is stored in a table with two columns, containing a date and a button identifier. An entry represents the first time a particular button was connected. The online algorithm works by querying online aggregate table for an existing entry with the same button identifier. If an entry is found, it is date is compared to that of the new event, and if the new event has an earlier date, the existing entry is updated with the earlier date. If no event is found, a new entry is inserted with the button identifier and date of the new event.

7.3.5 Problems

One limitation of this system is that every event is isolated. It is often useful to be able to compute a result from multiple events. To compute the average time a button remains connected, a connection event and the consecutive disconnect event could be used. However, it is impossible to write an aggregate that computes this result and fulfills the requirement of being order independent, as the system can not be sure if the consecutive disconnected event has been received. While this can be solved by using a custom event type like the one presented in the query analysis for query 9, it would result in a large number event types. A possible extension would be to introduce sequence numbers to each event. Each consecutive event of the same type would have a consecutive sequence number, which would allow the system to express computations based on consecutivity.

Another problem is that of event type selection. The set of event types and the data that is included in every event type is determined in the queries that needs to be implemented. As the system should support new queries to be implemented, a new query might require new data that is not included or deducible from any current event type. This would require a new event type that might in addition to the new data include data present in other event types, leading to duplicate information being stored. To prevent this, it is a good practice to include as much information as possible in the event types, including data not currently needed to compute any query. This minimizes the risk of having to implement a lot of similar event types.

A possible improvement suggested in previous research[5] is to make the presentation views visualization-aware. Currently, all data is transferred to the presentation tool which then renders the data. If the presentation tool performs projections or reductions, this could be done in the database instead to reduce data transfers.
7.4 Conclusions

The aim of this thesis is to research if it is possible to implement features commonly offered by specialized big data analysis frameworks within a RDBMS, and maintain good performance when the size of the data set reaches the big data realm. This paper presents a framework for storing updated results for queries for which online algorithms exist, and shows how this can be used to implement several data analysis methods and visual representations common in big data analysis. The framework is also shown to be flexible, supporting new queries to be added to the system to be retroactively applied to old data. A possible downside of the solutions is that the implementation of analytical queries is an analytical process that needs to be performed by a developer. Big data framework usually have interfaces that allows non-developers to perform analysis and visualization. The benefit of using a relational database comes from the fact that relational databases provide general purpose data storage. While big data frameworks provide good performance and usability but with limited functionality, a solution based on a relational database gives the users more power over their data. The ability to export data also means that this can be used in combination with other frameworks.
References


