Project Report Seminar 3

Data Storage Paradigms, IV1351

Teoman Köylüoglu and Teoman@kth.se

Date: 2023-12-04

1 Introduction

Concluding the previous seminar, wherein an OLTP database using row-based entries was created. In this task the objective is, with the use of OLAP queries and views, to query the OLTP database for analytical purposes to serve business reports for the Soundgood Music School. After the queries have been implemented, one will be subject to an "EXPLAIN ANALYZE" function in PostgreSQL to determine its run time efficiency.

Lastly, a historical database will be extended from the original database that utilizes the principle of denormalization. Some attributes will be merged with other tables to this, concluding with a comprehensive discussion on the strategis implemented throughout this report.

This project is a joint collaboration with students Julius Larsson and Victor Naumburg, and a higher grade is sought on this task.

2 Literature Study

In order to confidently traverse the task, sufficient knowledge with regards to SQL queries, databases and denormalization were needed.
As for SQL, this was covered in Paris Carbone's lecture, wherein he gives an overview of the declarative language. Having already studied Relational Algebra and Relational Calculus from chapter 8 in the course book, some insight with regards to the nature of SQL was apprehended as well.
For example, the SELECT statement in SQL is a projection of certain attributes, whereas Select in Relational Calculus is a condition placed upon the query, which represents the WHERE clause in SQL. Other insights such as the fact that a JOIN in SQL is simply a Cartesian Product that is not specified to a certain attribute in the SELECT clause.
This understanding of the connection between Relational Calculus and SQL was of good consequence when writing queries, such as having in mind that SQL does not work in

sets, rather multisets which permit duplicated values in the result. To specify that duplicates are prohibited one would need to use the "DISTINCT" function, or in the opposite case "UNION ALL".

Furthermore, chapters 6 and 7 provided clear explanations on the different functions of SQL, with clear examples on their intended use.
For example, the book starts by introducing JOINS, without naming them as such, rather the two tables are specified in the FROM clause, and the JOIN condition is in the WHERE clause. It is later revealed to be the definition of an INNER-join, subsequently expanding on this fact, NATURAL JOINS were easily introduced as a non-duplicate version wherein the JOIN condition may be omitted.

Laboration 1 which provided practice with queries on the IMDB database proved to be of great benefit when querying the OLTP database. With practice queries heavily emphasizing the use of JOINs, this knowledge was heavily relied upon in the following queries. The tips and trick document provided by Leif mentions a good methodology on how to approach complex queries. He also explains the purpose of using views, as well as how to use an "EXPLAIN ANALYSE" which proved to be utmost benefit for this task.
Lastly, in order to create the historical database, further inquiries with regards to denormalization was made. Not much in the course book is related on this topic, Page 18 in the the course book briefly mentions it, so further investigation on the web on its applications such as from TechTarget, and GeeksForGeeks gave more in-depth breakdown on the topic. Both sources heavily point to the fact that data may become inconsistent such as from potential update anomalies which puts heavier responsibility on the applications (that maintain it) and the database itself.

3 Method

The Soundgood Music school database is implemented using the DBMS called PostgreSQL. All queries are developed in pgAdmin 4, and a low level SQL Shell running is used for the subsequent SELECT-FROM queries to the database to further assistance in the coding.
To verify that the queries work as intended, SELECT-FROM queries performed on the database tables are used. It can be noted that the results are verified with the values fetched from the queries in accordance to the generated data from the previous task.

Tables possessing insufficient data are manually inserted into, using the same data generator tools as previously mentioned. Thereby, further inquiries as to the validity of the queries can be made.

4 Result

This section will consider the results of the queries, the queries themselves and their intended use. Finally an "EXPLAIN ANALYZE" with an in-depth analysis of one of the queries is made to consider its performance efficiency.
As mentioned, pgAdmin 4 was used in the development of the queries. As opposed to using a low level Shell, pgAdmin 4 lets the user see past queries as well as accessing the full query when line breaks are used.

4.1 Query 1

The first query, instructs the designers to show the number of lessons an instructor has given during a specific year. This year, is of course, arbitrary and it is thus sufficient to provide a function which allows for any year to be given as input in the query.
To proceed, the figure above show the expected output of the query. It is mentioned in

Month	Total	Individual	Group	Ensemble
Oct	2	1	0	1
Nov	3	0	2	1
Dec	10	4	4	2

Figure 1: Expected output of Query 1, included to illustrate sought table.

the description, as well as can be deduced from the table, that information with regards to a "month of the lesson", the "total" amount of lessons (from the three types", and the respective amount for each type of lesson is sought. Seeing as this information is spread across multiple tables, some JOINs have to be made. The specified "month" is extracted from the TIMESTAMP used in the "time_of_lesson" which lies in the "lesson" table. The master entity is in addition the "lesson" table with its Primary Key as "lesson_id" which uniquely identifying each type of lesson makes a primary JOIN candidate. Keeping the information in the lesson table is desirable, a LEFT JOIN, joining "lesson" with its subservient tables combines the columns of either tables where the lesson_id corresponding to a lesson type match.
A count function is needed to calculate the number of "lesson_id" occurrences of each lesson type, thus a total of four counts are needed (including the generic lesson entity's). A WHERE condition restricting the output to a given year is included. Furthermore, the lessons are grouped in their respective months and ordered as such.

4.2 Query 2

The second query expects an output showing the amount of students with siblings, ranging from zero to two. This means there ought to be a total of three rows in the resulting

```sql
SELECT EXTRACT(MONTH FROM time_of_lesson) AS month,
COUNT(lesson.lesson_id) AS total,
COUNT(solo_lesson.lesson_id) AS solo,
COUNT(group_lesson.lesson_id) AS group,
COUNT(ensemble.lesson_id) as ensemble
FROM lesson
LEFT JOIN solo_lesson ON solo_lesson.lesson_id = lesson.lesson_id
LEFT JOIN group_lesson ON group_lesson.lesson_id = lesson.lesson_id
LEFT JOIN ensemble ON ensemble.lesson_id = lesson.lesson_id
WHERE EXTRACT(YEAR FROM time_of_lesson) = '2024'
GROUP BY month
ORDER BY month;
```

Figure 2: Query 1.

	month numeric	total bigint	solo bigint	group bigint	ensemble bigint
1	1	29	9	8	12
2	2	20	7	8	5

Figure 3: Query 1 Result.

output, one for each number of siblings. This effectively tells the designers to group and order by "no_of_siblings". Satisfying the first case means that the "sibling_id" ought to be NULL for the student without any siblings. The latter case, is however, not as simple. The code is therefore divided into two sections wherein a "UNION ALL" ought to combine the two results into a greater table.
The first case is handled with a FULL JOIN between the "student" entity and the "student_sibling" entity. Consequentially, a WHERE clause checking for NULL values handles this case efficiently.

As for the latter case, wherein siblings exist for a given student, then seeing as the many-to-many entity "student_sibling" has to columns referring to one another, a select statement ought to be made for each seeing as a certain student_id may point to a sibling_id, but that sibling_id may point to another sibling as well. This means searching for either column would be incorrect, instead combining the results of each in a UNION ALL statement creates a resulting set of unique student_ids and sibling_ids.The inner Count, counts the amount of unique s_ids and the outer Count function counts the number of students and projects the result column wise.

```sql
SELECT COUNT(ss.sibling_id) AS no_of_siblings, COUNT(s.student_id) AS no_of_students
    FROM student AS s
    FULL JOIN student_sibling AS ss
    ON ss.student_id = s.student_id
    WHERE ss.sibling_id IS NULL

    UNION ALL

    SELECT no_of_siblings, COUNT(*) as no_of_students
FROM (
    SELECT COUNT(*) AS no_of_siblings
    FROM (
        SELECT student_id AS s_id FROM student_sibling

        UNION ALL

        SELECT sibling_id AS s_id FROM student_sibling
    )
    GROUP BY s_id
)
GROUP BY no_of_siblings
ORDER BY no_of_siblings;
```

Figure 4: Query 2, without views.

The query above however may, and has been, utilized with a "view". The query can be considered as the set of two sub-queries, wherein their respective results are combined to form a single table. This separation is possible as has been mentioned above, the first SELECT-FROM query considers only the case wherein the number of siblings equal to NULL. The first sub-query is thus reduced its own view "no_of_siblings_for_null".

```sql
CREATE VIEW no_of_siblings_for_null AS
SELECT COUNT(ss.sibling_id) AS no_of_siblings,
COUNT(s.student_id) AS no_of_students
FROM student AS s
FULL JOIN student_sibling AS ss
ON ss.student_id = s.student_id
WHERE ss.sibling_id IS NULL
```

Figure 5: Query 2 View 1.

Similarly, the second query, handling the case of existing siblings, is reduced to its own view.

```sql
CREATE VIEW no_of_siblings_for_not_null AS
SELECT no_of_siblings, COUNT(*) as no_of_students
FROM (
    SELECT COUNT(*) AS no_of_siblings
    FROM (
        SELECT student_id AS s_id FROM student_sibling

        UNION ALL

        SELECT sibling_id AS s_id FROM student_sibling
    )
    GROUP BY s_id
)
GROUP BY no_of_siblings
ORDER BY no_of_siblings
```

Figure 6: Query 2, View 2.

A materialized master view is then created, calling on both views and doing "UNION ALL" on their respective results and providing the expected output.

```sql
CREATE MATERIALIZED VIEW no_of_siblings_per_student AS
SELECT no_of_siblings, no_of_students
FROM no_of_siblings_for_null

UNION ALL

SELECT no_of_siblings, no_of_students
FROM  no_of_siblings_for_not_null

```

Figure 7: Query 2, Materialized Master View.

The query has thus been simplified and the master view can be called onto with a simple SELECT-FROM query, resulting in the query output below.

	no_of_siblings bigint	no_of_students bigint
1	0	428
2	1	32
3	2	2

Figure 8: Query 2 Result.

4.3 Query 3

Proceedingly, the third query is to list all of the instructor_ids and full names of instructor who have during the current month given a certain amount of lessons. Seeing as the objective is to see which instructors are overworking, sorting by the number of lessons is sensible. An INNER-JOIN between the records from the person table and the instructor table to fetch the name records. Another INNER-JOIN is made to fetch the instructor ids for each lesson. The Count function returns the number of tuples specified as no_of_lessons. The extract function takes the month date from the TIMESTAMP of time_of_lesson and matches it to the extracted current month. The "Having Count" is the condition which ultimately satisfies the requirement set forth by the description, selecting an arbitrary number of lessons by which the check ought to be made. The following result is obtained.

```
SELECT  i.instructor_id,
p.first_name, p.last_name,
COUNT(*) AS no_of_lessons FROM instructor AS i
INNER JOIN person AS p ON p.person_id = i.person_id
INNER JOIN lesson AS l ON i.instructor_id = l.instructor_id
WHERE EXTRACT(MONTH FROM l.time_of_lesson) = EXTRACT(MONTH FROM NOW())
GROUP BY i.instructor_id, p.first_name, p.last_name
HAVING COUNT(lesson_id) > 1
ORDER BY no_of_lessons DESC
```

Figure 9: Query 3.

	instructor_id integer	first_name character varying (500)	last_name character varying (500)	no_of_lessons bigint
1	1	Kato	Gardner	2
2	4	Rudyard	Barrett	2
3	7	Curran	Clarke	2
4	11	Lara	Cummings	2
5	12	Keelie	Beard	2
6	21	Sigourney	Lucas	2
7	25	Salvador	Walton	2

Figure 10: Query 3 Result.

4.4 Query 4

Lastly, Query 4 deals with listing all the ensembles held during the next week. From the suggested output in the table as provided by the description, the days are to be projected via their weekday names. This entails extracting information from the timestamps of each weekday in a case statement. Being a generic query, it is reduced to its own view.

```
CREATE VIEW dow AS
SELECT
CASE
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 0) THEN 'Monday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 1) THEN 'Tuesday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 2) THEN 'Wednesday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 3) THEN 'Thursday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 4) THEN 'Friday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 5) THEN 'Saturday'
    WHEN (EXTRACT(DOW FROM time_of_lesson) = 6) THEN 'Sunday'
END AS day,
l.time_of_lesson
FROM lesson as l;
```

Figure 11: Query 4 View.

In order to group the ensemble seats as by their conventions, additional case statements calculating the currently available seats are made. Similarly, as in previous queries, joining the tables becomes necessary, usage of the defined view, and the extracted "week" from the lessons' TIMESTAMP equaled to the current week's incremented by one. The following query and result is obtained.

```sql
SELECT
dow.day,
e.ensemble_genre AS genre,
CASE
    WHEN(CAST(ensemble_maximum AS INTEGER) -
         COUNT(DISTINCT sl.student_id)) > 2 THEN 'Many Seats'
    WHEN(CAST(ensemble_maximum AS INTEGER) -
         COUNT(DISTINCT sl.student_id)) > 0 THEN '1 or 2 Seats'
    WHEN(CAST(ensemble_maximum AS INTEGER) -
         COUNT(DISTINCT sl.student_id)) < 1 THEN 'No Seats'
END AS no_of_free_seats
FROM ensemble AS e
LEFT JOIN dow ON e.lesson_id = dow.lesson_id
LEFT JOIN lesson AS l ON l.lesson_id = e.lesson_id
LEFT JOIN student_lesson AS sl ON e.lesson_id = sl.lesson_id
GROUP BY e.lesson_id, dow.day, l.time_of_lesson
HAVING EXTRACT(WEEK FROM l.time_of_lesson) = EXTRACT(WEEK FROM NOW()) +1
ORDER BY EXTRACT(DOW FROM l.time_of_lesson), e.ensemble_genre;
```

Figure 12: Query 4.

	day text	genre character varying (500)	no_of_free_seats text
1	Tue	Blues	1 or 2 Seats
2	Tue	Soul	1 or 2 Seats
3	Fri	Blues	No Seats
4	Sun	Boyband	Many Seats
5	Sun	Delta Blues	Many Seats
6	Sun	Jazz	Many Seats
7	Sun	Jazz Fusion	Many Seats

Figure 13: Query 4 Result.

All queries pertaining to the task may be found here: A link to the Github repository

4.5 EXPLAIN ANALYZE

The third query was subject to an "EXPLAIN ANALYZE" using the PostgreSQL GUI tool pgAdmin. The purpose of this analysis is to question about the efficiency and understand the underlying nature of the execution plan.

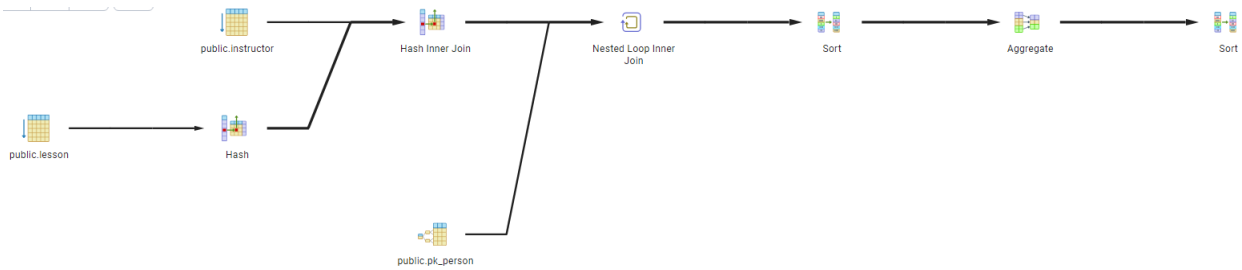

Figure 14: Query 3 Execution Planner.

Node type	Count	Time spent	% of query
Aggregate	1	0.021 ms	2.66%
Hash	1	0.008 ms	1.02%
Hash Inner Join	1	0.025 ms	3.17%
Index Scan	1	0.005 ms	0.64%
Nested Loop Inner Join	1	0.156 ms	19.75%
Seq Scan	2	0.482 ms	61.02%
Sort	2	0.097 ms	12.28%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	**Count**	**Sum of times**	**% of relation**
public.instructor	1	0.414 ms	52.41%
Seq Scan	1	0.414 ms	100%
public.lesson	1	0.068 ms	8.61%
Seq Scan	1	0.068 ms	100%
public.person	1	0.005 ms	0.64%
Index Scan	1	0.005 ms	100%

Figure 15: Query 3 Execution Chart.

4.6 Historical Database

This section considers the higher grade aspect of the project. It aims to create an OLAP historical database out of the original OLTP that had been previously created, with one single table, gather data, statistics and do aggregations on said table, and as an effect the music school. The historical database, in this case, primarily aims to gather data on students who have taken lessons, their types and their corresponding prices associated to those lessons. The historical database was created and named appropriately, and via using a scrip utilizing using the postgres_fdw extension that connection to another server "hist_server" using a foreign data wrapper. Thereafter the publiv schema from the "hist_server" was imported into the "sgms_historical_schema" and the table containing the desired attributes was created.

```
CREATE DATABASE sgms_historical_db;

\c sgms_historical_db;

CREATE TYPE difficulty AS ENUM
('beginner', 'intermediate', 'advanced');

CREATE EXTENSION postgres_fdw;

CREATE SERVER hist_server FOREIGN DATA WRAPPER
postgres_fdw OPTIONS (dbname 'soundgood', host 'localhost', port '5432');

CREATE USER MAPPING FOR POSTGRES
SERVER hist_server OPTIONS (user 'postgres', password '');

CREATE SCHEMA sgms_historical_schema;

IMPORT FOREIGN SCHEMA public
FROM SERVER hist_server INTO sgms_historical_schema;
```

Figure 16: Historical Database Script.

```
CREATE TABLE sgms_historical_schema.historical_table (
    historical_table_id INT GENERATED ALWAYS AS IDENTITY NOT NULL,
    lesson_id INT,
    student_id INT,
    lesson_type VARCHAR(500) NOT NULL,
    ensemble_genre VARCHAR(500),
    type_of_instrument VARCHAR(500),
    price INT NOT NULL,
    first_name VARCHAR(500),
    last_name VARCHAR(500),
    phone_number VARCHAR(500)
);
ALTER TABLE sgms_historical_schema.historical_table
ADD CONSTRAINT PK_historical_table PRIMARY KEY (historical_table_id);
```

Figure 17: Historical Database Table Script.

As evident from the table, the table is created with the information derived from the imported public schema. Moreover the lesson_types are set to "NOT NULL" disallowing for any such tuples. This, in combination with the LEFT JOINs made in the INSERT INTO statement (see repository) excludes irrelevant tuples generated from what would have otherwise been a FULL JOIN.
Denormalization, and in the case of a historical database, means storing redundant data into a singular tabloid form to enhance analytical query performance and to make possible for Soundgood Music School to gather statistics and commit to analytical queries. In the OLTP database, data was spread across multiple tables, requiring JOINS to gather relevant data. In the OLAP database however, the different types of lessons are merged into a singular attribute "lesson_type". Similarly, "ensemble_genre" is included in the table, which means there will be NULLs in the case of solo and group lessons. Implementations such as this, which in an OLTP database would considered undesirable are distinguishing characteristics of a historical database.

```
1 SELECT lesson_id, student_id,
2 lesson_type, ensemble_genre,
3 type_of_instrument, price, first_name,
4 last_name, phone_number
5 FROM sgms_historical_schema.historical_table
6 WHERE first_name = 'Hoyt'AND last_name = 'Finley';
```

Figure 18: Historical Database Query.

	lesson_id integer	student_id integer	lesson_type character varying (500)	ensemble_genre character varying (500)	type_of_instrument character varying (500)	price integer	first_name character varying (500)	last_name character varying (500)	phone_number character varying (500)
1	39	2	solo_lesson	[null]	Piano	75	Hoyt	Finley	+46688574466
2	66	2	solo_lesson	[null]	Violin	200	Hoyt	Finley	+46688574466

Figure 19: Historical Database Query Result.

5 Discussion

Throughout the development of the queries, it was noted that some queries had become more complex than others, necessitating, or at least, hinting to the use of views. For example, the second query uses two separate sub-queries combining their respective rules via a "UNION ALL". The goal then of using a a view here is to facilitate understanding via abstraction. It is not apparently clear from a first look how the second view's query is utilized and thus seeks to complicate where not needed. Instead, renaming such a query as its intended goal in a view achieves this end.

As for the question regarding if queries were made easier to understand, this remains true, especially for query 2. It is without a doubt, the most complicated query construed in this project, as it involves the use of two UNIONs. By storing parts of the query in views and naming them as by their intended meaning, the queries were effectively made easier to understand as well. Performance may also be improved; As the query is to be performed a few times per week, enchancing performance may become desirable. In the case of query 2, the nature of the query pertains to counting the amount of students for a number of siblings. It may be assumed that siblings are not oft signing up each week to the music school, and thus storing the view as a static table makes sense as the view need not often be refreshed. This would also lead to enchancing the performance as the

materialized view is saved on disk, leading to quicker run times. Additionally, via storing it on disk, this query does not affect the live database's state and as other queries are run daily, this leaves more room for other intensive queries to be run on the database instead. If there is a good estimate on the average "sibling sign up" for the music school, then periodic refreshing of the materialized view seeks to even further stress the benefit of storing the view on disk.

The possibility of some of the database lacking in sufficient information was considered before beginning the implementation of the queries, however as the queries were finished and executed, no lack of information was found, with the exception of ensemble lessons pertaining to the following week in query 4. Many of the lessons in the database were taking place farther ahead than the next week, and thus some of the lessons' timestamps were modified such that the query may present desirable values. Similarly, the "ensemble_maximum" had been previously set to 30, but was ultimately decided to be extravagant and was thus reduced to 15 as to not need necessitate further instances of ensembles as in such a case most ensembles would fall in the "many_seats" category.
These changes did not negatively affect the database in any case, and as such was not worsened by it, but instead modified to meet the desirable output of query 4. The structure and underlying mechanics of the database was not altered in any way to conform to query standards.

Proceeding to the "EXPLAIN ANALYSIS" section of the discussion. From the query planner in figure 14, an overview of the execution plan is given. It may be noted that the first operation in the query is the sequential scan on the "lesson" table. It is also evident as seen from figure 15 that it stands for %52 of the query's run time. This may be attributed to it being a sequential scan of linear time complexity O(n) where n is the number of rows in the table. Additionally, this sequential scan handles the Extract filter from the query, resulting in 61 rows omitted by the filter. Proceedingly, this scan is then hashed which evidently posits zero cost to the operation with its complexity of O(1). Thereafter, its sibling node commits to a sequential scan on the "instructor" table having an approximately %9 operation run time percentage. After both tables have been fetched, an index scan on the "person" table of minimum cost is committed, and a hash inner join is made which results in a hash table via joining the keys of the two tables. This operation posits a time complexity of O(n + m) where n, m are the number of rows in the respective tables. This explains the discrepancy between the hash operation and the hash inner join.
After both sibling nodes have executed, a "nested loop inner join" is performed on said tables representing a whole %20 percentage run time. This operation is nested due to its comparing the value of one tuple in the first tuple with each tuple in the second table resulting in a time complexity of O(n*m). Under grandeur circumstances, this operation would possibly become more costly, scaling more heavily than the previously mentioned complexities. Therefore, its lesser run time may be ascribed to the limited number of rows.
It should be noted that execution times may vary depending on hardware related specifics,

and for a greater estimate on the query execution time, operations such as this ought to be repeated over a longer period of time using an average aggregation of some sort to more accurately determine the nature of the query's run time operation.

To conform to the higher grade as pertaining to the project, an OLAP historical database was created. In order to realize this, several changes had to be made, not in the case of the OLTP database, but in the case of transitioning into the historical database, to achieve the desired denormalization. And - as mentioned in the result section - the subclasses pertaining to "lesson" were merged into one attribute inside the historical table. Via this implementation, querying a single individual, for information with regards to the specifics of said person, or the database in general, transforms such queries into a considerably more manageable endeavour. Having attributes of a schema contained into a single table, enables for simplified queries and omits the need for doing complex joins to get identical data. As the nature of an OLAP database is to do analytical queries, then this convention further enhances readability as well as query performance. JOINS, and sequential scans as already noted, takes time. By eliminating the need for JOINS, query performance may be significantly reduced, putting lesser performance constraints on the database and allowing for a streamlined query methodology.

Having discussed some of the obvious pros, keeping data denormalized, in the scope of this database is not without its cons. Denormalization, while leading to enhanced performance and faster query retrieval has encourages redundant data. And while doing so, puts additional constraints on data storage, as unnecessary redundant data is stored on disk. Moreover, data consistency in a denormalized database leads to challenges such as updates, inserts, and deletions wherein anomalies may occur over time should these not be met with meticulous care. As a result, should such anomalies exist, will inevitably lead to inaccurate data, confuting the intended meaning of a historical database in the first place.
Furthermore maintaining a denormalized database, should it over time expand into a larger more complex system, or in general as denormalized database itself, could lead to further limitations on the system. This is particularly evident in the case of keeping the historical database in a single relation, putting limitations on adaptability of the system into a larger relational schema. This could ultimately result in the need of altering the historical database to maintain such complexities, which may prove to be of necessity in subsequent times.