Statistical Analysis of the Effect of Denormalization on Query Resource Consumption and Performance

Ryan Killea, Nathanael Mercaldo, Edith Gomez, Agustina Maccio, Todd Moore

1. Introduction

If one spends any amount of time implementing or leveraging a database system, the constraints of disk usage, memory usage and query execution time tend to quickly manifest themselves. These constraints can have significant effects on the usability and applicability of a database system in real world contexts.

In this paper we explore how denormalization may affect the factors above by performing an in depth comparative analysis of query resource consumption and query execution performance before and after denormalization. Our hope is to derive general insights which may be used by practitioners to more easily determine whether denormalization would provide sufficient advantages in their particular context.

In order to obtain consistent results, we base our comparative analysis on the set of queries and relations provided by the TPC-H benchmark. These queries were executed by the SQLite database system on the entire TPC-H dataset. Note that usage of Sqlite is of particular interest given that it is designed to run in resource constrained environments and is thus likely not traditionally a target for denormalization.

During comparison of statistics we consider two sets of queries namely the original TPC-H query set and an alternative set (based on the original) wherein queries are restructured to target denormalized versions of commonly accessed relations. Finally in our analysis we consider firstly the time/space tradeoffs observed between normalized and denormalized queries then examine in more detail various statistics related to certain aspects of query memory usage and performance such as cache hit/miss frequency, largest single memory allocation and others.

2. Background and Related Work

Normalization is a technique in databases to eliminate data redundancy or inconsistent dependency. Normalizing is achieved by dividing larger tables of the database into smaller ones and defining relationships between them. By splitting the database in this way, data that is not strictly relevant to a particular query is not accessed. A database is said to go to a higher normal form upon normalization (Date, C. J., 2019). Although normalization is generally regarded as the rule for the statute of relational database design, there are still times when database designers may turn to denormalizing a database in order to enhance performance and ease of use. Even though normalization results in many benefits, there is at least one major drawback — poor system performance. For example, there can be particular queries that cause retrieval of data spread across distinct tables of the database, causing a rejoin of those tables, which can slow down responses to such queries.

Conversely, denormalization can be described as a process for reducing the degree of normalization with the aim of improving query processing performance. One of the main purposes of denormalization is to reduce the number of physical tables that must be accessed to retrieve the desired data by reducing the number of joins needed to derive a query answer (Date, C. J., 2019).

Schkolnick and Sorenson are the first researchers introducing the notion of denormalization in 1980. They argued that, to improve database performance, the data model should be viewed by the user because the user is capable of understanding the underlying semantic constraints. Hanus in 1994 developed a list of normalization and denormalization types, and suggested that denormalization should be carefully used according to how the data will be used. A limitation of this work is that the approach to be used in denormalization was not described in sufficient detail. Rodgers (1989) and Hahnke (1996) discuss some of the most common situations in which a database designer consider denormalization (e.g. when there are two entities with a One-to-One relationship and a Many-to-Many relationship with non-key attributes) and illustrates positive effects of denormalization on analytical business applications (e.g. development of facts and dimensions to solve the complexity of hierarchies in

multidimensional analysis). More recently, Sanders (2001) has proposed denormalization as an intermediate step between logical and physical modeling and concluded that denormalization can enhance query performance when it is deployed with a complete understanding of application's requirements. However, the methodologies used in this work are limited because it only compares computation costs. Finally, Lieponienė (2018) evaluated the impact of denormalization on SQL queries and achieved a 95% of reduction in time. Limitations of this paper are the amount of queries used for the tests (3 queries) and lack of other important parameters in terms of performance such as storage costs, memory usage costs, and communication cost.

On the other hand, Bolloju and Toraskar in 1997 presented an approach to avoid or minimize the need for denormalization. In the view of the current popularity of object-oriented techniques in information systems, they introduce the concept of data clustering as an alternative to denormalization. Another possible drawback of denormalization was raised by Coleman in 1989. In this article he argues denormalization decisions usually involve the trade-offs between flexibility and performance, and denormalization requires an understanding of flexibility requirements, awareness of the update frequency of the data, and knowledge of how the database management system, the operating system and the hardware work together to deliver optimal performance.

The inherent trade-offs in normalization/denormalization have not been as thoroughly studied as one would expect. Moreover, denormalization is still one issue that lacks solid principles and guidelines. There has been relatively little research related to illustrating how denormalization enhances database performance and reduces query response time. Therefore, this paper aims at providing a comprehensive analysis of denormalization on 16 SQL queries.

3. Problems Faced

We encountered two key roadblocks while gathering query execution information. Firstly, the original intent of this paper was to facilitate denormalization through leveraging of materialized views. The motivation for using materialized views was to prevent modification of original relations and automation of data synchronization between ground truth relations and denormalized variants. Unfortunately it was soon discovered that SQLite does not support materialized views in its current form and thus all denormalized tables were created manually. Although we suggest practitioners make use of materialized views whenever possible to reduce maintenance overhead, we do not expect that usage of manually created tables as we have done in this paper to affect the validity of our results

The final problem we encountered was related to query execution time. A number of TPC-H queries took longer to execute on available hardware than was feasible. Additionally extremely long running queries would introduce outlier data thus increasing difficulty of direct comparison of query statistics.

4. Approach

4.1 Overview

Our approach involved three stages namely denormalization, data gathering and data analysis. During the denormalization stage, a number of TPC-H relations were denormalized and relevant data was joined and copied from source relations into the new denormalized relations. Next 16 TPC-H queries were refactored such that they directly queried desired attributes of the denormalized tables instead of relying on joins. Next we executed both the original query set and denormalized variants with SQLite statistics enabled. The following key variables were captured for 1 run of all queries in both sets. Note that queries were run on a single machine to reduce data variability.

Table 4.1.1: Query statistics fields used in analysis

Memory Used	Memory Used (Max)	Number of Outstanding Allocations	Number of Outstanding Allocations (Max)	Number of Pcache Overflow Bytes	Number of Pcache Overflow Bytes (Max)
Largest Allocation	Largest Pcache Allocation	Lookaside Slots Used	Lookaside Slots Used (Max)	Successful lookaside attempts	Lookaside failures due to size
Lookaside failures due to OOM	Pager Heap Usage	Page cache hits	Page cache misses	Page cache writes	Page cache spills
Schema Heap Usage	Statement Heap/Lookaside Usage	Fullscan Steps	Sort Operations	Autoindex Inserts	Virtual Machine Steps
Reprepare operations	Number of times run	Memory used by prepared stmt	Bytes received by read()	Bytes sent to write()	Read() system calls
Write() system calls	Bytes read from storage	Bytes written to storage	Canceled write bytes	Query Run Time	

In order to facilitate extraction of statistics from SQLite we created a simple python script which reads terminal output from the sqlite process and writes extracted statistics to a csv file.

To briefly elaborate on what the non-self-explanatory statistics mean, we will review some of them. Largest Pcache allocation is the largest allocation to the pager cache that SQLite does, which is indicative of queries where long sequences are likely to repeat. Lookaside is a technique wherein the data pre-allocates a large amount of memory, using a fixed sized “slot” for small allocations in this pool. Using equal size slots makes the allocation and deallocation faster with this memory at the cost of overhead from wrong-sizing. Currently SQLite has two such pools with different slot sizes. Lookaside may fail if there isn’t a slot available or if the size is incorrect for using the technique. Likewise the Page cache is another technique for improving efficiency. SQLite will cache data as it is paged and attempt to avoid reading data written to disk by keeping a copy in memory of pages that are frequently read. Because reading from disk is expensive, page cache hits are very important for disk-heavy queries. Fullscan steps are the steps taken when scanning through a full table. Sort operations are the number of sorts that a query performs.

4.2 Queries and Relations

The general technique used to denormalize each of the 14 selected TPC-H benchmark queries was to eliminate as many of the joins of tables as reasonable during the original query, and including sub-queries. This was done by creating a denormalized table that joined all pertinent tables related by their key columns. In an effort to make the table have maximum utility we did not filter these joins past equality, and omitted query “parameters” such as country names, so our denormalized table could be useful on all queries of similar form even if it was used with different locations or time ranges (for example). This table was then used as a reference in the *from* clause of the query which was intended to drastically lower required resources for query processing. Not all queries in the TPC-H benchmark fit the requirements for this optimization method and therefore those that did not fit were omitted. Additionally, due to a constraint of resources, any query that took longer than 30 minutes to execute was omitted. This criteria left us with 14 of the 22 queries provided by the benchmark. The 14 selected TPC-H queries chosen for modification are exhibited in *Appendix 7.1*.

In order to compare these statistics meaningfully, each of the original queries, create tables (handling joins) and new queries were clocked for run time, memory usage, cache statistics, and more using SQLite’s built-in methods:

1. Explain Query Plan as .eqp
2. CPU time measurements as .timer
3. Resource usage statistics as .stats

Denormalizing tables, in most cases, did not require a drastic change in the original query to accommodate new denormalized tables. Generally only joins in the From clause or the Where clause were required to be handled in this optimization as exemplified in query 14 below.

```
select
       100.00 * sum(case
       when p_type like 'PROMO%'
       then l_extendedprice * (1 - l_discount)
       else 0
       end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
       lineitem,
       part
where
       l_partkey = p_partkey
       and l_shipdate >= date ('1994-12-01')
       and l_shipdate < date ("1994-12-01", "+1 months");
```

Figure 4.2.1: Example Query of business question 14 as provided by the official TPC-H benchmark documentation

In the above there is a simple join in the 'where' clause between lineitem and part tables using the relationship between l_partkey and p_partkey. Additional to that, other tables and columns required to satisfy the query above include l_extendedprice, l_discount and l_shipdate. It would be possible to denromalize further by removing the aggregation in the select statement and even filtering the rest of the where clause, but this would make it generally incompatible for comparison across the whole. A table was created with the required joins using the standard format seen in figure 4.2.2.

```
Create table query14tab as
   Select
       p_type,
       l_extendedprice,
       l_discount,
       l_shipdate
   from
       lineitem,
       part
   where
       l_partkey = p_partkey;
```

Figure 4.2.2: Denormalized table creation for example query 14 of the TPC-H benchmark

The required information is selected and stored for the new query.

```
select
       100.00 * sum(case
       when p_type like 'PROMO%'
       then l_extendedprice * (1 - l_discount)
       else 0
       end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
       query14tab
where
       l_shipdate >= date ("1994-12-01")
       and l_shipdate < date ("1994-12-01", "+1 months");
```

Figure 4.2.3: New query for query example 14 of the TPC-H standard

The query is now run with the joined data, and generally remains unaltered if able. The run time and resource statistics are captured separately during these 3 processes and are documented for comparison and analysis.

Some queries during the third stage of this optimization required further refinement in the new query. One example of this is during Query 4 there was unwanted replication of records causing a read output that was more than expected, assumedly from a cross join mechanism used in Sqlite3. This was corrected by using the 'distinct' function during the aggregation in the select clause. Due to this addition, it was noticed that this greatly decreased performance of the query.

5. Analysis

In order to maximize the number of useful insights gleaned from our data, we leverage multiple varied types of statistical techniques in our analysis. We begin by applying descriptive analysis methods namely A/B comparison of unprocessed statistics. We then proceed to more indirect methods involving detection of deeper relationships between individual statistics. At each step in analysis we attempt to logically deduce explanations for observed differences between the execution statistics of denormalized and normalized queries.

5.1 Direct Analysis

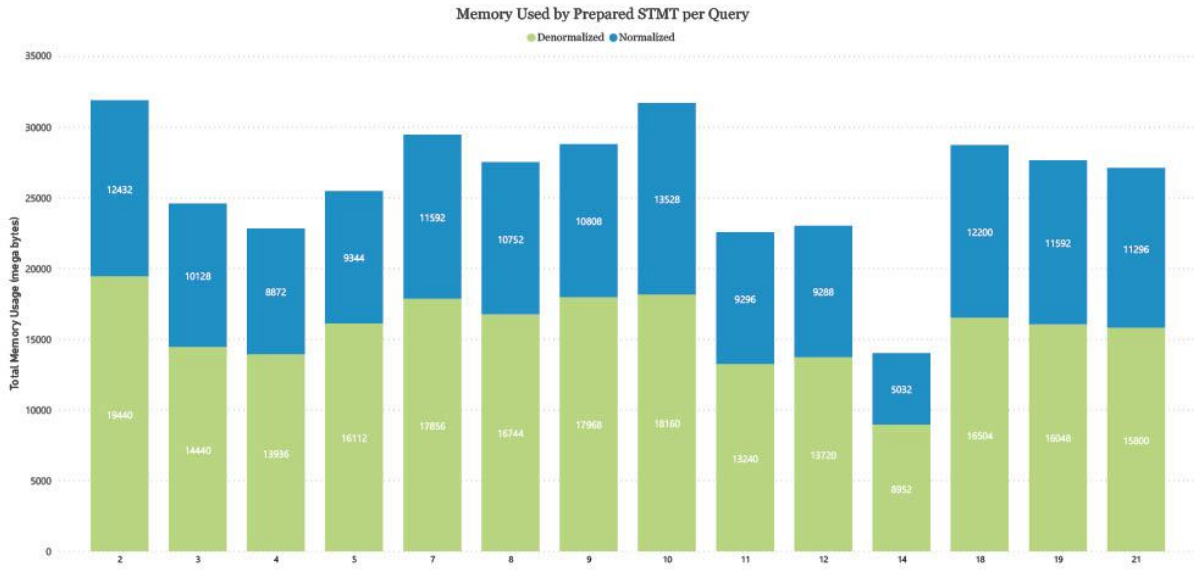

Figure 5.1.1: Comparing memory used (after the query plan has been prepared)

For normalized and denormalized queries, we can see that the denormalized queries tend to use more memory than normalized, but are roughly on the same order of magnitude and the two correlate. This could be because less complicated logic (fewer joins) are being performed, but tuples are not able to be eliminated as early in these joins, as conditions outside of equality between tables are not performed. As a result these rather large tables are being loaded into memory and scanned.

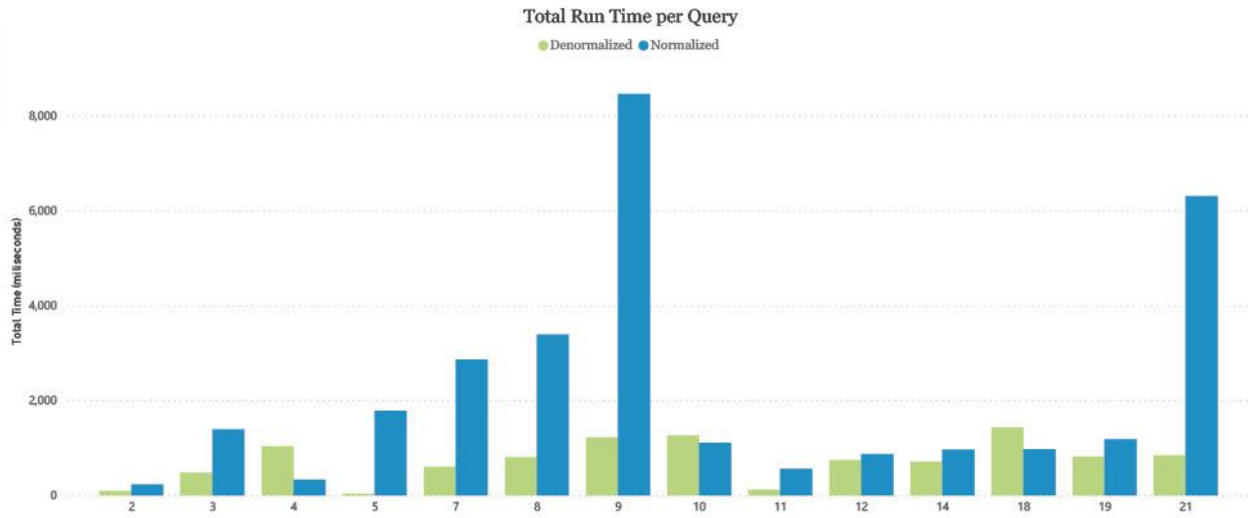

Figure 5.1.2: Comparing Total Run Time per Query between Normalized and Denormalized.

Examining Fig. 5.1.2, we can see the benefits of denormalization on the data at rest. Outside of specific queries where denormalization performs poorly such as Queries 18 and 4, Denormalization drastically improves query performance in runtime. What makes it so spectacular for queries 5, 7, 8, 9, and 21, is that these produce large denormalized tables encompassing joins of at least 4 large tables each. These joins, being precomputed, trade disk space (in the form of super-large tables) for time efficiency. The original normalized variations of Query 18 and 4 both join 3 or fewer tables and do not pre-filter by membership and date range respectively. As a result, checking these conditions in the scan with duplicated entries for matches on the join, (as is the case in the denormalized query variations) leads to inefficiency.

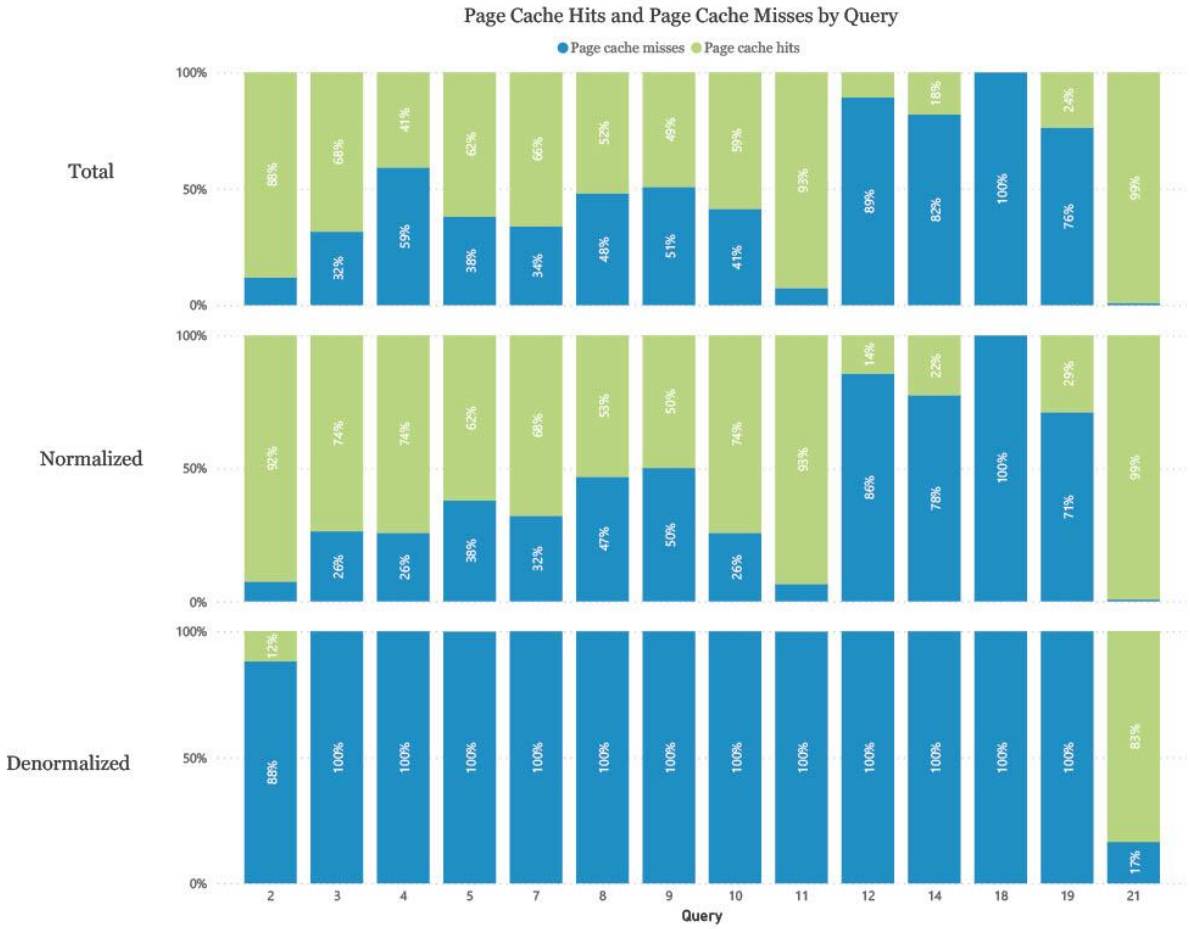

Figure 5.1.3: Comparing the fraction of cache hits and misses between normalized and denormalized.

We can see that denormalization leads to reduced page cache performance on all queries and 0 hits on all but two. This can potentially be explained because Denormalized avoids revisiting cached data from our preconstructed denormalized tables. The revisiting would all occur while building the joins that have already been performed. This can be viewed as a tradeback of denormalization.

5.2 Indirect Analysis

In order to discover subtle connections between different statistics we now train a decision tree classifier with the goal of classifying each query in our dataset as either belonging to the normalized or denormalized set of queries. Below is a decision tree resulting from said training process. The tree in question achieved 88% classification accuracy under 10 fold cross validation.

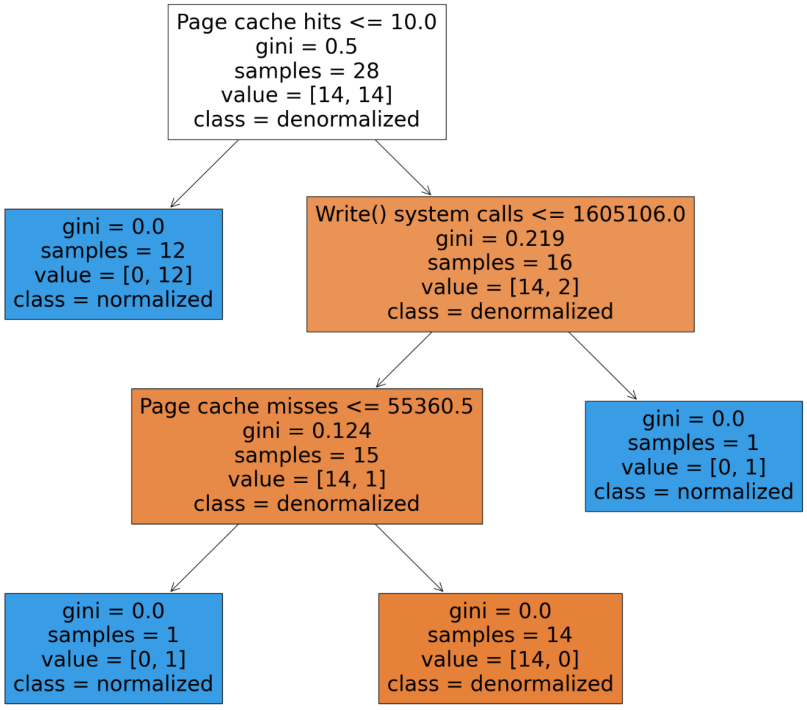

Figure 5.2.1: Decision tree trained on dataset containing both normalized and denormalized queries and corresponding class labels

Here a number of interesting results are exemplified. Firstly the single greatest predictor of query class membership appears to be the number of page cache hits. This result aligns with previous analysis. Next we see that normalized queries appear to involve more writes to disk then denormalized variations. This result could be caused by the need to cache data on disk due to the greater memory requirements of the cross joins present in normalized queries. Finally we observe the involvement of page cache misses, however this result is less interesting given that page cache misses should be inversely correlated with page cache hits.

In general the decision tree analysis approach leveraged here revealed a number of interesting properties of our queries that would be challenging to discover through brute force exploration of the data. We expect that a larger dataset and more involved tuning of the decision tree training process would result in the discovery of more interesting relationships between query execution statistics. Such a methodology would certainly be interesting to explore in future work.

6. Insights and Conclusion

We believe that the results of our study clearly demonstrate the costs and benefits of leveraging denormalization in a real world database context. In general, we observed that usage of denormalization increased memory consumption during query execution in nearly all cases and consumed additional disk space due to data duplication; however performance improvements from denormalization were seen to be quite substantial, exhibiting on average a 2x improvement in query performance.

We now end our study by presenting a number of recommendations regarding application of denormalization in real world contexts based on our results. Firstly, given the increased memory/disk usage of denormalization, we would like to emphasize that denormalization should only be applied in contexts where query execution performance is a significant bottleneck and/or an obstacle to realization of functional business requirements. Additionally, given the overhead associated with denormalized table creation and maintenance, we recommend only performing denormalization on read heavy tables/tables accessed by read heavy queries.

Next, given the negligible performance gains observed from the denormalization of queries which did not perform many joins, we reinforce the commonly mentioned suggestion to prioritize denormalization of relations/queries which perform a large number of joins.

Finally we present what we believe is a novel observation derived from our study, namely that denormalization should be avoided when operating in an environment with limited/restricted disk io throughput. This suggestion is based on the observation that denormalized queries appear to make far less effective use of the page cache thus forcing the database engine to access the disk more often. (Note that such a drawback could be avoided in contexts where an in-memory database is used.)

7. Appendix

7.1 Collaboration Strategy

This study was a highly collaborative effort. Team communication was primarily accomplished through the Discord messaging platform. Additionally collaborative editing of documents was enabled through the use of Google Docs and Google sheets.

Project work was divided between teammates as a function of individual expertise, however frequent communication between members was encouraged to ensure proper goal alignment.

During the project a number of “all hands” meetings were scheduled to verify task progress and brainstorm solutions to various problems encountered. Additionally a few 1 on 1 focus meetings were scheduled.

7.2 Modified Queries

Num	Original Query	Denormalized Relation Creation	Denormalized Query
Q2	`select` `  s_acctbal,` `  s_name,` `  n_name,` `  p_partkey,` `  p_mfgr,`	`create table query2tab as` `  select` `  s_acctbal,` `  s_name,` `  n_name,` `  p_partkey,`	`select` `  s_acctbal,` `  s_name,` `  n_name,` `  p_partkey,` `  p_mfgr,`

	s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 15 and p_type like '%BRASS' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE') order by s_acctbal desc, n_name, s_name, p_partkey;	p_mfgr, s_address, s_phone, s_comment, p_size, p_type, r_name, ps_supplycost from Part, Partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey;	s_address, s_phone, s_comment from query2tab where p_size = 15 and p_type like '%BRASS' and r_name = 'EUROPE' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE') order by s_acctbal desc, n_name, s_name, p_partkey;
Q3	select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'MACHINERY' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date ('1995-03-21') and l_shipdate > date ('1995-03-21') group by	create table query3tab as select o_orderkey, o_orderdate, o_shippriority, c_mktsegment, l_extendedprice, l_discount, l_shipdate from orders, lineitem, customer where l_orderkey = o_orderkey and c_custkey = o_custkey;	select o_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from query3tab where c_mktsegment = 'MACHINERY' and o_orderdate < date ('1995-03-21') and l_shipdate > date ('1995-03-21') group by o_orderkey, o_orderdate, o_shippriority order by

	l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate;		revenue desc, o_orderdate;
Q4	select o_orderpriority, count(*) as order_count from orders where o_orderdate >= date ('1996-03-01') and o_orderdate < date ('1996-03-01', '+3 months') and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority order by o_orderpriority;	create table query4tab as select * from lineitem,orders where l_orderkey=o_orderkey;	select o_orderpriority, count(distinct o_orderkey) as order_count from query4tab where o_orderdate >= date ('1996-03-01') and o_orderdate < date ('1996-03-01', '+3 months') and l_commitdate < l_receiptdate group by o_orderpriority order by o_orderpriority;
Q5	select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE' and o_orderdate >= date ('1997-01-01') and o_orderdate < date ('1997-01-01', '+1 years') group by n_name order by	create table query5tab as select C_custkey l_orderkey, L_suppkey, c_nationkey, n_regionkey, l_extendedprice, l_discount, n_name, r_name, o_orderdate from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey;	select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue from query5tab where r_name = 'EUROPE' and o_orderdate >= date ('1997-01-01') and o_orderdate < date ('1997-01-01', '+1 years') group by n_name order by revenue desc;

	`` revenue desc; ```		
Q7	``` select supp_nation, cust_nation, l_year, sum(volume) as revenue from (select n1.n_name as supp_nation, n2.n_name as cust_nation, strftime('%Y',l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as volume from supplier, lineitem, orders, customer, nation n1, nation n2 where s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey = o_custkey and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey and ((n1.n_name = 'PERU' and n2.n_name = 'IRAQ') or (n1.n_name = 'IRAQ' and n2.n_name = 'PERU')) and l_shipdate between date ('1995-01-01') and date ('1996-12-31')) as shipping group by supp_nation, cust_nation, l_year order by supp_nation, cust_nation, l_year; ```	``` create table query7tab as select n1.n_name as supp_nation, n2.n_name as cust_nation, l_shipdate, l_extendedprice, l_discount, s_suppkey, o_orderkey, c_custkey from Supplier, Lineitem, Orders, Customer, nation n1, nation n2 where s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey = o_custkey and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey; ```	``` select supp_nation, cust_nation, strftime('%Y',l_shipdate), sum(l_extendedprice * (1 - l_discount)) as revenue from query7tab where ((supp_nation = 'PERU' and cust_nation = 'IRAQ') or (supp_nation = 'IRAQ' and cust_nation = 'PERU')) and l_shipdate between date ('1995-01-01') and date ('1996-12-31') group by supp_nation, cust_nation, strftime('%Y',l_shipdate) order by supp_nation, cust_nation, strftime('%Y',l_shipdate); ```

Q8

```
select
        o_year,
        sum(case
        when nation = 'IRAQ' then volume
        else 0
        end) / sum(volume) as mkt_share
from
        (
        Select
        strftime('%Y',o_orderdate) as
o_year,
        l_extendedprice * (1 - l_discount)
as volume,
        n2.n_name as nation
        From
        Part,
        Supplier,
        Lineitem,
        Orders,
        Customer,
        nation n1,
        nation n2,
        Region
        Where
        p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and s_nationkey = n2.n_nationk
          and o_orderdate between date
('1995-01-01') and date ('1996-12-31')
        and p_type = 'STANDARD ANODIZED
BRASS'
        ) as all_nations
group by
        o_year
order by
        o_year;
```

```
Create table query8tab as
 Select
       o_orderdate,
       l_extendedprice,
       l_discount,
       N2.n_name,
       R_name,
       p_type
 from
      part,
      supplier,
      lineitem,
      orders,
      customer,
      nation n1,
      nation n2,
      region
 where
      p_partkey = l_partkey
      and s_suppkey = l_suppkey
      and l_orderkey = o_orderkey
      and o_custkey = c_custkey
      and c_nationkey = n1.n_nationkey
      and n1.n_regionkey = r_regionkey
      and s_nationkey = n2.n_nationkey;
```

```
select
       o_year,
       sum(case
               when nation = 'IRAQ' then
volume
               else 0
               end) / sum(volume) as
mkt_share
from
       (
       select

strftime('%Y',o_orderdate) as o_year,
               l_extendedprice * (1 -
l_discount) as volume,
               n_name as nation
       from
               query8tab
       where
               o_orderdate between date
("1995-01-01") and date ("1996-12-31")
               and r_name = 'MIDDLE
EAST'
               and p_type = 'STANDARD
ANODIZED BRASS'
       ) as all_nations
group by
       o_year
order by
       o_year;
```

Q9

```
select
      nation,
      o_year,
      sum(amount) as sum_profit
from
      (
               select
              n_name as nation,
              strftime('%Y',o_orderdate)
as o_year,
              l_extendedprice * (1 -
l_discount) - ps_supplycost * l_quantity
as amount
               from
                       part,
                       supplier,
```

```
create table query9tab as
  select
      n_name,
      O_orderdate,
      l_extendedprice,
      l_discount,
      ps_supplycost,
      l_quantity,
      p_name,
      s_suppkey,
      ps_suppkey,
      ps_partkey,
      p_partkey,
      o_orderkey,
      s_nationkey
  From
```

```
select
       nation,
       o_year,
       sum(amount) as sum_profit
from
       (
               select
               n_name as nation,
           strftime('%Y',o_orderdate)
as o_year,
                       l_extendedprice
* (1 - l_discount) - ps_supplycost *
l_quantity as amount
               from
               query9tab
               where
```

	lineitem, partsupp, orders, nation where s_suppkey = l_suppkey and ps_suppkey = l_suppkey and ps_partkey = l_partkey and p_partkey = l_partkey and o_orderkey = l_orderkey and s_nationkey = n_nationkey and p_name like '%antique%') as profit group by nation, o_year order by nation, o_year desc;	part, supplier, lineitem, partsupp, Orders, nation Where s_suppkey = l_suppkey and ps_suppkey = l_suppkey and ps_partkey = l_partkey and p_partkey = l_partkey and o_orderkey = l_orderkey and s_nationkey = n_nationkey;	p_name like '%antique%') as profit group by nation, o_year order by nation, o_year desc;
Q10	select c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment from customer, orders, lineitem, nation Where c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= date ('1993-12-01') and o_orderdate < date ('1993-12-01', '+3 months') and l_returnflag = 'R' and c_nationkey = n_nationkey group by c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment order by revenue desc;	Create table query10tab as Select c_custkey, c_name, c_acctbal, n_name, c_address, c_phone, c_comment, o_orderdate, l_returnflag, l_extendedprice, l_discount From customer, orders, Lineitem, nation Where c_custkey = o_custkey And l_orderkey = o_orderkey And c_nationkey = n_nationkey;	select c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment from query10tab where o_orderdate >= date ("1993-12-01") and o_orderdate < date ("1993-12-01", "+3 months") and l_returnflag = 'R' group by c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment order by revenue desc;

Q11

```
Select
       ps_partkey,
       sum(ps_supplycost * ps_availqty) as
value
from
       partsupp,
       supplier,
       nation
where
       ps_suppkey = s_suppkey
       and s_nationkey = n_nationkey
       and n_name = 'CHINA'
group by
       ps_partkey having
                sum(ps_supplycost *
ps_availqty) > (
       select

sum(ps_supplycost * ps_availqty) *
0.0001000000
       From
       Partsupp,
       Supplier,
       Nation
       Where
       ps_suppkey = s_suppkey
       and s_nationkey = n_nationkey
       and n_name = 'CHINA')
order by
       value desc;
```

```
create table query11tab as
   select
       ps_supplycost,
       ps_availqty,
       n_name,
       ps_suppkey,
       s_nationkey,
       ps_partkey
   from
       partsupp,
       supplier,
       nation
   where
       ps_suppkey = s_suppkey
       and s_nationkey = n_nationkey;
```

```
Select
       ps_partkey,
       sum(ps_supplycost * ps_availqty)
as value
from
       query11tab
where
       n_name = 'CHINA'
group by
       ps_partkey having
                sum(ps_supplycost *
ps_availqty) > (
       felect
       sum(ps_supplycost * ps_availqty)
* 0.0001000000
       from
       Query11tab
       where
       n_name = 'CHINA')
order by
       value desc;
```

Q12

```
select
       l_shipmode,
       sum(case
       when o_orderpriority = '1-URGENT'
       or o_orderpriority = '2-HIGH'
       then 1
       else 0
       end) as high_line_count,
       sum(case
       when o_orderpriority <> '1-URGENT'
       and o_orderpriority <> '2-HIGH'
       then 1
       else 0
       end) as low_line_count
from
       orders,
       lineitem
where
       o_orderkey = l_orderkey
       and l_shipmode in ('AIR', 'RAIL')
       and l_commitdate < l_receiptdate
       and l_shipdate < l_commitdate
       and l_receiptdate >= date
('1994-01-01')
       and l_receiptdate < date
('1994-01-01', '+1 years')
group by
```

```
create table query12tab as
   select
       l_shipmode,
       o_orderpriority,
       l_receiptdate,
       l_commitdate,
       l_shipdate
   from
       orders,
       lineitem
   where
       o_orderkey = l_orderkey;
```

```
select
       l_shipmode,
       sum(case
       when o_orderpriority = '1-URGENT'
       or o_orderpriority = '2-HIGH'
       then 1
       else 0
       end) as high_line_count,
       sum(case
       when o_orderpriority <>
'1-URGENT'
       and o_orderpriority <> '2-HIGH'
       then 1
       else 0
       end) as low_line_count
from
       query12tab
where
       l_shipmode in ('AIR', 'RAIL')
       and l_shipmode in ('AIR', 'RAIL')
       and l_commitdate < l_receiptdate
       and l_shipdate < l_commitdate
       and l_receiptdate >= date
("1994-01-01")
       and l_receiptdate < date
("1994-01-01", "+1 years")
group by
```

	`` l_shipmode order by l_shipmode; ```		``` l_shipmode order by l_shipmode; ```
Q14	``` select 100.00 * sum(case when p_type like 'PROMO%' then l_extendedprice * (1 - l_discount) else 0 end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue from lineitem, part where l_partkey = p_partkey and l_shipdate >= date ('1994-12-01') and l_shipdate < date ('1994-12-01', '+1 months'); ```	``` Create table query14tab as Select p_type, l_extendedprice, l_discount, l_shipdate from lineitem, part where l_partkey = p_partkey; ```	``` select 100.00 * sum(case when p_type like 'PROMO%' then l_extendedprice * (1 - l_discount) else 0 end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue from query14tab where l_shipdate >= date ("1994-12-01") and l_shipdate < date ("1994-12-01", "+1 months"); ```
Q18	``` select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, sum(l_quantity) from customer, orders, lineitem where o_orderkey in (select l_orderkey from lineitem group by l_orderkey having sum(l_quantity) > 313) and c_custkey = o_custkey and o_orderkey = l_orderkey group by c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice order by o_totalprice desc, o_orderdate; ```	``` Create table query18tab as select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, l_quantity From customer, orders, lineitem where c_custkey = o_custkey and o_orderkey = l_orderkey; ```	``` select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, sum(l_quantity) from query18tab where o_orderkey in (select l_orderkey from lineitem group by l_orderkey having sum(l_quantity) > 313) group by c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice order by o_totalprice desc, o_orderdate; ```

Q19

```
select
        sum(l_extendedprice* (1 -
l_discount)) as revenue
from
        lineitem,
        part
where
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#43'
                and p_container in ('SM
CASE', 'SM BOX', 'SM PACK', 'SM PKG')
                and l_quantity >= 3 and
l_quantity <= 3 + 10
                and p_size between 1 and 5
                and l_shipmode in ('AIR',
'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#25'
                and p_container in ('MED
BAG', 'MED BOX', 'MED PKG', 'MED PACK')
                and l_quantity >= 10 and
l_quantity <= 10 + 10
                and p_size between 1 and
10
                and l_shipmode in ('AIR',
'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#24'
                and p_container in ('LG
CASE', 'LG BOX', 'LG PACK', 'LG PKG')
                and l_quantity >= 22 and
l_quantity <= 22 + 10
                and p_size between 1 and
15
                and l_shipmode in ('AIR',
'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        );
```

```
create table query19tab as
   select
       p_brand,
       p_container,
       l_quantity,
       l_extendedprice,
       p_size,
       l_shipmode,
       l_shipinstruct,
       l_discount
   from
       lineitem,
       part
   where
       p_partkey = l_partkey;
```

```
select
        sum(l_extendedprice* (1 -
l_discount)) as revenue
from
        query19tab
where
        (
                p_brand = 'Brand#43'
                and p_container in ('SM
CASE', 'SM BOX', 'SM PACK', 'SM PKG')
                and l_quantity >= 3 and
l_quantity <= 3 + 10
                and p_size between 1 and
5
                and l_shipmode in
('AIR', 'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        )
        or
        (
                p_brand = 'Brand#25'
                and p_container in ('MED
BAG', 'MED BOX', 'MED PKG', 'MED PACK')
                and l_quantity >= 10 and
l_quantity <= 10 + 10
                and p_size between 1 and
10
                and l_shipmode in
('AIR', 'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        )
        or
        (
                p_brand = 'Brand#24'
                and p_container in ('LG
CASE', 'LG BOX', 'LG PACK', 'LG PKG')
                and l_quantity >= 22 and
l_quantity <= 22 + 10
                and p_size between 1 and
15
                and l_shipmode in
('AIR', 'AIR REG')
                and l_shipinstruct =
'DELIVER IN PERSON'
        );
```

Q21

```
select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate >
l1.l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey =
l1.l_orderkey
                        and l2.l_suppkey
<> l1.l_suppkey
        )
        and not exists (
                select
                        *
                from
                lineitem l3
                where
                l3.l_orderkey =
l1.l_orderkey
                and l3.l_suppkey <>
l1.l_suppkey
                and l3.l_receiptdate >
l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = 'INDIA'
group by
        s_name
order by
        numwait desc,
        s_name;
```

```
create table query21tab as
   select
       s_name,
       s_nationkey,
       n_nationkey,
       o_orderkey,
       l_receiptdate,
       l_commitdate,
       o_orderstatus,
       s_suppkey,
       n_name
   from
       supplier,
       orders,
       nation,
       lineitem
   where
       s_nationkey = n_nationkey
       and o_orderkey=l_orderkey
       and s_suppkey = l_suppkey;
```

```
select
        l1.s_name,
        count(*) as numwait
from
        query21tab l1
where
        l1.o_orderstatus = 'F'
        and l_receiptdate > l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey =
l1.o_orderkey
                        and l2.l_suppkey
<> l1.s_suppkey
        )
        and not exists (
                select
                *
                from
                lineitem l3
                where
                l3.l_orderkey =
l1.o_orderkey
                    and l3.l_suppkey <>
l1.s_suppkey
                    and l3.l_receiptdate
> l3.l_commitdate
        )
        and l1.n_name = 'INDIA'
group by
        l1.s_name
order by
        numwait desc,
        l1.s_name;
```

8. References

[1] Bolloju, N., & Toraskar, K. (1997). Data clustering for effective mapping of object models to relational models. Journal of Database Management (JDM), 8(4), 16-24.

[2] Coleman, G. (1989). Normalizing not only way. Computerworld, 12(1989), 63-64.

[3] Date, C. J. (2019). Denormalization. Database Design and Relational Theory: Normal Forms and All That Jazz, 161-182.

[4] Hahnke, J. (1996). Data model design for business analysis. Unix Review, 14(10), 6.

[5] Hanus, M. (1993). To normalize or denormalize, that is the question. In Int. CMG Conference.

[6] Lieponienė, J. (2018). A study of relational and document databases queries performance. Informatika. Scientific review of the Dennis Gabor College, 12-22.

[6] Rodgers, U. (1989). Denormalization: why, what, and how. Database Programming and Design, 2(12), 46-53.

[7] Sanders, G. L., & Shin, S. (2001). Denormalization effects on performance of RDBMS. In Proceedings of the 34th Annual Hawaii International Conference on System Sciences (pp. 9-pp). IEEE.

[8] Schkolnick, M., & Sorenson, P. (1980). Denormalization: a performance oriented database design technique. In AICA Congress, Bologna, Italy.