A Study on Query Optimization

Anjali(anjali@wisc.edu) and Sakshi Bakshi(sbansal8@wisc.edu)

Abstract

Network functions typically need to be visited in a specific order to meet certain objectives, giving rise to the notion of Service Function Chaining. Software-Defined-Networking enables fine-grained traffic routing optimization while satisfying correct traversal of network functions. In this work, we investigate the problem of maximizing throughput in SDN-enabled networks with respect to service chaining specifications under both traditional and new constraints. Besides the algorithm design, we also derive rigorous performance bounds. In the offline traffic routing case, we propose Traffic-Merging-Algorithm and prove that, although the underlying optimization problem is generally NP-hard, our algorithm can efficiently compute the optimal solution in practical settings. In the online traffic routing case, we propose the Primal-Dual-Update-Algorithm, which comes with a system parameter that trades off the algorithm's throughput competitiveness and its meeting of QoS requirements, and prove that our online algorithm achieves optimal tradeoff. We demonstrate that our solutions can be used to address practical problems by conducting simulation-based evaluation over backbone and data center topologies.

1 Introduction

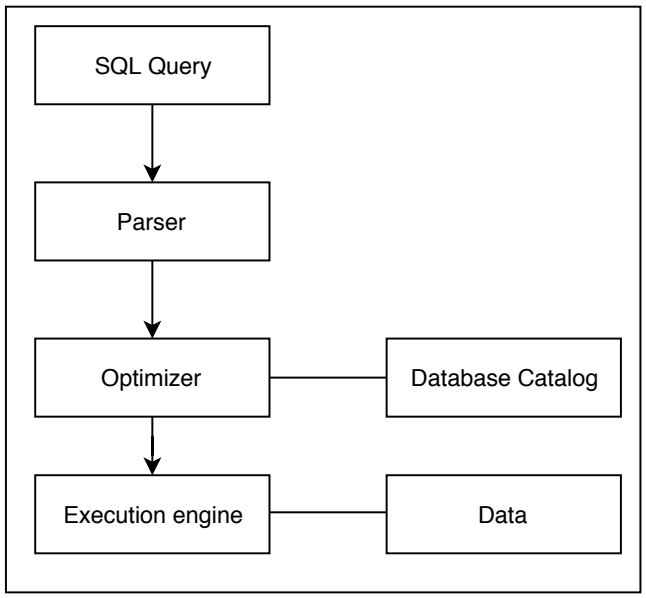


Figure 1. Stages of Query processing

Query optimization is part of query processing in many relational database management systems. It determines the most efficient way to execute a particular query by evaluating the various query plans for it. Once the query is submitted to the database server, it is then passed to the parser and then forwarded to the query optimizer as shown in Figure 1.

There are many ways to implement a particular query depending on the schema and the complexity of the query. Different databases use different query optimizer which might result in different execution time for the same query when executed on different platforms. The fundamental task of a query optimizer is to select an algorithm from among the many available options that provides the answer with a minimum of disk I/O and CPU time.

Query optimizer [2] frees the programmer from the task of selecting a particular query plan and allows the programmer to focus on high-level application issues. For simple query the choice of plan is mostly obvious but as schema and queries become important, the optimizer plays an inmportant role in simplifying the work of application development for the programmer.

The rest of the paper is organised as follows: Section 2 gives an overview of the architecture for PostgreSQL and SQLite, Section 3 defines the benchmarks in detail, in Section 4 we discuss the results. Finally, we discuss future work and conclude in Section 5 and Section 6 respectively.

2 Databases

2.1 PostgreSQL

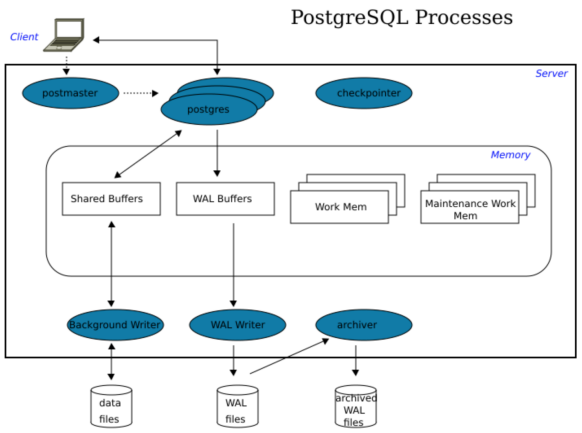


Figure 2. PostgreSQL Architecture

PostgreSQL [3] is a relational database management system following a client-server architecture. At the server side an instance is created comprising of PostgreSQL's processes and shared memory which handles the access to the data.

Client programs connect to the instance sending read and write operations. The overview of the architecture is shown in Figure 2.

An instance consists of multiple processes, postmaster process, multiple postgres processes (one for each connection), WAL writer process, background writer process, checkpointer process and other optional processes like autovacuum launcher process, logger process , archiver process, stats collector process, WAL sender process (when streaming replication is active), WAL receiver process (when streaming replication is active) and background worker processes (in case a query gets parallelized). The client program sends the request to the instance. The instance itself does not directly write to disk instead it buffers the requested data in shared buffer. The flushing to disk is done at a later stage.

2.2 SQLite

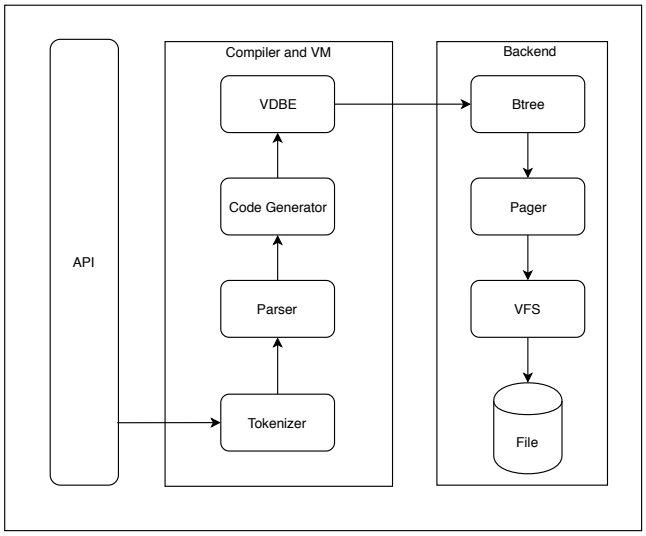


Figure 3. SQLite Architecture

SQLite [1] is an embedded file based relational database management system which can be linked statically or dynamically with the application program. It does not have a client-server database engine which makes the SQLite applications require less configuration. SQLite is called zero-conf as it does not require service management or access control based on password and GRANT.

SQLite complies with ACID (atomicity, consistency, isolation, durability) properties and implements the SQL standard. It stores the entire database as a single disk file and all reads and write takes place directly on this file.

SQLite database architecture as shown in Figure 3 is divided into two different sections called core and backend. Core contains the Interface, Tokenizer, Parser, Code generator, and the virtual database engine (VDBE), which create an execution order for database transactions. Backend contains B-tree, Pager and OS interface to access the file system. Tokenizer, Parser and code generator is together named as the compiler which generates a set of opcodes that runs on a virtual machine.

3 Benchmarks

3.1 TPC-H

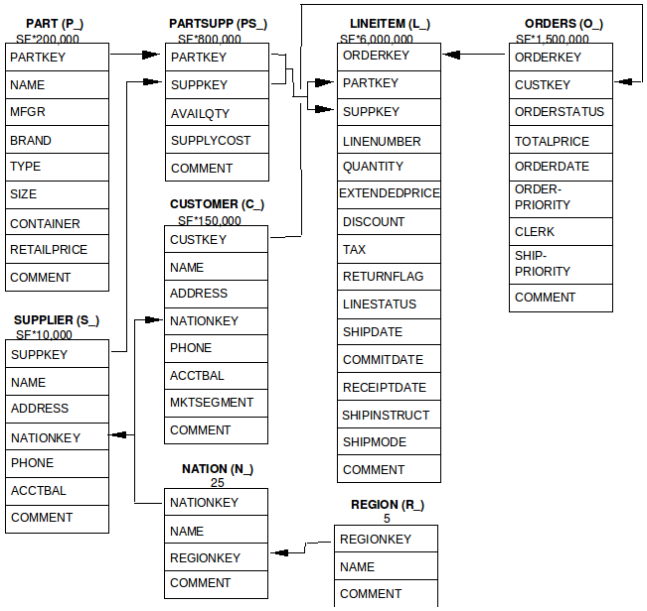


Figure 4. TPC-H schema

TPC-H [5] is a decision support benchmark that consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. It evaluates the performance of various decision support systems by the execution of sets of queries against a standard database under controlled conditions.

The purpose of this benchmark is to reduce the diversity of operations found in an information analysis application, also retaining the application's essential performance characteristics, i.e, the level of system utilization and the complexity of operations. A large number of queries of various types and complexities need to be executed to completely manage a business analysis environment.

The components of the TPC-H database consists of eight separate and individual tables known as the Base Tables. The relationships between these tables are illustrated in Figure 4.

3.2 SSB

The Star Schema Benchmark (SSB) [6] is designed to test star schema optimization to address the issues of TPC-H along with measuring performance of database products and test a new materialization strategy. The SSB is a simple benchmark that consists of four query flights, four dimensions,

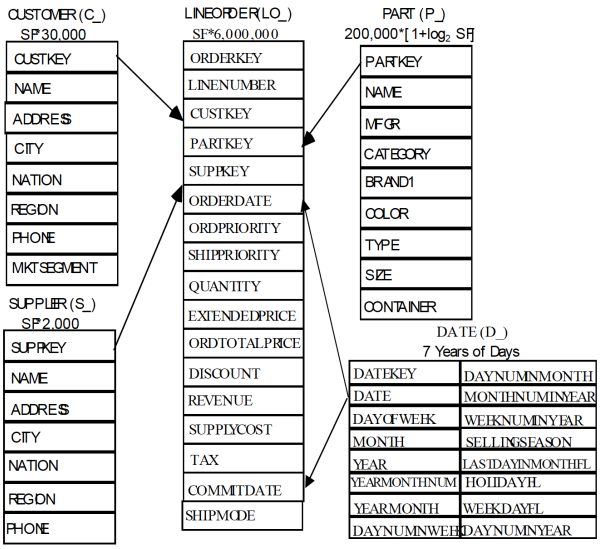


Figure 5. SSB schema

Table name	Size
Customer	24M
LineItem	720M
Nation	4K
Orders	169M
Part	23M
Partsupp	121M
Region	4K
Supplier	2.1M

Table 1. TBC-H tables size

Table name	Size
Customer	11M
Ddate	224K
Lineorder	2.3G
Part	49M
Supplier	648K

Table 2. SSB tables size

and a simple roll-up hierarchy. The SSB is largely based on the TPC-H benchmark with improvements implementing a traditional pure star-schema and allowing column and table compression.

The SSB is designed to measure the performance of database products against a traditional data warehouse scheme. It implements the same logical data in a traditional star schema whereas TPC-H models the data in pseudo 3NF schema.

Modifications to the TPC-H schema were made to transform it into a star schema form. The TPC-H tables LINEITEM and ORDERS are combined into one sales fact table named LINEORDER. The PARTSUPP table is dropped. The comment attributes for LINEITEMS, ORDERS, and shipping instructions are also dropped. A dimension table called DATE is added to the schema as is in line with a typical data warehouse. LINEORDER serves as the fact table. Dimension Tables are created for CUSTOMER, PART, SUPPLIER and DATE.

SSB concentrates on queries that select from the LINEORDER table exactly once. It avoids the use of self-joins or subqueries as well as or table queries also involving LINEORDER. The classic warehouse query selects from the table with restrictions on the dimension table attributes. SSB supports queries that appear in TPC-H. SSB consists of one large fact table (LINEORDER) and four dimensions tables (CUSTOMER, SUPPLIER, PART and DATE).

4 Results

4.1 Method

We run all our experiments on Cloudlab [5] xl170 machine, with a ten-core Intel E5-2640v4 running at 2.4 GHz, 64GB ECC Memory (4x 16 GB DDR4-2400 DIMMs), Intel DC S3520 480 GB 6G SATA SSD and 10Gbps NIC. We run on Ubuntu 18.04 (4.15.0-55-generic). We performed all the experiments on PostgreSQL version 10.10 and SQLite3 version 3.22.0. The buffer pool for each system is set to 2MB. The table size for each benchmarks are shown in Tables 1 and 2.

PostgreSQL is configured to run with zero parallelism to get a baseline comparison between the two systems since SQLite runs in a single thread. We also make sure that each query runs on a single core. To achieve zero paralleism the following configurations are set in PostgreSQL:

`max_worker_processes`: Maximum number of background processes that the system can support. We have set this parameter to 1.

`max_parallel_workers_per_gather`: Maximum number of workers that can be started by a single Gather or Gather Merge node. Gather node has one child plan which requires some number of background workers to execute the rest of the portion in parallel. We set this parameter to 0.

We first measure the execution times for both the benchmarks and then compare the query plans generated for few queries from TPC-H to understand the difference in execution time.

We also compare the results of TPC-H benchmark on PostgreSQL from previous setting with results obtained by turning parallelism on. In the new setting, `max_worker_processes` is set to 30 and `max_parallel_workers_per_gather` is set to 20.

We have changed the SSB and TPC-H suite to work on SQLite and have also automated the whole benchmarking process for both databases.

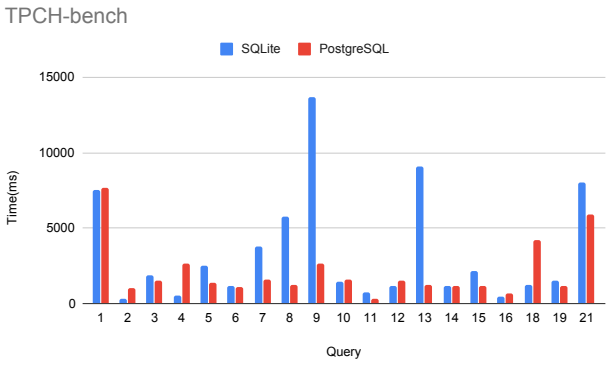


Figure 6. TPC-H result

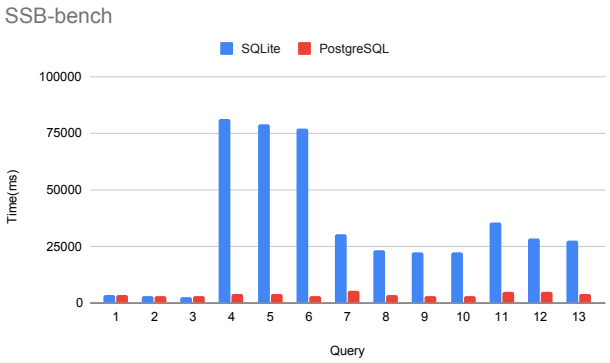


Figure 7. SSB result

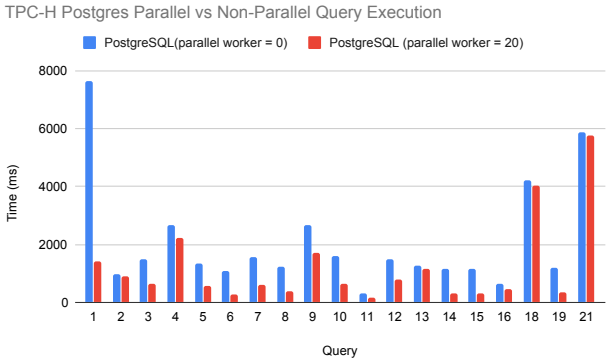


Figure 8. TPC-H result

4.2 Execution Time

The total execution time for TPC-H is shown in Figure 6. We observe that even though PostgreSQL is a much more complex system than the light-weight SQLite, it is not always faster. SQLite is competitive in some cases. The same behaviour is observed for SSB in Figure 7. These results give us an overall picture of the execution time of one system in comparison with the other.

Figure 8 compares the results of PostgreSQL with max_parallel_workers_per_gather set to 0 versus max_parallel_workers_per_gather = 20. The results show that the run times of almost all the queries decrease significantly upon changing this parameter. It was noted that the queries for which run times didn't change (Q2, Q13, Q18, Q21) were the ones that didn't use Gather node in their plans at all. This further strengthens our asssumption that parallelism affects PostgreSQL queries significantly and this parameter should be set to 0 for analysis with SQLite.

4.3 Query Plan Comparison

4.3.1 Almost same time

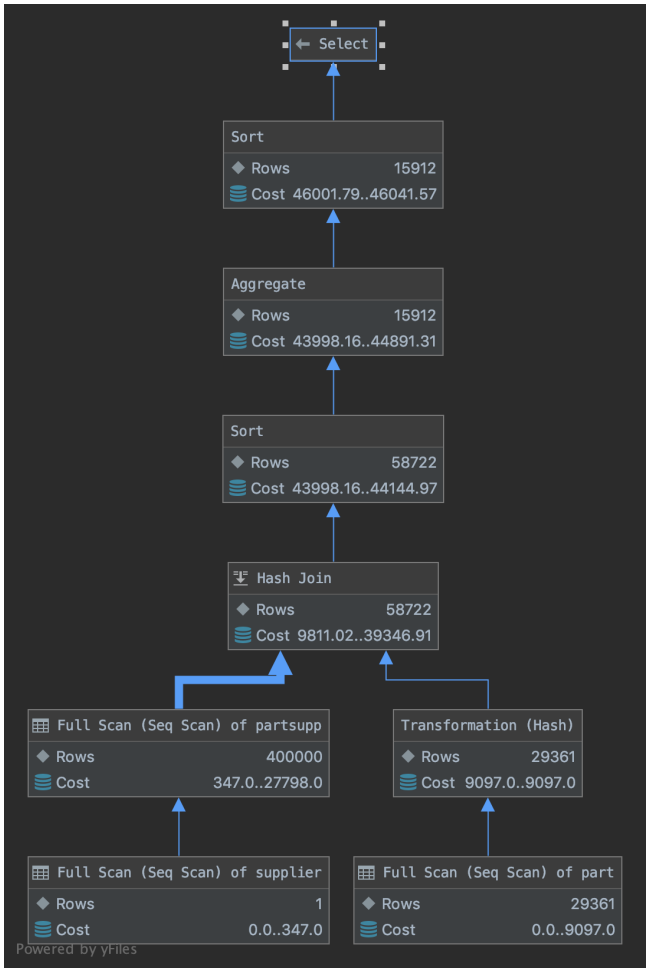
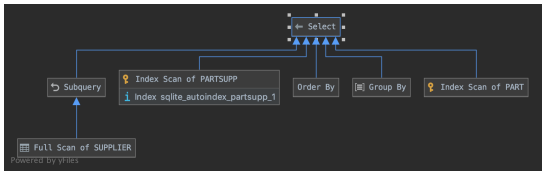


Figure 9. PostgreSQL plan for query 16

We now look at query 16 represented in Listing 1 from the TPC-H benchmark suite. This is called the Parts/Supplier Relationship Query that returns how many suppliers can supply parts with given attributes. This query has almost same time for both, PostgreSQL and SQLite.



(a) SQLite query plan 16 steps

```
QUERY PLAN
|--SCAN TABLE PARTSUPP USING COVERING INDEX sqlite_autoindex_PARTSUPP_1
|--LIST SUBQUERY
|  `--SCAN TABLE SUPPLIER
|--SEARCH TABLE PART USING INTEGER PRIMARY KEY (rowid=?)
|--USE TEMP B-TREE FOR GROUP BY
`--USE TEMP B-TREE FOR ORDER BY
```

(b) SQLite query plan 16

Figure 10. SQLite plan for query 16

Figure 9 shows the query plan generated for this query in PostgreSQL, we observe that it does a full scan of all the tables followed by some hash joins, sort, aggregate and then sort again. If we look at the query plan for SQLite in Figure 10, we observe that it also does a scan of all the tables and execute a similar query plan. Since the query plans generated by both are significantly similar and this query does not contain the big table, LINEITEM, therefore the scans take similar time even though SQLite does an index scan. All this adds up to almost same time for this particular query in both the databases.

```
select p_brand, p_type, p_size, count(distinct
    ps_suppkey) as supplier_cnt
from partsupp, part
where p_partkey = ps_partkey
and p_brand <> '[BRAND]'
and p_type not like '[TYPE]%'
and p_size in ([SIZE1], [SIZE2], [SIZE3], [SIZE4],
     [SIZE5], [SIZE6], [SIZE7], [SIZE8])
and ps_suppkey not in (
select s_suppkey
from supplier
where
s_comment like '%Customer%Complaints%'
)
group by p_brand, p_type, p_size
order by supplier_cnt desc, p_brand, p_type,
    p_size;;
```

Listing 1. Query 16

4.3.2 PostgreSQL faster

Query 9 shown in Listing 2 called the Product Type Profit Measure Query determines how much profit is made on a given line of parts, broken out by supplier nation and year. This query takes longer execution time SQLite.

Figure 11 shows the query plan for PostgreSQL where we observe that it does full scan of all the tables followed by hash join and aggregate at the end. In SQLite shown in Figure 12, we see many index scans. We also see that this query has LINEITEM which is a big table and hence, selectivity for this particular query is very high. Therefore, index scan in this scan becomes expensive over full scan which is seen by the lower execution time for PostgreSQL.

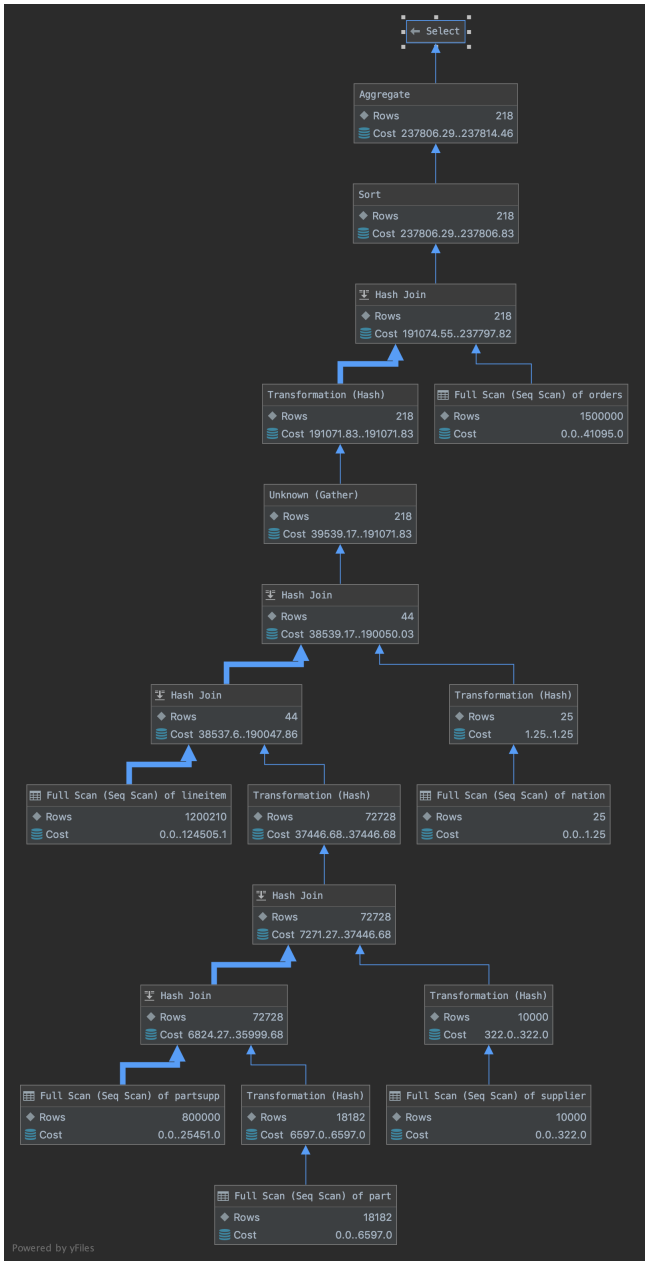


Figure 11. PostgreSQL plan for query 9

(a) SQLite query plan 9 steps

```
QUERY PLAN
|--SCAN TABLE LINEITEM
|--SEARCH TABLE PART USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE SUPPLIER USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE PARTSUPP USING INDEX sqlite_autoindex_PARTSUPP_1 (PS_PARTKEY=? AND PS_SUPPKEY=?)
|--SEARCH TABLE ORDERS USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE NATION USING INTEGER PRIMARY KEY (rowid=?)
|--USE TEMP B-TREE FOR GROUP BY
`--USE TEMP B-TREE FOR ORDER BY
```

(b) SQLite query plan 9

Figure 12. SQLite plan for query 9

```
select nation, o_year, sum(amount) as sum_profit
from ( select
n_name as nation,
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost
     * l_quantity as amount
from 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
and p_name like '%[COLOR]%'
) as profit
group by nation, o_year
order by nation, o_year desc;
```

Listing 2. Query 9

4.3.3 SQLite faster

Query 4 shown in Listing 3 is called Order Priority Checking Query which finds how well the order priority system is working and gives an assessment of customer satisfaction. This query runs faster in SQLite.

The query plan for PostgreSQL is shown in Figure 13 and SQLite is shown in Figure 14. This query has an inner query which is called a correlated subquery [4] which depends on the outer query. The selectivity of this query is around 3%. In case of PostgreSQL we observe that it does a full scan of all the tables whereas SQLite uses an index scan. In queries where selectivity is low, index scan is faster, therefore, we see that SQLite performs better than PostgreSQL. Since the table in question is Lineitem table which occupies 750 MB on disk out of 1GB of the entire TPC-H database, the effects seem more significant in this query than the others.

```
select o_orderpriority, count(*) as order_count
from orders
where o_orderdate >= date '[DATE]'
and o_orderdate < date '[DATE]' + interval '3'
    month
and exists (
select * from lineitem where l_orderkey =
    o_orderkey and l_commitdate < l_receiptdate
) group by o_orderpriority, order by,
    o_orderpriority;
```

Listing 3. Query 4

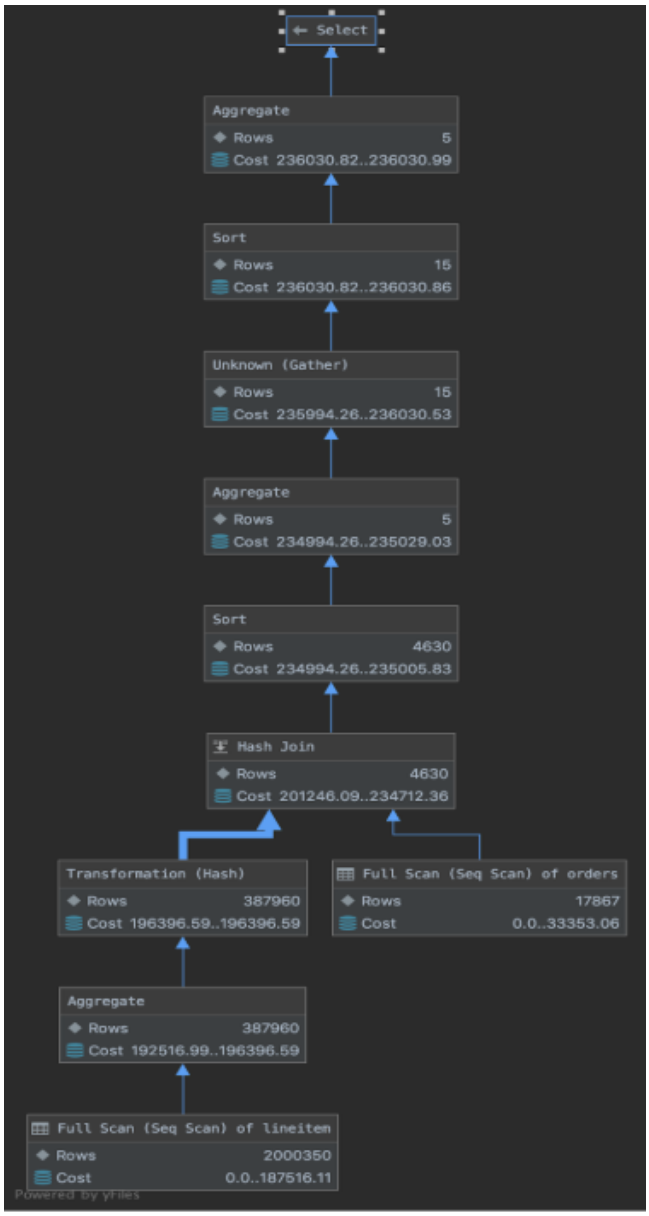
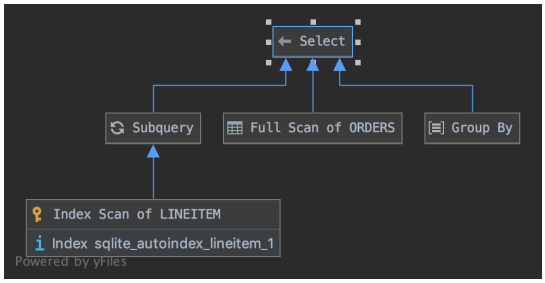


Figure 13. PostgreSQL plan for query 4

5 Future Work

In this project we looked at two benchmarks. It can be extended to more benchmarks which are more closely suited for application where SQLite is used over PostgreSQL. Databases



(a) SQLite query plan 4 steps

(b) SQLite query plan 4

Figure 14. SQLite plan for query 4

with bigger size can also be tested, in this project we have used DB sizes of around 1.17GB for TPC-H and 2.5GB for SSB. The project can also extended to create a standalone service that takes SQLite queries and outputs a generic optimised physical plan in a DAG form for other services like Hustle to ingest and execute.

Also, the query plan generated by SQLite is not as thorough and detailed as PostgreSQL, so it was very difficult to make concrete conclusions in some cases. One can also think about implementing this part so that SQLite outputs better and more detailed logical plan that helps in further analysis.

6 Conclusions

In the preceding sections we compared the performance of SQLite and PostgreSQL on two benchmarks, TPC-H and SSB. We observed that SQLite was slow in most of the cases but it did perform almost as same and even better in some cases. We can attribute this performance difference between the two databases to the difference in the architecture and the use-case for each in different scenarios.

PostgreSQL follows the client/server architecture whereas SQLite is file based (embedded). Although PostgreSQL has more complex architecture but it performs efficiently in most cases. Also, the benchmarks we used TPC-H and SSB are for decision support for business and data warehouse respectively which are more suited to the use-case for PostgreSQL, which also explains its better performance in these benchmarks. SQLite [7] is light-weight and suited for applications like IoT, embedded devices, low-medium traffic websites, etc.

When we look at the difference in their query plans we can see that the index scan that is used by SQLite in most cases becomes expensive as the selectivity increases. Also, the current implementation of SQLite uses only loop joins, i.e, joins are implemented as nested loops which also add to this overhead most of the time.

PostgreSQL does a sequential full scans followed by hash-joins in most cases. In comparision to nested loop joins, hash-joins are better which is evident from the better performance of PostgreSQL in most cases. It only performs worse in cases where the selectivity is low.

References

[1] Internal architecture of the sqlite database. https://dzone.com/articles/the-internal-architecture-of-the-sqlite-database.

[2] The next-generation query planner. https://www.sqlite.org/queryplanner-ng.html.

[3] Postgresql/architecture. https://en.wikibooks.org/wiki/PostgreSQL/Architecture.

[4] The sqlite query optimizer overview. https://www.sqlite.org/draft/optoverview.html.

[5] Tpc-h. http://www.tpc.org/tpch/.

[6] J. Sanchez. A review of star schema benchmark. *CoRR*, abs/1606.00295, 2016.

[7] A. Yigal. Sqlite vs. mysql vs. postgresql: A comparison of relational databases, 2018. https://dzone.com/articles/sqlite-vs-mysql-vs-postgresql-a-comparison-of-rela.