A Study on Query Optimization

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

Abstract

Query optimization is the part of the query process in which the database system compares different query strategies and chooses the one with the least expected cost. The query optimizer, which carries out this function, is a key part of the relational database and determines the most efficient way to access data. It makes it possible for the user to request the data without specifying how these data should be retrieved.

The cost of accessing a query is a weighted combination of the I/O and processing costs. The I/O cost is the cost of accessing index and data pages from disk. Processing cost is estimated by assigning an instruction count to each step in computing the result of the query.

In this project, we study the optimization done by PostgreSQL and SQLite by running the queries present in the SSB and TPCH benchmarks. The results obtained help us to further investigate the query optimization techniques implemented by the two database systems.

1 Introduction

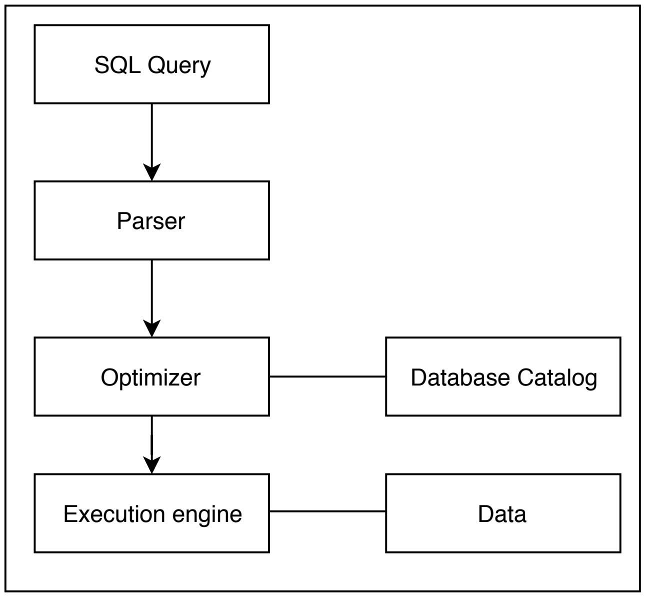


Figure 1. Stages of Query processing)

Query optimization is part 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 passed 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. 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 overhead.

2 Databases

2.1 Postgresql

PostgreSQL is a relational database management system following a client-server architecture. At the server side the PostgreSQL's processes and shared memory work together and build an instance, which handles the access to the data. Client programs connect to the instance and request read and write operations.

2.2 SQLite

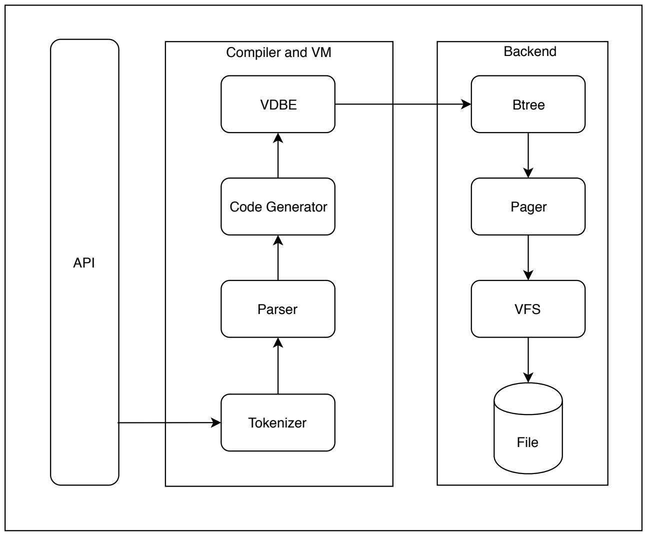


Figure 2. SQLite Architecture)

SQLite is a relational database management system embedded in a C library 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 because it does not require service management or access control based on password and GRANT.

SQLite is ACID (atomicity, consistency, isolation, durability) compliant and implements the SQL standard. It stores the entire database as a single cross platform file on a host machine.

SQLite database architecture split into two different sections named as core and backend. Core section contains Interface, Tokenizer, Parser, Code generator, and the virtual machine, 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 altogether named as the compiler which generates a set of opcodes that runs on a virtual machine.

3 Benchmarks

3.1 TPCH

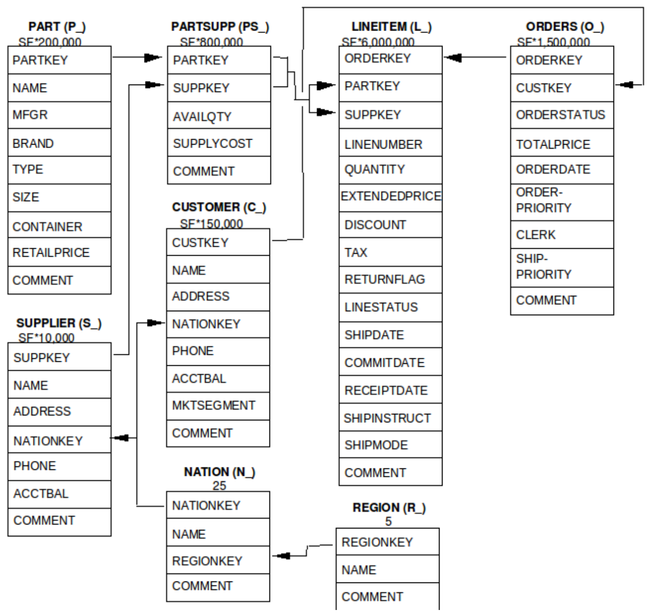


Figure 3. TPCH schema

TPCH is a decision support benchmark that consists of mixed queries. These queries are

The TPC Benchmark (TPC-H) is a decision support benchmark. It 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, while retaining the application's essential performance characteristics, namely: the level of system utilization and the complexity of operations. A large number of queries of various types and complexities needs to be executed to completely manage a business analysis environment.

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 3.

3.2 SSB

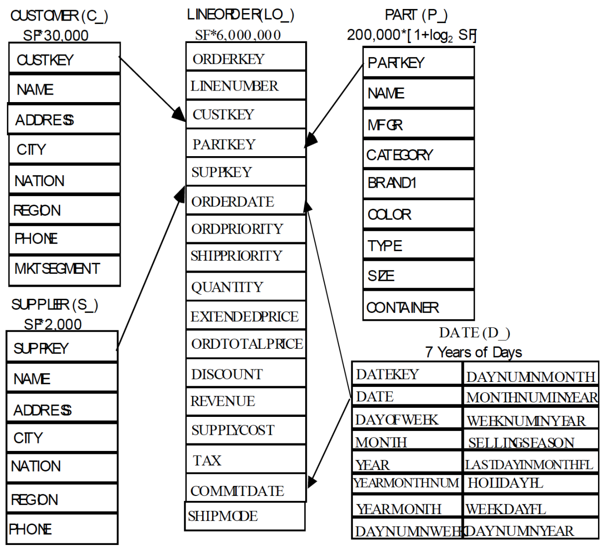


Figure 4. SSB schema

The Star Schema Benchmark (SSB) was designed to test star schema optimization to address the issues outlined in TPC-H with the goal of measuring performance of database products and to test a new materialization strategy. The SSB is a simple benchmark that consists of four query flights, four dimensions, and a simple roll-up hierarchy . The SSB is significantly based on the TPC-H benchmark with improvements that implements a traditional pure star-schema and allows 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.

Schema modifications were made to the TPC-H schema 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 as a data warehouse does not store such information in a fact table, they can't be aggregated and take significant storage space. A dimension table called DATE is added to the schema as is in line with a typical data warehouse. LINEORDER serves as a central fact table. Dimension Tables are created for CUSTOMER, PART, SUPPLIER and DATE.

SSBM 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. SSBM 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 and SQLite3. Postgres was configured to run with a single thread by changing `max_parallel_workers` = 1, `max_worker_processes` = 1 and `max_parallel _workers_per_gather` = 0. The buffer pool for each system was set to 20MB.

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

4.2 Execution Time

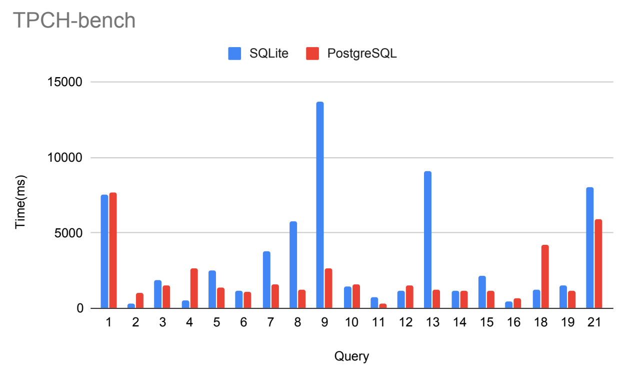


Figure 5. TPCH result

The total execution time for TPCH is shown in Figure 5. We observe that postgres is not always faster, sqlite is competitive in some cases. The same behaviour is observed for SSB in Figure 6. These results give us an overall picture of the execution time of one system in comparison with the other. .

4.3 Query Plan Comparison

4.3.1 Almost same time

We now look at query 16 represented in Listing 1 from the TPCH benchmark suite. In this query is the Parts/Supplier Relationship Query that returns how many suppliers can supply parts with given attributes. This query has almost same time for both, postgres and sqlite.

Figure 7 shows the query plan generated for this query in postgres, we observe that it does a full scan of all the tables followed by some hash joins, sort, aggregate and then sort

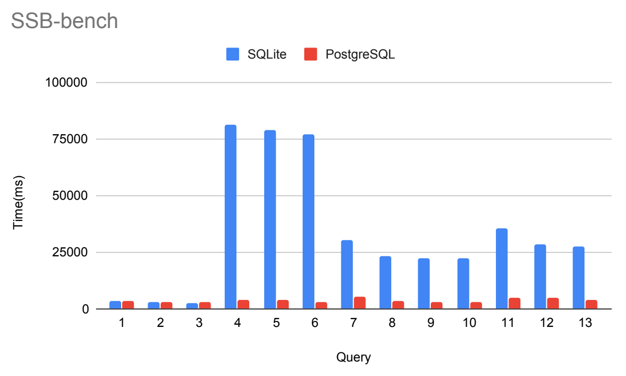


Figure 6. SSB result

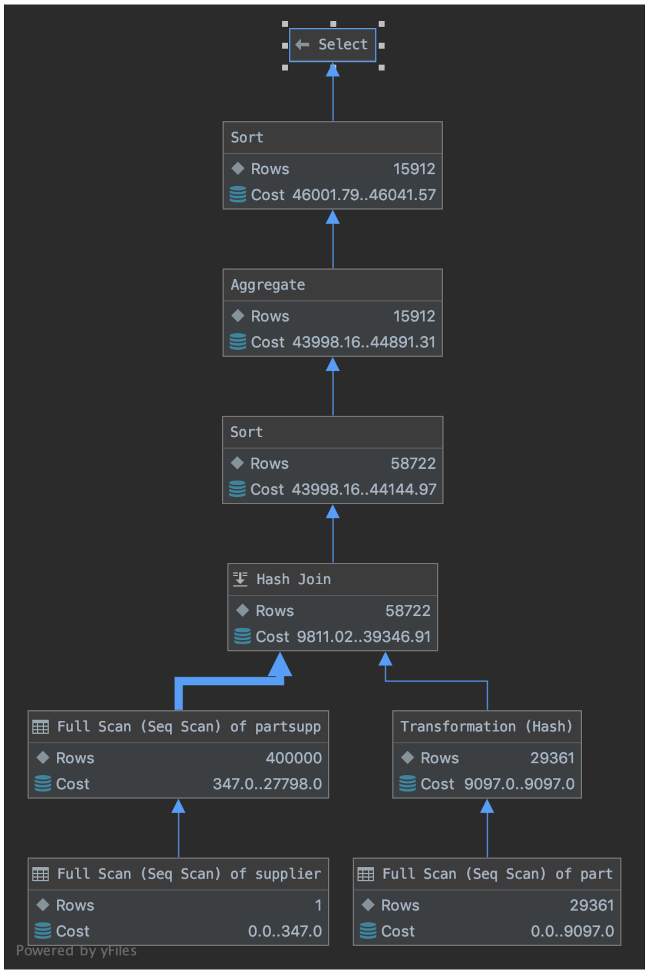
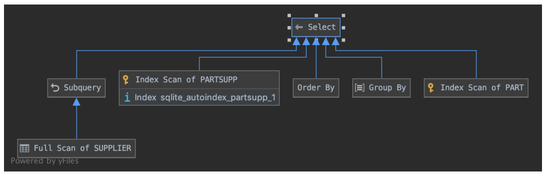
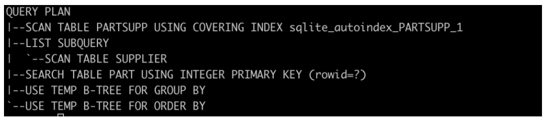


Figure 7. Postgres plan for query 16



(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 8. SQLite plan for query 16

again. If we look at the query plan for SQLite in Figure 8, 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 add 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 Postgres faster

Query 9 shown in Listing 2 called Product Type Profit Measure Query which 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 9 shows the query plan for Postgres 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 10, we see many index scans. We also see that this query has LINEITEM which is a big table and hence, selectively 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 Postgres.

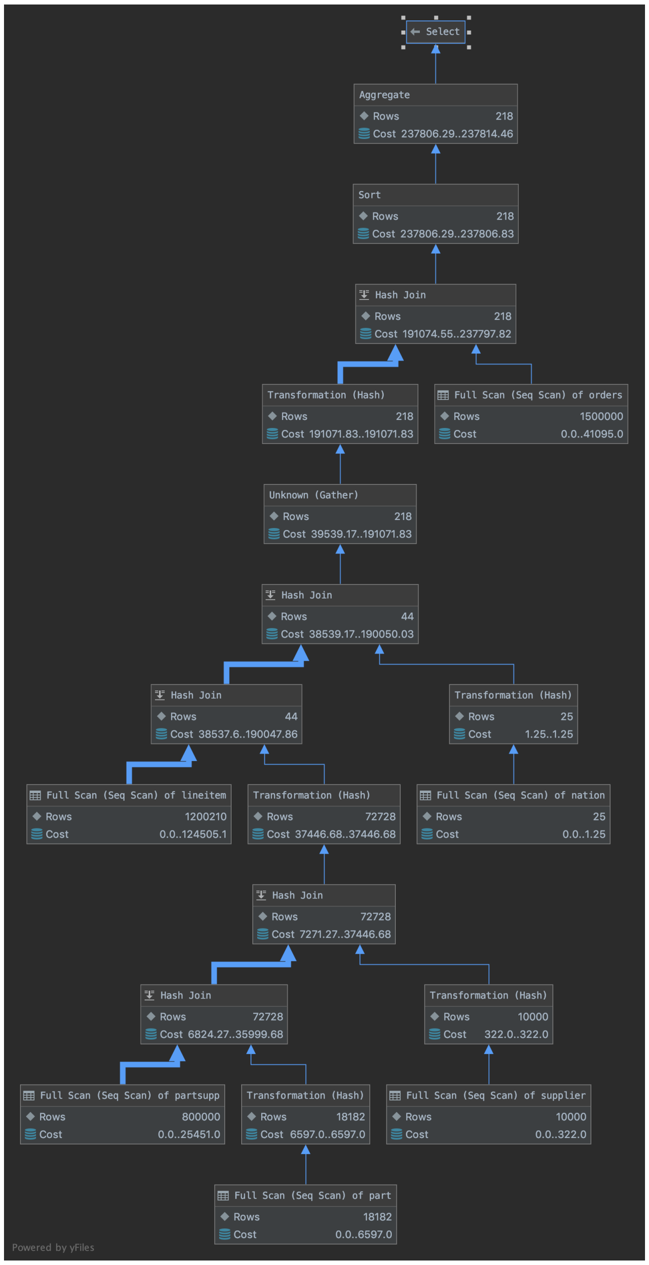
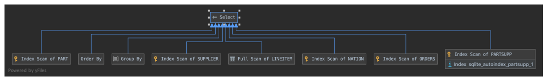
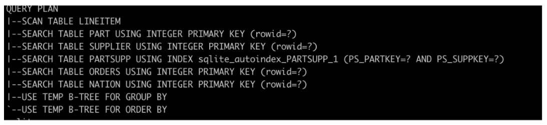


Figure 9. Postgres plan for query 9



(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 10. 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 Postgres is shown in Figure 11 and SQLite is shown in Figure 12. This query has an inner query which is called a correlated subquery [1] which depends on the outer query. The selectively of this query is less that 5%. In case of Postgres we observe that it does a full scan of all the tables whereas SQLite uses an index scan. In queries where selectively is low, index scan is faster, therefore, we see that SQLite performs better than Postgres.

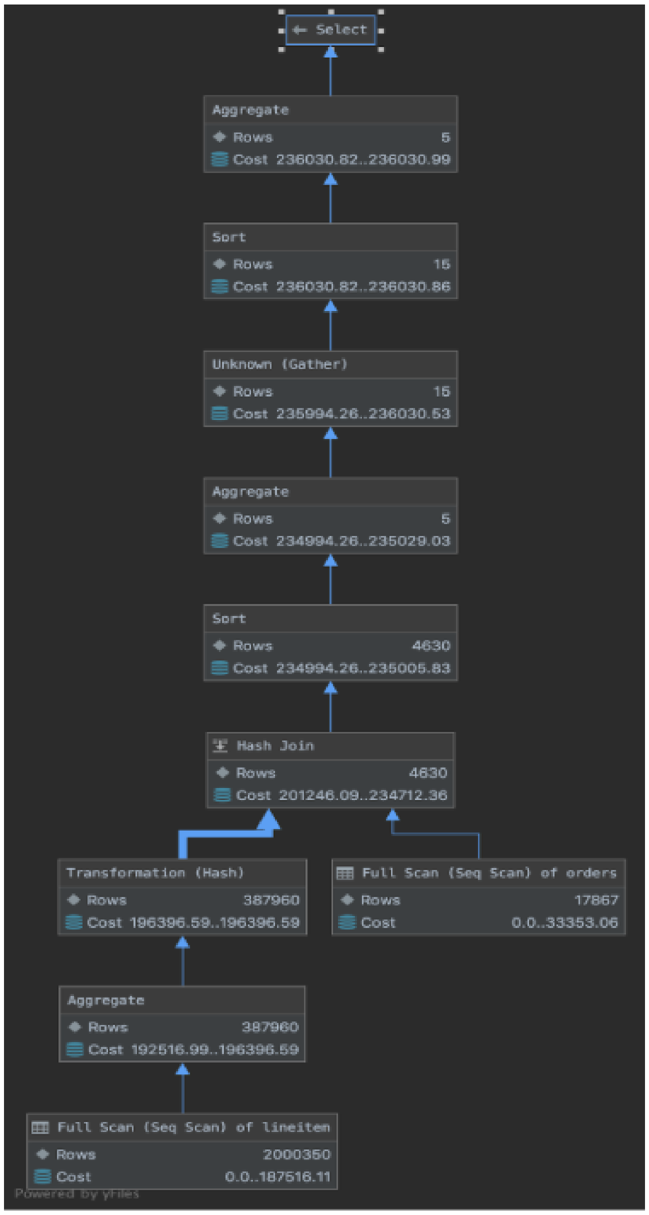
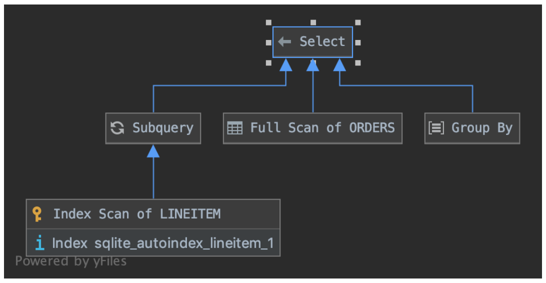
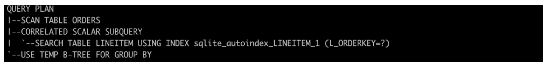

Select
Aggregate
Rows 5
Cost 236030.82..236030.99
Sort
Rows 15
Cost 236030.82..236030.86
Unknown (Gather)
Rows 15
Cost 235994.26..236030.53
Aggregate
Rows 5
Cost 234994.26..235029.03
Sort
Rows 4630
Cost 234994.26..235005.83
Hash Join
Rows 4630
Cost 201246.09..234712.36
Transformation (Hash)
Rows 387960
Cost 196396.59..196396.59
Full Scan (Seq Scan) of orders
Rows 17867
Cost 0.0..33353.06
Aggregate
Rows 387960
Cost 192516.99..196396.59
Full Scan (Seq Scan) of lineitem
Rows 2000350
Cost 0.0..187516.11
Powered by yFiles

Figure 11. Postgres plan for query 4

```
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,
```



(a) SQLite query plan 4 steps



```
QUERY PLAN
|--SCAN TABLE ORDERS
|--CORRELATED SCALAR SUBQUERY
|  `--SEARCH TABLE LINEITEM USING INDEX sqlite_autoindex_LINEITEM_1 (L_ORDERKEY=?)
`--USE TEMP B-TREE FOR GROUP BY
```

(b) SQLite query plan 4

Figure 12. SQLite plan for query 4

5 Future Work

[TBA]

6 Conclusions

[TBA]

References

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