DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Query Processing Benchmark (SSB) Implementation and Performance Analysis

Author: Xuedong Jin, xuedong.jin@epfl.ch
Supervisor: Prof. Anastasia Ailamaki, anastasia.ailamaki@epfl.ch
Responsible collaborator(s): Manos Athanassoulis, manos.athanassoulis@epfl.ch
Duration: 12 weeks (September 21, 2011 - January 14, 2012)

Keywords: Query Processing, Performance Benchmarks for DBMS, system profiling

1. Introduction

The multi-core trend and the recent popularization of Flash storage change the hardware that traditional Query-Processing applications assume. We employ a staged architecture which exploits the available parallelism (and we implement QPipe over an efficiently multi-threaded storage engine like Shore-MT). Existing query processing benchmarks like TPCH and SSB will be implemented and tested over our prototype.

This project aims at completing the implementation of the SSB benchmark over our prototype storage manager (Shore-MT). The student will spend the first one-two week(s) familiarizing her/him-self with the system. The integration of the benchmark is already in place and the student will be guided through concrete implementation steps in order to implement the benchmark's queries and measure and profile their performance.

During this project, I first manage to get familiar with the API used to integrate the benchmark. At the meantime, I generate the plain data file and loaded into SQL Server 2008, then run the SSB queries on SQL Server 2008 in order to get the query plans. These query plans are optimized based on the optimizer's analysis of the SSB Schema, the size of the corresponding table files, as well as the dimensional coverage of the queries. With these query plans, I implement the SSB queries on the experimental QPipe environment over the multi-threaded storage engine (Shore-MT). Finally, I make a basic experimentation, measuring the response of each query of the SSB benchmark and comparing it against the open-source DBMS PostgreSQL.

2. QPipe

Relational DBMS execute the concurrent queries independently, by allocating a set of operator instances for each of the queries[1]. Within the query engine, the same-operator is expected to execute among concurrent queries accordingly, exploiting common accesses to the memory and disks and common intermediate results, so as to maximize the data and work sharing across concurrent queries at execution time.

There are many limitations in run-time sharing of modern execution engines, such as paradigms of invoking an independent set of operator instances per query, potentially missing sharing opportunities when the caches and buffer evict data pages early, etc.[1]. Furthermore, the degree of sharing the buffer pool is sensitive to the time of other queries which have the same type of operation or potential data to be shared.

For the traditional database systems, there are three main stages for an engine to check the reusability of data and work sharing in the processing of the query:

- When the query is submitted, the system will firstly look up the cache of recent completed queries to get the matching data executed by other queries which are reusable.
- Within the execution engine, the new query could also reuse the previously computed intermediate results in condition that there are matching materialized views.
- For the operator, it will look up the buffer pool to fetch the tuple which could be reused.

However, there are also great limitation in this mechanism for data and work sharing, for example, the missed opportunity of not examining concurrent queries for potential data and work overlapping.

QPipe is a new operator-centric relational engine that supports on-demand simultaneous pipelining (OSP).It would enable proactive, dynamic operator sharing by pipelining the operator's output simultaneously to multiple parent nodes. In this model, each operator is executed by being encapsulated into a micro-engine to serve the query tasks in the queue, with the maximization of data and work sharing in the execution periods. The performance of QPipe is up to 2 times speedup over DBMS X. This is heavily due to the fact that QPipe can proactively share the disk pages for all the concurrent executed queries. The ability of on-demand simultaneous pipelining (OSP) is fully explored by QPipe for all operators to pipeline data from a single query node to multiple parent nodes simultaneously.

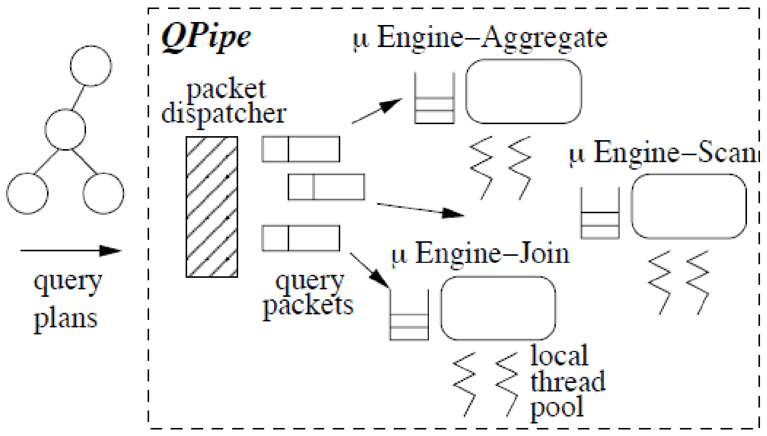

Figure 1. QPipe micro-engine model [3]

In the QPipe model, a monitor is implemented in each relational operator for all the active queries so as to detect the overlap in time. If the monitor has detected the overlap, the operator engine will pipeline the result to all participating nodes simultaneously.

QPipe is developed based on the principles of the *Staged Database System* design[2], following the "one-operator, many-queries" design philosophy. In this model, as shown in Figure 1, there are several micro-engines managing a set of threads in order to serve the queuing queries. This would ensure the detection and maximization of overlapping operation.

3. Star Schema Benchmark

Star Schema Benchmark (SSB) is designed to measure the database system performance of star schema data warehouse queries. It is based on the TPC-H benchmark [TPC-H] with major modifications. The columns in the SSB tables are designed with the capability of being compressed by any approach in the database system. Compared with TPC-H, SSB has done some of the schema changes in order to get the efficient star schema. The SSB is very important to be used to measure a number of major commercial database product on Linux.

Figure 2 shows the schema layout of TPC-H benchmark, and Figure 3 shows the schema layout of SSB benchmark, from which we can see the main changed of SSB from TPC-H are [4]:

- Create SSB LINEORDER Table
- Drop PARTSUPP Table
- Drop Some TPC-H columns of LINEITEM and ORDER and Add Some to LINEORDER
- Drop Tables NATION and REGION
- Further Changes Resulting from Grain Mismatches
- Dropping, Adding, and Changing Columns
- Adding Date Dimension

The detailed changes are further explained in [4]. After committing these changes, SSB is created with the table LINEORDER in the middle linked with other dimension tables as CUSTOMER, PART, SUPPLIER, and DATE.

The SSB queries are also modified based on the TPC-H queries, which are designed concentrating on queries that select from the LINEORDER table exactly once without self-joins or sub-queries or table queries involving LINEORDER. The principles of the design of SSB queries are *Functional Coverage* and *Selectivity Coverage*[5], further explained in [SETQ]. With these two principles, the performance of the Star Schema warehouse database will be maximally explored by the SSB queries. Some of the SSB queries are modified from TPC-H queries, while others have no counterparts in TPC-H as they are only needed in SSB.

There are four main categories of SSB queries, and for each category, there are 3 to 4 queries which are slightly different with each other. Q1 have restriction on only one dimension while Q2 on two dimensions. Q3 and Q4 are implemented to place selectivity restrictions on three dimensions and four dimensions correspondingly.

DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

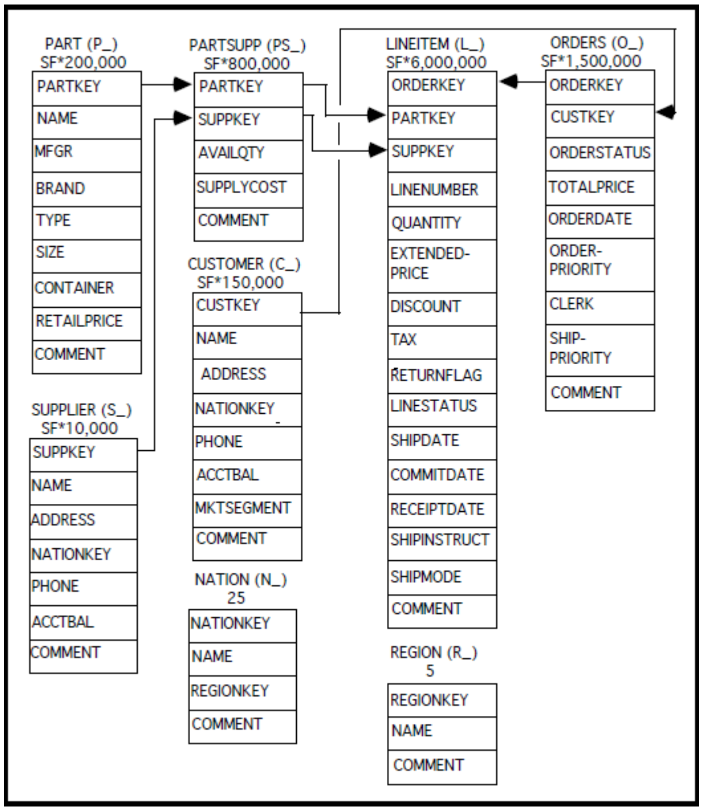

Figure 2. TPC-H Benchmark Schema [4]

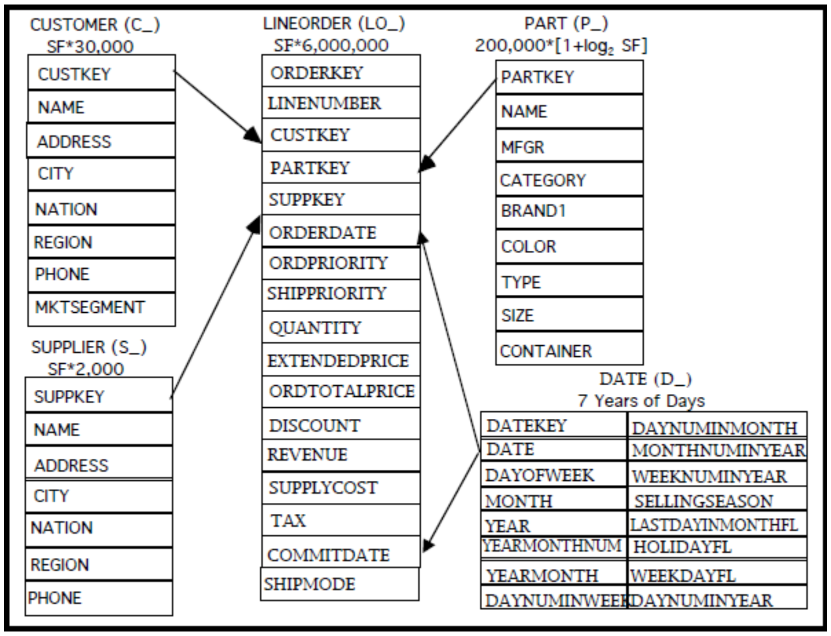

Figure 3. Star Schema Benchmark (SSB) Schema [4]

4. Implementation of Query Processing Benchmark (SSB)

The query processing benchmark is consist of four categories of queries, each of them have three to four similar queries with slightly different constrains and parameters. The purpose of designing and implementing these different queries is to maximize the functional coverage and selectivity coverage of the benchmark. This can help us fully explore the performance of the database system under-tested.

QPipe is a staged relational query engine, which drives us to implement all the benchmark queries by splitting them into sub-request. This will ensure the system with the capability of processing a group of sub-request at each stage, consequently maximize the data and work sharing across concurrent queries[6].

As QPipe is designed with "one-query, many-operators" model, there are several micro-engines implemented for different staged functions, such as µEngine-Scan, µEngine-Join, µEngine-Aggregate, µEngine-Sort as shown in Figure 1. In our case, we will split the queries into different stages, and after implementing them one by one, we finally integrate them together with the whole functionality according to SSB queries.

We choose to implement the benchmark queries from the easiest ones, namely Q1. The SSB queries is shown in the Appendix B, from which we can see that, for query Q1_1, Q1_2, Q1_3, we only use two tables (LineOrder and Date), with different constrains of lo_discount, lo_quantity, d_year, d_weeknuminyear, d_yearmonthnum correspondingly, and with the same joining condition lo_orderdatekey = d_datekey.

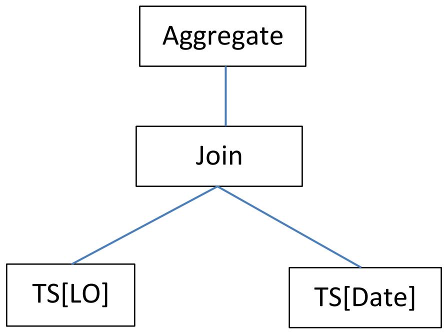

Figure 4. Query plan of Q1

After analyzing the structure of Q1, we start to implement them with the following stages, according to the query plan of Q1 shown in Figure 4 (here we take Q1_1 as example):

- Stage of T-Scan: in this stage, we scan all the tuples of table [lineorder] and [date], and project the columns of lo_extendedprice, lo_orderdate, lo_discount and d_datekey correspondingly. During the T-Scan of table [lineorder], we need to discard all those tuples which do not meet the requirement of "lo_discount between 1 and 3"; and for the T-Scan of table [date], we need to discard all those tuples which do not meet the requirement of "d_year = 1993".

- Stage of Joining scanned table [lineorder] and [date]: in this stage, we join the already scanned and projected table [lineorder] and [date] with the condition of "lo_orderdatekey = d_datekey". And project the joined result into a new table called q11_join_tuple with two elements "lo_extendedprice" and "lo_discount".
- Stage of Aggregation: in this stage, the table [q11_join_tuple] is processed with the requirement of "lo_extendedprice*lo_discount" and project the result as revenue.

During the implementation, in order to get the right result of the queries, we have to rule out the unexpected bugs by running and testing each stage one by one, and at last test them as an integrated query. In each stage, we will trace the result of output to check the reasonability of the code. In this way, we can efficiently fix the bugs and get the expected query results.

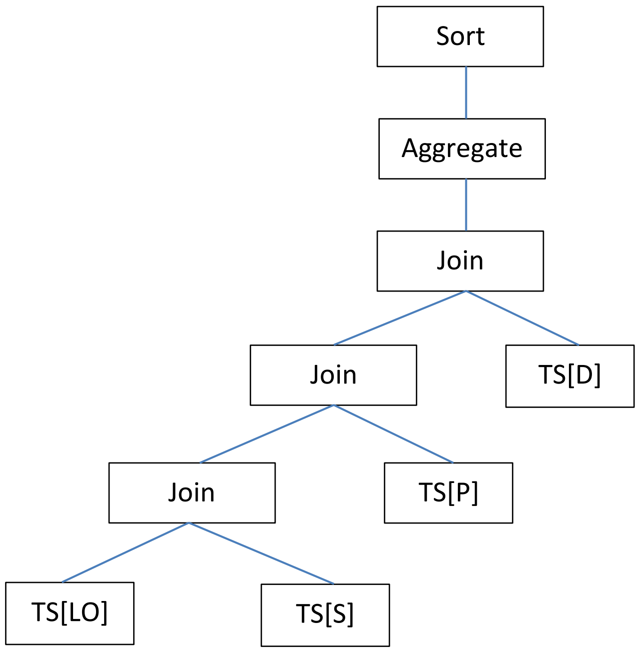

Figure 5. Query plan of Q2

Q2 is more complex compared with Q1, not only because they are using two more tables, but also that the condition is enriched with fixed strings. We implement them with the following stages, according to the query plan of Q2 shown in Figure 5 (here we take Q2_1 as example):

- Stage of T-Scan: in this stage, we scan four tables [lineorder]. [supplier], [part], [date] and project the corresponding columns according to the queries, with the same mechanism mentioned above in Q1.
- Stage of Joining: in this stage, we need to join the table [lineorder] and [supplier] first and project an intermediate table which would be used to join the scanned table [part] later on. And this result will be used to join table [date] afterwards. All the joining are committed under the

requirement of joining in the query, such as "lo_orderdatekey = d_datekey, lo_partkey = p_partkey, lo_suppkey = s_suppkey".

- Stage of Aggregate: another difference between Q2 and Q1 is that Q2 need to be sorted with "group by" and "order by" condition. We choose to use partial aggregate in this stage not only to finish the aggregate mission but also the "group by" mission.
- Stage of Sort: in this stage, we will order the table by d_year and p_brand.

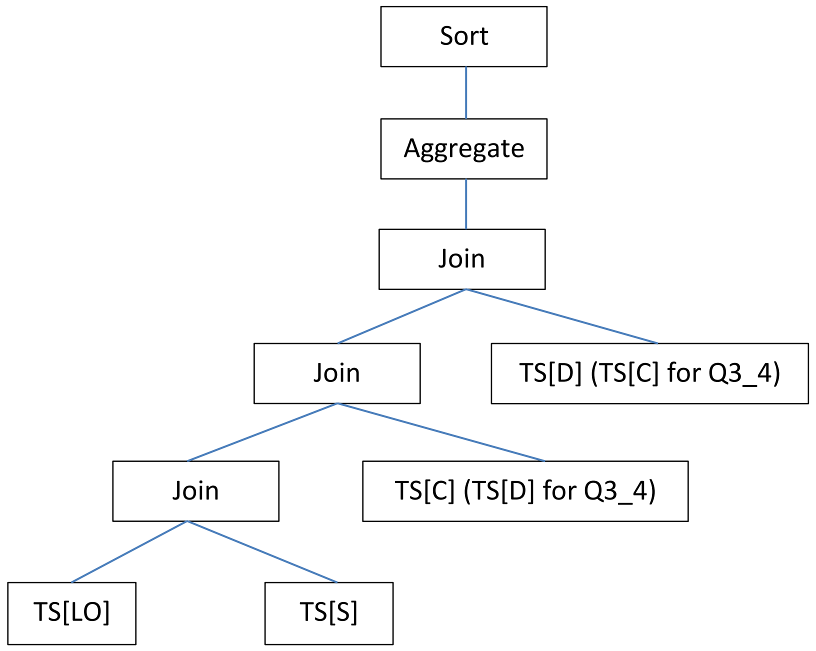

Figure 6. Query plan of Q3

Q3 have the similar complexity with Q2, the main difference is the "order by" round. In Q3, the tuples should be ordered ascending with d_year firstly and then ordered descending with revenue. So we have to use different key_extractor in this case.

Q4 are the most complex SSB queries compared with others, and the query of Q4 is shown in Figure 7. Q4 has five T-Scans, four Joining, one Aggregation and one Sorting process. Each stage is doing the similar working as mentioned in previous queries. The only difference is that the amount of process to be executed is larger.

There are several technical difficulties I have met when implementing SSB benchmark. The first one is that when I try to get the query plan from SQL Server 2008, I met the errors of loading data into the server. It takes me some time to figure out it was because that the data file is needed to be changed into the format which is compatible with SQL Server system. And also, the data generated by dbgen is also not always perfect to be used. There are duplicated tuples in lineorder.tbl, which should be fixed in order to get data properly loaded into the server.

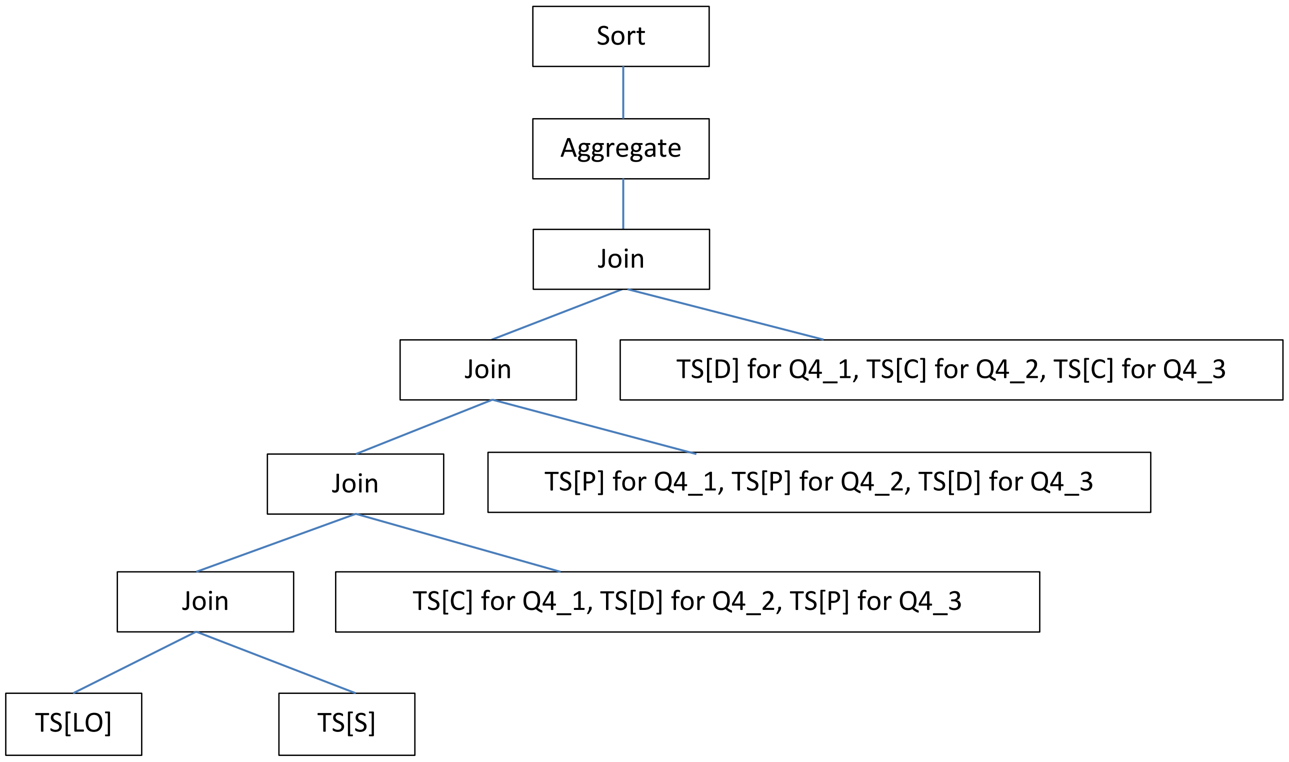
Sort
Aggregate
Join
Join
TS[D] for Q4_1, TS[C] for Q4_2, TS[C] for Q4_3
Join
TS[P] for Q4_1, TS[P] for Q4_2, TS[D] for Q4_3
Join
TS[C] for Q4_1, TS[D] for Q4_2, TS[P] for Q4_3
TS[LO]
TS[S]

Figure 7. Query plan of Q4

After implementing the queries, I also met the some problems when I was doing the test with the plain data file. After testing and debugging, I found that whenever did I make the projection for the tables, I should use “STRSIZE()” to control the size of the projected strings. Otherwise, the program would be messed up with memory fault.

5. Query Performance

In order to get the performance of the SSB benchmark queries, we should get the execution time of SSB queries running on QPipe supported Shore-SM system, compared with the performance of the same queries running on PostgreSQL as the reference system. From Figure 8 (with y-axis as performance time, second as unit) we can see theperformance of SSB queries running on both QPipe and PostgreSQL system.

For Q2_3, Q3_4, and Q4_3, they are clearly taking less time than others within the same category correspondingly. The reason is that Q2_3 has the condition “p_brand = ‘MFGR#2239’”, Q3_4 has the condition “d_yearmonth = ‘Dec1997’” and Q4_3 has the condition “d_year = 1997 or d_year = 1998, and p_category = ‘MFGR#14’, and s_nation = ‘UNITED STATES’”, which would greately shrink the amount of tuple to be processed for these three queries when they are committing the joining executions after T-Scans. Except these three queries, the order of average execute time is Q4>Q3>Q2>Q1 because of the query complexity.

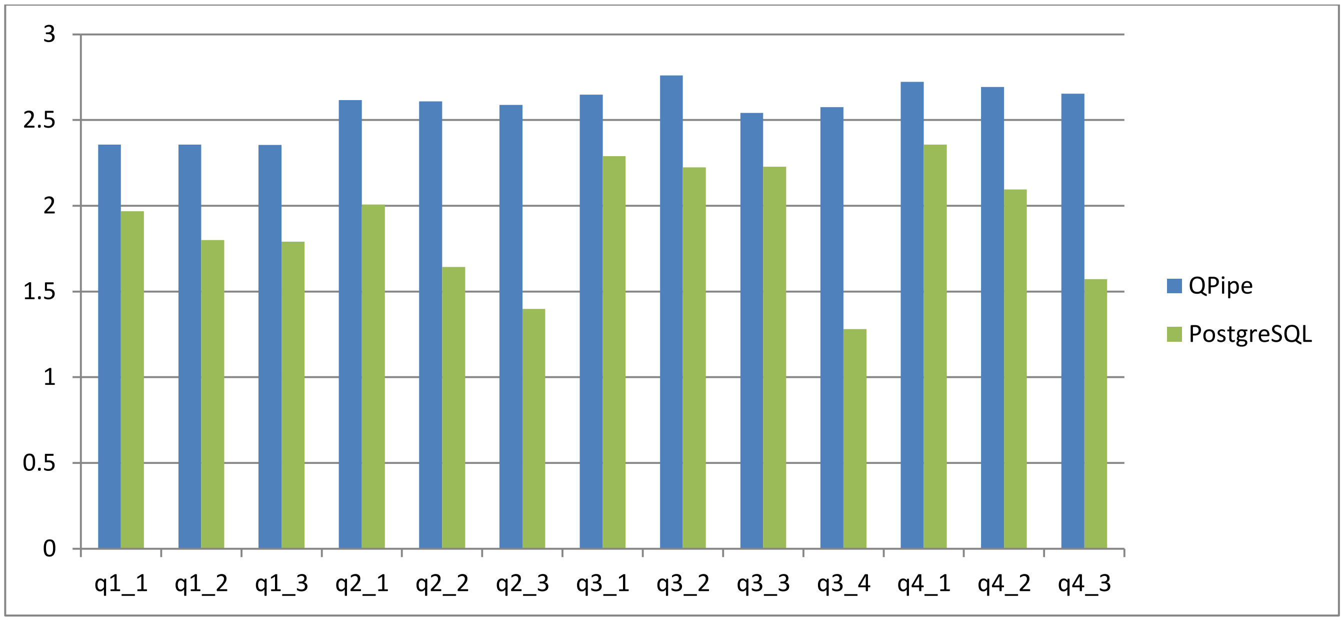

Figure 8. Performance of SSB Queries on QPipe and PostgreSQL (statistics can be found in Appendix C)

6. References

[1] S. Harizopoulos, V. Shkapenyuk and A. Ailamaki, "QPipe: A Simultaneously Pipelined Relational Query Engine." In *SIGMOD*,2005.
[2] S. Harizopoulos and A. Ailamaki, "A Case for Staged Database Systems." In *Proc. CIDR*, 2003.
[3] S. Harizopoulos and A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware." In *IEEE*, 2005.
[4] P. O'Neil, Betty O'Neil and X. Chen, "The Star Schema Benchmark (SSB)." Download this text from http://labs.inovia.fr/tracker/projects/pgbench/repository/revisions/2/raw/trunk/StarSchemaB.pdf
[5] P. O'Neil, Betty O'Neil and X. Chen, "Star Schema Benchmark (SSB) Revision 3." Download this text from http://www.cs.umb.edu/~poneil/StarSchemaB.PDF
[6] S. Harizopoulos, A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware." In *IEEE*, 2005.

DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Appendix A. SSB Tables [5]

LINEORDER Table Layout SF*6,000,000
LO_ORDERKEY numeric (int up to SF 300) first 8 of each 32 keys populated
LO_LINENUMBER numeric 1-7
LO_CUSTKEY numeric identifier FK to C_CUSTKEY
LO_PARTKEY identifier FK to P_PARTKEY
LO_SUPPKEY numeric identifier FK to S_SUPPKEY
LO_ORDERDATE identifier FK to D_DATEKEY
LO_ORDERPRIORITY fixed text, size 15 (See pg 91: 5 Priorities: 1-URGENT, etc.)
LO_SHIPPRIORITY fixed text, size 1
LO_QUANTITY numeric 1-50 (for PART)
LO_EXTENDEDPRICE numeric ≤ 55,450 (for PART)
LO_ORDTOTALPRICE numeric ≤ 388,000 (ORDER)
LO_DISCOUNT numeric 0-10 (for PART, percent)
LO_REVENUE numeric (for PART: (lo_extendedprice*(100-lo_discnt))/100)
LO_SUPPLYCOST numeric (for PART)
LO_TAX numeric 0-8 (for PART)
LO_COMMITDATE FK to D_DATEKEY
LO_SHIPMODE fixed text, size 10 (See pg. 91: 7 Modes: REG AIR, AIR, etc.)
Compound Primary Key: LO_ORDERKEY, LO_LINENUMBER

PART Table Layout 200,000*floor(1+log2SF)
P_PARTKEY identifier
P_NAME variable text, size 22 (Not unique)
P_MFGR fixed text, size 6 (MFGR#1-5, CARD = 5)
P_CATEGORY fixed text, size 7 ('MFGR#'||1-5||1-5: CARD = 25)
P_BRAND1 fixed text, size 9 (P_CATEGORY||1-40: CARD = 1000)
P_COLOR variable text, size 11 (CARD = 94)
P_TYPE variable text, size 25 (CARD = 150)
P_SIZE numeric 1-50 (CARD = 50)
P_CONTAINER fixed text, size 10 (CARD = 40)
Primary Key: P_PARTKEY

Supplier Table Layout
SUPPLIER Table Layout (SF*2,000 are populated):
S_SUPPKEY numeric identifier
S_NAME fixed text, size 25: 'Supplier'||S_SUPPKEY
S_ADDRESS variable text, size 25 (city below)
S_CITY fixed text, size 10 (10/nation: S_NATION_PREFIX||(0-9)
S_NATION fixed text, size 15 (25 values, longest UNITED KINGDOM)
S_REGION fixed text, size 12 (5 values: longest MIDDLE EAST)
S_PHONE fixed text, size 15 (many values, format: 43-617-354-1222)
Primary Key: S_SUPPKEY

Customer Table Layout
CUSTOMER Table Layout (SF*30,000 are populated)
C_CUSTKEY numeric identifier
C_NAME variable text, size 25 'Cutomer'||C_CUSTKEY
C_ADDRESS variable text, size 25 (city below)
C_CITY fixed text, size 10 (10/nation: C_NATION_PREFIX||(0-9)
C_NATION fixed text, size 15 (25 values, longest UNITED KINGDOM)
C_REGION fixed text, size 12 (5 values: longest MIDDLE EAST)
C_PHONE fixed text, size 15 (many values, format: 43-617-354-1222)
C_MKTSEGMENT fixed text, size 10 (longest is AUTOMOBILE)
Primary Key: C_CUSTKEY

Date Table Layout
DATE Table Layout (7 years of days)
D_DATEKEY identifier, unique id -- e.g. 19980327 (what we use)
D_DATE fixed text, size 18: e.g. December 22, 1998
D_DAYOFWEEK fixed text, size 8, Sunday..Saturday
D_MONTH fixed text, size 9: January, ..., December
D_YEAR unique value 1992-1998
D_YEARMONTHNUM numeric (YYYYMM)
D_YEARMONTH fixed text, size 7: (e.g.: Mar1998
D_DAYNUMINWEEK numeric 1-7
D_DAYNUMINMONTH numeric 1-31
D_DAYNUMINYEAR numeric 1-366
D_MONTHNUMINYEAR numeric 1-12
D_WEEKNUMINYEAR numeric 1-53
D_SELLINGSEASON text, size 12 (e.g.: Christmas)
D_LASTDAYINWEEKFL 1 bit
D_LASTDAYINMONTHFL 1 bit
D_HOLIDAYFL 1 bit
D_WEEKDAYFL 1 bit
Primary Key: D_DATEKEY

Appendix B. SSB Benchmark Queries [4]

Q1.1
```
select sum(lo_extendedprice*lo_discount)
as
revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between1 and 3
and lo_quantity < 25;
```

Q1.2
```
select sum(lo_extendedprice*lo_discount)
as
revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_yearmonth = 199401
and lo_discount between4 and 6
and lo_quantity between 26 and 35;
```

Q1.3
```
select sum(lo_extendedprice*lo_discount)
as
revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_weeknuminyear = 6
and d_year = 1994
and lo_discount between 5 and 7
and lo_quantity between 26 and 35;
```

Q2.1
```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q2.2
```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1between 'MFGR#2221
and 'MFGR#2228'
and s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q2.3
```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1= 'MFGR#2239'
and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Q3.1
```
select c_nation, s_nation, d_year,
sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_region = 'ASIA'
and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Q3.2
```
select c_city, s_city, d_year,
sum(lo_revenue)
as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_nation = 'UNITED STATES'
and s_nation = 'UNITED STATES'
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q3.3
```
select c_city, s_city, d_year,
sum(lo_revenue)
as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1'
or c_city='UNITED KI5')
and (s_city='UNITED KI1'
or s_city=’UNITED KI5')
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q3.4
```
select c_city, s_city, d_year,
sum(lo_revenue)
as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1'
or c_city='UNITED KI5')
and (s_city='UNITED KI1'
or s_city='UNITED KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Q4.1
```
select d_year, c_nation,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part,
lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1'
or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

Q4.2
```
select d_year, s_nation, p_category,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part,
lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1'
or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

Q4.3
```
select d_year, s_city, p_brand1,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part,
lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and s_nation = 'UNITED STATES'
and (d_year = 1997 or d_year = 1998)
and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1;
order by d_year, s_city, p_brand1;
```

Appendix C. Query Performance

Performance of QPipe:

	1	2	3	4	5	6	7	8	9	10	Average
q1_1	2.35	2.36	2.35	2.35	2.36	2.36	2.35	2.36	2.36	2.37	2.357
q1_2	2.34	2.36	2.35	2.36	2.36	2.36	2.36	2.35	2.36	2.35	2.355
q1_3	2.35	2.37	2.34	2.36	2.35	2.34	2.37	2.36	2.36	2.35	2.355
q2_1	2.55	2.64	2.62	2.63	2.64	2.57	2.63	2.63	2.62	2.6	2.613
q2_2	2.64	2.63	2.59	2.63	2.61	2.63	2.57	2.64	2.61	2.59	2.614
q2_3	2.58	2.62	2.57	2.54	2.6	2.59	2.62	2.54	2.65	2.6	2.591
q3_1	2.64	2.64	2.64	2.65	2.65	2.64	2.65	2.64	2.65	2.66	2.646
q3_2	2.78	2.76	2.78	2.78	2.69	2.75	2.77	2.76	2.78	2.77	2.762
q3_3	2.61	2.65	2.55	2.5	2.55	2.53	2.5	2.57	2.56	2.55	2.557
q3_4	2.67	2.53	2.66	2.67	2.53	2.58	2.52	2.56	2.54	2.54	2.58
q4_1	2.74	2.76	2.68	2.76	2.74	2.74	2.73	2.71	2.71	2.71	2.728
q4_2	2.72	2.72	2.71	2.71	2.74	2.74	2.66	2.65	2.63	2.69	2.697
q4_3	2.8	2.7	2.65	2.71	2.66	2.63	2.61	2.64	2.68	2.63	2.671

Performance of PostgreSQL:

	1	2	3	4	5	6	7	8	9	10	Average
q1_1	1.956	1.964	1.937	1.954	1.953	1.962	1.948	2.014	2.001	1.975	1.9664
q1_2	1.809	1.802	1.793	1.796	1.793	1.803	1.82	1.803	1.799	1.796	1.8014
q1_3	1.786	1.779	1.801	1.782	1.779	1.808	1.79	1.778	1.781	1.804	1.7888
q2_1	2.03	1.993	1.995	1.998	1.985	2.014	2.028	2.008	2.027	1.999	2.0077
q2_2	1.649	1.647	1.638	1.638	1.645	1.641	1.649	1.647	1.645	1.637	1.6436
q2_3	1.444	1.382	1.411	1.441	1.385	1.404	1.387	1.376	1.396	1.382	1.4008
q3_1	2.281	2.296	2.291	2.287	2.278	2.292	2.291	2.285	2.291	2.297	2.2889
q3_2	2.225	2.227	2.223	2.234	2.226	2.216	2.222	2.226	2.222	2.227	2.2248
q3_3	2.226	2.225	2.228	2.224	2.237	2.215	2.228	2.224	2.231	2.232	2.227
q3_4	1.284	1.278	1.278	1.279	1.276	1.278	1.297	1.282	1.277	1.274	1.2803
q4_1	2.37	2.302	2.292	2.369	2.363	2.367	2.358	2.363	2.377	2.371	2.3532
q4_2	2.098	2.099	2.095	2.098	2.093	2.089	2.11	2.101	2.088	2.085	2.0956
q4_3	1.569	1.568	1.569	1.569	1.568	1.571	1.569	1.579	1.578	1.575	1.5715