Semester Project in Computer Science

Query processing benchmark (TPCH)

Implementation and performance analysis

Andreas Schaedeli

EPFL
School of Computer and Communication Sciences

Autumn 2011

Professor: Anastasia Ailamaki

Supervisor: Data Intensive Applications and Systems Lab
Manos Athanassoulis

Contents

1 Introduction **3**

2 TPC Benchmark™H (TPC-H) **3**

3 The QPipe Query Engine **5**
3.1 Staged Database Systems 5
3.2 QPipe 5

4 Implementation of the TPC-H benchmark in QPipe **6**
4.1 Query Plan extraction 6
4.1.1 Example: TPC-H Query 18 7
4.2 Data Structures 9
4.3 Table Scans 9
4.3.1 Example: TPC-H Q18 10
4.4 Join 10
4.4.1 Example: TPC-H Q18 11
4.5 Aggregation 11
4.5.1 Example: TPC-H Q18 11
4.6 Sorting 11
4.6.1 Example: TPC-H Q18 12

5 Implementation Challenges **12**

6 Experiments and Analysis **13**
6.1 Experimentation environment 13
6.2 Results & Analysis 14
6.3 Problematic Queries 14
6.3.1 Data specification error 16

7 Future Work **19**
7.1 Operator availability 19
7.2 Query plan variation 19
7.3 Further Experiments 19

8 Conclusion **19**

Abstract

This project's main goal is to implement the TPC-H benchmark within the QPipe query engine. QPipe, in contrast to today's commercial Database Management Systems (DBMSs), uses an operator-centric instead of the commonly used query-centric approach. This staged database system design attempts to better coordinate queries using the same or similar operators, and to reuse previously calculated results and in-progress operations. Furthermore, it improves scheduling by switching context at operator boundaries instead of the usual random context switching.

Staged database system design being a rather recent approach, it is unavoidable to implement benchmarks in order to compare QPipe and other DBMS (commercial and open-source). QPipe uses pre-compiled query execution plans, implemented in C++ and concatenating stages, each of which implementing one single operator, passing data to each other using FIFO queues.

1 Introduction

QPipe with its staged, operator-centric design breaks the monolithic, query-centric paradigm of commonly used commercial DBMSs. It is, consequently, of high interest to compare the performance of this system, proposed in [1], with the performance of current DBMSs. In order to do this comparison, several benchmarks, such as the TPC benchmark suite, are available. These benchmarks provide a data model as well as several queries. As it is relatively easy to create queries that are biased in a manner to overemphasize a system's strengths, these benchmarks should be used in order to get acceptance for experimentation results by the academic community. The queries provided by the benchmark should therefore be balanced, aiming at using as many different parts of a system as possible. Due to this character of the queries, benchmarks give a relatively good overview of the strengths and weaknesses of a system.

In order to get an even better estimate of the system, several benchmarks should be implemented. Currently, the TPC-H [2] (this document) and SSB [3] (Star Schema Benchmark based on TPC-H) benchmarks are being implemented for QPipe at DIAS Lab. I will first give a short overview of the TPC-H benchmark 2, before presenting a brief description of staged database systems, particularly QPipe 3. I will then describe the implementation of the TPC-H query plans for QPipe 4, as well as challenges encountered during the implementation phase 5. Before coming to the conclusion 8, I will present the results of the comparison of QPipe and PostgreSQL [4] for isolated single-query execution 6, as well as some ideas for future work with respect to the TPC-H benchmark implementation on QPipe 7.

Initially, it was planned to lay more stress on the analysis part. However, as I did do this project for 8 instead of the initially scheduled 12 credits, I had only 2/3 of the time. This lead to the project's content being cut back. In the final section on Future Work 7, I will present possible experiments as well as improvements of the query plans.

2 TPC Benchmark^{TM}H (TPC-H)

TPC-H is one of several benchmark published by the *Transaction Processing Performance Council* (TPC). It is designed to have broad industry-wide relevance. More precisely, the data scheme reflects a general data scheme in industry, and the data population was made to be as realistic as possible. The 22 queries provide a broad coverage of queries used in real applications. The complete specification may be found in [5]. It contains the data scheme to be used, as well as the queries and a manual of how to construct the data population. A program to generate the data (*dbgen*) is also available.

The relational data scheme consists of eight tables, as shown in 1. Note that the size of the data population is given in the form $SIZE = BASE\_SIZE * SF$, where SF denotes the scaling factor used to generate the data, and $BASE\_SIZE$ is the approximate number of tuples with scaling factor 1. Table 1 gives a brief overview of the eight relations.

Relation Name	*Description*
REGION	Contains 5 regions of the world
NATION	Contains 25 nations of the world, associating them with the region they belong to
PART	Contains parts of 5 different segments and various types
SUPPLIER	Contains suppliers associated with the nation where they are located
CUSTOMER	Contains customers associated with their home country and the market segment of which they buy parts
PARTSUPP	Contains information about which supplier supplies what parts, along with the available quantity and the cost per piece
ORDERS	Contains orders associated with the customer that placed them
LINEITEM	Contains information about every item (represented by part and supplier keys) belonging to an order

Table 1: Overview of TPC-H relations

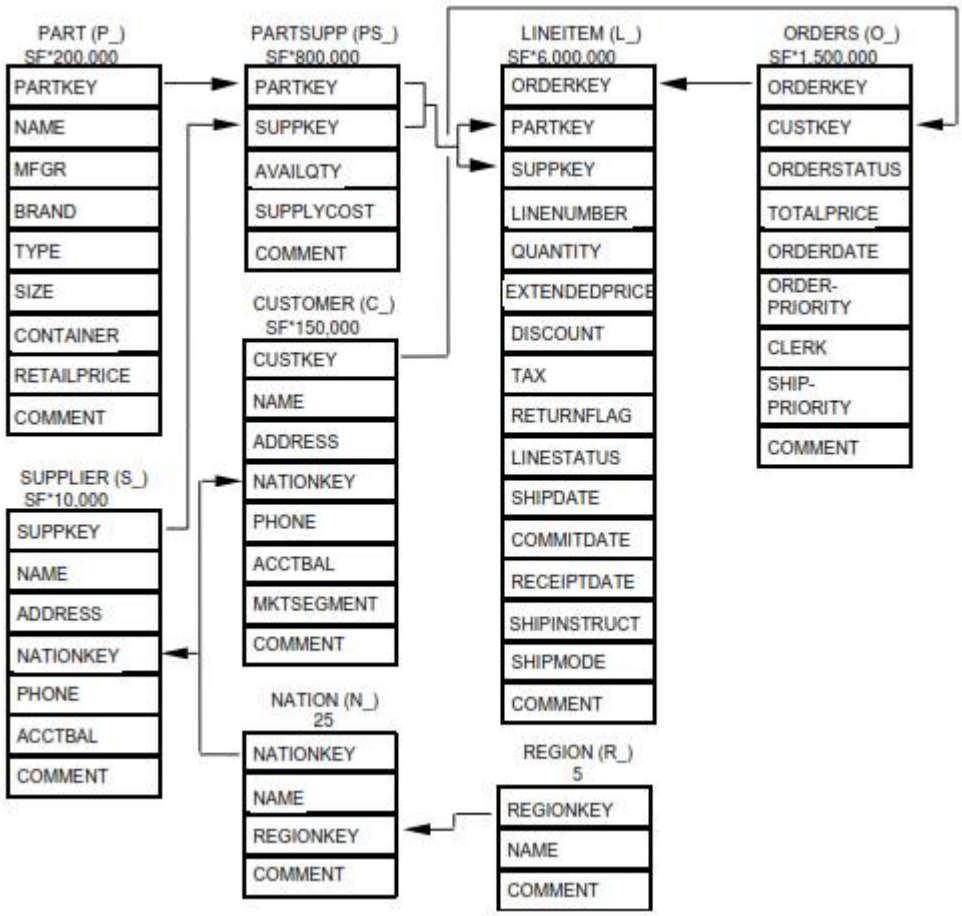

Figure 1: The TPC-H schema

3 The QPipe Query Engine

QPipe, a simultaneously pipelined relational query engine, was proposed and implemented at Carnegie Mellon University, and was later integrated with Shore-MT [6] at DIAS Lab at EPFL. DIAS Lab is maintaining QPipe on top of the in-house storage manager, Shore-MT [6], which is able to scale to utilize today's highly parallel multicore hardware.

QPipe breaks the commonly used query-centric paradigm and instead focuses on the operators needed to serve a query. Instead of executing a query at once, it breaks up the query into several stages, each of which can be executed by a single operator. QPipe is therefore one implementation of the Staged Database Systems design [7].

3.1 Staged Database Systems

Today's commercial DBMSs use a monolithic approach, which means that a query is considered to be a single unit of work. Staged database systems use a "Divide-and-conquer"-like approach by dividing a query into several stages. Every stage then implements a single operator, which is mostly independent of the other ones. The data is passed from one stage to another through queues, containing blocks of tuples. An operator dequeues tuples from the queue of the incoming packet, and enqueues the resulting tuples to the queue of the outgoing packet. The next operator can therefore start its work as soon as it receives some data. This staged architecture, depicted in Figure 2 has several advantages compared to the traditional, monolithic architecture:

- *Distribution of work:* In traditonal DBMSs, there is often a predefined number of worker threads. Each of these usually handles one query. It is rather difficult to find the optimal number of threads with changing workloads, as resources are wasted in the presence of too many threads, and concurrency is restricted when there are too few threads. In the staged approach, there is one pool of worker threads per stage. It is much easier to monitor thread utilization for a single stage and to auto-tune the stage's parameters, than it is for the whole query engine.
- *Random preemption vs. Yielding:* In most commercial DBMSs, a thread executes during a previously defined time quantum. Once this time elapses, the thread is preempted, and a context switch occurs. This is likely to happen in the middle of a logical operation. The whole thread state then needs to be saved and later restored, which is a very costly operation. In the staged system, there is no preemption. A thread yields the CPU when it is done with the work on one stage, i.e. it has written all the output tuples. This means that its current state contains way less information, and it is therefore much less costly to switch context.
- *Data reusing:* In current DBMSs, almost no data can be shared among different queries. This occurs only if, by chance, some data used by another query, e.g. resulting from a tablescan, is still cached when a new query needs it. This is completely different for a staged database system. If queries execute concurrently, the query engine identifies same or similar operations among all the queries. Due to the scheduling, which can for instance do context switching only among threads belonging to the same stage, data can be reused. This leads to fewer cache misses and therefore less total I/O delay.

3.2 QPipe

QPipe is one implementation of the previously described staged database system design. It is written in C++ and built on top of DIAS Lab's *Shore-MT* storage manager. It is, however, only a query execution engine, i.e. it does not (yet) contain a built-in query parser and optimizer. This means that in order to implement queries, the query plans need to be built by hand or taken from another DBMS, using the 'explain' command. Query plans taken from several commercial and open-source systems are available and documented on-line.

The QPipe framework then offers then various operators. Not all commonly used SQL operators are however currently available (e.g. '(not) exists', 'index scan'), or could not be used due to input compatibility reasons (e.g. '(not) in', 'hash join', 'merge join'). This obviously drastically reduces the number of possible query plans, making it often impossible to use the most efficient plan (based on the static decision of the query optimizer of a system which assumes the existence of these operators).

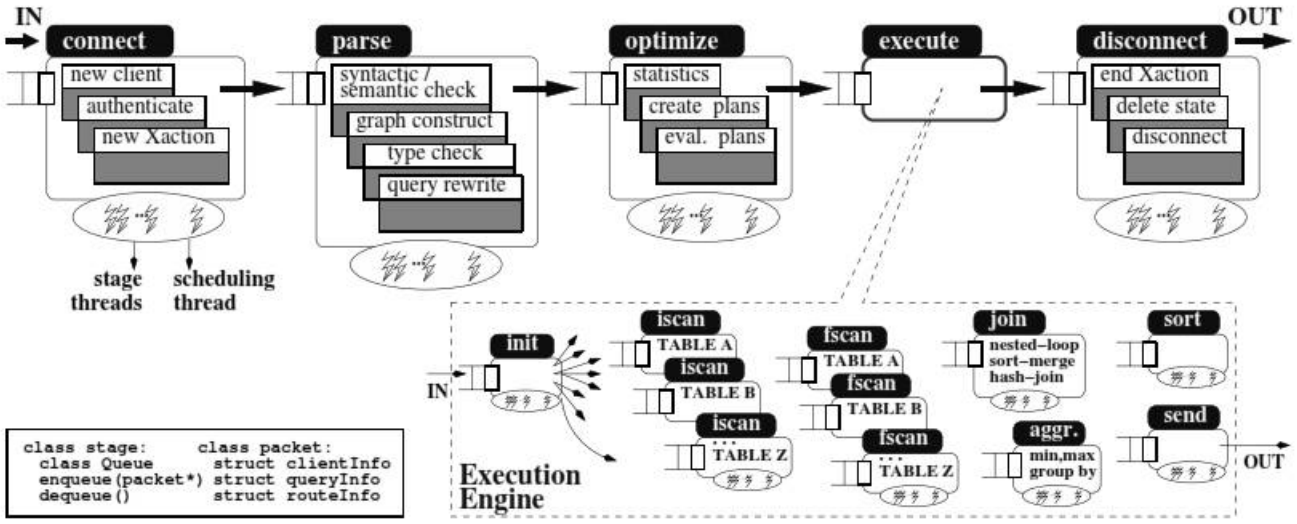

Figure 2: Staged Database System design

In order to implement a query plan, packets need to be created. A packet always contains one or two input packets and an output queue, as well as the stage operator itself. Furthermore, a filter, allowing for selection and projection, is applied after each operator. For certain operators, further functionality needs to be provided, e.g. a key extractor and comparator for the sort operator. Passing packets as input of other packets finally results in creating an execution tree. The root is then passed to the execution engine, which will recursively call the operators of the children packets, and finally output the result tuples.

4 Implementation of the TPC-H benchmark in QPipe

QPipe is implemented in C++ programming language. This means that the query plans of TPC-H had to be transformed into C++ code as well, in order to run them. Each query consists of several stages, all of which, as described above, consist of various components, which have to be implemented. This chapter describes the work to be done from the query's SQL code until it can be run in QPipe. A concrete example is given for TPC-H query 18. This query was chosen because most of the techniques applied for implementation were used there.

4.1 Query Plan extraction

As the QPipe query execution engine does not contain a parser and optimizer, query plans need to be found elsewhere. One possibility is to run the queries in another DBMS using the 'explain' command, which allows the user to find the query execution plan. The approach chosen for this project was however a different; for most queries, the plans used are those found on the internet [8] for SQL Server 2008. As mentioned earlier, QPipe's current state did not allow to use all existing SQL operators, which lead to a simplification or adaptation of many query plans. More precisely, all index lookups and index scans were replaced by full table scans, and hash joins are used for all join operations. Furthermore, the 'in' and 'exists' operators were simulated by using an aggregation and a join operator.

While index scans were simply not available at the time this project was done, the reason for using hash joins is different. In fact, block nested loop joins used a different an input structure that was not a packet, and we did not figure out how to easily adapt the input. Using sort-merge joins would have been possible; however, in most cases it was assumed that the input was already sorted due to a previous index scan. As we could not use index scans, this was not true, which means that an additional sort stage would have been needed. It is however questionable whether a hash join or a merge join with previous sorting would have

been more efficient. For this reason, hash joins were implemented everywhere. However, as soon as more operators become available, it is strongly encouraged to try different query plans, especially different join operators, as the currently implemented plans are most probably not the most efficient.

The structure of the query plans, however, was preserved, which means that the table joins were implemented in the same order as given in the aforementioned plans. For a few queries, a different execution plan was chosen, more precisely a query plan extracted from a DB2 query engine. As we did not use sort-merge joins, however, some sorting operators were not implemented. But in general, [8] is a good reference to see which plans were used for QPipe.

4.1.1 Example: TPC-H Query 18

The SQL code for this query is as follows:

```
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) >[QUANTITY])
        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;
```

In this case, the DB2 query plan 3 was taken as a reference for the QPipe implementation, which can be seen in Figure 4. One thing to notice is that this query plan scans the *lineitem* relation only once (compared to two scans in the SQL Server 2008 query plan). This is possible, as the result is grouped by *l_orderkey* in both the main and the sub query, and the *sum of l_quantity* is used in the end as well. Therefore, this sum does not need to be calculated twice. Here is an explanation of what is precisely done in the DB2 and the adapted QPipe query plans:

- First, there are index scans for all three relations, namely *LINEITEM, ORDERS, CUSTOMER*. In QPipe, we use full table scans.

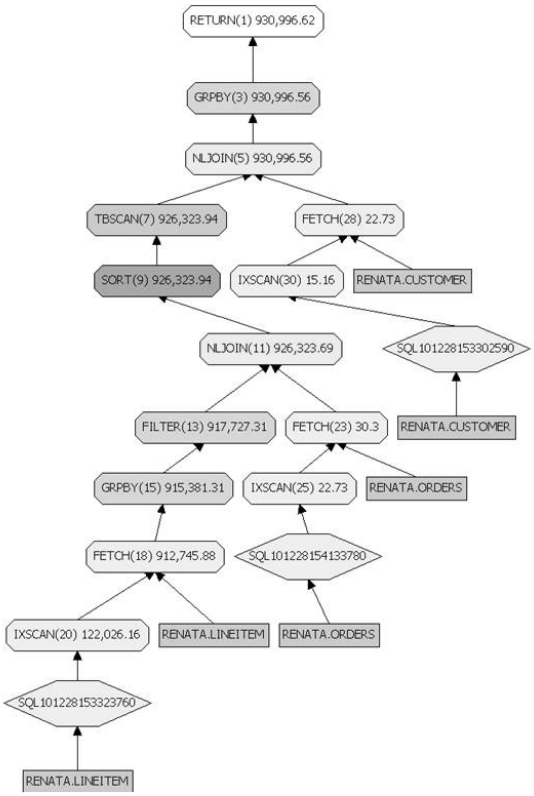

Figure 3: DB2 query execution plan for Q18

- The lineitems are then grouped by *l_orderkey*, and the sum of *l_quantity* is calculated. In QPipe, this is done the same way, i.e. by using an aggregation operator.
- The resulting tuples are then filtered, and only those where *sum(l_quantity) >[QUANTITY]*, where [QUANTITY] is a randomly selected value within [312..315], pass the filter. In QPipe, every stage may contain a filter. Therefore, this filter is implemented within the aggregation stage.
- The remaining, aggregated *LINEITEM* tuples are then joined with the whole *ORDERS* relation, using a natural join on the *order key*. While DB2 proposes to do a *block nested loop join (BNL*, we have to use a *hash join* in QPipe.
- It can be noticed that no two tuples of the join output can have the same order key. This means that the final group by clause will do nothing. As the two fields on which the result will be sorted are already present as well, this new relation can already be sorted on *o_totalprice desc, o_orderdate*. This is possible as a BNL join, preserving the tuple order of the left relation, will later be used. As we have to use hash join, which is not order preserving, in QPipe, sorting would be a waste of time. Therefore, nothing is done here.
- The left relation, i.e. the joint *LINEITEM and ORDERS* relation, is then joined with *CUSTOMER*, using a natural join on the *customer key*. Again, a *BNL join* is used in DB2, while a *hash join* has to be done in QPipe.
- Then, in both query plans, the *group by aggregation* is applied. However, as the sum aggregation was already calculated before, and all order keys are distinct, this group by aggregation will not do anything useful.
- While the DB2 query plan ends after the group by aggregation, we still have to do the sorting in QPipe.

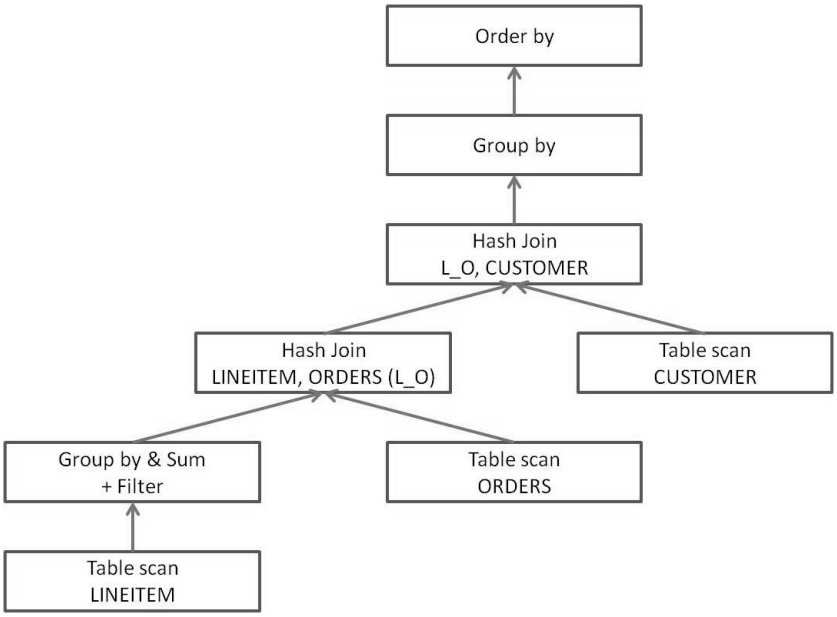

Figure 4: QPipe query execution plan for Q18

4.2 Data Structures

It would be possible to always use the complete tuple in QPipe. However, as this is not desirable, due to huge resource wasting (by doing avoidable memory copying), additional data structures need to be put in place. In general, a new data structure is needed for the output of each stage, as the content of in- and output usually differs. For Q18, six additional data structures are used; one for each projection after the table scan, one for each join, and one representing the sort key. It has to be noticed, however, that the structure for the projected lineitem tuples is used twice, namely after the table scan and the sum aggregation. The tuple called 'Final Tuple' is used for the second join, the group by and the order by stages. The key structure is only for simplicity reasons, as casts may become complicated when the key consists of several fields.

Note that the names of these structures were chosen to express where they are used. For structures containing values of different relations, only the identifiers of these relations occur in the name. The name usually contains as well the operator which produces tuples having this structure. For example, the operator joining lineitem and orders tuples produces tuples of the structure *q18_l_join_o_tuple*.

4.3 Table Scans

The first step in a query execution is the scanning of the relations involved in the query. In order to do this, a *tscan packet* had to be created. Such a packet takes the following parameters:

- *packet_id:* A string describing the packet
- *output_buffer:* The queue where the output tuples will be stored. The size of the output tuples has to correspond with the size given at the creation of this queue. The previously created data structures, along with the *sizeof* function, is very useful in this case.
- *output_filter:* A filter letting pass only tuples that qualify (selection; not used in Q18 table scans), and projecting the initial tuples to the custom data structures, keeping only fields that are used later on
- *db:* A reference to the storage manager
- *table:* Descriptor of the table to be scanned
- *pxct:* The transaction to be used
- *lm:* Lock mode

In order to do table scans, a filter is needed. It is possible to use a trivial filter, letting pass all tuples and doing no projection. However, it is often the case that a filtering on specific data fields needs to be done. Furthermore, as previously explained, only those attributes needed later on should be projected. In order to all of this, there exists a basic, abstract class named *tuple_filter_t*, which can be extended in order to create custom filters. The following methods can or must be implemented:

- *Constructor:* The constructor can be used to pass the generated random values, needed for selection, to the filter. Furthermore, the super class constructor needs to be called, giving the input tuple size as a parameter
- *Destructor:* The destructor may be implemented in order to free allocated memory
- *bool select():* This method takes a tuple as an argument and returns true if the tuple qualifies, i.e. passes the filter, and false otherwise
- *void project():* This method receives an input and an output tuple. It then copies the useful attributes from the input to the output tuple
- *Filter_Class* clone():* This method is mandatory and should return a clone, i.e. a deep copy, of the current object
- *c_str to_string():* This method is mandatory and returns a string containing a description of the filter

Constructor, Destructor, clone and to_string methods can or must be implemented for all operators. They will therefore not be presented again.

4.3.1 Example: TPC-H Q18

In Q18, three relations are read. Therefore, three filters had to be created. The select method, however, returns always true, as there are no selection predicates. The project method projects the inital tuples on the custom data structures used to carry the data to the next stage.

4.4 Join

Joins are used to combine two relations, thereby forming another one. As explained earlier, only hash joins are currently used in the TPC-H implementation of QPipe. In order to use such a hash join, a *hash_join_packet_t* object needs to be created. Besides the output queue and the filter, as well as the packet description, the following parameters are needed to create the packet:

- *left:* Left input packet
- *right:* Right input packet
- *join:* Reference to the joiner, defining what exactly needs to be joined, i.e. which attributes will be copied
- *outer:* Boolean indicating whether a left outer join will be performed or not (default: false)
- *distinct:* Boolean indicating whether a distinct join will be performed or not (default: false

For Q18, two joins are needed. Both of them have a trivial output filter. However, in some cases a condition can be verified only after a join. If this is the case, a custom filter can be used, such as described above. In addition to the filter, where it is optional to create a customized one, it is mandatory to implement a so-called *joiner*. This is a class extending *tuple_join_t*, defining how the join is exactly done. The following methods must or can be implemented in the custom joiner:

- *Constructor:* The constructor is mandatory for the joiner. It is needed to give to the super-class constructor details about the in- and output tuple sizes, the offset of the key within the input tuples, and the key size
- *void join():* This method receives three tuples; one output tuple and two input tuples. It is similar to the project method of a filter, as it copies the values needed later on from the in- to the output. This method is mandatory
- *void left_outer_join:* This method is called when, at creation of the join packet, the attribute 'outer' was set to true, and there is no right tuple matching a key of a left tuple. In case no outer join is desired, this method does not need to be implemented

4.4.1 Example: TPC-H Q18

Both joins implemented in Q18 are straight forward. The first one, joining *LINEITEM and ORDERS* relations on order keys, copies all four attributes of the projected orders tuple, as well as the quantity (which is in fact a sum) of the lineitem tuple, to the custom output tuple. The second join operator keeps both attributes coming from the *CUSTOMER* relation, and all but the customer key (which was used for the join, but is not used later on) of the joint lineitem and orders relation.

4.5 Aggregation

Aggregation operators can be used for a wide variety of tasks. Mainly for aggregation operations, such as 'SUM' or 'AVG', but it can also be used to implement the 'DISTINCT' operator. There are two different aggregation methods: *Stream Aggregates and Hash Aggregates*. Stream aggregates can only be used when all tuples have the same key, or are ordered on the aggregation key. A stream aggregate reads input tuples as long as it finds the same key, then does the finish operation and continues with the next tuple. If, however, tuples are not sorted on the aggregation key, the stream aggregate will not produce the desired result. In this case, a hash aggregate needs to be used. With this method, the aggregation key is hashed. If the same key had not be seen before, a new, usually empty, aggregate tuple is created and aggregated with the input tuple. If there is already an aggregation tuple corresponding to the input tuple's key, the two tuples are aggregated.

Both types of aggregation packets basically need the same parameters. These, besides the already know ones (input packet, output buffer, filter, description), are the following:

- *aggregate:* Custom aggregator class specifying what exactly the aggregation stage operator does
- *extractor:* Key extractor. The extractor class contains methods to generate an 'extract hint', and to extract the whole key
- *compare:* Key comparator. The comparator is only needed for hash aggregation and specifies how exactly two keys need to be compared

In order to do an aggregation, at least an aggregator class, extending *tuple_aggregate_t*, needs to be implemented. This class requires the following methods (besides clone and to_string):

- *Constructor:* This can be used to pass generated random values to the operator, and to call the super-class constructor giving the input tuple size
- *void init():* This method is called upon creation of a new aggregate tuple. It can be implemented whenever, besides the key, there should be non-zero values in the initial aggregate tuple, e.g. a high value if a minimum needs to be extracted afterwards
- *void aggregate():* This method is always called with an input tuple and the corresponding aggregate tuple. Here, values can be summed up, divided, compared - whatever the aggregation requires
- *void finish():* This method is called with two aggregate tuples - an intermediate tuple, created by calls to aggregate(), and the designated output tuple. This sub-stage is needed e.g. for average operations. Usually, however, the intermediate tuple's content is just copied to the output tuple

The key extractor and comparator will be described in the next subsection.

4.5.1 Example: TPC-H Q18

Two aggregations are needed in Q18. The first one two calculate the summed-up quantity of all lineitems with a given order key. The second one to group by various attributes and to calculate a sum. However, as written before, as order keys are distinct, this second aggregation stage is actually useless.

4.6 Sorting

Tuples need to be sorted in order to use sort-merge joins. Furthermore, users often desire to get query results ordered on one or several attributes. For this purpose, a sort packet can be created. Besides the in/output arguments, the filter and description, takes the following parameters:

- *extract:* Key extractor. The extractor class contains methods to generate an 'extract hint', and to extract the whole key
- *compare:* Key comparator. The comparator is used to compare two keys. This is useful to define the precise ordering, and is especially needed when tuples should be ordered on descending values

In order to extract the key from a tuple and compare it to another key, two classes are needed. For the key extractor, the *default_key_extractor_t* can be used. This default extractor is especially useful when the key is a simple integer, as it then gives this integer as an extract hint, which at the same time is the complete key. If, however, the key is more complex, and a non-zero hint is desired, a custom key extractor needs to be implemented, extending the *key_extractor_t* class and containing the following method (besides the mandatory clone method):

- *int extract_hint:* This method receives the extracted key and can return a key hint of the size of an integer (i.e. four characters, a double cast to integer, or an integer). The hint must however satisfy the condition that if it is greater than another hint, then the whole key must be greater than the other one, and vice versa. If two hints are equal, the full key needs to be compared

As the extract hint is only 4 bytes long, it is not always possibly to completely sort a relation using only the key extractor. If two extract hints are the same, the keys need to be compared. For this purpose, a key comparator class, extending *key_compare_t*, needs to be implemented. This class contains the following method:

- *int operator()()* : This method receives two keys and has to return an integer. If the first key is smaller than the second one, the integer must be negative. Consequently, the integer is positive if the first key is larger, and is zero if the keys are equal

4.6.1 Example: TPC-H Q18

The output tuples in Q18 should be ordered on two values: First, on *descending order total price* values, and second, *on ascending order dates*. In the extractor, the hint is the total price. As this is a decimal value, it first has to be transformed into integer. As the sorting should be done on descending values, a negative sign needs to be added to the hint.

The key comparator then first compares the total price values of both keys. If the first one is smaller, this means, it will occur later in the result, +1 is returned. -1 is returned if the first value is larger. If both values are equal, then the difference of the subtraction of the first and second order date is returned, as the dates should again occur in ascending order.

5 Implementation Challenges

As I had never programmed in C++ before beginning this project, getting used to this language was the first minor challenge. Having a solid Java and C background, however, this did not take very much effort. Other topics were much more challenging, as this chapter describes.

As the QPipe system already existed when I started the project, I had to get to know the API in order to implement queries. Furthermore, some queries were already implemented, which proved to be helpful. Code did already exist for the random input generation, as well as for various utility methods.

To use or to extend existing code helps saving a lot of work - under the condition that this code is correct. This was, however, not always the case and cost me dozens of hours of work. The following list shows program bugs encountered during the implementation phase:

- *Data Generation:* When testing queries and comparing the results with the ones received by executing the same query on a reference database, generated by dbgen, inconsistencies could be spotted concerning order keys. In fact, by digging deeper into the code, we found out that, in the order relation, there were indeed duplicate order keys, which violates the primary key constraint of this table.

  Furthermore, some decimal values, such as *l_extendedprice, l_discount or ps_supplycost*, are 100 times larger than what their actual values should be. This, in contrast to the order key bug, could however

not be corrected easily, so that the concerned values need to be divided by 100 in every query they are used.

Another error concerns the two date fields *l_commitdate and l_receiptdate*, which seem to have been inverted.

- *Random input generation:* Most errors were encountered within these methods. In fact, the person who implemented the random input generation created enumerations for all possible values, always adding a final item of the form END_FIELD_NAME, e.g. *END_N_NAME* for the nation names. When generating a random value, we can then give 0 as the lower and END_FIELD_NAME as the upper boundary. The problem, however, is that this final element only exists virtually, i.e. there will never be a 'real' value that corresponds to it (e.g. there will be no nation key with value END_N_NAME = 25). What needs to be used instead as upper boundary is END_FIELD_NAME - 1.

  As I assumed the already existing code to be correct, I did not check for such an error when implementing the queries. I then wondered why QPipe crashed during query execution. Finding all occurrences of this bug, which was frequent and occurred in about every second query input, again took me hours of work.

- *Bugous utility methods:* In order to compare with the tuple fields, the previously mentioned random input values had to be converted to strings, or strings were converted to integer values. Unfortunately, not all of these methods were correctly implemented. An example is that strcat, a method to concatenate two strings, was called with an uninitialized string, which could therefore contain random data. What happened then was that random data was concatenated with a useful value, giving a useless result. Again, finding all bugs of this type took a lot of time.

As this project was mainly about implementation, virtually all challenges laid in precisely this implementation part. Sometimes the query plan implementation was challenging as well, especially if a need operator, such as '(not) exists' did not exist yet. Some queries (21, 22) had to be simplified due to this reason. This, however, were minor issues, compared to the time wasted on debugging, which finally made probably half the total working hours.

6 Experiments and Analysis

Now, as most queries are completely and correctly implemented, they can be tested and the executions times can be compared to another DBMS. For this project, experiments were done in QPipe and PostgreSQL, a freeware DBMS. As written before, the project content had to be cut back, and therefore only few experiments could be done during this project. More precisely, queries were executed individually on both systems, and the execution time was measured. In order to execute exactly the same queries, they were first run on QPipe, where random input parameters were generated. These parameters were then copied into a .sql file, which could then be run with PostgreSQL.

6.1 Experimentation environment

All experiments were run on *diascld9.epfl.ch* server. This is a Sun X4140 server with 2 64-bit quad core CPUs of type AMD Opteron running at 2.7 GHz. The server has 32 GB RAM and a 1 Gbps network connection, as well as 8 SAS disk drives having a total storage capacity of 1.5 TB. The server runs Kubuntu with a 2.6.28-18-generic kernel.

The 8 SAS disk drives, all running at 10,000 RPM, are used as follows:

- *OS:* Two mirrored disks of 146 GB each
- */home directory:* Two 146 GB disks, having a stripe size of 256 KB
- *cloud data:* Four disks of 300 GB each, having a stripe size of 256 KB

All disks use read caching, but no write caching. Hardware level RAID1 is used for the OS disks, RAID0 for all other disks.

The environment was configured such that all data was in memory, i.e. no I/O operations were required for these experiments.

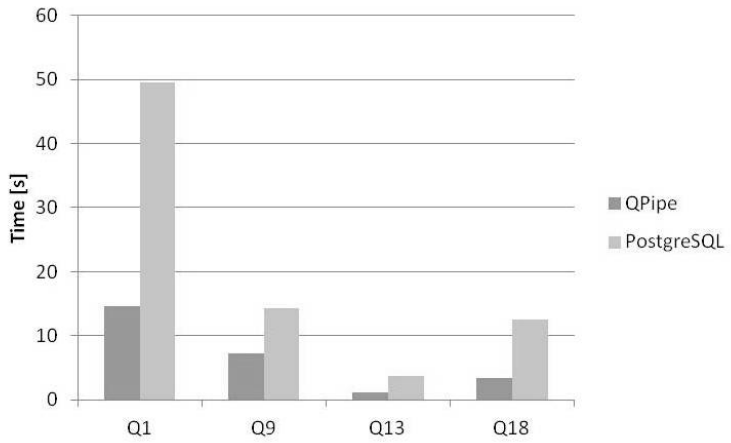

Figure 5: Queries for which QPipe is faster than PostgreSQL

6.2 Results & Analysis

The experimentation results can be roughly divided into four groups, as can be seen in the corresponding figures.

- QPipe faster than PostgreSQL: Figure 5
- PostgreSQL faster than QPipe: Figure 6
- Both systems take roughly equally long: Figure 7
- A query terminates only in one system: Figure 8

Unfortunately, the current TPC-H implementation on QPipe does not seem to be very competitive compared with the freeware DBMS PostgreSQL. The structure of the queries where QPipe is faster differs quite a lot, so we can not identify a type of queries where the system is especially efficient. However, the results have to be looked at while keeping in mind several facts:

- The query plans implemented in QPipe are most probably not the best ones. The plans were taken from other system, which used various join methods, which were then all replaced by hash joins. While the query plans were efficient in the original systems, using merge and block nested loop joins, they may be bad in case hash joins are used.
- Some operators, such as 'in' or 'exists', either did not exist or could not be used due to input requirements being different from the standard packets. They had to be simulated by using aggregations and joins, which is certainly less efficient than using a single dedicated operator.
- All relations were always entirely scanned, instead of using index scans, as this method was not available yet.
- QPipe is designed to be especially efficient when queries are executed concurrently. For isolated queries, a staged database system may be slower, as not a single thread executes the query, but intermediate results have to be passed from one stage to another, and different threads are used for each stage, possibly leading to more context switches than in a traditional system.

These factors may, despite the underlying, very powerful storage manager, contribute to the rather deceiving results. However, there is much potential for improvement, especially concerning the query plans. The next section talks about these possibilities to improve the benchmark's performance.

6.3 Problematic Queries

As can be seen in Figure 8, some query execution times could not be compared. There are currently two queries, namely *Q2 and Q21*, which do not correctly run in QPipe. Even hours of debugging work did not

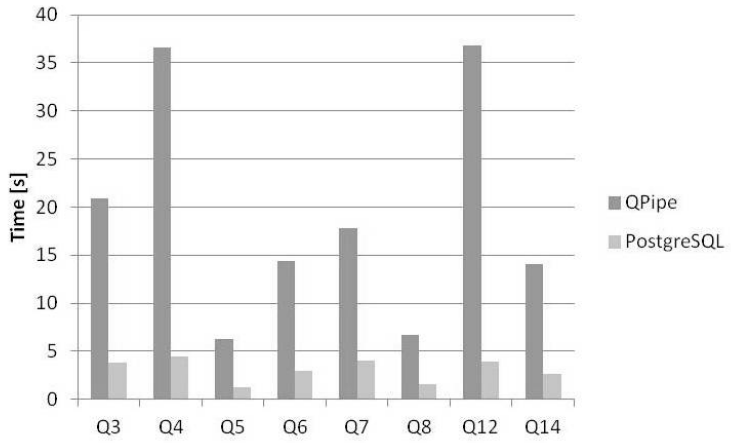

Figure 6: Queries for which PostgreSQL is faster than QPipe

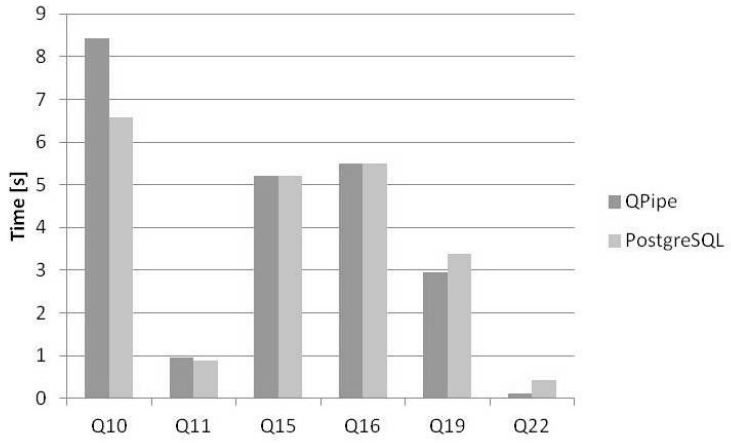

Figure 7: Queries which run roughly equally fast on both systems

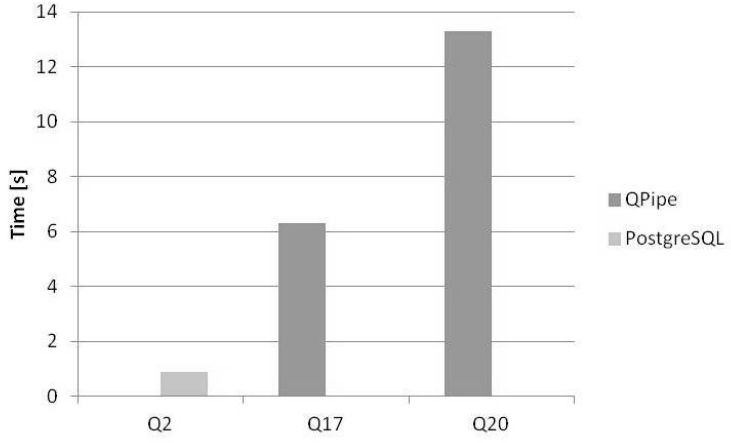

Figure 8: Queries which only terminate on one system

reveal the bug in these two query implementations. Queries *Q17 and Q20* take extremely long in PostgreSQL, or do not even terminate at all. Therefore, none of these queries could be classified among the three other categories. Q21 was not even classified at all, due to the reason I am going to explain in the following subsection.

6.3.1 Data specification error

On one hand, Q21 did not terminate at all in QPipe. In PostgreSQL, however, it did terminate, but always returned an empty result. When searching for the reason for this, we found out the following:

In clause 4.2.3 of TPCH specificaion 2.14.3 (the current version as of January 2011) we can see that for a given tuple of the table Lineitem we have the following information (page 86): L_COMMITDATE = O_ORDERDATE + random value [30 .. 90]. L_RECEIPTDATE = O_ORDERDATE + random value [1 .. 30].

Clearly this means that always L_COMMITDATE $\geq$ L_RECEIPTDATE; Which will lead to several queries (e.g. Q4, Q12, Q21) having always empty results.

In order to reason about why the results will be empty, I will present the SQL code of these three queries.

Query 4 code:

```
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;
```

The condition 'l_commitdate <l_receiptdate' will therefore never be satisfied, which means that the 'exists' clause will always be empty. This finally leads to having an empty result. However, due to the switching of l_commitdate and l_receiptdate in the data generation algorithm for QPipe, the condition mentioned before is always true, which means the result was non-empty, which means that we were not able to discover this specification error when testing Q4. The same holds for Q12, which is presented next.

Query 12 code:

```
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 ('[SHIPMODE1]', '[SHIPMODE2]')
        and l_commidate <l_receiptdate
        and l_shipdate <l_commitdate
        and l_receiptdate ≥ date '[DATE]'
        and l_receiptdate <date '[DATE]' + interval '1' year
group by
        l_shipmode
order by
        l_shipmode;
```

Again, the condition 'l_commitdate <l_receiptdate' can never be satisfied, which implies that the result set will be empty. However, as written for query 4, we did not get an empty result, which means we still could not find the specification error. This error was however found when query 21 was executed.

Query 21 code:

```
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'
        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 s_name = '[NATION]'
group by
        s_name
order by
        numwait desc,
        s_name;
```

In query 21, the condition 'l_commitdate >l_receiptdate' appears twice. The first time, it needs to be satisfied, but not the second time. Obviously, as the specification says that l_commitdate $\geq$ l_receiptdate, the first condition will almost always be satisfied. However, the SQL code says that there must exist another lineitem having the same order key but a different supplier key, but that there must not exist such a lineitem where the initially mentioned condition is satisfied. According to the specification, the only case in which this is respected is when both values are equal. However, in our data set, there is no tuple having this property. This means that the result set will always be empty.

On the other hand, both the sample outputs and the exeuctions of the queries were not empty. We discovered a discrepancy between the dbgen provided by the TPC and the standard, because the dbgen create tuples in lineitem which commitdate can be either larger, smaller or equal to the receiptdate. This contradicts the standard BUT it gives non-empty results for the aforementioned queries.

Here are some tuples created by dbgen for scaling factor 1. Only the fields l_orderkey, l_commitdate and l_receiptdate are listed.

L_ORDERKEY	L_COMMITDATE	L_RECEIPTDATE
1	1996-02-12	1996-03-22
1	1996-03-05	1996-01-31
32	1995-08-27	1995-10-26
32	1995-10-07	1995-08-27

This shows that indeed both kinds of tuples exist; tuples where l_commitdate >l_receiptdate and tuples where l_commitdate <l_receiptdate.

Thus, we believe that there is an error in the above constraint of clause 4.2.3.

7 Future Work

The performance analysis of QPipe shows that the response time of single queries using QPipe versus a similarly tuned postgreSQL does not yield benefits. This, however, is hardly surprising since we expect more performance benefits when there are several queries executed concurrently and there is opportunity for work sharing. As this project was a first step of TPCH integration in QPipe there is a lot of room for improvement. This section presents work that may be done in order to complete and improve the current implementation.

7.1 Operator availability

Several important operators do not yet exist or can not be used the 'standard' way. Missing operators are '(not) exists' and index scans. Furthermore, the 'distinct' operator has to be implemented by using an aggregation, which may not always be the most efficient way to do.

Operators which currently require a different input format (tuple_source_t instead of the standard packet_t) are the block nested loop (BNL) operators, such as 'BNL join' or 'BNL in'. It might be helpful to change the input format to the standard one, or to add an explanation of how to use the tuple_source_t.

7.2 Query plan variation

Once all necessary and important operators will be available and can be used, the number of possibilities to create plans for a given query will increase even more. It will then be possible to implement exactly the same query plans as in the systems QPipe is compared with (e.g. PostgreSQL), which would allow to get more significant experimentation results. Furthermore, by varying the query execution plans, it may be possible to find out whether there exist execution patterns that are specially efficient in QPipe, which would later on allow to build a query optimizer for QPipe. However, to reach this point, a lot more of experiments will need to be done.

7.3 Further Experiments

The execution of isolated queries can not reveal the estimated strengths of QPipe, which are the concurrent use of operators and data. One possible experiment would therefore be to execute queries concurrently. It would as well be interesting to see whether QPipe scales well when the database size grows. Therefore, another experiment to do would be to try scaling factors greater than 1, which was used for the experiments in this project. As these experiments only required in-memory operations and no I/O, the strengths of the storage manager QPipe relies on could not be exploited. This would be different when the database size grows, and scaling factors in the order of 50 or more are used.

8 Conclusion

The project's main goal was to implement the TPC-H benchmark in QPipe, a staged database system, in order to compare its performance with other DBMSs. As QPipe uses pre-compiled query plans and does not contain a query parser and optimizer, the first step was to find query plans that were used by other systems.

The query plan then had to be translated into C++ code, using operator classes and packets provided by QPipe. The queries were then executed in an isolated manner in QPipe and PostgreSQL, the reference system used in the scope of this project.

The first experimentation results did not meet the expectations. However, due to the fact that the initially planned project content had to be cut back by about 1/3, only few experiments have already been done. More tests in different conditions could reveal different results, which makes it impossible to draw a definitive conclusion about QPipe's performance compared with other systems. It can only be said that for the execution of isolated queries, where all data is in memory, QPipe executes most queries slower than the PostgreSQL reference system.

The project helped as well to debug the data creation code, as well as the random input generation and some utility functions. Furthermore, we probably revealed an error in the TPC-H specification document, concerning commit and receipt dates in the lineitem table.

Personally, this project helped me to learn another programming language (C++), and to get useful and interesting background about a new, promising database system design. Even though the single-query results do not show always performance benefits, I believe the current basic QPipe implementation will be useful start to optimize the system and run future performance tests.

References

[1] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In Fatma Özcan, editor, *SIGMOD Conference*, pages 383–394. ACM, 2005.

[2] TPC Benchmark^{TM}H. `http://www.tpc.org/tpch/default.asp`.

[3] Star Schema Benchmark. `www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[4] PostgreSQL - The world's most advanced open source database. `http://www.postgresql.org/`.

[5]

[6] Shore-MT: A Scalable Storage Manager for the Multicore Era. `http://diaswww.epfl.ch/shore-mt/`.

[7] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[8] TPC-H SF100 Non-parallel Plans, SQL Server 2008. `http://www.qdpma.com/tpch/TPCH100_Query_plans.html`.