JCC-H: adding Join Crossing Correlations with Skew to TPC-H

Peter Boncz[1], Angelos-Christos Anatiotis[2], and Steffen Kläbe[3]

[1] CWI, peter.boncz@cwi.nl
[2] EPFL, angelos.anadiotis@epfl.ch
[3] TU Ilmenau, steffen.klaebe@tu-ilmenau.de

Abstract. We introduce JCC-H, a drop-in replacement for the data and query generator of TPC-H, that introduces Join-Crossing-Correlations (JCC) and skew into its dataset and query workload. These correlations are carefully designed such that the filter predicates on table columns in the existing TPC-H queries now suddenly can have effects on the value-, frequency- and join-fan-out-distributions, experienced by operators in the query plan. The query generator of JCC-H is able to generate parameter bindings for the 22 query templates in two different equivalence classes: query templates that receive "normal" parameters do not experience skew and behave very similar to default TPC-H queries. Query templates expanded with the "skewed" parameters, though, experience strong join-crossing-correlations and skew in filter, aggregation and join operations. In this paper we discuss the goals of JCC-H, its detailed design, as well as show initial experiments on both a single-server and MPP database system, that confirm that our design goals were largely met. In all, JCC-H provides a convenient way for any system that is already testing with TPC-H to examine how the system can handle skew and correlations, so we hope the community can use it to make progress on issues like skew mitigation and detection and exploitation of join-crossing-correlations in query optimizers and data storage.

1 Introduction and Motivation

The past four decades of research into data storage and indexing, query execution and query optimization have yielded many research contributions, but also impacted a wealth of systems in broad ICT use, whose reach significantly surpasses database systems alone, as shown by the popularity of big data frameworks, such as Spark, for data science, ETL, machine learning and stream processing, which at heart are also powered by these techniques.

Benchmarks have helped significantly to quantitatively evaluate the properties of such techniques and have arguably played an important role in maturing the state-of-the-art in systems. By now, a scalable data management system with a SQL-like query language needs to meet a high bar of user expectations, set by previous database systems, and also codified in a number of database

benchmarks that it will be expected to be able to run. Significant benchmarks that have influenced the field of analytical database systems are TPC-H [1], TPC-DS [9], the Star Schema Benchmark [8] and BigBench [5].

Database benchmarks use synthetic data produced by data generators. This allows controlled generation of any desired dataset scale-factor (SF), which is useful for scalability analysis; yet regrettably so far this synthetic data has been rife with uniformity, in terms of (a) value distributions, (b) frequency distributions and (c) join fan-out distributions.[4] Any practitioner knows that in deployed use, as opposed to in benchmark tests, database systems face data that is typically skewed in all these aspects. To make matters worse, in real data, data tends to be highly correlated. A well known example of correlation would be a `CAR(brand, model)` table, where the predicate `brand=Porsche` and `model=Panamera` are correlated: after the selection on Panamera, there is 100% certainty that remaining tuples are Porsche. This type of correlations was long elusive for query optimizers using the independence assumption, but thanks to ample CPU power nowadays available, cardinality estimation is increasingly done by executing predicates on table samples, which catches any correlation within a single table. It was recently confirmed [7] that faulty cardinality estimation is the main problem for join-order optimization (which arguably is the most important query optimization problem), and as such the frontier for systems and for database research into this are correlations not within the same table, but across different tables. To continue the example, in a join of Panameras towards a `SALES(date,price,brand,type)` table, the optimizer would probably mis-estimate the cardinality of `extract(year from date) between 2000 and 2010` because the Panamera was introduced only in 2009. Between different referenced items, there can be a hugely different number of join partners (e.g. Panamera vs Golf or iPhones vs. Nokia handsets, lately). These sales examples exemplify *Join-Crossing-Correlations* (JCC), which is as far as we know a poorly supported aspect of reality in current data management systems, and certainly unsupported in the current generation of database benchmarks.

In this paper we describe a non-invasive variant of the well-known TPC-H benchmark, that makes it a much harder benchmark to execute efficiently by introducing *join crossing correlations* and *skew* in its data and queries. As we explained above, a join-crossing-correlation means that values occurring in tuples from one table, can influence the behavior of operations (joins,filters,aggregations) involving data from other tables, in a query that joins these tables. Barring the recent Interactive Workload for the LDBC Social Network Benchmark (SNB [3]), which is focused on short-running graph traversal queries rather than ad-hoc OLAP queries, there do not exist database benchmarks that test join-crossing correlations; and none that contain join skew.

The goals of JCC-H are as follows:

[4] The join fan-out distribution is the distribution of amount of join partners for values in a primary key (PK) column, towards a particular foreign key (FK) column.

1. to be a drop-in replacement of TPC-H in terms of data generator and query generator. The only difference being a single flag `-k`, that when passed to `dbgen` generates skewed/correlated data, and for `qgen` generates the skewed query variants (see point 5). The advantage of being a drop-in replacement is that many existing products and research prototypes already have TPC-H testing suites that can be leveraged, and also, the nature of the TPC-H queries is already well-understood by their development teams [1].
2. to introduce severe skew in all foreign key joins: for each referencing table, 25% of all tuples refer to just a handful of PKs (typically: 5). Having a handful of very frequent values is a known practical issue and one of the effects it causes is that if table partitioning is used, then the partition in which such a frequent value happens to fall will be larger than others. Another effect is that when joining or aggregating in a shuffled fashion, the worker responsible for the frequent values will be overloaded (receive a lot of network traffic, and have a lot of CPU work), which will lead to poor load balancing, unless specific anti-skew measures are taken by the system. A deterioration of speedup when e.g. comparing single-core to parallel execution is a good indication of the adverse effects of skew.
3. to correlate the join-fan-out skew created by our modifications to the data generator to (join-crossing) filter-predicates in the query. The correlation is carefully generated to create as much effect as possible on the existing 22 query templates. This required a thorough understanding of all 22 TPC-H queries and drawing up a plan how each query, given its existing filter predicates, could be affected by join-crossing correlations.
4. to be able to generate different query parameters that cause the queries to touch different data but behave identically performance-wise. Having such multiple parameter bindings for usage in concurrent query stream tests is a useful benchmark feature, as it helps guard against inflating the score of an ad-hoc query benchmark using query result caching: a query variant can be executed multiple times in a (throughput) test run, with different parameter bindings, but with equivalent results in terms of performance, so the results remain comparable.
5. to create for all or most of the 22 TPC-H queries two *query variants*:[5] one **normal** variant whose behavior closely resembles the behavior of the query on **default** TPC-H and one **skewed** variant that causes the skew to surface during runtime.
6. to make the single-table statistics of the columns from which the query parameters are derived look innocuously uniform. That is, it should force the query optimizer to understand join-crossing-correlations for it to predict that a different parameter value leads to very different behavior, as the values (used in equi- or range-comparisons) have similar frequencies in the column accessed by the filter predicate.
7. to design the "skewed" parameter bindings such that evaluating the query takes typically much more effort than for a "normal" parameter binding.

[5] As in [6] the two variants stem from exactly the same query template: the only thing that makes them different are the parameters that get pasted into the template.

It has been observed that for systems, workload scheduling could be eased if queries that affect very large volumes of data ("whale" queries, as opposed to normal "fish" queries) could be detected and handled differently. However, due to errors in cardinality estimation (which are often caused by join-crossing correlations and skew [7]) this is non-trivial.

A very important **non-goal** in JCC-H is to make TPC-H more "realistic", as has been done for instance in [2] by having the order-customer distribution over nations more real-life-like. While this may also interesting, we think that correlations that lead to unexpected and severe skew is a phenomenon that has been observed in practitioner lore so often that we consider introducing such correlation and skew a more important step in making TPC-H more "realistic" than trying to have the value and frequency distributions of it regions, nations, suppliers, customers, and orders to resemble real life more closely.

In order to fulfill all goals above, we must introduce skew and correlations primarily based on the predicates found in the 22 queries of TPC-H rather than on any overarching realism concerns.

This paper is organized as follows: in Section 2 we provide a detailed design of the JCC-H benchmark, while in Section 3 we describe our experiments we ran on multiple database systems, both single-server and MPP. In Section 4 we outline future work and conclude the paper.

2 Benchmark Design

In the remainder we assume the reader to be familiar with the TPC-H benchmark, and if not, advise the reader to first study its specification and/or [1].

In the JCC-H data generator we make use of *bijective permutation functions* based on a *linear permutation polynomial*, as also described in [4]. Given a key domain $K \in [0..N\rangle$, and a fixed, chosen, prime number P, these functions find a number X where $X * P \bmod N = 1$. This number is easily found using linear search and leads to a hash function: $h(K) : K * P \bmod N$. This is a perfect hash in that it delivers an outcome $\in [0..N\rangle$. Further, the function can be inversed simply using $h^{-1}(H) : H * X \bmod N$. The hash function can be made more random by adding and subtracting a constant: $h(K) : (K * P + C) \bmod N$ and $h^{-1}(H) : (H + N - C) * X \bmod N$. This is slightly different from [4], in that instead of $(H - C)$ we do $(H + N - C)$ and choose $C < N$ such that $(H + N - C)$ never underflows. This allows to use the fast C/C++ % operator (which is not a pure mod, but just a remainder).

We chose to modify the existing TPC `dbgen`, rather than to write a new data generator from scratch. The reason is that we want the tool to be an exact drop-in replacement, with exactly the same options and functionality. The TPC-H `dbgen` is arguably of a dated design, but it *can* generate data in parallel, or rather, it can generate all of its main tables in pieces, and thus with scripting that starts multiple data generators generating different pieces at the same time, parallelism is achieved (in a way that is independent of the parallelism framework).

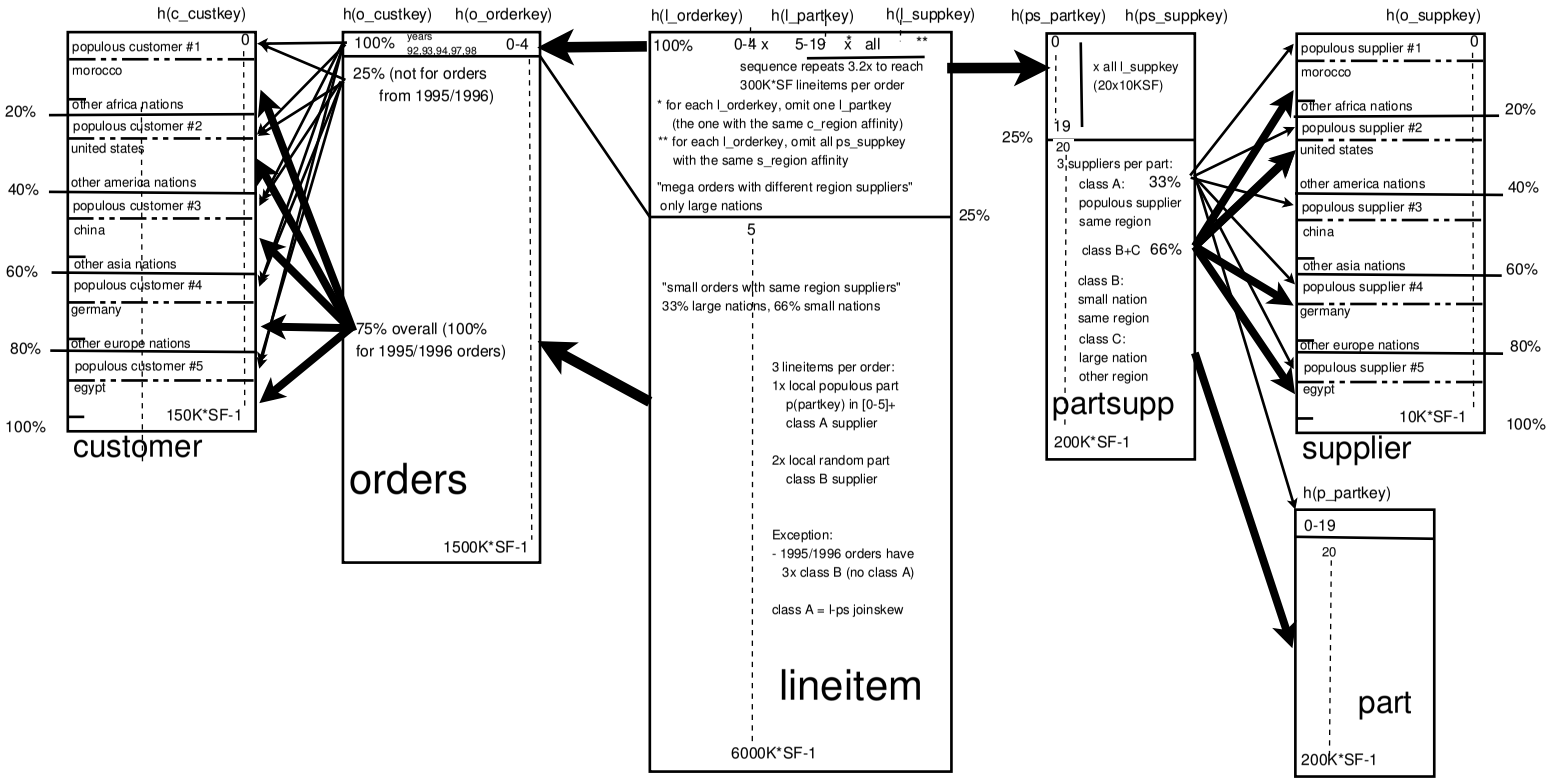

Fig. 1. JCC-H join connectivity

In order to introduce correlations and skew, in the data generator we decide what to generate based on the identity of the tuple we are generating. The TPC-H `dbgen` does this by passing the primary key of the table to the routine that generates the record. This number fulfills a similar function as the random seed used in modern parallel data generators such as PGDF [4]. PDGF introduces the concept of hierarchical seeds that follow the schema as the seed of each table referenced by a parent table depends on the seed of the parent. The issue of table dependencies is only partially addressed in TPC-H, because `dbgen` generates the part and partsupp tables at the same time, as well as orders and lineitem. For our purposes, though, this does not generate enough context to insert all correlations, and therefore we designed an elaborate mechanism of dependencies that start with `orderkey` and propagate down to all other keys, as described next.

2.1 Join Skew and Aggregation Skew

In JCC-H we have introduced join skew in all major joins[6]:

c-n there are 25 nations, evenly divided in 5 regions. We identify for each region a "large" nation to which 18% of all customers belong, and 4 "small" nations to which 0.5% of customers belong (5*(18+4*0.5)=100). The h(`c_custkey`) determines to which, by dividing the hash range in regions proportional to these percentages, as displayed in the left side of Figure 1. Further, each large nation has one customer (the first in the hash range) that is a "populous customer": it will have very many orders. These populous customers have a special country nation code in their `c_phone` phone numbers (the first two digits have values 40,50,60,70,80 – normally country codes in TPC-H are <40), to make them recognizable in Q22.

s-n similar to customer, suppliers are mapped to nations based on h(`s_suppkey`). There are also 5 populous suppliers, but they are not marked with a correlated column (which we did with `c_phone`).

o-c there are 5 populous orders, namely those with h(`o_orderkey`) < 5; these orders will have very many lineitems. They are recognizable in that their `o_comment` contains the string "1mine2 3gold4". For the other orders, in 25% of the cases a populous customer key is chosen (the decision is made determined by h(`o_orderkey`), in the other cases a normal customer is chosen). In all cases, the customer is chosen in such a way that h(`o_orderkey`) determines in which region the customer is located. Thus, by knowing `o_orderkey`, the data generator knows from which region the customer stems. We choose the customers from only 100K*SF out of the total 150K*SF customers, because in default TPC-H, one third of customers also does not have any orders.

ps-p according to our targets, 25% of partsupp should refer (via `ps_partkey`) to only 5 distinct parts. This mean every such part must have many distinct suppliers. But even if we take *all* suppliers, this would give 5*10K*SF=50K*SF

[6] In this paper, we abbreviate the foreign key joins of TPC-H (and JCC-H) using the first letters of the table name (ps for partsupp to distinguish it from p for part)). For example, with l-o we mean the join between linetem and orders.

different partsupps. Since (ps_partkey,ps_suppkey) must be unique, we use 20 populous parts, which link with all suppliers: 20x10K*SF=200K*SF, i.e. 25% of partsupp (whose size is 800K*SF).

ps-s For all non-populous parts, we generate three partsupps, i.e. choose three suppliers. This is done using a dependency of ps_suppkey on h(ps_partkey). We create an *affinity* between h(ps_partkey) and the supplier region. The three combinations are generated with three different formulas (called class-A, class-B and class-C suppliers), which guarantee that for a given partkey, class-A,B,C suppliers are distinct. To be exact, class A selects a populous supplier (which is always from a large nation) from the affinity region. Class-B selects a supplier from a small nation from the affinity region. Class C selects a supplier from a large nation from a distinct region. These three suppliers are evidently from different nations and therefore distinct.

l-o according to our target, 25% of the lineitem tuples have just 5 distinct l_orderkey values. For this, the 5 populous orders must consist of a lot of items (300K*SF).[7] To generate these, we use the 15 higher populous parts, and generate all suppliers from a large nation in a *different* region. As this amounts to (15 parts)*(0.8*5 regions)*(0.8*0.2*10K*SF suppliers) = 96K*SF, we fall short of the desired 300K*SF lineitems. Therefore, we repeat this sequence 3.2 times to get there. The skewed lineitems that we generate like this, have a few extra characteristics: l_quantity=51 (just above the normal maximum value), l_shipmode = "REG AIR", l_shipinstruct = "DELIVER IN PERSON" and l_returnflag = R.

l-ps Given that 25% (i.e. 0.25*6000K*SF = 1500K*SF) of the lineitems belong to populous orders, the other 1500K*SF-5 orders must consist of 3 lineitems (3*1500K*SF = 4500K*SF). For each of the 3 lineitems in an order, we must generate a partsupp reference. The two latter partsupps are so-called class-B partsupps, generated from a random partkey. The first partsupp in each non-populous order is the populous supplier matching the customer region, paired with one of the 5 populous parts (the one with matching region affinity). As such, all these first lineitems form just 5 different partsupp combinations, which is what we want for l-ps join skew.

Please note that when generating a table, we often choose a foreign key based on certain dependencies or conditions. These dependencies are computed in the hashed space of the parent key, and lead to choosing a hashed child key. For instance, as described above (o-c) in the generation of orders, we choose a customer such that from the orderkey we know the customer region (e.g. #region = h(o_custkey) mod 5). In order to actually generate a key (e.g. o_custkey) we use the inverse hash function $h^{-1}()$. This exploits the property in TPC-H that the key space is dense, so given any number H in $[0..N\rangle$ then $h^{-1}(H)$ must be a valid, existing key. The only exception to this rule are in fact orderkeys: there are holes in the space of orderkeys that TPC-H uses to generate inserts and deletes.

[7] A huge order indeed, and realism is not our primary target. However, if one orders all parts of an entire airplane, or aircraft carrier, it might still be realistic ;-)

However, JCC-H never needs to compute an orderkey as it is the root of the key hierarchy, therefore this is not an issue.

Finally, we introduce some correlated columns in the part table, for populous parts (p(p_partkey)<20). These have `p_brand` = "Brand#55", `p_size` = 1, `p_container` = "LG BOX", `p_type` = "SHINY MINED GOLD" and `p_name` = "shiny mined gold". Just doing this would introduce rather infrequent values for the latter two columns, as they would occur only 20 times. This would be easily picked up by the query optimizer in those TPC-H queries that have equality predicates on `p_type` and `p_name`. In order to hide these values in the statistics, we also give some non-populous parts these values, such that all individual column frequency distributions remain uniform. However, we guarantee that no non-populous parts have multiple of these values. Thus, only when selecting on a conjunction of these, the 20 populous parts will come out. This hiding of infrequent combinations is an example of a "simple" anti-correlated columns inside the same table. The fact that only 20 results come out, might be found by a multi-column histogram or using sampling (though likely the sample would be too small to contain a populous part). Still, even if query optimizers could detect this, this would only be stage one, as the second stage is to recognize the very different join-fan-out in the ps-p join that these populous parts have.

2.2 Filter Skew

In TPC-H the date dimension has a uniform value distribution. There is in fact a correlation between `o_orderdate` and the lineitem dates (the latter dates are within four months of the former). But, during the years, orders and lineitems are generated at the same pace.

JCC-H introduces a so-called Black Friday, which is one day in the year where there are many more orders. We actually chose to have this on Memorial Day, which is a fixed day (May 29), and on this day, 50% of all orders are placed. Please recall that absolute realism is a non-goal of JCC-H. But, we do want to test the effects of strong time skew in table generation. After generating `o_orderdate` we use the normal TPC-H mechanism to generate all lineitem dates based on it, so they follow after it within four months.

All 5 populous orders (25% of lineitem) are generated on Black Friday. Hence, even more than 50% of lineitems get ordered on Black Friday, because also 50% of the non-populous orders are from that date . This is done by moving a fraction of non-populous orders from their original random date to the Black Friday of that year. However, we do not do this in 2 out 7 the years, namely 1995 and 1996 (so 25% + 5/7 * 50% of 75% = 52% of lineitems was ordered on Black Friday). The reason is that 1995 and 1996 should be sanctuaries from join skew. These two years appear as constants (non-parameters) in default TPC-H Q7 and Q8. Recall that we want to generate two sets of parameter bindings: **skewed** bindings and **normal** bindings. By omitting generation of skew (i.e. class-A) in the lower 75% of lineitem (the area labeled "small orders from same-region suppliers" in Figure 1) during 1995-1996, we make it possible to avoid the o-c, l-c, and l-ps join skew by choosing date ranges from 1995 and 1996.

2.3 Query Parameter Generation

We now discuss how the JCC-H version of `qgen` substitutes parameter values into the 22 TPC-H query templates. We generate for each template parameters for two *query variants* [6]: one **normal** set of parameters where JCC-H tries to behave as close as possible to default TPC-H (no correlations, uniform distributions) and a **skewed** set of parameters, where all forms of correlation and skew come into play. Skewed variant generation is triggered using the `-k` flag of `qgen`.

Q1 because the query has no joins and is well-known for its full-scan behavior (it selects more than 95% of lineitem) and has few group-by values in the aggregation, there is no real opportunity for join-, aggregation- or filter-skew, so Q1 was left unmodified. Both normal and skewed queries use default parameters.

Q2 is a p-ps-s join with a `p_type` LIKE predicate. For the "skewed" query variant we set the parameter to suffixes of "SHINY MINED GOLD" (of at least 6 characters, e.g. "%INED GOLD". The "normal" parameters use default bindings. The effect of this is that skewed queries will select populous parts, and normal parameters non-populous parts.

Q3 has a date range that is lower-bounded on `o_orderdate` and upper-bounded on `l_shipdate`. Please recall that `l_shipdate` is always within four months of `o_orderdate`. Certain existing database systems that take join-crossing statistics into account and store the tables clustered or partitioned on date, will be able to unify these bounds on both lineitem and orders into bounded ranges in both tables (e.g. using MinMax indexes and noting which MinMax ranges of rows in orders and lineitem join with each other). This is the case already in TPC-H.
For the "skewed" query variants, we always choose a date range in 1993 around Black Friday. This will include the populous order from 1993[8], and thus join skew in l-o. For the "normal" variants, the date range lies in 1995. For orders from this date, there is no join skew in l-o and o-c. Please be aware that if a system uses table partitioning for lineitem, then the 5 partitions (or less) in which the populous orderkeys happen to fall will be larger than the rest. Therefore, the "normal" query variant will also experience scan-skew just for that reason, even if the tuples turn out to be not selected by the query. If additional measures are taken, such as clustering within the partition on a date, or sub-partitioning on date, then all other table areas than those corresponding to 1995 will be *skipped*, e.g. using partition pruning or by exploiting MinMax indexes. In that case, the "normal" variants of Q3 can avoid all skew.

Q4 contains a 3-month range restriction on `o_orderdate`. For the "skewed" query variant, this range is picked from the years 1993 and 1994. For the "normal" variant from 1995 and 1996, which avoids the l-o join skew.

Q5 identical to Q4, except that the range is one year long.

Q6 identical to Q5, except that `l_shipdate` is involved.

[8] Because there are seven years (1992-1998) and 5 populous orders, there are two years without populous order and these are 1995 and 1996.

Q7 in default TPC-H, there is a hard-coded (non-parametrized) two-year range restriction on `l_shipdate`. In JCC-H, the range boundaries become parameters, but for the "normal" query variant retain their old value (1995-1996). In the "skewed" variant the range is 1993-1994. The existing parameters are two nation names (between which trade is measured). In the "skewed" variant we pick two different large nations (from different regions, because there is only one large nation per region in JCC-H). For the "normal" variant, we pick two small nations from the same region. As has been described previously under l-ps join skew (and depicted in Figure 1), the class-B partsupps match up suppliers from small nations with customers from the same region (10% of which are from a small nation, 2% out of 20%). Hence the two variants both produce results, but their joins traverse disjunct joined tuples, where the "skewed" variant will hit strong o-c,l-o,l-ps join skew, but the "normal" variant not.

Q8 similar to Q7, the hard-coded date restriction on 1995-1996 (on `o_orderdate`) was turned into a parameter that for the "normal" query variant remains 1995-1996 and for the "skewed" variant is 1993-1994. In that case, the `p_type` equi-restriction becomes "SHINY MINED GOLD". This will select populous parts, hence focus on join-skew. This skew is absent in 1995-1996.

Q9 contains all joins, and only has a `p_name` LIKE restriction. Similar to Q2, it is set to a suffix of "shiny mined gold", for the "skewed" query variant.

Q10 the "normal" variant selects a 3-month `o_orderdate` interval starting on a day in the first two months of 1995 (this means it misses Black Friday, which in JCC-H is on May 29), and there is no join skew. The "skewed" query variant uses other years than 1995-1996 (so there is join skew) and uses a week enclosing Black Friday.

Q11 the "normal" variant uses a small nation whereas the "skewed" variant uses a large nation.

Q12 the "normal" variant uses a 1-year `l_receiptdate` restriction of 1995 or 1996, whereas the "skewed" variant uses 1993-1994 and includes "REG AIR" in the `l_shipmode` restriction.

Q13 the "normal" variant in the `o_comment` NOT LIKE restriction uses a variation of "%1mine2%3gold4%" where any of the digits can be omitted. This will eliminate all populous orders. The "skewed" query variant uses the normal parameter bindings.

Q14 and Q15 the "normal" query variant uses a 3-month `l_shipdate` restriction in 1995 or 1996 that excludes the months May-August, whereas the "skewed" variant uses 1992, 1993, 1994, 1997 or 1998 where the range includes these four months (hence it experiences both Black Friday filter skew and join skew).

Q16 the "skewed" query variant makes sure that the `p_size` IN range includes 1, whereas the "normal" variant ensures it never includes it. In both cases, asking for `p_brand` in-equality on "Brand#55" is avoided. The result is that the

"skewed" variant homes in on the populous parts, whereas the "normal" query variant only select non-populous parts.

Q17 the skewed variant ask for `p_brand` = "Brand#55" and `p_container` = "LG BOX" with an identical effect as in Q16.

Q18 this query cannot be easily parametrized, so a WHERE `l_quantity` < :2 parametrized restriction in the inner subquery was added, that in the normal case limits until 50 (which in default TPC-H is always the case) and in the skewed case until 100, so it will include the lineitems with join-skew (which have value 51 there).

Q19 the "skewed" variant restricts `p_brand` in the last disjunction to "Brand#55" and `l_quantity` to a range that includes 51. The effect is similar to Q16.

Q20 the "normal" variant uses a 1-year `l_shipdate` restriction of 1995 or 1996, whereas the "skewed" variant uses 1993-1994 and includes prefix of "shiny mined gold" in the `l_name` LIKE restriction.

Q21 the "normal" variant uses a small nation whereas the "skewed" variant uses a large nation.

Q22 the "skewed" variant include the `c_phone` area codes 30, 40, 50, 60, 70, 80 in the IN restriction, which selects populous customers. The "normal" variant uses the default parameter values which never will select such a customer.

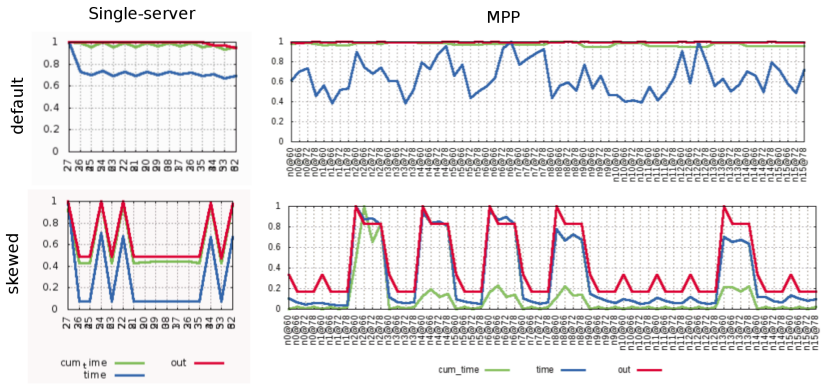

Fig. 2. Query 9: l-ps join load balance comparing JCC-H "skewed" with TPC-H "default" behavior on single-server and MPP systems (X-axis is workload per core).

3 Experiments

We ran experiments on Actian Vector (VectorWise) on a single-server machine and also using its MPP version VectorH (Vector on Hadoop) on a small cluster with relatively slow network. We also ran experiments on a faster cluster

with Hive. The single-server machine is a dual-socket Intel Xeon E5-2650 v2 @ 2.60GHz with in total 32 vCores (16 real), 256 GB RAM and four disks in RAID0. The disk configuration is not very relevant as all our results are with the data cached in-memory. The slow cluster consists of 16 nodes, each having a single-socket small machine i5-4590S CPU @ 3.00GHz (4 vCores each), a single 1TB magnetic disk and 16GB of memory and 1Gb ethernet. Actian Vector was version 5.1 and VectorH version 4.2. We used Hive 1.2.2 (Tez 0.8.5) and the fast cluster it ran on consisted of 8 nodes of each dual Intel Xeon X5660CPUs, 48GB RAM, and 10Gb ethernet. In all cases, the OS is Linux and both clusters ran Hadoop 2.7.3.

The Vector and VectorH results are listed in Table 2. Figure 3 summarizes the single-server numbers by normalizing query runtimes towards the default TPC-H query runtimes of the specific parallelism level (i.e. single-threaded or using 16 cores). We can observe at first that the green and blue bars, expressing runtimes of the 'normal" query variants, are always near 1, which means they behave very similar to the default TPC-H queries they are normalized to. This is an important requirement to fulfill goal 5.

Furthermore we can observe, comparing the "normal" queries with the "skewed" ones, that skewed query variants take significantly more effort to run than normal ones. One reason for this absolute difference is that indeed the skewed variant selects more data (goal 7: "whale" queries).

The disparity between normal and skewed gets worse when using all 16 cores. We have looked into detailed query profiles to establish the reasons for this. In certain cases, like Q2, Q17 and Q20, the reason is wrong optimizer choices in the "skewed" query variant. This is caused by cardinality estimates which are very much off due to the optimizer missing the join-crossing predicate correlations. When looking at the profile of the skewed variant of query 17 in detail, we can notice these estimation failures. The selection on part returns 20 tuples, which is about 0.1% of the estimated cardinality, but as we know, these are the "whale" tuples of the part relation. As a consequence, joining lineitem with these heavy hitters produces 1000 times more tuples than estimated. Aggregating on this join result produces again just 0.03% of estimated tuples, which is also a result of the wrong initial estimation on the part selection. So the used query plan does not seem to be the optimal one. Alternatively, in Q2 and Q20, the decision of query parallelization seem wrong. The system we test with determines the parallelism strategy during query optimization, and when it estimates (wrongly) that intermediate results will be very small it (mistakenly) chooses not to parallelize certain query subtrees anymore, because for small data volumes, the overheads of parallel execution tend not to pay off.

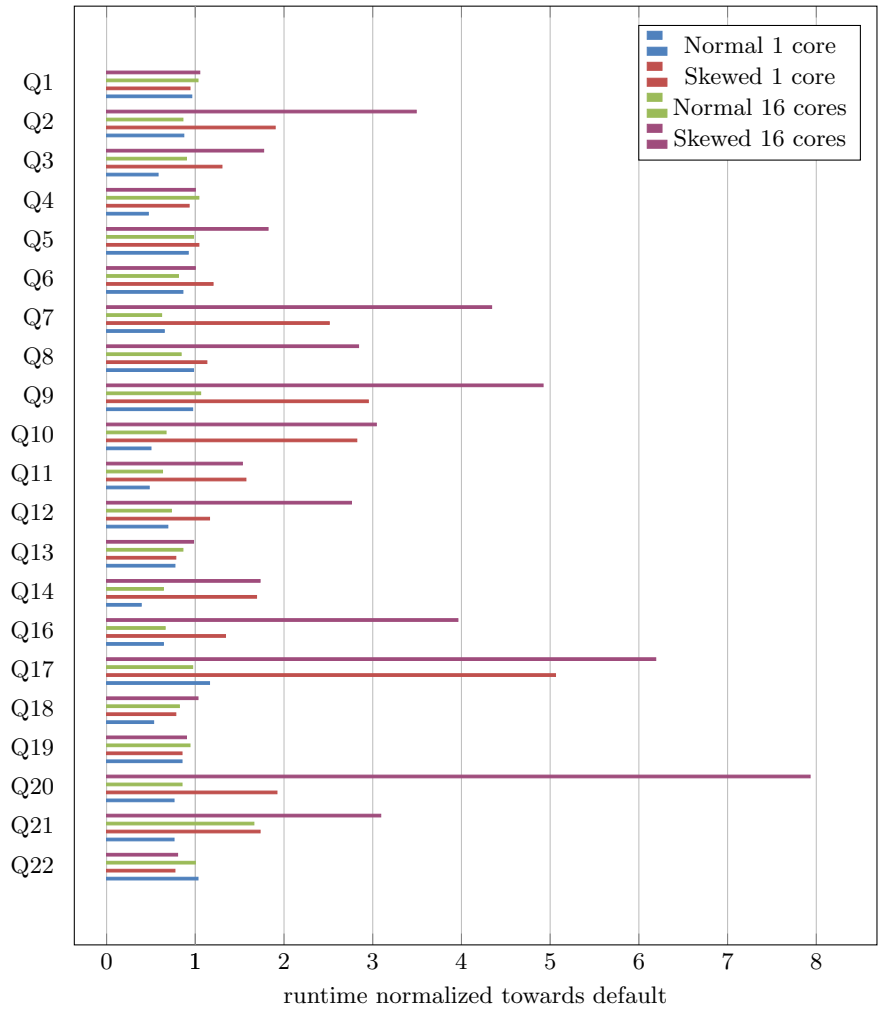

Fig. 3. single-server Vector(Wise) experiments with JCC-H: runtime normalized towards default TPC-H with the same amount of cores

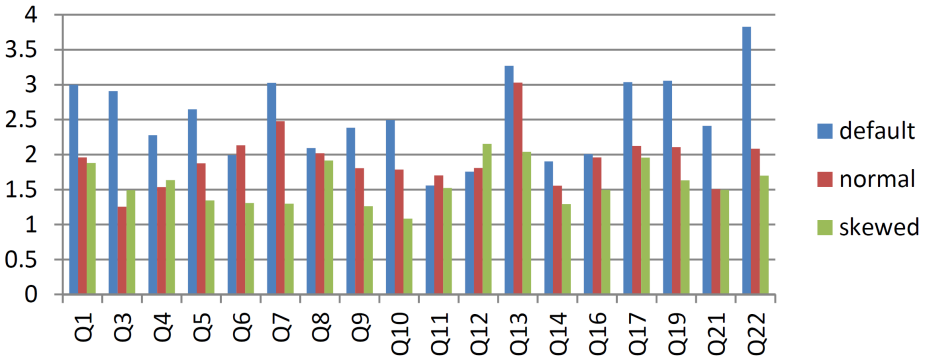

Fig. 4. Hive: scalability achieved when hardware is scaled x4 (2 to 8 nodes) for 100GB TPC-H (default) and JCC-H (resp. normal and skewed query sets).

The second reason why the difference between skewed and normal gets bigger with more parallelism is indeed skew. The query profiles we examined had very strong scan skew, filter skew, aggregation skew and join skew. As an example, we have a more detailed look at query 9 in Figure 2. These plots show various characteristics (time and output size) of the execution of the most expensive join operator (the l-ps join), per active core. On the single-server system as well as on the MPP system, the join in default TPC-H produces a balanced join fan-out, shown by the red lines. In contrast to that, the join of the skewed variant exhibits five peaks. While 5 threads return about 58 million tuples, the remaining 11 threads return only about 27 million results. The same can be observed in the MPP system: five nodes produce about 11 million tuples with each of their four threads, while the remaining 11 nodes return about 2 million results per thread. So, in the skewed query variants, the overall runtime of the join operator is dominated by the threads/nodes that process the parts with the peak cardinalities, causing the whole operator to run slower than in the default query variant. Overall, we observe in Figure 3, that the impact of the skewed query variants on the parallel experiments (purple bar) is much higher than on the single-threaded runs (red bar).

The most striking aspect of the VectorH results is that they show that the skewed queries relatively become much more expensive, with Q2,8,9,10,17,20 running into tens of seconds. Also, scalability on these queries is fully gone and some even run slower on more hardware. We think this shows that when data movement over the network becomes a factor, the penalties for skew become higher. Further, MPP systems must do static partitioning of tables, and the fact that e.g. 5 of the `lineitem` partitions are much larger due to skew even makes queries without joins affected: even though Q1 on any database system tends to scale perfectly, the fact that 5 tasks must now scan a significantly bigger partition than the other tasks causes imperfect scaling.

The Hive results are in table 1, where we tested scalability by running the JCC-H and TPC-H query sets on 2, 4 and 8 nodes. Figure 4 shows that while TPC-H scales reasonably from 2-8 nodes (though less than a factor 4), especially the JCC-H "skewed" query set scales very badly. The "normal" query set scales less badly, but still not so good. The fact that table partitioning is affected by skew, even affects queries with non-skewed query processing behavior, since the skewed partitions (read as Parquet files [9]) need to be scanned and this is an important factor in Hive performance.

4 Conclusion

In this paper, we have introduced a new variant of TPC-H, named JCC-H, that adds correlations and skew to TPC-H.[10] JCC-H was carefully designed to include

[9] We used the same DDL as in the VectorH SIGMOD paper: https://github.com/ActianCorp/VectorH-sigmod2016.

[10] The code for JCC-H can be downloaded from: http://github.com/ldbc/dbgen.JCC-H

very severe join skew as well as filter skew. Moreover, these skewed effects are observed by the original 22 TPC-H queries only if special parameters are given to them. That is, for each of the 22 queries there is a "skewed" query variant and a "normal" query variant (the normal variant is generated by default by `qgen`, and the skewed variant when passing `-k`). The decision to make JCC-H a drop-in replacement for TPC-H has a number of advantages, as JCC-H can be dropped into any existing benchmark testing pipeline, and its queries are well understood by practitioners.

A disadvantage of focusing on the existing 22 TPC-H queries is that there may be interesting and relevant query patterns where join-crossing correlations and related skew have even more significant effects. This belief is informed by the fact that access path selection for TPC-H is relatively straightforward. As such, it is also of immediate interest to devise additional query patterns for JCC-H where join-crossing correlation will affect the join execution order, or the (non-)use of unclustered indexes.

Acknowledgments. This paper is a result of the "Parallelism and Skew" working group at Dagstuhl seminar 17222 (Robust Performance in Database Query Processing). We would like to thank group members Johann-Christoph Freytag (HU Berlin), Alfons Kemper (TU Munich), Glenn Paulley (SAP Canada) and Kai-Uwe Sattler (TU Ilmenau) for their contributions. The research of A.C. Anadiotis was partially funded by the Swiss National Science Foundation, Project No.: 200021_146407/1 (FN–X–Core).

References

1. Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, 2013.
2. Alain Crolotte and Ahmad Ghazal. Introducing skew into the TPC-H benchmark. In *TPCTC*, 2011.
3. Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network enchmark: Interactive workload. In *SIGMOD*, 2015.
4. Michael Frank, Meikel Poess, and Tilmann Rabl. Efficient update data generation for DBMS benchmarks. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 169–180, 2012.
5. Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, 2013.
6. Andrey Gubichev and Peter Boncz. Parameter curation for benchmark queries. In *TPCTC*, pages 113–129, 2014.
7. Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
8. Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, 2009.
9. Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. Why you should run TPC-DS: a workload analysis. In *VLDB*, 2007.

	TPC-H 2 nodes	JCC-H 2 nodes normal	JCC-H 2 nodes skewed	TPC-H 4 nodes	JCC-H 4 nodes normal	JCC-H 4 nodes skewed	TPC-H 8 nodes	JCC-H 8 nodes normal	JCC-H 8 nodes skewed
Q1	817.93	985.09	966.68	531.55	696.46	700.11	272.89	502.67	514.47
Q3	306.69	311.02	415.10	177.00	287.87	344.12	105.49	247.68	278.32
Q4	82.62	83.86	92.39	52.47	57.69	81.99	36.30	54.66	56.45
Q5	183.27	159.29	291.59	114.55	111.35	226.38	69.17	84.87	216.59
Q6	60.53	53.63	75.89	46.6	34.63	54.84	30.39	25.12	58.02
Q7	822.66	690.25	1237.07	530.13	410.23	1014.77	271.98	278.45	952.37
Q8	1306.56	1568.00	2402.17	882.11	1150.03	1667.44	624.6	776.87	1254.27
Q9	449.14	406.67	1945.30	371.25	278.72	1789.56	188.48	225.3	1540.54
Q10	243.05	232.23	448.57	191.21	168.56	414.79	97.40	130.04	413.74
Q11	41.27	29.52	68.58	31.92	24.05	49.37	26.47	17.34	45.08
Q12	37.48	34.49	42.33	84.04	25.35	24.53	21.35	19.07	19.67
Q13	139.15	137.98	142.43	95.27	74.26	97.69	42.57	45.58	69.85
Q14	58.73	39.73	82.54	68.38	33.86	58.68	30.85	25.57	63.88
Q16	74.17	73.14	126.20	51.28	44.11	104.99	36.95	37.32	84.34
Q17	855.21	815.78	909.09	438.17	501.61	622.31	281.64	384.19	464.98
Q19	1263.05	1332.35	1484.97	751.50	818.59	1041.23	413.44	632.50	908.99
Q21	1076.77	1023.01	1051.54	631.46	634.80	645.62	446.68	679.46	704.04
Q22	86.34	81.57	84.39	52.93	48.31	56.92	22.55	39.15	49.64

Table 1. Experiments on the faster cluster with Hive: query runtime in seconds (SF=100)

	Vector, 16-core 256GB single-server						VectorH, 16x(2-core,4SMT 16GB) cluster					
	TPC-H 1 core	JCC-H 1 core normal	JCC-H 1 core skewed	TPC-H 16 cores	JCC-H 16 cores normal	JCC-H 16 cores skewed	TPC-H 16x1	JCC-H 16x1 normal	JCC-H 16x1s skewed	TPC-H 16x4	JCC-H 16x4 normal	JCC-H 16x4 skewed
Q1	14.96	14.40	14.11	1.12	1.16	1.18	0.95	1.36	1.37	0.30	0.91	0.96
Q2	3.23	2.80	6.13	0.87	0.75	3.04	0.62	0.66	33.45	0.59	0.46	29.18
Q3	1.17	0.68	1.52	0.39	0.35	0.69	1.43	1.72	1.21	0.57	2.36	2.57
Q4	0.73	0.34	0.69	0.23	0.24	0.23	0.06	0.10	0.08	0.06	0.05	0.07
Q5	3.53	3.24	3.66	0.66	0.65	1.20	1.49	1.08	4.62	1.09	1.51	3.91
Q6	0.94	0.81	1.13	0.27	0.22	0.27	0.14	0.10	0.23	0.07	0.07	0.22
Q7	3.54	2.29	8.90	0.64	0.40	2.78	1.74	0.69	4.97	1.02	0.79	6.66
Q8	3.94	3.86	4.46	0.69	0.58	1.96	0.67	0.79	29.43	0.62	1.45	48.02
Q9	18.80	18.23	54.32	2.10	2.22	10.34	6.64	5.98	39.68	3.85	5.67	39.02
Q10	2.62	1.31	7.40	0.55	0.37	1.67	1.43	3.36	20.75	4.63	5.48	20.47
Q11	1.84	0.89	2.89	0.38	0.24	0.58	0.49	0.12	0.44	0.10	0.10	0.55
Q12	1.88	1.29	2.18	0.33	0.24	0.91	0.55	0.21	0.60	0.10	0.10	2.78
Q13	19.42	15.04	15.17	1.27	1.09	1.24	1.69	2.18	3.23	1.93	3.99	4.05
Q14	2.44	0.96	4.13	0.45	0.29	0.78	0.70	0.28	3.06	0.26	1.06	2.44
Q16	6.27	3.99	8.41	0.82	0.54	3.25	1.01	0.74	7.73	1.02	2.58	8.75
Q17	4.33	5.04	21.9	0.58	0.56	3.39	0.56	0.54	19.81	0.72	1.39	18.24
Q18	5.43	2.89	4.21	0.60	0.49	0.62	0.76	0.46	0.73	0.25	0.40	0.39
Q19	11.48	9.72	9.73	1.05	0.99	0.95	1.01	1.00	0.70	1.06	0.92	0.81
Q20	4.45	3.39	8.53	0.60	0.51	4.76	3.88	1.91	77.34	4.77	2.08	75.72
Q21	16.21	12.33	27.85	1.18	1.96	3.65						
Q22	4.77	4.90	3.67	0.75	0.75	0.60	1.52	1.81	1.60	1.85	3.99	3.69

Table 2. Vector (left) and Vector-on-Hadoop (right) query runtime in seconds (SF=100). Note: the Vector and VectorH results are not comparable in absolute terms (and this is not the point of these experiments), since the machines in the Hadoop cluster used for VectorH are older and slower than the single-server machine used for Vector. We can see that relatively, the skewed queries on MPP system VectorH bear a heavier performance and scalability penalty than on the single-server system.