Trading off Accuracy for Speed in PowerDrill

Filip Buruiană[§], Alexander Hall[*], Reimar Hofmann[†], Thomas Hofmann[‡], Silviu Gănceanu[*], Alexandru Tudorică[§]
[*]Google {alexhall, silviu}@google.com [†]HS Karlsruhe. reimar.hofmann@hs-karlsruhe.de
[‡]ETH Zurich. thomas.hofmann@inf.ethz.ch [§]Wholi {filip.buruiana, tudalex}@gmail.com

Abstract—In-memory column-stores make interactive analysis feasible for many big data scenarios. In this paper we investigate two orthogonal approaches to optimize performance at the expense of an acceptable loss of accuracy. Both approaches can be implemented as outer wrappers around existing database engines and so they should be easily applicable to other systems.

For the first optimization we show that memory is the limiting factor in executing queries at speed and therefore explore possibilities to improve memory efficiency. We adapt some of the theory behind *data sketches* to reduce the size of particularly expensive fields in our largest tables by a factor of 4.5 when compared to a standard compression algorithm. This saves 37% of the overall memory in PowerDrill and introduces a 0.4% relative error in the 90th percentile for results of queries with the expensive fields.

We additionally evaluate the effects of using *sampling* on accuracy and propose a simple heuristic for annotating individual result-values as accurate (or not). Based on measurements of user behavior in our real production system, we show that these estimates are essential for interpreting intermediate results before final results are available. For a large set of queries this effectively brings down the 95th latency percentile from 30 to 4 seconds.

I. Introduction

In recent years with the steep rise of "big data" generated by websites and services, the interest across the industry to do large-scale data analysis has skyrocketed. This has fueled the large amount of exciting research in the area of column-stores—databases which store each column of a table separately to enable faster analytics. In-memory column-stores have pushed out even further the limits with regards to speed, scalability and flexibility. However, even with today's affordable RAM sizes it is not always possible to fit all relevant data into memory.

This is the situation of the data analysis tool PowerDrill [1], available for internal users across Google since 2009. Each month it is used by about 1800 users sending out 15 million SQL queries. One of our top users, after a hard day's work, spent over 6 hours in the tool, triggering roughly 12 thousand queries. This may amount to scanning as much as 525 trillion cells in (hypothetical) full scans.

PowerDrill is used within Google for a large variety of data exploration purposes, e.g. for identifying causes of alerts and spam analysis, where newly emerging patterns need to be identified and judged. This process is inherently iterative: An analyst defines an initial query, looks at the results, develops a hypothesis, then defines a new query to (in-)validate the hypothesis and so on. Often it takes many iterations until a conclusion is reached, so query response times directly translate into efficiency and satisfaction of the analyst.

A. Contributions

In this paper we propose two orthogonal approaches for reducing memory usage and user perceived query latencies. Both performance optimizations become possible if one is willing to sacrifice accuracy. This is motivated by the observation that for data discovery tasks often a qualitative picture is sufficient, i.e., finding patterns, trends and correlations are enough for making conclusive arguments. In the generic case, no strong guarantees are provided on the preciseness of the results. However, the methods can be employed on a large range of SQL-like queries, whenever the analyst prefers faster results which can be used to already validate hypotheses.

Both optimizations had to be easy to integrate in existing systems with little overhead in complexity. The simplicity with which our approaches can be implemented around existing production grade systems can be seen as the main contribution of this paper. In more detail:

- In Section II we adapt some of the theory behind data-sketches to approximate certain types of queries with "expensive" fields by replacing their values with hashes. In our production environment, this introduces insignificant losses of accuracy (0.01% median relative error, 0.4% error in the 90th percentile) for these queries while leading to huge RAM savings of 4.5x compared to an industry standard, high performance compression algorithm. Since memory is the limiting factor in executing queries at speed, the 13 TB of RAM made available can be used to improve the overall efficiency.
- In Section III we show usability related aspects of using sampling with variable sample size (in the case of our system, we have intermediate query results over successively growing subsamples). We propose a heuristic to obtain accuracy estimates at virtually no extra computational cost. Based on measurements of user behavior, we show that marking results as trustworthy or not is essential for enabling users to start with the interpretation based on intermediate results, without waiting for the final ones. For a large set of queries this effectively brings down the 95th latency percentile from 30 to 4 seconds.

Generally, since PowerDrill operates as a (Google-internal) service, we are able to perform many measurements and observations on the real, operational system and real users. These results provide insights that cannot easily be predicted from theory, because—as in many other systems with complex compression and memory caching taking place— performance depends on usage patterns and statistical distributions of the underlying data in complex ways.

B. Query Constraints for High Accuracy Results

When high accuracy is required, the trade-off between efficiency and results' quality should be done in a controlled manner. In such a case, our optimizations can be employed for some classes of queries only. More specifically, they can be applied on group-by queries aggregating only by sum, count or average (the latter can be inferred from the first two). It is also important that the query sorts in decreasing order by either count or some other metric in the query that increases with count (e.g., a sum of a field with positive values), for reasons that will be detailed later. A query satisfying all these constraints can be written in SQL as:

```
SELECT dimension as gby,
       COUNT(*) AS metric_1,
       SUM(field) as metric_2,
       AVG(other_field) as metric_3,
       ...
 FROM table
 WHERE <some_conditions>
 GROUP BY gby
 ORDER BY <some_metric_above> DESC
 LIMIT K
```

An arbitrary number of metrics can be used. Throughout this paper, such a query will be referred to as a TOP-K query, and the values of the group-by-field will be referred to as *entities*. In our case, and probably in others as well, this still has wide applicability: In PowerDrill, of the 15 million analyzed queries in the usage logs, 66% were TOP-K.

A special note needs to be made about joins. In our system and at our scale, for efficiency reasons data-sources are usually pre-joined. Every record in the resulting data set is represented by a hierarchical data structure, typically encoded using protocol buffers ([2]). Joining at serving time is also possible in real-time in PowerDrill but is limited in scope and not frequently used. The next sections thus make the assumption that data sources are pre-joined, meaning that each query can be considered to have a single table only.

C. Related Work

For an introduction to OLAP and basic techniques applied in data-warehouse applications, see Chaudhuri and Dayal [3].

For an overview of recent work on column-store architectures, see the concise review [4] and references therein. The excellent PhD thesis by Abadi [5] can serve as a more in-depth introduction on the topic.

Recent research in this area includes, e.g., work on how super-scalar CPU architectures affect query processing [6], tuple reconstruction [7], compression in column-stores [8], [6], [9], and a comparison to traditional row-wise storage [10]. Kersten et al. [11] give a more open ended outlook on interesting future research directions.

The ever growing set of commercial column-store and open-source systems, e.g., [12], [13], [14], [15], [8], [16] is further indication of the effectiveness of this paradigm.

Melnik et al. [17] introduced Dremel, another data analysis system used at Google, to a wider audience. Similar to PowerDrill, Dremel's power lies in providing interactive responses to ad hoc SQL queries. While PowerDrill is optimized for keeping major parts of the relevant data in memory, and thus is only used for a few selected data sources, Dremel is optimized for streaming over petabytes of data from disk (stored, e.g., on GFS [18]) in a highly distributed and efficient manner. Therefore Dremel is being used over thousands of different datasets, but with higher latency than PowerDrill on its supported data sets. Melnik et al. also give a nice overview of data anlysis at Google and how interactive approaches like Dremel's complement the offline MapReduce [19] framework.

Returning intermediate results to the users has been explored in the literature from diverse directions, for a selection of related papers see [20], [21], [22], [23], [24]. For more on data sketches, see [25], [26], [27].

II. Efficiently encoding expensive columns

This section is divided into five parts. In subsection II-A we show that memory is the limiting factor in executing queries at speed. In order to serve more from memory and thus decrease query execution times, we aim to improve the data encoding scheme. In II-B we identify a class of fields which fill a significant portion of the caches and proceed by looking into ways of reducing their size. Replacing values with IDs is one option, discussed in II-C1. Unfortunately, due to constraints mentioned in II-C2, this is challenging, so we propose using hash values instead. We show in II-D1 how a *data sketches* approach could be employed to reduce the effects of collisions introduced by hashing. In II-D2 we describe the final approach, which requires only half of the memory needed in the data sketches solution, but deviates considerably from the way sketches are normally implemented. The last subsection, II-E, presents the results we obtained in practice, from a real query load, both in terms of accuracy and memory savings. A more theoretical analysis is conducted in Appendix A.

A. Memory As Performance Bottleneck

Even though PowerDrill runs distributed across thousands of machines and uses a column-store architecture, acceptable response times cannot be achieved if data needs to be read from disk. Therefore in memory caching is used. Data is kept in RAM in a compressed format, but nevertheless the available memory is not sufficient to hold entire tables. So while PowerDrill is not a pure in-memory tool, one does not have to read all relevant data from disk in processing a query either. Table I shows that under its real usage conditions, 68% of the queries can be answered completely from memory. For the remaining 32% some amount of data needs to be read from disk. The latter queries are *26 times slower* in the 95th latency percentile than pure in-memory queries. Obviously, obtaining more memory or increasing memory efficiency directly translates into large speedups.

B. Expensive fields

Even within the same table, the amount of memory needed to store different fields varies greatly, depending on their number of distinct values, compressability and type. With regards to type, one could classify fields as follows:

TABLE I
LATENCIES FOR 15 MILLION QUERIES EXECUTED BY POWERDRILL, TOUCHING AND NOT TOUCHING DISK. TYPICAL TABLE SIZES: RANGING FROM 10^8 TO 10^{11} (100 BILLION) RECORDS

Query	frequency	latency	
		median	95th percentile
fully in-memory	68%	0.04 s	2.96 s
touching disk	32%	0.54 s	78.80 s

TABLE II
MEMORY (IN TB) FOR SELECTED COLUMNS, USING NO COMPRESSION (ORIGINAL), STANDARD LEMPEL-ZIV-OBERHUMER COMPRESSION (LZO), POWERDRILL'S PROPRIETARY FORMAT (PD), AND LZO COMPRESSION ON THE PROPRIETARY FORMAT (PD+LZO)

Field type	size in TB			
	original	LZO	PD	PD+LZO
numeric				
average memory (per field)	0.96	0.22	0.26	0.19
total memory (all fields)	8.64	1.97	1.83	1.67
low cardinality				
average	0.94	0.23	0.14	0.05
total	33.12	7.95	4.89	1.86
high cardinality				
average	3.35	1.21	1.87	0.94
total	63.69	22.91	35.54	17.85
all fields				
average	1.67	0.52	0.69	0.34
total	105.45	32.83	43.66	21.38

- Numeric fields, representing metrics like `cost`.
- Low cardinality (string) fields. These are mostly what would be called *dimensions* in OLAP, i.e., something by which results are grouped or filtered. A typical example is `country`.
- High cardinality (string) fields. A typical example is `user_query`, i.e., the text a user enters in Google's search box. These fields take on a large number of possible distinct values and are usually longer strings.

When a query groups by a high cardinality field, returning all groups is infeasible in practice. In this case, meaningful queries always order the result according to some metric and limit it to a certain number K of rows (so called TOP-K-queries). Example: 'The ten most frequent `user_queries`'.

PowerDrill uses a proprietary, optimized format to keep data in memory (see [1]). A direct comparison between this and other compression algorithms is not fair because unlike the latter, PowerDrill's format allows indexed access to subsets of the data without having to decompress everything. When data is kept in memory for longer periods of time without being accessed, it is additionally compressed using LZO. This further reduces the size of the data structures in caches but requires an extra decompression step at access time. The size of the PD+LZO format determines the cache capacity and is the relevant quantity for the purposes of this chapter, minimizing disk access. Table II shows the memory consumption for representative fields of different types, in PowerDrill's most important tables, including the PD+LZO size.[1]

[1]For practical reasons not all of the 1000's of fields in PD were evaluated but rather for each field type a subset of the most frequently used fields (i.e. that appear in at least N queries, where N was arbitrarily set) was selected for evaluation. The compiled list has 9 numeric, 35 low cardinality and 19 high cardinality string fields.

It is observed that high cardinality fields are the largest in uncompressed format. Unfortunately, compression rates are much worse for them, too, in all evaluated formats. In PD+LZO-format such a field occupies on average about *18 times more memory* than a low cardinality field. For the rest of this chapter we will therefore refer to the high cardinality fields as *expensive*. In absolute numbers, the size of all selected expensive fields combined amounts to 18 TB in PD+LZO compressed format—a disturbing number considering that they reflect only 19 out of thousands of fields for which in total there are only 35 TB of memory available in PowerDrill. Optimizing the RAM consumption for these fields is therefore key to reducing the pressure on the in-memory caches.

C. Hashing Expensive Fields

1) Mapping Strings to IDs: To reduce the memory consumption of expensive fields one can replace their values with IDs, by defining a mapping function from the set of string values occurring in the field to the ID range.

For each expensive field, a new ID-field is added to the table, obtained by applying the function on all the original string values. If the expensive field is used in transformations that require computations on the string values themselves (e.g., regular expressions, *contains* operations), then queries are executed as received from users. From over 200,000 checked queries, only 18% fall into this category. Otherwise, all references to the expensive field are replaced with references to the corresponding ID-field.

On disk, the ID column is stored *in addition* to the original column, increasing the disk space. This is needed to still support queries doing string transformations. For all the other queries (which, as presented, are the large majority), only the ID column needs to be fetched into RAM. Since the number of bits required to encode IDs is much smaller than the number of bits required for lengthy strings, less is loaded into RAM (and thus memory is saved).

Before showing the results to the user, IDs need to be translated back to their corresponding string values. To achieve this, an inverse function is needed, based on a reverse dictionary from all the string values that appear in the data. One dictionary is created per *(table, expensive_field)* and does not need to be query specific: Indeed, just mapping all the IDs to their corresponding strings is enough for the translation. To allow for extensibility (i.e., supporting new fields easily), the map is built and maintained "on the fly", during query execution time, instead of being pre-processed. The first query over a table snapshot for which the map hasn't been updated pays the price of updating it and will be (much) slower. All subsequent queries don't need to do any extra work and can be very fast.

Access to the reverse dictionary is required only at the very end of the query execution, and only for those values which are actually handed to the user. Since these are bounded by the limit in the query, only few lookups are needed, which can be handled quickly - even from disk. For this reason, this dictionary does not need to be stored in memory: It can be kept in a persistent storage instead.

2) Trade-offs in Choosing the Function: The properties of the function used above raises different problems that need to be addressed. For instance, an injective function greatly simplifies the translation, since each integer has exactly one associated string, but makes it hard to create the reverse dictionaries. In our tables, some fields have tens of billions of unique values. These fields are processed in parallel, from thousands of machines. Preserving uniqueness in a distributed setup processing such a large volume of data and making this fast enough to work in real-time is extremely challenging: ID consistency needs to be preserved in case of machine failures, there is a significant overhead per value processed (i.e., for each value there needs to be an extra communication over network with the entity managing IDs), etc.

For these reasons, we use a function that is not injective, i.e., a hash function. This immediately raises the problem of handling collisions. The probability of collisions can be decreased if large output ranges are used, but this increases the number of bits needed to encode the values, and thus the memory that is being used. A reasonable trade-off between memory that is consumed and accuracy of the results produced needs to be made, and doing this consciously first requires an understanding of what the effects of collisions are.

3) Effects of Hash Collisions: The collisions have two effects. First, the values of the metrics (count, sum, etc) can "by-catch" other entities with the same hash—potentially overestimating the correct result. Second, when translating back a query result, it is to be expected that some hash values cannot be translated back uniquely. In such a case, all possible string values have to be presented to the user by concatenating them, e.g., with *OR*.

Both effects make the query results approximate. In practice collisions can not be made an extremely rare event without using prohibitively long hashes which use too much memory. However, there is a way to diminish effects of collisions while still obtaining significant memory savings.

D. Multiple Hashes For Resolving Collisions

1) Data Sketches Inspired Approach: There is a relationship between the mapping approach described in section II-C1 and a class of data structures called data sketches ([25], [26], [27]). More specifically, a *count-min-sketch* [27] (short: *CM-sketch*) is one of these data structures; it is used to represent a multiset. The result of a TOP-k query using *count* for aggregation can be stored in a CM-sketch—the entities being the members of the set, and the counts being their multiplicities. A CM-sketch uses one or more hash tables for storing the multiplicities of its members. Its main advantage is that it can cheaply be updated incrementally as elements stream in, and that—depending which hash size one chooses—it may require far less memory than alternative data structures.

Like the mapping approach from section II-C1, count-min-sketches suffer from hash collisions: Each cell of each hash table stores the *sum* of the multiplicities of all multiset members mapped to the cell. This value is used as an estimator of the multiplicity of each member mapped to the cell—potentially overestimating it. If multiple hash tables with independent hash functions are used, then the multiplicities are retrieved from each hash table, and the *minimum* of these multiplicities is used as the estimator—reducing the overestimation.

It is easy to see that a CM-sketch with only one hash table delivers the same results and thus the same estimation errors as the mapping approach. For the CM-sketch there is a well known error bound [27], which can be rewritten as:

$$err \geq 0 \text{ (with certainty), and}$$
$$P\left(err \geq k \cdot \frac{N}{2^b}\right) \leq \left(\frac{1}{k}\right)^d \quad \forall k > 0 \tag{1}$$

where err is the returned minus the correct result, N is the cardinality[2] of the multiset, d is the number of hash tables used, and 2^b is the size of each of the hash tables.

One can transfer the method of using multiple independent hashing functions to the mapping approach by adding multiple ID-fields based on different hashing functions for a single expensive field. Query results are then computed for each of the ID-fields, and then for each entity in the result set the *minimum* of the counts of all corresponding ID-fields is returned as the result.

For CM-sketches the optimal tradeoff between b and d for achieving a given accuracy, that is between fewer larger hash tables and more smaller ones, is well known as a result of equation 1. These results are not applicable to the mapping approach, however: In a CM-sketch, the hash values are used as an index into a hash table for which memory has to be allocated, and the total required memory scales with $d \cdot 2^b$. In the mapping approach from section II-C1, the hash values are stored in a database field. The required disk space (which also determines the memory required for caching this field in memory)[3] scales with $d \cdot b$ (but with a large constant factor of N), so the price for increasing b is much lower than in a CM-sketch.

It is easy to show that for the mapping approach the optimal tradeoff between b and d is always using only one hash table: One can rewrite equation 1 to bound for a *given amount of available memory* m and a given desired accuracy h the probability with which this accuracy is exceeded—as a function of the number d of hash tables between which the available memory is split. Denoting the available memory in bits by $m := N \cdot d \cdot b$, and $h := k \cdot \frac{N}{2^b}$ equation 1 becomes:

$$P(err \geq h) \leq \left(\frac{N}{h}\right)^d \cdot 2^{-m/N} \quad \forall h > 0$$

Since $h << N$ for all meaningful levels of accuracy h, it is obvious that for all values of h and m the bound is tightest for $d = 1$ (d being a natural number).

2) Multiple Partitioned Hashes: The idea from count-min-sketches of using multiple hash functions can be adapted in our case, in a way that does not increase the memory requirements while making it possible to better resolve collisions. To achieve

[2] i.e., the sum of the multiplicities of all members

[3] The memory required for storing intermediate results when processing a query on such a field might scale differently, depending on the query. However, we are only interested in queries with low *limits*, where there is a low cap on the required intermediate.

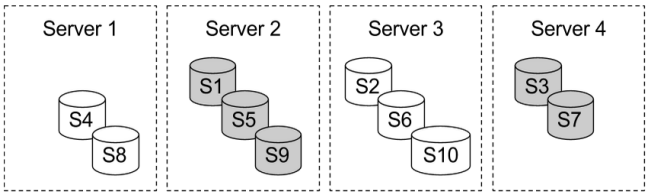

Fig. 1. Distributing 10 shards of data to 4 servers. The white shards, assigned to servers 1 and 3, are encoded with the first hash, and the grey ones, assigned to servers 2 and 4, are encoded with the second. Since there is an equal number of shards of each color and since the shards are of the the same size, each hash encodes exactly half of the data.

TABLE III
POSSIBLE RESULTS AFTER TRANSLATING IDS FOR BOTH HASH FUNCTIONS AND HOW THEY CAN BE MERGED TOGETHER. I.E., THE FIRST ROW FOR $h1$ CAN BE COMBINED WITH THE SECOND ROW FOR $h2$, BY KEEPING THE COMMON ENTITY AND ADDING UP THE COUNTS

Results for h1		*Results for h2*		*Combined results*	
user query	#	user query	#	user query	#
fuzzy *OR* Sport	5	Shopping	6	Shopping	10
Shopping	4	Sport *OR* isotope	3	Sport	8
Computer	1	abc *OR* Computer	2	Computer	3

this, a hash function $h1$ can be used to encode a random partition containing *half of the data*, and a different hash function $h2$ can be used on the partition containing the other half. This sounds difficult but can be easily achieved in a distributed system. Typically in such a system, records of the data are randomly and evenly split into shards, which are then distributed to servers for processing, usually in a round-robin fashion. In this setup, half of the servers can always (independently of the query they are computing) encode their assigned shards with $h1$, and the other half with $h2$. If there is an even number of shards of equal size, then each hash encodes exactly half of the records. The process is illustrated in Figure 1.

Each server fetches into memory only (its local part of) one ID column, and is responsible for updating only one of the reverse dictionaries, corresponding to its assigned hash function. In total, this consumes exactly the same amount of memory as when using a single function. Results encoded with $h1$ are mixed (combined) separately than the ones encoded with $h2$: Two intermediate results are obtained, corresponding to running the same computation on half of the data, in which the group-by keys are values of different hash functions. To further merge the two intermediate results into a final one, the group-by keys are translated to strings, using the corresponding reverse dictionary. The translation might not be unique, so a list of string values can be obtained for each group-by entry in each of the results. Collisions can be resolved by intersecting the lists of strings on each side and aggregating the metrics for the keys where an intersection occurs (i.e., adding up corresponding values). An example is shown in Table III.

Having 50% samples on each side helps by ensuring the "real" result appears on both sides, such that intersections can be done. When this is not the case, i.e., for very infrequent entities, colliding strings can still be concatenated with OR.

Some theoretical problems that might arise, like two lists having more than one string in the intersection, or a list on one side intersecting with more than one list on the other, have such low probability that they can be safely ignored in practice.

E. Experimental Validation

1) Setup: To validate the effectiveness of our approach in practice, in cases where high result accuracy is required, we selected real queries with expensive fields from a few days of PowerDrill's logs. The selected queries additionally satisfy the constraints presented in Section I-B. Since applying a function on the hashes instead of on the original values gives meaningless results, we further excluded queries in which the expensive field is used in any transformations. The final remaining set contained 160,000 queries. Each of them was then executed twice: Once in the version using the original expensive field, and once with the expensive field replaced by its "partitioned hashes". For both versions, memory consumption was recorded. Additionally, after both queries finished, results were compared in order to determine the errors introduced by hashing.

For the experiments, for all expensive fields a hash length of 31 bits was used. This length was a natural choice, since it entirely fits in a primitive type (e.g., an *int*), for which, as opposed to variable-length values, PowerDrill's dictionaries offer builtin support. Even more, alongside the hash value, an extra bit can be enconded in an int, to discriminate between $h1$ and $h2$. Because the value spaces of the two functions are now non-intersecting, the implementation is simplified—results from individual servers can be merged together. Also, we empirically validated that chosing this length provides significant memory savings, while still producing results with small relative errors.

2) Memory Savings: Table IV shows the memory benefits of using the "partitioned hashing-approach" on the expensive fields identified in Section II-B. One can see that memory consumption is reduced massively, on average over all expensive fields by a factor of 3.5. When comparing to using the standard, production grade compression algorithm LZO, we have saved an impressive factor of 4.5.

In absolute numbers, the total memory needed for the considered expensive fields reduced from 18 TB to 5 TB. Since they only represent a fraction of the expensive fields and since only 35 TB of RAM are available to PowerDrill, this is a massive improvement. It is also worth mentioning that these memory savings are achieved with almost no additional cost on the execution speed of the queries—the 95th latency percentile for the queries increases from 6.5 to 6.6 seconds when using the hashed encoding.

3) Accuracy: For accuracy measurements, results of real queries in our production environment were compared, in the "partitioned hashes" approach and in the version using the original expensive fields. Errors were computed for all numeric columns of these results. The standard definition of the relative error $|v' - v|/|v|$ with respect to the L2 norm was used, where v' is a column from a result using the former approach and v

TABLE IV
MEMORY FOR STORING THE REPRESENTATIVE SET OF EXPENSIVE FIELDS INTRODUCED AT THE BEGINNING OF THE SECTION, ALSO ADDING THE MEMORY REQUIRED FOR PARTITIONED HASHES ENCODED WITH PD+LZO

	size in TB			
	original	LZO	PD+LZO	hash+PD+LZO
average	3.35	1.21	0.94	0.27
total	63.69	22.91	17.85	5.05

the corresponding column from a result using the latter. The median error for all the columns was less than 0.01%, the 90th percentile of the error was 0.4%.

We observed that except for very rare items, the merging strategy proposed in Section II-D2 resolves collisions uniquely and correctly: When ignoring rows having a count smaller than 5, the relative error for columns goes down from 0.4% to 0.3% in the 90th percentile. Also, in more than 90% of queries, all the entities with a count greater than 5 were resolved correctly.

For very infrequent items, it can happen that all the rows that contribute to the aggregated result end up in only one of the two random partitions. In such situations, all contributing strings are concatenated (with a separator, such as *OR*) and displayed to the user. We call this a *loose match*. The errors for infrequent items are totally tolerable, especially in data discovery tasks, which require more a qualitative picture rather than a fine grained, exact view of very rare events. However, it is worth mentioning that even when collisions for the rare events cannot be resolved programatically, most of the time they might still convey some information to the analyst. As shown in Table V, in some cases the string to choose can be selected based on the context.

TABLE V
A RESULT FOR WHICH THE BOTTOM ENTRIES ARE LOOSE MATCHES. AN ANALYST CAN GUESS THE CHOICE BASED ON THE CONTEXT (THE TOP ENTRIES). THE EXAMPLE HERE IS JUST FOR ILLUSTRATION PURPOSES AND DOES NOT REFLECT A REAL RESULT.

user query	Count
hotel in Paris	15411
Paris trip	9806
Eiffel tower tickets	3009
Paris must see 4 kids *OR* top sci-fi books	4
volatile Java *OR* honeymoon in Paris worth it	3

Overall, our method saves an important 37% of all the memory available in PowerDril (13 out 35 TB), while preserving high result accuracy (only 0.4% relative errors in the 90th percentile). The gains in terms of resources achieved by using the partitioned hashes approach open new possibilities in the future, by allowing us to both serve more data sets and improve the query execution speed for the existing ones.

III. SAMPLING

In the last section we saw how a data-sketches inspired approach can be used to achieve tremendous memory savings for certain types of queries. Overall, in our production environment (in Sept 2014) we still observed a 95th latency percentile of about 30 seconds for queries classified as "normal". (About 42% of all queries are in this class; it is important for many in-depth investigations.)

Prendse and Creeth in their first OLAP-report [28] postulated an average latency of 5 seconds and "very few" queries taking more than 20 for interactive data analysis—a definition which has been widely accepted since and also matches qualitative feedback which we received from our users. Latencies of 30 seconds are therefore far beyond our target.

In order to speed up these "normal" queries (and other still unsatisfactory cases), we had a closer look at a completely orthogonal approach: sampling. By simply using a subsample of the data and scaling up the result values appropriately, one can trivially reduce both latencies and memory usage by arbitrary factors. This is easy to do for queries using COUNT, SUM, or AVG as aggregators—the same type of queries as considered in chapter II, but this time without limiting ourselves to queries containing expensive fields or not containing certain types of WHERE-conditions (covers $> 98\%$ of all queries).

Initial experiments with providing 3% subsamples to our users showed very little adoption. The sampling feature—which was easy to switch on for any dataset—was only used for 0.6% of all UI queries. Talking to our users it turned out that they often end up slicing data down to very small subsets where the sample gave bad estimates. Users that ran into such cases quickly got frustrated and stopped using the subsample entirely.

A. Sampling made practical

In this section we describe a novel approach which tries to overcome these problems which prevented standard sampling from being useful in PowerDrill's usage scenario. The approach enhances standard sampling with the following key elements:

Intermediate results as data is being processed. *Estimates continuously keep updating with increasing sampling rate.* Thus the first estimate will appear quickly and keep improving—all automatically. At any time the user can start the next query (cancelling the remains of the old query) if the results are already sufficiently clear—or otherwise wait for more accurate results to appear.

Accuracy of intermediate results. The accuracy is difficult to judge intuitively, because it depends not only on the sampling rate, but also, e.g., on the selectiveness of the query or the statistical properties of the underlying data. It is therefore important to *clearly mark the accuracy of intermediate results*.

Display accuracy per shown value. The results of a query typically consists of a table with multiple rows and columns, and the accuracy of an estimated number varies from row to row depending on the amount of data backing it, and from column to column depending on the statistical properties of the underlying field. Usually the user does not need to wait for all the numbers to become accurate: Often the rows with more data are the important ones for the analysis and there is no need to wait for the less accurate results in rows with very little data, because it is sufficient to know that these rows

correspond to little data. It is therefore essential to *mark the accuracy of each displayed value*, so users can proceed before everything has become accurate.
We visualize this by showing numbers with accuracy below a predefined threshold greyed out and sufficiently accurate numbers in regular black.

While sampling is generally well understood and straightforward to implement, the challenge in our case was implementing continuous updates and accuracy estimates on top of the existing complex distributed system. We eventually arrived at a very simple solution which in our eyes is compelling and elegant. With few changes to the existing system it achieves surprising productivity gains by giving very clear accuracy indications and without compromising on the accuracy of the final result. The changes do not add the extra load and complexity on the system that common approaches of computing confidence intervals would.

For a brief explanation of how relational data / joins are handled in our setup, please see B.

B. Accuracy Estimation

It is well understood how to estimate the accuracy of estimates computed from samples under various circumstances (consult [29] for an overview): Typically for the types of queries we are considering, as defined in Section I-B, the accuracy of an aggregated number in a result set is estimated based on the variance of the underlying variable and on the size of the sample, i.e., the number of records that was aggregated into the number. Both the variance and the number of records are query specific: They need to be computed only on the subset of rows satisfying the WHERE and GROUP BY clauses. Based on PowerDrill's logs we verified that for the real query mix, the variance varies greatly from field to field as well as from query to query. This means that precomputations are not possible. Instead, the variance would need to be computed along with the (sampled) query. Unfortunately, this is prohibitively expensive, leading to additional full-scans of the data and extra calculations that are not "materialized" in the table itself. Because of this, a different approach had to be taken.

An heuristic was developed with the goal of providing as tight as possible probabilistic error guarantees for the sampled estimates, under the condition that their computation must not add significant computation time or memory consumption. We took a large representative set of queries from PowerDrill's query logs and compared sampled and exact results. Empirically we found that more than 90% of all those values that had been estimated based on more than *100* records in the sample, had a relative error of less than 10%. This held for values computed by COUNT-, SUM- and AVG-aggregation alike.

So according to the heuristic, in a result set we mark a number as reliable if and only if it was aggregated over a group of more than 100 data set rows. This criterion can be evaluated at basically no extra cost.

The correctness of this heuristic criterion depends on the query mix issued by users and on the statistical properties of the data—in particular on the variance of the expressions over which aggregations are computed. These could change over time. Therefore, we constantly monitor the quality of the estimates displayed as reliable (i.e., displayed in black). We do this by always computing the exact result in the background for queries that are not canceled. More on this in Section III-D below.

C. Implementation

In this section we briefly describe how the system is extended to show intermediate result estimates including accuracy annotations (i.e., which values are to be marked black for being trustworthy).

It is obviously quite easy to automatically extend user queries to contain the "cheap heuristic" for accuracy estimation per result value. Similarly the appropriate scaling of results over sampled data is straightforward. The more interesting question may be how to efficiently obtain intermediate results over increasingly larger subsamples. The answer is that this comes for free by the highly distributed design of PowerDrill.

Queries are computed in parallel by a large number of servers, currently 2000 machines. The data is partitioned across these machines with each of these partitions being a random subsample, cf. [1]. Since the machines are shared by many (arbitrary) other services, the load on the individual machines varies heavily. As is common in such distributed systems, the response times of individual servers therefore varies tremendously (the execution times on the fastest and the slowest machines can differ by even an order of magnitude). As soon as at least one of the servers has finished, we have results for a subsample of the data. The size of the subsample keeps growing with every additional finishing server. These intermediate results over growing subsamples can be scaled appropriately and shared with the user. Other approaches for computing intermediate results have been studied in the literature, see, e.g., [20], [21], [22], [23], [24].

In some cases, waiting even only for the first of the 2000 servers to finish may be inconveniently slow. To have a very quick first intermediate answer, we additionally compute the same query over a small (currently 3%) subsample of the entire data which is distributed across all machines as any other dataset would be.

D. Validation and Monitoring Setup, Metrics

As mentioned in Section III-B, we eventually always compute the final, fully accurate result for queries that were not canceled. Note that this is important since users often drill into small subsets of the data where subsamples may perform poorly. Below in Section III-E we will show numbers which verify this with our live query-load.

Since we have the luxury of obtaining the final result for each completed query, we can nicely validate our "cheap heuristic" for accuracy estimation and the general quality of the intermediate results in our production environment. This can be used to fine-tune and monitor the approach and thresholds chosen. Please note that for cases where the user cancels running queries (e.g., to move on quickly before

everything has been computed), we do not obtain the final result. I.e., for these cases we cannot validate the "cheap heuristic". As per Sept 2015 about 30% of all queries are canceled and we therefore obtain final results in about 70% of all cases.

For the latter we measure 1) the latencies of all intermediate and final results, 2) the percentage of result values / cells which are determined to be trustworthy (i.e., marked black), 3) the per-column relative error of all result values, and 4) the per-column relative error restricting only to trustworthy / black values. For the computation of the relative error we treat individual columns of the result as vectors. Note that each column of the result represents a single aggregator across multiple group-by values. As for our data-sketches related optimization, we compare intermediate vector v' with the final vector v using the standard definition of relative error $|v' - v|/|v|$ with respect to the L2 norm. This nicely captures the way users interact with the data, viewing a column as a whole (e.g., trying to figure out which groups / rows accumulate high costs), rather than caring in detail about each individual value.

The collected data also is the basis for the experimental results / measurements which we show in the following section.

E. Measurements on Production Data

We can show that the approach gives an enormous productivity gain to our users on actual queries. For the first intermediate result users wait only *4 seconds*. This compares to 30 seconds for the final result.[4] Note that this is measured over all queries which are eligible for these intermediate estimates, no matter how restrictive. In this first intermediate result on average 62% of the values are marked black (i.e., trustworthy); and indeed when considering these values only, the relative error compared to the final result is quite low with only 9%.

TABLE VI
95TH LATENCIES PERCENTILES OF THE FIRST INTERMEDIATE AND THE FINAL RESULT, THE PERCENTAGE OF "BLACK", I.E., TRUSTWORTHY RESULT VALUES (ON AVERAGE) AND THEIR RELATIVE ERROR (MEDIAN AND 90TH PERCENTILE).

			relative error black	
	latency	% black	median	90th percentile
First result	3.89 secs	62%	0.17%	8.99%
Final result	30.3 secs	100%	0%	0%

We summarize the key results in Table VI. The measurements are based on about 60'000 live queries. This shows an impressive 7.7x speedup (30.3 vs. 3.89 secs) for cases where users can deduce useful insights from the "black cells" only.

Next we were interested to see how important for actual live queries it is to estimate the accuracy and annotate result values as trustworthy or not. Figure 2 shows the relative error of both all result values and of black result values in comparison for intermediate results (in both cases we look at the 90th percentile of the errors). On the x-axis we plot the relative

[4]As mentioned, all latencies are given with respect to the 95th percentile.

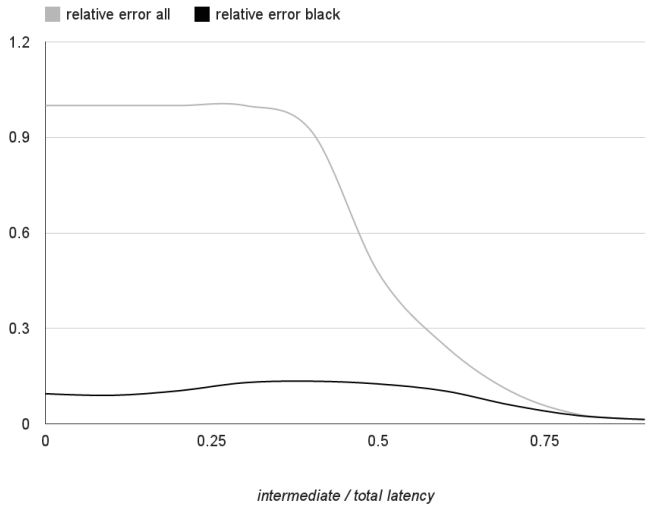

Fig. 2. The relative error of all result values compared to the relative error of values which are marked "black" for trustworth. The 90th percentiles are plotted over all intermediate results and with respect to the relative latency, i.e., intermediate latency / total latency.

latency when the intermediate result arrived, i.e., the latency of the intermediate result divided by the latency of the final result.

The chart shows that 1) the relative error of the "black values" is consistently low, independent of how large the subsample is backing the computation: up to a relative latency of 0.6 the relative error is between 9% and 13%. From a relative latency of 0.7 on the error decreases to 0 gradually (note that over time the number of black values obviously increases). In other words, the "cheap heuristic" to estimate the accuracy is doing a consistenly good job independent of the sample size. The other equally important insight here is that 2) the error of *all values* (= all result cells, including the ones which are not marked as trustworthy) is very bad for low relative latencies: Up to a relative latency of 0.4 the relative error for all values is larger than 90%. Put simply, the initial results are unusable without proper annotation.

We believe that these numbers nicely show that a suprisingly simple extension of an existing column-store can give users large speedups, even though results over the full dataset are still computed in about 70% of the cases. In our production environment this pushes the large set of "normal" queries from being "borderline interactive" down to a latency of below 5 seconds in the 95th latency percentile which has been accepted as being reasonable for OLAP type investigations (originally proposed in [28]). It also becomes clear that the intermediate results are only useful when annotated in some manner to hint their accuracy.

Obviously, it is of large interest whether these advancements actually enabled users to interact more quickly with their data. To quantify this, we looked at the number of UI interactions per user & hour. The sooner users see useful results, the quicker they can react and continue with their analysis. I.e., more interactions per hour translate into a more interactive and productive experience—the goal of trading off accuracy for speed in the first place.

Fig. 3. The average number of UI interactions per user & hour over 1.5 years (from Jan 2014 to Sep 2015). Each iteraction (e.g., restricting the WHERE statement) triggers the computation of one or many queries. Being able to do more iteractions per hour speaks for a more interactive and productive experience.

A first version of the sampling-based approach was launched in September 2014. Figure 3 shows that the average number of UI interactions per user & hour doubled between August 2014 and September 2015. We believe that showing trustworthy estimates early on has significantly contributed to users over time becoming more comfortable with moving on a lot more quickly with their analyses.

IV. Conclusions

In this paper we examined ways to improve performance, both in terms of compute resources and execution speed, to allow for interactive analysis over very large datasets when perfect accuracy is not a strict requirement. We observed that for data discovery work, for which PowerDrill is being used, the qualitative picture is mostly sufficient, and showed that by sacrificing a little accuracy in a controlled manner, high performance gains could be achieved.

Using an approach inspired by data sketches, we achieved a decrease in the memory usage of queries with expensive, high cardinality fields, by a factor of 4.5, which saves a significant amount of PowerDrill's available resources. However, our approach required us to deviate considerably from the way data-sketches are normally implemented, to benefit from better reuse of the data structures across a variety of free-form queries.

We also explored how we can gain significant speedups by using sampling. Our initial implementation based on sampled data sets had a very low adoption, due to the fact that results were often innacurate for investigations over very small subsets of data. We solved this by having continuous estimates based on automatically increasing sampling rates, in conjunction with displaying the level of accuracy of these estimates for each individual number. This made possible to achieve a 7.7x speedup while still showing insightful results. It also saves users from having to deal explicitly with sampling rates and yet allows them to skip waiting for more precise estimates with confidence, when the current estimates are already sufficient for a clear picture. Over the course of a year, this led to an impressive doubling of the number of UI interactions per user & hour.

PowerDrill has been developed and improved over many years, and as a result the underlying code is quite complex, incorporating multiple optimizations such as compression, parallelization and multiple layers of caching. Therefore large deep-down changes are expensive and risky. One of the main contributions of this paper is presenting approaches that can be implemented as an outer wrapper around the core software, without interfering with its internal affairs. This should make them easily applicable to other systems.

Implementing changes on a live system heavily used for business critical operations raised many restrictions, but on the other hand it gave us the opportunity to run experiments on realistic input data and examine usage patterns and user behavior. Our observations and findings in a real environment are the other main contribution of this paper, and we believe they can be applied to other data discovery systems as well.

Acknowledgments

The authors would like to thank to Ashley Brown, Marko Ivanković, Martin Wimmer and Maksym Zavershynskyi from Google, who reviewed our paper, to our engineering director, Olaf Bachmann, who supported this initiative, and to the entire PowerDrill team.

References

[1] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser, "Processing a trillion cells per mouse click," *PVLDB*, vol. 5, no. 11, pp. 1436–1446, 2012.
[2] Protocol Buffers: Developer Guide, https://developers.google.com/protocol-buffers/.
[3] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.
[4] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column oriented database systems," *PVLDB*, vol. 2, no. 2, pp. 1664–1665, 2009.
[5] D. J. Abadi, "Query execution in column-oriented database systems," Ph.D. dissertation, MIT, 2008.
[6] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *CIDR*, 2005, pp. 225–237.
[7] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *SIGMOD Conference*, 2009, pp. 297–308.
[8] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented dbms," in *VLDB*, 2005, pp. 553–564.
[9] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.
[10] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD Conference*, 2008, pp. 967–980.
[11] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou, "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds," *PVLDB*, vol. 4, no. 12, pp. 1474–1477, 2011.
[12] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2094114.2094126
[13] MonetDB, www.monetdb.org, last accessed 2016-05-18.
[14] Netezza, www.netezza.com, last accessed 2016-05-18.
[15] QlikTech, www.qlik.com, last accessed 2016-05-18.

[16] VectorWise, www.actian.com/products/vectorwise, last accessed 2016-05-18.
[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.
[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003, pp. 29–43.
[19] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[20] D. Fisher, I. Popov, and S. Drucker, "Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 1673–1682.
[21] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: The control project," *Computer*, vol. 32, no. 8, pp. 51–59, 1999.
[22] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol, "The sort-merge-shrink join," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1382–1416, 2006. [Online]. Available: http://doi.acm.org/10.1145/1189775
[23] S. Joshi and C. M. Jermaine, "Materialized sample views for database approximation," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 3, pp. 337–351, 2008. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2007.190664
[24] D. Shasha, "Review - ripple joins for online aggregation," *ACM SIGMOD Digital Review*, vol. 1, 1999. [Online]. Available: db/journals/dr/Shasha99c.html
[25] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," in *Proceedings of the 31st Symposium on Principles of Database Systems*, ser. PODS '12. New York, NY, USA: ACM, 2012, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/2213556.2213562
[26] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00778-009-0172-z
[27] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, pp. 29–38, 2004.
[28] N. Pendse, R. Creeth, and B. (Firm), *The OLAP Report: Succeeding with On-line Analytical Processing*. Business Intelligence, 1995. [Online]. Available: http://books.google.de/books?id=JDxDOgAACAAJ
[29] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 13, pp. 1–294, 2011. [Online]. Available: http://dx.doi.org/10.1561/1900000004

APPENDIX A
THEORETICAL ACCURACY BOUNDS FOR THE MULTIPLE PARTITIONED HASHES METHOD

As discussed in Section II-D1, when using a single non-partitioned hash, the known accuracy bounds of count-min sketches [27] can also be used to bound the numerical errors caused by hash collisions in our approach, for queries using *count*-aggregation. The core idea behind Cormode's derivation of the bound can be generalized to our *multiple partitioned hashes* approach, presented in Section II-D2, not only for queries aggregating by *count* but also for ones using *sum*, as we will show here. Note that we are only looking at the numerical errors caused by by-catching other entities because of hash-collisions—not at potential errors in translating hashes back to the original strings.

In the multiple partitioned hashes approach, a query like:

```
SELECT SUM(X) AS m FROM table GROUP BY G
```

is computed by executing

```
SELECT SUM(X) as m
FROM partition_1 GROUP BY h1(G)
```

on one half of the data and an analogous query using $h2$ on the other half. The result for a specific entity, e.g. for G='computer' is then computed by adding the values of the metric from the group *h1('computer')* from the result set of the first and from group *h2('computer')* from that of the second query. In order to derive an error bound, we use the following identifiers:

$e_0, ..., e_{n-1}$: List of all distinct entities occurring in field G
e_0: Entity we are interested in (e.g. 'computer').[5]
r_j: SUM(X) over all lines in the first partition where $G = e_j$
s_j: SUM(X) over all lines in the second partition where $G = e_j$
err: Difference between the correct and returned value of m for the group containing e_0 (correct: $r_0 + s_0$).

We assume that both hash functions $h1$ and $h2$ have an output range size of 2^b and are *uniform*, i.e. that for $h1$ and h_2 each entity other than e_0 has a probability of 2^{-b} of colliding with e_0. The expected error from these by-catches is therefore

$$E(err) = \sum_{i=1}^{n-1} \left(r_j 2^{-b} + s_j 2^{-b}\right) \leq \sum_{i=0}^{n-1} (r_j + s_j) 2^{-b} = \frac{n}{2^b} \widetilde{X}$$

where $\widetilde{X} := \frac{1}{n} \sum_{i=0}^{n-1} (r_j + s_j)$ is the average over all entities of SUM(X) over both partitions.

Let's assume for the moment that all values of X are non-negative. Applying Markov's inequality to $E(err)$ then yields the desired error bound:

$$P(err \geq k) \leq \frac{n}{2^b \cdot k} \widetilde{X} \quad \forall k > 0 \tag{2}$$

and on the other hand obviously $err \geq 0$ with certainty.

If X is a signed field then it is clear that $|err|$ can not be larger than the error on the modified query where X has been replaced with $Y := abs(X)$. Therefore equation 2 applied to $abs(X)$ instead of X can be used as an upper bound for $|err|$.

A bound for queries aggregating by count instead of sum can be easily derived as a special case, because counting lines is equal to summing 1's. The result is identical to the bound for a single non-partitioned hash in equation 1 (with $d = 1$).

The above probabilistic error bounds refer to the *absolute* error and are identical for all groups of the result set (i.e. are identical for 'computer' and 'phone'). This means that the *relative* error of the query result is lowest for those entities with the largest correct values. Fortunately, TOP-k-queries return just the entities with the highest counts, which typically will have large sums associated with them as well.

A relative error bound for averages can be derived from the bounds for sum and count (which is probably significantly larger in theory, the errors of sum and count will reinforce each other).

We are showing these error bounds mainly to illustrate that the multiple partitioned hashes approach used on TOP-k-queries using count, sum or avg is similarly sound as the non-partitioned version on count-queries which is equivalent to count-min-sketches in terms of approximation accuracy. At the moment, we are not using these results for optimizing the memory-accuracy tradeoff, and rely on the experimental validation instead.

[5] without loss of generality

APPENDIX B
DEALING WITH RELATIONAL DATA FOR SAMPLING

In general, dealing with relational database operators like joins is one of the more difficult aspects of sampling. However, in our use case the situation is quite specific: The structure of the data sets on which PowerDrill is used, is basically fixed, and there is only one meaningful way of joining the underlying tables. All datasets are dominated by one big table with billions of rows, one row representing, e.g., one user query. There are other tables representing sub-entities to user-queries like clicks, but the number of clicks per user query is limited and at most in the hundreds.

In the given case, the total size of all sub-entities associated with one user query is small. It is therefore sufficient to sample on the level of user queries only, and then select all sub-entities of the sampled user queries into the sample as well.

This means that on the level of sub-entities, rows are not sampled independently at random, but rather all rows corresponding to the same main entity are either in or out. This dependency between the sampled rows potentially increases the variance of estimates computed on sub-entity fields but introduces no bias. (Having more rows of the sub-entity than of the main entity would normally lead to estimates for sub-entity fields having higher accuracy than for main entity fields due to a higher sample size. This advantage is neutralized by the dependency, but the accuracy is not lower than for main entity fields.)