Data Science and Engineering (2021) 6:1–19
https://doi.org/10.1007/s41019-020-00144-y

Parrot: A Progressive Analysis System on Large Text Collections

Yazhong Zhang[1,2] · Hanbing Zhang[1,2] · Zhenying He[1,2] · Yinan Jing[1,2] · Kai Zhang[1,2] · X. Sean Wang[1,2,3]

Received: 1 June 2020 / Revised: 20 August 2020 / Accepted: 5 October 2020 / Published online: 22 October 2020

Abstract
The size of textual data continues to grow along with the need for timely and cost-effective analysis, while the growth of computation power cannot keep up with the growth of data. The delays when processing huge textual data can negatively impact user activity and insight. This calls for a paradigm shift from blocking fashion to progressive processing. In this paper, we propose a sample-based progressive processing model that focuses on term frequency calculation on text. The model is based on an incremental execution engine and will calculate a series of approximate results for a single query in a progressive way to provide a smooth trade-off between accuracy and latency. As a part, we proposed a new variant of the bootstrap technique to quantify result error progressively. We implemented this method in our system called Parrot on top of Apache Spark and used real-world data to test its performance. Experiments demonstrate that our method is 2.4×–19.7× faster to get a result within 1% error while the confidence interval always covers the accurate results very well.

Keywords Approximate query processing · Text data analytics · Term frequency · Bootstrap

1 Introduction

A huge amount of textual data is increasingly produced on the Internet. In twitter, for example, more than 500 million tweets were published per day in 2017.[1] These data are of great analytic values across many fields including hot topic analysis, social public sentiment, etc. Compared to structured data, textual data contains more semantic information such as term frequency and tf-idf whereas existing SQL aggregation functions focused mainly on numerical values, and, thus, are not suitable. And due to the non-correlated relationship between documents, people have much less priori about the distribution of words, especially on a subset. Analyzing textual data through a collection of fixed workload becomes unrealistic. Therefore, the way of interactive exploration becomes popular. The interactive exploration tool gives the user opportunities to continuously approach the final goal by iteratively executing queries using varying predicates [7]. A key requirement of these tools is the ability to provide query results at "human speed". Previous literature [31] has demonstrated that a great delay can negatively impact user activity and insight discovery. However, the term frequency calculation on a 100GB text collection costs more than 10 minutes in our experiment.

For analyzing structured data, lots of previous works attempt to speed up query execution through Data Cube or AQP (Approximate Query Processing) techniques. For data cube [10] and its successors, e.g., imMens [21] and NanoCubes [20], they either suffer from the curse of dimensionality or restrict the number of attributes that can be filtered at the same time. When limited by response time and computing

The preliminary version of this work was published at the International Conference on Database Systems for Advanced Applications (DASFAA) 2020.

✉ Yinan Jing
jingyn@fudan.edu.cn

Yazhong Zhang
zhangyz17@fudan.edu.cn

Hanbing Zhang
hbzhang17@fudan.edu.cn

Zhenying He
zhenying@fudan.edu.cn

Kai Zhang
zhangk@fudan.edu.cn

X. Sean Wang
xywangcs@fudan.edu.cn

1 School of Computer Science, Fudan University, Shanghai, China

2 Shanghai Key Laboratory of Data Science, Shanghai, China

3 Shanghai Institute of Intelligent Electronics and Systems, Shanghai, China

[1] http://www.internetlivestats.com/twitter-statistics/.

resources, AQP systems (e.g., AQUA [2], IDEA [9], VerdictDB [23]) only return a single approximate result regardless of how long the user waits. However, there is an increasing need for interactive human-driven exploratory analysis, whose desired accuracy or the time-criticality cannot be known a priori and change dynamically based on unquantifiable human factors [29]. Besides, due to the difference in data structure, these technologies cannot be migrated to apply to text data easily. For semi-structured and unstructured data, state-of-the-art solutions are based on the content management system or cube structures, such as ElasticSearch [1] and Text Cube [19]. ElasticSearch supports simple queries with key-value based filtering as well as full-text searching for fuzzy matching over the entire dataset. However, it doesn't have good support for ad-hoc queries on subset and cannot return an accurate term frequency on a subset through the *termvectors* API. Text Cube uses techniques to pre-aggregate data and gives the user the possibility to make a semantic navigation in data dimensions [5]. Text Cube can significantly reduce query latency, but it requires extensive preprocessing and suffer from the curse of dimensionality.

The universality and the demand for performance motivate us to utilize sampling techniques to return approximate answers to shorten response latency. However, approximate answers are most useful when accompanied by accuracy guarantees. Most commonly, the accuracy is guaranteed by *error estimation*, which comes in the form of the confidence interval (a.k.a., "error bound") [18]. The error estimation can be reported directly to users, who can factor the uncertainty of the query results in their analysis and decisions. Many methods have been proposed for producing reliable error bounds—the earliest is closed-form estimates based on either the central limit theorem (CLT) [26] or large deviation inequalities such as Hoeffding bounds [12]. Unfortunately, these techniques either compute an error bound much wider than the real which lost guidance to users or require data to follow the normal distribution while the distribution of terms frequency often obeys the Zipf law [6]. This has motivated the use of resampling methods like bootstrap [30], which requires no such normal distribution and can be applied to arbitrary queries. However, traditional bootstrap and its variant, variational subsampling technique proposed by VerdictDB [23] remain high complexity in our progressive execution model due to lots of duplicate computation.

In this paper, we first present a new query formulation by extending SQL grammar with UDF (user-defined function) for term frequency analysis on text data. Then, we propose a sample-based progressive process model to continuously refine the approximate result in the user-think period. Longer the waiting time becomes, the more accurate the result will be. As a part of our progressive execution model, we present a new error estimation method, progressive bootstrap. Moreover, to achieve a good performance over rare words, we present a new low-overhead sampling method, *Tail Sampling*. Compared to our previous work, we make the following extensions: (1) we propose a method to maintain the pre-computed samples for data updates; (2) we propose an optimized progressive bootstrap method for efficiency; (3) we introduce the workflow and the components of Parrot in more detail and extend the experiment to study the convergence rate of Parrot. In summary, this paper makes the following contributions:

- We propose a new query formulation that extends SQL grammar with UDF to support term frequency calculation on text data.
- We apply AQP techniques to get the approximate result to shorten the response latency on large text datasets.
- We present a sample-based progressive execution model and a progressive bootstrap method to continuously refine the approximate result. We also propose a method to maintain the pre-computed samples to adapt to the data updates.
- We integrate these methods into the system called Parrot. Experiments show that Parrot can provide a smooth trade-off between accuracy and latency while the quantified error bound covers the accurate result well.

This paper is organized as follows. Section 2 introduces an overview of Parrot. Section 3 describes Parrot's sample-based progressive processing. Section 4 explains how our progressive error estimation works. Section 5 presents our experiments. Finally, we review the related work in Sect. 6 and conclude this paper in Sect. 7.

2 Overview

2.1 System Architecture

Parrot is placed between the user and an off-the-shelf database. The user submits queries through any application that issues SQL queries to Parrot and obtains the result directly from Parrot without interacting with the underlying database. Parrot communicates with the underlying text collection for accessing and processing data when sampling. Figure 1 shows the workflow and internal components of Parrot, which contains two stages, *online* and *offline*. In the offline phase, the sample preparation module first normalizes data into a unified format which is JSON-based with a "text" attribute to store text, a "desc" object to store other attributes, and a "words" array to store words that appeared in the text. In Parrot, we use Stanford NLP[2] and Jieba[3] to

[2] https://stanfordnlp.github.io/CoreNLP/.

[3] https://github.com/fxsjy/jieba.

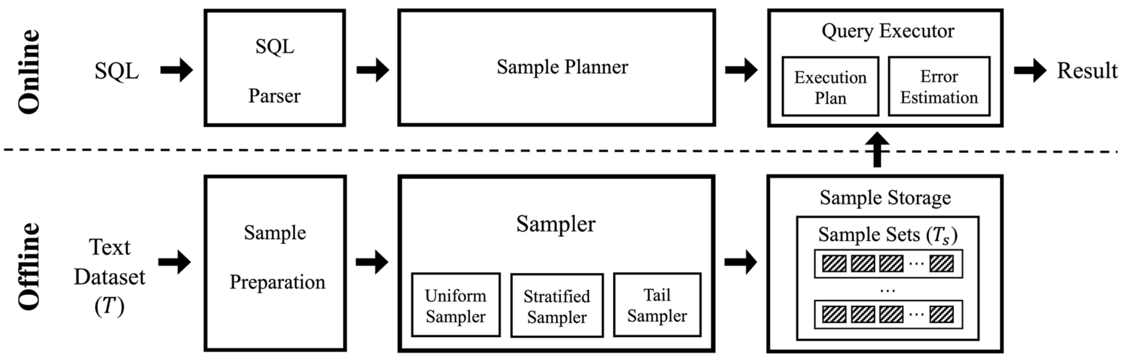


Fig. 1 System overview of Parrot

do word segmentation. Then samplers build different types of *sample set* which is a logical concept and composed of multiple *sample blocks*. Each diagonal filled box in Fig. 1 represents a sample block that stores a part of the original data. The data stored in each block of one sample set are distinct. At runtime, the query parser and analyzer analyze the issued SQL and generate a progressive execution plan. The execution plan contains two parts—a reference to the best sample set which is chosen according to our sample planner and an error estimation instance. Then the execution engine fetches blocks from the best sample set and calculates the result in the progressive mode which means that the user will receive an approximate result within a short time and the approximate result will be continuously refined until the whole sample set has been processed or Parrot is stopped by manual. Meanwhile, Parrot uses progressive bootstrap to estimate the error by the confidence interval according to a given confidence level. Generally, the width of the confidence interval can reflect the accuracy of the current result.

In the remainder of this paper, we use T to represent the underlying text collection, T_s to represent a sample set, $|T|$ to represent the cardinality of T and b_i to represent the i-th block. Actually, a sample set T_s is built by a sampler with a group of specified parameters applied on the underlying text data T.

2.2 Query Formulation

We extend the standard SQL grammar with UDF (user-defined function) to support analysis for text collections. Here is an example of inquiring the frequency of word *bank* in the *text* attribute. The date range is limited between Jan. 1, 2018 and Jan. 31, 2018.

```
SELECT FREQ('text', 'bank')
FROM news
WHERE date BETWEEN '2018-01-01' AND
'2018-01-31'
```

The FREQ function is to get the frequency of one word. It needs two parameters—the first one refers to the text attribute and the other refers to the issued word. The FREQ function is similar to a standard *count* aggregation after *group-by*. We support selecting multiple terms frequencies or using other select clauses in a mixture of FREQ in one single SQL. In addition to the FREQ function, we also support the TF-IDF which is a numerical statistic widely used in information retrieval. Since the core of the two functions is both about term frequency, we will focus on FREQ function in the following sections.

Parrot also integrates some third-party algorithms and systems to process accurate queries. Taking the TOP-K function as an example, the operator aims to find the k most frequent words. To tackle this operator, we integrate the ListMerge algorithm [32] into Parrot. ListMerge is based on threshold-style (TA-style) algorithms to answer top-k aggregation queries and has a superior performance on a large number of lists.

As a middleware, Parrot also supports to tackle queries directly to the backend database. To achieve that, the user just needs to precede the query with the *bypass* keyword, e.g., BYPASS SELECT FREQ('text', 'bank') FROM news. Then, Parrot sends the query to the backend database without any query rewriting or extra processing. This approach can be used for all other queries, such as "create table", etc.

2.3 Quantifying Result Error

The query engine refines the approximate result as more data is proceeded. Our error estimation is in the form of confidence interval with a given confidence level associated with the continuously updated result. For example, the confidence interval [3.5, 5.5] with the confidence level 95% means that we have 95% confidence to ensure that the accurate result will fall into the interval [3.5, 5.5]. In our progressive execution model, the expected performance is that the width of

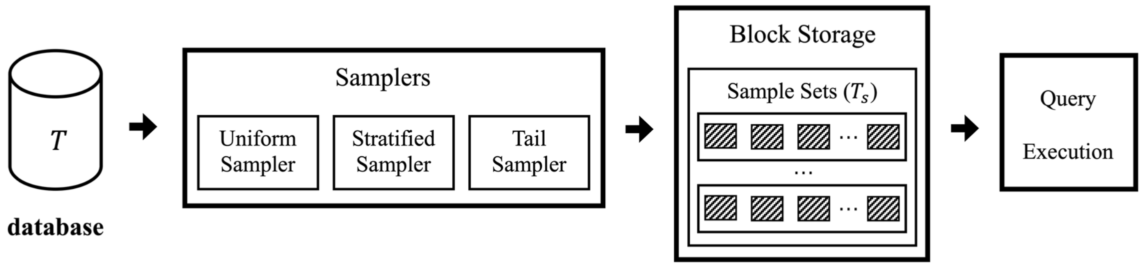


Fig. 2 The storage form of samples in Parrot

confidence intervals asymptotically converges from a relative large value to 0.

Currently, there are three widely-used methods in AQP system to do error estimation: closed-form estimates based on either the central limit theorem (CLT) [26], large deviation inequalities such as Hoeffding bounds [12], and the bootstrap [8, 30]. As discussed before, the Zip-F law of natural languages motivated us to use bootstrap techniques in our method. However, the traditional bootstrap needs to recompute the aggregate on many resamples (usually a large number, e.g., 100 or 1000), where each resample is a simple random sample (with replacement) of the original sample. Suppose the original sample size is n and the number of resamples is m, then the time complexity for one round of traditional bootstrap is $O(n \cdot m)$. VerdictDB [23] proposed a variational subsampling technique, which uses subsamples instead of resamples, to embed the entire bootstrap process into the single query with a bit more overhead. The time complexity for one round of variantional subsampling bootstrap reduces to $O(n)$. However, in our progressive execution model, the query will be executed for many rounds (e.g., 100 or 1000), where each round means a new block of data is being proceeded. Suppose there are p rounds in total, then the Verdict bootstrap has a high time complexity of $O(n \cdot p)$ through the whole process.

Considering that re-generating subsamples in each round needs to scan all past blocks, it's very time-consuming and contains lots of duplicate computations. In fact, we can apply some optimizations to avoid re-generating all subsamples when new block data has been proceeded. Our new bootstrap method, progressive bootstrap, has a lower time complexity of $O(\sqrt{n} \cdot m \cdot p)$. At the expense of that, the progressive bootstrap requires an additional memory cost (i.e. $O(\sqrt{n} \cdot m)$ to store the entire subsamples. Since the size is proportional to the square root of n, the memory cost is usually a limited value. The progressive bootstrap algorithm will be described in detail in Sect. 4.

3 Sample-Based Progressive Processing

In this section, we first show how we prepare samples by three samplers offline. Then we will introduce our online sample planner about how to pick the best sample set for execution. Finally, we explain the workflow of our execution engine, which utilizes delta computation to minimize re-computation.

3.1 Sample Storage

Sample storage is the module responsible for sample storage and provides the foundation for execution engine to progressively and parallel proceed sample data. As shown in Fig. 2, the structure of block storage contains two levels. The first level is called *sample set*, which is a logical concept that organizes data into block form. When executing, the driver fetches blocks from one specified sample set and calculate the result progressively. The second level is *sample block*. Each diagonal filled box in Fig. 2 represents a sample block which stores a part of original data. The sample blocks are distributed separately from each other among the storage devices. The data stored in each block is distinct. In this section, we use T to represent the underlying text collection, T_s to represent a sample set, $|T|$ to represent the cardinality of T, b_i to represent the i-th block and B to represent the number of blocks in the sample set. Actually, a sample set T_s is built by a sampler with a group of specified parameters applied on the underlying text data T.

3.2 Offline Sample Preparation

Uniform Sampler

Give the number of blocks B, uniform sampler scans the underlying dataset and generate a random integer in $[1, B]$ for each document. The random integer represents the block to which the document will be assigned. Each document will be attached with an extra attribute—ω, to represent the weight of that document. Then the sampler groups document by the generated integer and output into the corresponding block in block storage. The set of all outputing blocks is called a uniform sample set. In uniform sample set, the ω of each document has the same value, i.e., $1/|T|$. Both each single block and any combinations of these blocks could be seen as an independent uniform sample. The union data of all blocks is equivalent to the original dataset.

In the query execution phase, when the uniform sample is chosen, the query engine reads blocks sequentially in the sample set and calculates them in parallel. Whenever a block calculation ends, the partial result of current block will be merged into the global result. And then move to the next block. This process is repeated until all blocks have been calculated or manually interrupted.

Stratified Sampler

Stratified sampler optimizes queries over rare subpopulations by applying a biased sampling on different groups [4]. Given a column set $\mathcal{C}$ and a number k, stratified sampling ensures that at least k documents pass through for every distinct value of the columns in $\mathcal{C}$[4].

To satisfy progressive query execution, our stratified sampler needs to ensure that each sample block has almost the same and enough number of documents. If we use a fixed k value for all blocks, the block size will be less and less as there exist less and less groups of $\mathcal{C}$. Uncertain sizes among sample blocks will cause unsmooth in progressive execution.

Our approach is to dynamically adjust k to an appropriate value to make the block size be the most close to the expected. Suppose the cardinality of each block is set to $|b|$. At first, we collect the underlying dataset's cardinality of each group on the given column set $\mathcal{C}$. Suppose there are m groups and the cardinality of i-th group is represented by $|c_i|$. Then we try to minimum the value of *diff* in Formula 1 by binary searching an appropriate value $k = \hat{k}$ in the range of $[1, |b|]$. The time complexity of this step is $O(m \cdot \log_2 |b|)$. As in practical, there are usually hundreds or thousands of groups, the method is effective. Since we get the appropriate value of k (i.e., $\hat{k}$), we apply a Bernoulli sampling for each group individually. For the i-th group, the sampling probability is set to $min(|c_i|, \hat{k})/|c_i|$. All the chosen documents by the Bernoulli sampling will output into the new block with an extra attribute w setting to $1/|c_i|$ to record the weight. And they will be removed from the original dataset. Then update the left cardinalities for each group. The above process will be repeated until there is no left document (i.e. the cardinality of all groups is 0).

$$\text{diff} = \left| \sum_{i=1}^{m} \min(|c_i|, k) - |b| \right| \tag{1}$$

[4] Precisely, at least min(k, number of documents for that distinct value).

Besides, the stratified sampler will record the *disappearance block* (abbr. *d-block*) for each group in the above process. The *d-block* of a group refers to the block number that after outputting to that block, the remaining cardinality of this group reduces to 0. The reason to record it is to maximize the number of documents selected by the query in the sample within the same time. It's based on a simple observation that the cardinality of a group in the i-th block (i.e., b_i) is no more than that in the j-th block (i.e., b_j) if (1) $i \leq j$, and (2) $j < d - block$. Thus, for a specified group Ω and its $d - block_\Omega$, traversing from $d - block_\Omega$ to 1 (in reverse order) will maximum the query selecting number.

In the query execution phase, when stratified sample is chosen, the execution will first retrieve the maximum *d-block* ($d - block_{max}$) of all groups included in the query. Then the execution reads all blocks from $d - block_{max}$ to 1 (in reverse order) and calculates them in parallel. For example, suppose the query is SELECT FREQ('bank') FROM news WHERE date='2018-01-01' OR date='2018-01-02' and the *d-block* values for groups '2018-01-01' and '2018-01-02' are 100 and 150 respectively. Then the traversal order is from $d - block_{max} = 150$ to 1.

Tail Sampler

Queries over rare words occur frequently in the interactive data exploration, but often lead to large errors due to small hitting size. Many existing indexing techniques (e.g., B-trees, sorting database cracking [14]) organize all document without regarding for the semantic frequency in textual data, resulting in unnecessary overhead. While before mentioned samplers, uniform sampler and stratified sampler, can already provide good results even over rare subpopulations. Besides, these indexing techniques destroy the randomness property that our progressive execution model requires. It's also prohibitively expensive to mess up the documents order at runtime. Thus, we propose a low-overhead partial sampler which works offline to optimize query accuracy over rare words.

In general, Tail Sampler aims to improve the performance for queries over rare words. That is, if a word is frequent enough for uniform or stratified sample to provide a good performance, the word will not be included in tail sample. Otherwise, tail sampler collects these rare words and the documents in which these words appear. As different rare words may appear in the same document, it's unnecessary to pick documents for each rare word individually. Tail Sampler first determines whether a given document is rare. Then the sampler constructs the sample set on the subset of all rare documents. We use two parameters tail threshold τ and overhead λ, to quantify the limit of tail tampler. Before introducing the construction process, we need to prepare an inverted index for each word.

Table 1 Inverted index

	a	Bank	Is	It	What
Index	{1}	{1}	{1, 2, 3}	{1, 2, 3}	{2, 3}
Times	1	1	3	3	2

Table 2 Rarities of the document

Document	Rarest word	Rarity (δ)
D_1	Bank	1/3
D_2	What	2/3
D_3	What	2/3

Suppose the collection contains 3 documents:

D_1 = “it is a bank”

D_2 = “it is what it is”

D_3 = “what is it”

The inverted index and appearance times will be built as following (Table 1):

After that, we get not only an inverted index, but also the statistics information of each word's appearance, i.e., the appearance times for each word. We define the rarity of word γ, as the proportion of appearance document cardinality to the total document cardinality. The rarity of document δ equals to the minimum word rarity it contains (Table 2).

Then to return, tail threshold τ refers to the maximum rarity δ of document allowed to be included into tail sample. An intuition is that the bigger τ is, the more document and rare words would be included into sample. But it may also cause the increment of sample size which will slow down the interaction execution speed. However, when τ is set to a small value, though sample size is under control, the improvement is limited, as lots of rare words might not be included. In fact, it is much relevant to textual data distribution and workload about how to set an appropriate τ value. Another parameter is sample overhead λ. It's a value in [0, 1] and indicate the upper bound of the tail sample size. That is, whatever τ is, the final size ratio of tail sample set to original dataset is less or equal than λ. The reason why we need this parameter is make a trade-off between the coverage of rare words and the execution performance. The detailed construction algorithm of tail sampler is shown in Algorithm 1. The input contains text collection T and two parameters—tail threshold τ and sample overhead λ. With these inputs, tail sampler firstly builds the inverted index for each word in T. Then for each word, we calculate γ_{word} (line 3–5). Next, we scan words in the ascending order of γ. For each word, if $\gamma \leq \tau$, and the size ratio not exceeds λ, then add the word into the set of rare words and add documents in which the word appears into the sample document set, D_s, without duplicates (line 7-12). Finally, the tail sampler builds the tail sample set T_s based on the D_s by the uniform sampler. In the query execution phase, when tail sample is chosen, the execution reads blocks sequentially and calculates them in parallel, similar to uniform sample.

Algorithm 1: Tail Sampling

```
   Input: text collection $T$, tail threshold $\tau$, sample overhead $\lambda$
   Output: tail sample set $T_s$, rare words set $W$
1  initialize $T_s = \{\}$, $W = \{\}$ $D_s = \{\}$;
2  build inverted index for each word in $T$;
3  foreach word $\in$ inverted index do
4      $n$ = number of documents which this word appears in;
5      $\gamma_{word} = n/|T|$ ;
6  end
7  sort words by $\gamma$ in ascending order;
8  foreach word in ascending order of $\gamma$ do
9      if $\gamma_{word} <= \tau$ then
10         $D_w$ = documents which this word appears in;
11         if $|D_s$ union $\mathrm{D_w}|/|\mathrm{T}| \leq \lambda$ then
12             $D_s = D_s + D_w$;
13             $W = W + word$;
14         end
15     end
16 end
17 build $T_s$ based on $D_s$ by uniform sampler;
18 return $T_s$ and $W$
```

3.3 SQL Parser

The SQL parser in Parrot is responsible for parsing the issued SQL and fetching the logical operators (e.g., projections, selections, joins, etc.). Then the SQL will be rewritten to a collection of new SQL statements that can be executed on the underlying database to perform progressive processing.

The parsing process contains three stages. The first stage is to generate the AST (abstract syntax tree). This part is implemented through Antlr4 [24]. We extend Spark SQL grammar with our customized rules to support new operators introduced by Parrot. The second stage is to fetch logical operators from the issued SQL. In Spark, the AstBuilder is the class for parsing SQL. It uses Antlr4 visitor tree-walking mechanism to perform walk on the generated AST. In Parrot, we inherit AstBuilder and override the visitFunctionCall method to extract logical operators and route the AST of different operators to different handlers. For example, the SQL with TOP-K operator will be passed to the ListMerge [32] algorithm, while FREQ and TF-IDF will be passed to the third stage.

In the third stage, the issued SQL will be rewritten. Here is a Parrot's query rewriting example for a simple FREQ query. Given the below input query:

```
SELECT FREQ('text', 'bank')
FROM news
WHERE date BETWEEN '2018-01-01' AND '2018-01-31'
```

Parrots rewrites the above query as follows:

```
SELECT FREQ('text', 'bank') / SUM (ω) AS freq
FROM %news%
WHERE date BETWEEN '2018-01-01' AND '2018-01-31'
```

In such queries, the result of the aggregation function will be scaled up according to the value of SUM(ω) which means the total weight of documents calculated in the sample. The parameter ω is generated in the offline sample preparation phase which has been introduced in Sect. 3.2. The table name "news" in the "from" clause will be enclosed in percent signs. After selecting a sample plan, Parrot will generate a collection of queries[5] based on the issued SQL. In each generated query, the table name will be rewritten to the block name.

3.4 Online Sample Planning

A *sample plan* is composed of a reference to a sample set with some extra information (e.g., traversal order of the chosen sample set). Parrot's sample planner aims to find the *best* sample set for the query, i.e., the sample plan that results in the lowest approximation errors within the same latency. Our strategy is based on the *selectivity* which is defined as the ratio (1) the number of documents selected by the query, to (2) the number of documents read by the query. At runtime, the response time increases with more number of documents being read and the error decreases with more number of documents WHERE/GROUP BY clause selects. For this, Parrot generates many possible sample plans (called *candidate plans*) and selects the *best*.

3.4.1 Candidate Plans

In the first step, Parrot's sample planner generates candidate plans. Candidate plans are the set which contains all sample sets that can be used to answer the issued query.

[5] The number of generated queries is equal to the number of blocks of the selected sample plan.

For the uniform sample, it is always a candidate sample even if it may cause poor performance. There are two situations for a sample set that cannot be used to answer the query: (1) For a stratified sample set, if SCS $\supseteq$ QCS, then it is a candidate sample set. The SCS (sample column set) is the column set that stratified sampler builds on. The QCS (query column set) is the set of all columns that appears in the WHERE/GROUP BY clauses; (2) For the tail sample set, if the concerned word of the query belong to the rare word of the sample, then it is a candidate sample set.

3.4.2 Selecting a Plan

Parrot's plan selection relies on the selectivity. We measure the selectivity of a sample set based on two criteria: sampling strategy and matching degree. First we discuss the sampling strategy. In brief, we say that tail sample is better than stratified sample and both of them are better than uniform sample. Here are the reasons: (1) Tail sampler only include documents that have rare words and can reduce the total size of the sample set. Thus, for rare words, the selectivity is the highest; (2) Stratified sampler gives different groups different traveral orders. The traversal order of stratified sample set is from $d-block_G$ to 1. For a rare subpopulation, its $d-block_{rare}$ is a small value (only appear in the first few blocks). Thus the total size of the sample data for the query will decrease. For a popular subpopulation, its $d-block_{popular}$ is a large value. As the number of blocks increases, the number of groups decreases and the proportion of that popular group increases. The selectivity also increases. Therefore, for all sorts of groups, stratified sample can provide a higher selectivity than uniform sample; (3) Uniform sampler simply spilts original dataset into multiple blocks to support progressive execution. Thus the selectivity equals to that of original dataset and is the lowest.

We then take matching degree into account. For stratified sample, the matching degree is defined as |QCS|/|SCS|. The planner chooses the sample set which has the highest degree. If there exists more than one stratified sample set that have the same degree, the planner chooses the sample set which has the smallest $d-block$ for the query. For tail sample, we choose the sample set which has the smallest overhead λ as it has the highest selectivity among them.

Here is an example to explain our sample planning workflow. Suppose we have built 4 samples—one uniform sample, one stratified sample on *location*, one stratified sample on *location* and *date*, one tail sample with $\tau = 0.001$ and $\lambda = 0.1$. And suppose the issued query is SELECT FREQ('bank') FROM news WHERE location='Tokyo' and the concerned word "bank" doesn't not belong to rare words of the tail sample. Then the candidate plans will include three references which are to the uniform sample and two stratified samples, as they all can be used to answer the query. In plan selecting, first we take sampling strategy into account. As stratified samples have higher selectivity than the uniform sample, we filter out the uniform sample at first. Then we consider the matching degree of two stratified samples. Obviously, the stratified sample on *location* has a higher matching degree and is picked with the traversal order from $d-block_{Tokyo}$ downto 1.

3.5 Query Execution Model

Since processing a large dataset in blocking fashion can easily exceed interactive requirements, our execution engine needs to use techniques to compute a frequency result $\hat{\theta}^S_{\text{word}}$ on a relatively small sample. Here the *word* refers to the concerned word in FREQ function, S refers to the sample, and $|S|$ refers to the cardinality of S. Then, by *maximum likelihood estimation* (MLE) [27], we can produce an approximation $\hat{\theta}^{\text{MLE}}_{\text{word}}$ of the accurate result θ_{word}:

$$\hat{\theta}^{\text{MLE}}_{\text{word}} = \frac{\hat{\theta}^S_{\text{word}}}{|S|} \tag{2}$$

Intuitively, this approximation $\hat{\theta}^{\text{MLE}}_{\text{word}}$ represents the estimated frequency of the word in each document of the dataset. Multiplying a frequency estimate $\hat{\theta}^{\text{MLE}}_{\text{word}}$ by the total number $|T|$ of documents in the full dataset will therefore yield an approximate frequency for the word:

$$\hat{\theta}^T_{\text{word}} = \hat{\theta}^{\text{MLE}}_{\text{word}} \cdot |T| \tag{3}$$

Our execution engine works in progressive fashion. The process contains many *rounds*, where each round means a new block data is being proceeded and a refined result will be returned after proceeding. The number of rounds equals to the number of blocks in the *best* sample set for the query. The guiding design principle behind Parrot is to take full advantage of delta computation to minimize re-computation. In other words, before the i-th round, suppose we have finished proceeding data S_{i-1} (i.e., $S_{i-1} = b_1 \cup b_2 \cup ... \cup b_{i-1}$ where b_i refers to the i-th block) and get an approximate result $\hat{\theta}^{S_{i-1}}_{\text{word}}$. Then, instead of computing query on S_i, we utilize the fact that $S_i = S_{i-1} + b_i$ and calculate $\hat{\theta}^{S_i}_{\text{word}}$ from the previous result $\hat{\theta}^{S_{i-1}}_{\text{word}}$ by a delta query $\hat{\theta}^{b_i}_{\text{word}}$. The intuition is that computing θ_{b_i} and merging into previous result would be much faster than directly computing $\hat{\theta}^{S_i}_{\text{word}}$ since $\hat{\theta}^{S_{i-1}}_{\text{word}}$ has been computed before. The result refinement formulation is shown as Formula 4:

$$\hat{\theta}^T_{\text{word}} = \left(\frac{\hat{\theta}^{S_{i-1}}_{\text{word}} + \hat{\theta}^{b_i}_{\text{word}}}{|S_{i-1}| + |b_i|} \right) \cdot |T| \tag{4}$$

Similar intuition is shared by online aggregation (OLA) [13, 29] and incremental view maintenance [11, 17, 22], with slight differences in the definition of b_i: for Parrot, b_i is the data of the i-th block which contains documents in the same format; while for delta view maintenance and streaming systems, b_i can also include deletion of old data.

3.6 Data Appends

When new textual data is arriving, Parrot needs to keep the sample fresh and results accurate enough. As resampling is a very costly process, we try to adopt incremental update as far as possible. All three types of sample sets, i.e., uniform sample set, stratified sample set, and tail sample set, are amenable to the data append.

3.6.1 Uniform Sample Set

For the uniform sample set, the sample updating strategy is straightforward, since given the number of blocks B, Parrot samples all documents independently. When a new batch of data arrives, Parrot can simply generates a random number in $[1, B]$ for each document and append to existing sample blocks.

3.6.2 Stratified Sample Set

For the stratified sample set, the sample updating strategy is more complex. When a new batch arrives, there are two cases for updating a stratified sample set. The first is that the number of documents increases more, and the number of groups increases less or not, which means that the k value becomes larger. We difine the new k value as k' and k' is greater than k. For this case, we first group the documents in the batch based on the column set $\mathcal{C}$. Then for each block in the stratified sample set, we perform Bernoulli sampling in parallel according to the increased number $(k' - k)$ of each group in the new batch. The second case is that the number of documents increases less, but the number of groups increases more, which means that the k value becomes smaller. We difine the new k value as k' and k' is less than k. For this case, we also group the documents in the batch based on the column set $\mathcal{C}$. As the k becomes smaller, there is no need to append documents for each group from the new batch to the sample block. However, it is very costly to remove documents for each group according to the decreased number $(k - k')$ as it needs to scan the whole sample set. Thus, we divide the documents in the new batch into two parts based on whether the group has appeared before. For documents in groups which have appeared before, we append them to sample blocks starting from the $d - block$ -th block. For documents in groups which have not appeared before, we append them to sample blocks starting from the 1st block. The Bernoulli sampling probability is based on the k' and the population of each group and updating process is similar to stratified sampling described in Sect. 3.2.

3.6.3 Tail Sample Set

For the tail sample set, the sample updating strategy contains two steps: (1) include all rare words of the new batch into the rare words list; and (2) include all documents of the new batch which contains at least one rare word into the tail sample set. In the first step, the tail sampler scans all words that hasn't appeared before in the new batch. For each word, the tail sampler calculates its rarity and compares it with the tail threshold τ. If it is less, tail sampler will append it into the rare words list. In the second step, the tail sampler picks out all documents which contains at least one rare word. Then update the original tail sample set by assigning a block number to each rare document of the new batch and append them to existing sample blocks correspondingly.

However, there may still exist one problem. Due to when updating, only the original rare words list and the words in the new batch are considered, the tail sample set may miss some rare words which should be included in the rare words list. But this will not affect the correctness, because it will only reduce the coverage of the rare words list. In the online phase, only when queries are related to the rare words list, Parrot chooses to use tail sample set to answer. This problem cannot be completely solved by incremental updating unless by periodic resampling.

4 Error Estimation

In this section, we describe Parrot's novel error estimation technique. Previous interactive exploration system engines, especially those that support skewed data analysis, have relied on bootstrap [15], which belongs to a family of error estimation techniques called resampling [16]. Resampling techniques, despite various optimizations [3, 25], are still too expensive to be implemented at a middleware layer.

Inspired by variational subsampling bootstrap [23] in VerdictDB, we propose a new variant, called progressive bootstrap, to apply a fast bootstrap method for progressive query execution. In the remainder of this section, we will start from traditional bootstrap and subsampling bootstrap (Sect. 4.1). Then we introduce our method, progressive bootstrap (Sect. 4.2) in detail.

4.1 Traditional Bootstrap and Variational Subsampling Bootstrap

Bootstrap is the state-of-the-art error estimation mechanism used by previous AQP engines. The key idea of Bootstrap is that, in order to use $\boldsymbol{S}$ (sample) to replace $\boldsymbol{D}$ (origin dataset), one can also draw samples from $\boldsymbol{S}$ instead of $\boldsymbol{D}$ to compose the distribution of $\boldsymbol{S}$.

Let θ be the result of an aggregate function (e.g., mean, sum) on N real values $x_1, ..., x_N$ (e.g., values of a particular column). Let a random sample with size n of these N values and $\hat{\theta}$ be an estimator of θ. In error estimation, we need to measure the quality (i.e., expected error) of the estimate. To achieve that, bootstrap recomputes the aggregate on many resamples, where each resample is a random sample (with replacement) of the original sample. In traditional bootstrap, the size of a resample is the same as the sample itself. Some of the elements might be missing and some might be repeated, but the total number remains as n. Bootstrap will construct m such resamples (m is usually a large number, e.g., 100 or 1000). Let $\hat{\theta}_j$ be the value of the estimator computed on the jth resample. Then bootstrap uses $\hat{\theta}_1, \ldots, \hat{\theta}_m$ to construct an empirical distribution of the sample statistics, which can then be used to compute a confidence interval. Let $\hat{\theta}_0$ be the estimator's value on the original sample itself, and t_α be the α-quantile of $\hat{\theta}_0 - \hat{\theta}_j$. Then, the $1 - \alpha$ confidence interval can be computed as:

$$[\hat{\theta}_0 - t_{1-\alpha/2}, \quad \hat{\theta}_0 - t_{\alpha/2}] \tag{5}$$

Variational subsampling bootstrap, which relaxes some of the requirements of traditional bootstrap, are presented by VerdictDB. It still retains the statistical correctness of traditional subsampling, while becomes significantly more efficient. It follows a procedure similar to bootstrap, but with three key differences: (1) instead of full resamples, it uses subsamples which are much smaller, (2) instead of drawing tuples from the original sample with replacement, subsampling draws tuples without replacement, and (3) allowing each tuple to belong to, at most one subsample. In other words, a subsample is also a random sample of the original sample, but without replacement, and of size n_s where $n_s \ll n$. In general, n_s must be chosen such that it satisfies the following two conditions: (1) $n_s \to \infty$ as $n \to \infty$, and (2) $n_s/n \to 0$ as $n \to \infty$. Thus the bootstrap process is simplified as: for each tuple, system only needs to generate a single random number to determine which subsample it belongs to (if any), and then perform the aggregation only once per tuple, instead of repeating m times. Computing the $1 - \alpha$ confidence interval is similar to traditional, but requires a scaling:

$$[\hat{\theta}_0 - t_{1-\alpha/2} \cdot \sqrt{n_s/n}, \quad \hat{\theta}_0 - t_{\alpha/2} \cdot \sqrt{n_s/n}]. \tag{6}$$

4.2 Progressive Bootstrap

The both two bootstrap methods introduced before, can obtain the confidence interval on a given sample and an unknown distribution of dataset. However, they only present the bootstrap for single time. While handling data increment, they need to repeat the whole bootstrap process entirely and causes lots of duplicate computation. Suppose the cardinality of the sample set is n, the number of resamples is m, and there are p rounds in total, traditional bootstrap and variational subsampling bootstrap have unaffordable time complexity $O(n \cdot m \cdot p)$ and $O(n \cdot p)$ respectively. Inspired by Verdict, we propose the progressive bootstrap by maintaining all needed sub-samples through the progressive process to avoid duplicate resampling from past blocks. VerdictDB has proved that the bootstrap has the lowest error when the cardinality of each subsample equals to $\sqrt{n}$. Thus, in the best case, the subsampling ratio is $r = m \cdot \sqrt{n}/n = m/\sqrt{n}$. As n grows (more and more data has proceeded), the ratio will drop. Therefore, if we have preserved all subsamples s (composed of s_1, s_2, ..., s_m) and when new block b_i is being proceeded, we can directly update subsamples based on s with b_i instead of re-subsampling overall past data. For documents in b_i, we apply a Bernoulli sampling with ratio $r_i = m/\sqrt{n}$. For maintained subsamples, let E_i refers to the event of a document being picked into subsamples at i-th round and we can also apply a Bernoulli sampling with ratio Δr calculated by conditional probability as shown in Formular 7.

$$\Delta r = P(E_i | E_{i-1}) = \frac{P(E_i \cap E_{i-1})}{P(E_{i-1})} = \frac{r_i}{r_{i-1}} \tag{7}$$

Algorithm 2 gives a detailed illustration. The input includes the issued word set *words* in FREQ functions, the cardinality of underlying dataset N, proceeded data S_{i-1} (i.e., $b_1 \cup b_2 \cup ... \cup b_{i-1}$), new block b_i, past sub-samples s (composed of s_1, s_2,......, s_m), past counters c ($c[word]$ for *word* and composed of $c[word]_1$, $c[word]_2$, ..., $c[word]_m$), confidence level α (e.g., 0.95), and the approximate result $\hat{\theta}$. Our algorithm first filters out documents which should be excluded from the past subsamples (line 3–7) and update counters of each word for each subsample (line 8–10). Then it picks out documents which should be included into sub-samples from the new block (line 11–15) and update counters (line 16–18). Then for each word, we sort m counters in ascending order and collect the error bound σ_{word} according to the subsamples distribution (line 19–21). Finally, return the updated subsamples, counters and error bound (line 22). Progressive bootstrap has a lower time complexity $O(\sqrt{n} \cdot m \cdot p)$.

```
Algorithm 2: Progressive Bootstrap
  Input: issued word set $words$, dataset cardinality $N$, proceeded data $S_{i-1}$, new
         block $b_i$, past subsamples $s$, past counters $c$, confidence level $\alpha$,
         approximate result $\hat{\theta}$
  Output: updated subsamples $s'$, updated counters $c'$, error bound $\sigma$
1  $n' = |S_{i-1}| + |b_i|$;
2  $n'_s = \sqrt{n'}$;
3  foreach document $doc \in s$ do
4      suppose the document is in the $\varepsilon$-th subsample $s_\varepsilon$;
5      $r$ = a random number in $[0, n')$;
6      if $r < |b_i|$ then
7          $s_\varepsilon = s_\varepsilon$ - $doc$;
8          foreach $word \in words$ do
9              $f = word$ frequency in $doc$;
10             $c[word]_\varepsilon = c[word]_\varepsilon$ - $f$;
11         end
12     end
13 end
14 foreach document $doc \in b_i$ do
15     $r$ = a random number in $[0, n')$;
16     $\varepsilon = r/n'_s + 1$;
17     if $\varepsilon < m$ then
18         $s_\varepsilon = s_\varepsilon + doc$;
19         foreach $word \in words$ do
20             $f = word$ frequency in $doc$;
21             $c[word]_\varepsilon = c[word]_\varepsilon + f$;
22         end
23     end
24 end
25 foreach $word \in words$ do
26     sort $c[word]$ in ascending order;
27     $\sigma_{word} = [\hat{\theta} - c[word]_{\alpha/2} \cdot \sqrt{n'_s/n'} \cdot N/n'_s,\ \hat{\theta} - c[word]_{1-\alpha/2} \cdot \sqrt{n'_s/n'} \cdot N/n'_s]$;
28 end
29 return updated subsamples $s' = s$, updated counters $c' = c$ and error bound $\sigma$;
```

4.3 Progressive Bootstrap with Rearrangement

Regardless of traditional bootstrap, variational bootstrap or progressive bootstrap, the core method is to estimate the error of the approximate result through resampling (subsampling). For the first two bootstrap algorithms, each time of bootstrap is a complete execution of the entire process of resampling (subsampling) to construct the empirical distribution on the whole past sample documents. For progressive bootstrap, it maintains all sub-samples continuously during the calculation process. In other words, every time the error is estimated, re-sampling will not be performed on the current entire sample. This is the core reason why progressive bootstrap improves performance. However, in the actual experiment, we observe that due to the nonuniform distribution of word frequency and the randomness of resampling, at certain times, the difference of results between subsamples is very large. Moreover, this difference will continue throughout each iteration and makes the error bound difficult to converge.

For this reason, in each iteration, we introduce a process of rearrangement for progressive bootstrap. That is, each time the subsamples are maintained, progressive bootstrap rearranges subsamples to which each document belongs. This can avoid the imbalance of sub-samples caused by randomness, and will not cause the additional time overhead meanwhile. Progressive bootstrap after rearrangement optimization is shown in Algorithm 3.

Table 3 Selectivity of Q_1–Q_9

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
Selectivity (%)	17.5	0.42	0.98	0.48	0.16	0.038

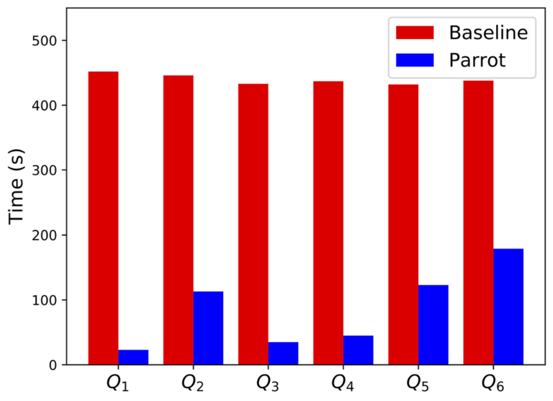
Table 4 Selectivity of Q_1–Q_9

	Q_7	Q_8	Q_9
Selectivity (%)	6.58	2.95	0.53

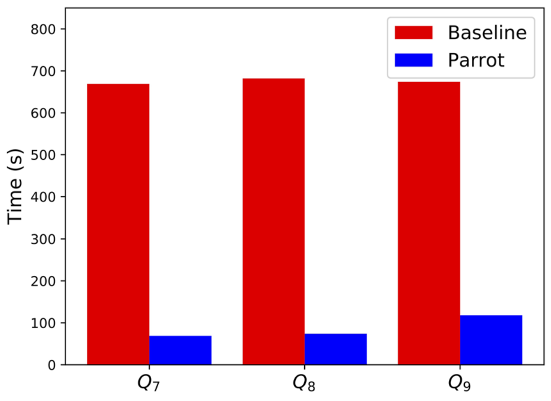
Algorithm 3: Progressive Bootstrap with Rearrangement

```
Input: issued word set $words$, dataset cardinality $N$, proceeded data $S_{i-1}$, new
       block $b_i$, past subsamples $s$, past counters $c$, confidence level $\alpha$,
       approximate result $\hat{\theta}$
Output: updated subsamples $s'$, updated counters $c'$, error bound $\sigma$
1  $n' = |S_{i-1}| + |b_i|$;
2  $n'_s = \sqrt{n'}$;
3  foreach document $doc \in s$ do
4      suppose the document is in the $\varepsilon$-th subsample $s_\varepsilon$;
5      $r$ = a random number in $[0, n')$;
6      $\varepsilon' = r/n'_s + 1$;
7      $s_\varepsilon = s_\varepsilon$ - $doc$;
8      if $r \geq |b_i|$ then
9          $s_{\varepsilon'} = s_{\varepsilon'} + doc$;
10     end
11     foreach $word \in words$ do
12         $f = word$ frequency in $doc$;
13         $c[word]_\varepsilon = c[word]_\varepsilon$ - $f$;
14         if $r \geq |b_i|$ then
15             $c[word]_{\varepsilon'} = c[word]_{\varepsilon'} + f$;
16         end
17     end
18 end
19 ... (the same as Algorithm 2 line 14 - 28) ...
20 return updated subsamples $s' = s$, updated counters $c' = c$ and error bound $\sigma$;
```

Algorithm 3 gives a detailed illustration. The input and output are same as Algorithm 2. The main difference is the part of filtering out documents which should be excluded from the past subsamples (line 3–18). While filtering documents out, for each retained document, we will reassign the subsample which it belongs to (line 8–10). Then when updating word counters by subtracting the frequency of words contained in the document (line 11–17), we also update word counters by increasing the frequency to the reassign subsample counter (line 14–16). The following process is the same as line 14–28 of Algorithm 2. Finally, return the updated subsamples, counters and error bound (line 20). Progressive bootstrap with rearrangement has the same time complexity $O(\sqrt{n} \cdot m \cdot p)$ as before. For the correctness of bootstrap with rearrangement, since it's completely random to assign the subsample sequence number for each document, it doesn't



(a) on the synthetic dataset



(b) on the real-world dataset

Fig. 3 Time to get the first acceptable result

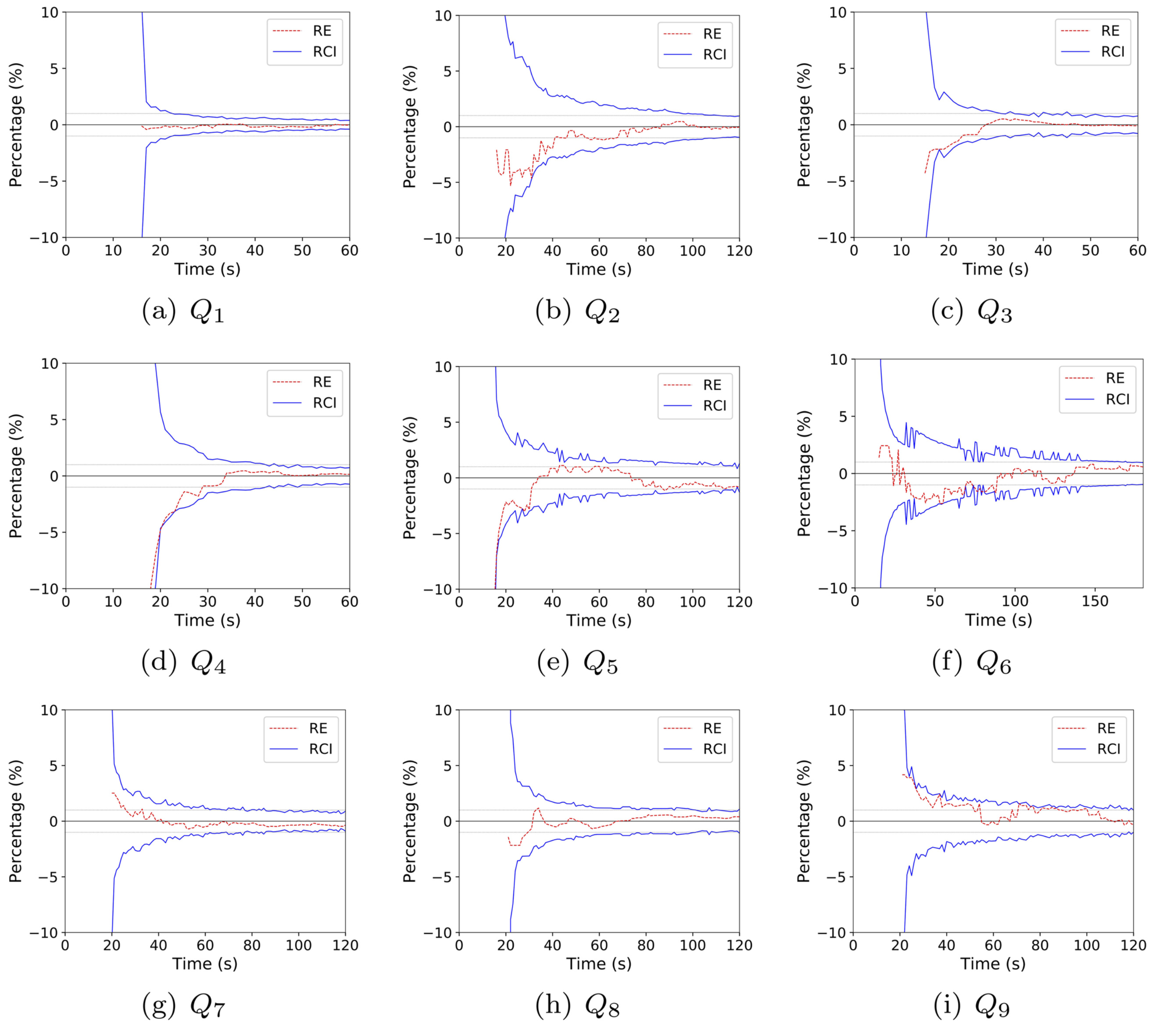


Fig. 4 The convergence of RE and RCI for Q_1–Q_9

matter whether there is a rearrangement or even multiple times of rearrangement in theory.

5 Experiment

5.1 Experiment Setup

We implemented the methods in our system—Parrot, and the baseline—blocking, on top of Spark 2.4.3. The baseline implements the same SQL parser and UDF as Parrot while reads documents from the underlying collection directly. The baseline processes the data in a blocking fashion which means the user cannot get the feedback result until all data has been processed. When the SQL is submitted, the baseline will parse the SQL and submit the job to the Spark cluster. The data source of this job is the entire text collection. On the contrary, the data source of our method is a bunch of samples, which are much smaller than the entire text collection. All the following experiments are performed on a 10-node cluster (each with Intel Xeon E5-2620, 64GB RAM, and 1.77TB HDD) under Apache Spark 2.4.3 and Ubuntu Linux 14.04 LTS. Our data is stored in Hadoop distributed file system and organized in JSON format.

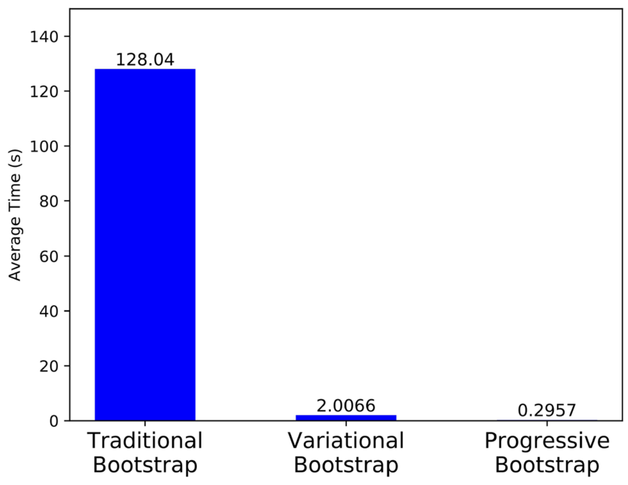


Fig. 5 Performance of bootstrap methods

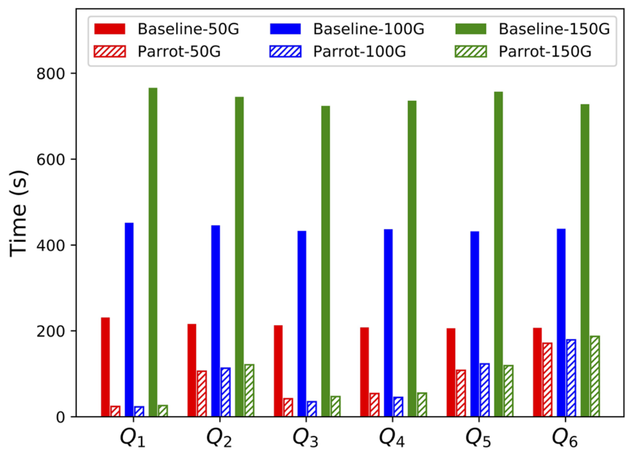


Fig. 6 Performance on different data sizes

5.1.1 Performance Metrics

Two metrics are used: (1) the relative error of the approximate result in each round; (2) the response latency for the first acceptable result. The relative error (*RE*) is calculated by $RE = |\hat{\theta} - \theta|/\theta$ where θ is the accurate result and $\hat{\theta}$ is the approximate result based on the sample. The relative confidence interval (*RCI*) is calculated by $RCI = (|\sigma_x - \sigma_y|/2)/\hat{\theta}$, where the $[\sigma_x, \sigma_y]$ represents the error bound. When the *RCI* of an approximate result is less than the given (1% as default in our experiment), we say it is an acceptable result. The confidence level is set to 95% as default.

5.1.2 Synthetic Dataset

We use the mix of Reuters news dataset[6] and Webhose English articles[7] as our dataset. After data cleaning and word segmentation, this dataset is about 10.9GB and contains about 3 million documents. Then we scale up the data to 100GB in proportion to ensure that the distribution and skewness are similar to the original. We built a uniform sample, a stratified sample on the *date* column, and a tail sample with $\tau = 0.01$ and $\lambda = 0.5$. The users could set these two values based on their requirements, while the smaller parameters make the faster convergence speed. We fix the block size of samples to 128MB. To evaluate the performance, we use the following 6 queries:

Q_1: **SELECT FREQ**('bank') **FROM** news;

Q_2: **SELECT FREQ**('bank') **FROM** news **WHERE** location='LONDON';

Q_3: **SELECT FREQ**('government') **FROM** news **WHERE** date **BETWEEN** '2015-10-01' **AND** '2015-10-31';

Q_4: **SELECT FREQ**('president') **FROM** news **WHERE** date **BETWEEN** '2008-01-01' **AND** '2008-01-31';

Q_5: **SELECT FREQ**('top-asia') **FROM** news;

Q_6: **SELECT FREQ**('chronology-bird') **FROM** news **WHERE** location <> 'PARIS';

Among them, Q_1, Q_2 run on the uniform sample, Q_3, Q_4 run on the stratified sample as their *where* clauses are on *date* column and Q_5, Q_6 run on the tail sample as words "*top-asia*" and "*chronology-bird*" are very rare words. Selectivity of Q_1 - Q_6 is shown in Table 3. The lower the selectivity, the larger the error and oscillation of the approximate result may occur. As Q_2–Q_6 are on very rare sub-populations, these queries can test the performance of all three kinds of samples comprehensively.

5.1.3 Real-World Dataset

The second dataset is a real-world Chinese dataset, which comprises about 60 million articles from the Sina website with size 64GB. We built a uniform sample, a stratified sample on the *channel* column and a tail sample with $\tau = 0.01$ and $\lambda = 0.5$. We fix the block size of samples to 128 MB. Then we use three queries, Q_7–Q_9,[8] to evaluate the performance. Selectivity of Q_7–Q_9 is shown in Table 4. In this experiment, Q_7, Q_8, Q_9 run on the uniform sample, stratified sample and tail sample respectively.

[6] https://trec.nist.gov/data/reuters/reuters.html.
[7] https://webhose.io/free-datasets/english-news-articles/.

[8] For convenience, we use English words with the same meaning in the paper.

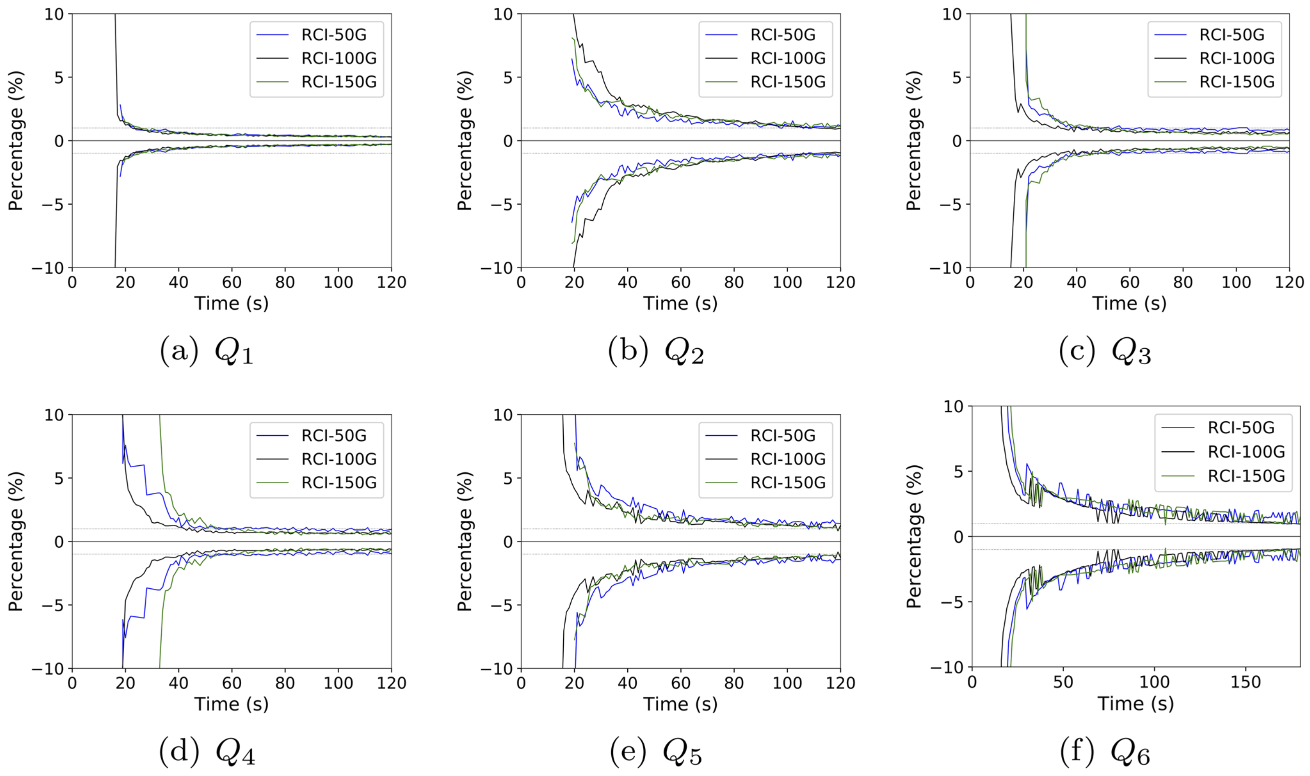


Fig. 7 The convergence of RCI for Q_1–Q_6 on different data sizes

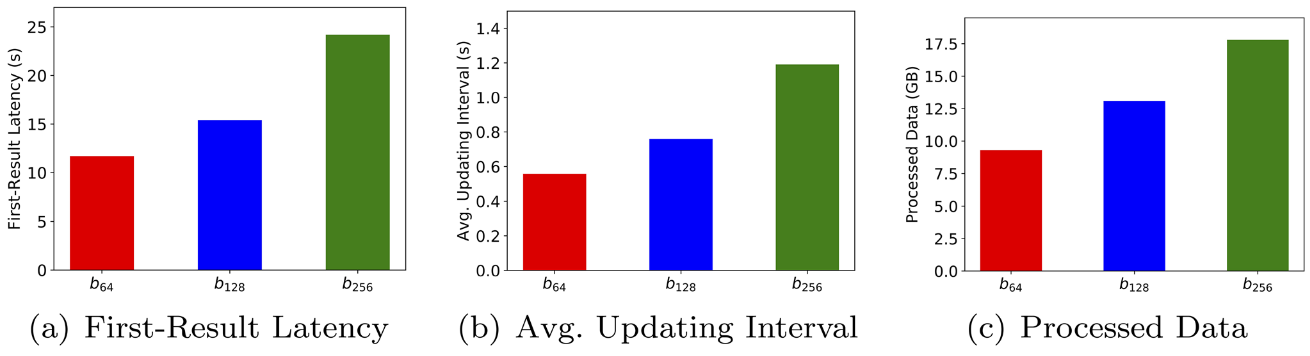


Fig. 8 Performance on different block sizes

Q_7: **SELECT FREQ**('Netizen') **FROM** article **WHERE** **LEN**(text) < 1000;

Q_8: **SELECT FREQ**('Female') **FROM** article **WHERE** channel='Health';

Q_9: **SELECT FREQ**('Guangxu') **FROM** article;

5.2 Experiment Results

5.2.1 Performance on Synthetic Dataset

In this experiment, we compare the performance between Parrot and baseline on the 100GB synthetic dataset by Q_1–Q_6. As shown in Fig. 3a, the time cost to get the first acceptable result of Parrot is much shorter than the baseline since Parrot uses a sample-based progressive execution model. Taking Q_1 as an example, Parrot costs about 23 seconds whereas blocking takes more than 450 seconds. However, Parrot brings different improvements for different queries. As we can see, Q_1, Q_3, Q_4 have more improvement than the other three. It's mainly due to two reasons: (1) for queries running on the same sample set, queries with higher selectivity may faster converge to 1% RCI as more documents can be selected within the same time; and (2) for queries of very low selectivity, RCI by bootstrap is much harder to converge to 1% as too few documents may cause unstable distribution of subsamples. Low selectivity may also cause fluctuations of RCI through the progressive process. Figure 4a–f show how RE and RCI converge during the execution process. The horizontal dotted line marks the 1% RCI. We can see that the *RE* and the *RCI* converge smoothly and fast except for some fluctuations when executing queries of very low selectivity (e.g. Q_6). Besides, Parrot can return the first result in about 15 s for all of the six queries. The length of this period mainly depends on block size and we will evaluate it in later experiments. Furthermore, the red line in these figures represents the real relative error, while the two blue lines show the estimated error interval by our progressive bootstrap. If the red line is in between the two blue lines, we can say the error estimation is accurate. Thus, as shown in Fig. 4a–f, our progressive bootstrap can give an accurate error estimation in most cases.

In summary, on the synthetic dataset, the first acceptable result of Parrot is 2.4×–19.7× faster compared with the blocking fashion. Besides, Parrot can provide a smooth trade-off between accuracy and latency. That means the proposed progressive execution model allows the user to get feedback results in a short latency and get more and more accurate results with waiting time getting longer and longer. Only queries with very small selectivity may lead to some fluctuations.

5.2.2 Performance on Rreal-World Dataset

In this experiment, we compare the performance between Parrot and baseline on the 64GB real-word dataset by Q_7–Q_9. As shown in Fig. 3b, Parrot can achieve 1% RCI at a very fast speed compared to the blocking fashion. For different queries, Parrot brings different improvement due to the same reasons as before. Figure 4g–i show how *RE* and *RCI* smoothly and fast converge. For all of the three queries, Parrot can return the first result in about 20 seconds. The latency is different from that of the first six queries because the two datasets have different deserialization costs. Through the execution process, the relative error falls into the confidence interval almost all the time, and hence our progressive bootstrap can give an accurate error estimation.

In summary, on the real-world dataset, the first acceptable result of Parrot is 5.7x–9.7x faster, compared with the blocking fashion. Besides, Parrot can provide a smooth trade-off between accuracy and latency.

5.2.3 Performance of Bootstrap

In this experiment, we compare the performance of three different bootstrap methods—traditional bootstrap, variational subsampling bootstrap, and progressive bootstrap. We run Q_1 on the 100GB synthetic dataset and record the time cost of the error estimation phase in each round from the beginning until the first acceptable result returned. The average time cost is shown in Fig. 5. We can see that traditional bootstrap costs more than 128 seconds on average while it's unacceptable in our interactive exploration scenario. The variational subsampling bootstrap costs from 0.307 to 4.14 seconds with an average of 2.0066 seconds. It's fast in the first few rounds and then becomes slower and slower with more and more data being processed as its time complexity is proportional to the amount of data that has been proceeded. Our progressive bootstrap gets the best performance with an average cost of 0.2957 seconds and can perform bootstrap within almost a fixed time cost. The cost mainly depends on the block size.

5.2.4 Effect of Data Size

In this experiment, we evaluate the effect of text data size. We use three text collections scaled from the synthetic dataset with size 50GB, 100GB, and 150GB. We generate samples of these three collections with the same parameters and the fixed block size (i.e., 128MB). Then we run Q_1–Q_6, both through Parrot and baseline. As shown in Fig. 6, the time cost for baseline increases with the data size increases (e.g., 100GB 452s vs. 150GB 766s for Q_1) since it needs to calculate on the entire dataset. We also find that the data size has a limited effect on the time cost of the first acceptable result (1% error bound) by Parrot (Fig. 6) and the confidence interval of Parrot converges quickly (Fig. 7), because Parrot mainly relies on a sufficient number of documents in the sample to be processed. Therefore, Parrot can provide a good and stable performance on large text collections.

5.2.5 Effect of Block Size

In this experiment, we evaluate the effect of different block sizes on the 100GB synthetic dataset. We construct three groups of samples under the same parameters except for the block size—64 MB, 128 MB, and 256MB, respectively. We run Q_1 by Parrot on the three groups of samples and record the time cost of the first estimate result, the average updating interval of result, and the processed data within 100 seconds. We use b_{64}, b_{128}, b_{256} to represent the block sizes with 64MB, 128MB, and 256MB respectively. Fig. 8a shows that smaller the block size, the shorter the latency of the first estimate result will be (e.g., b_{64} 11.7s vs. b_{128} 15.4s). Figure 8b shows that the increment of block size cause longer result updating interval (e.g., b_{64} 0.558s vs. b_{128} 0.759s). That's because smaller block size has less I/O and CPU cost for processing each single block. Figure 8c shows that smaller block size results in less data being processed within the same time (e.g., b_{64} 9.3GB vs. b_{128} 13.1GB) since smaller block size leads to more shuffle overhead. Therefore, it is a trade-off between first-result latency, updating interval and query accuracy.

6 Related Work

6.1 Interactive Exploration on Text

For structured data, lots of previous works attempt to speed up query execution through AQP (Approximate Query Processing) technique [2, 9, 23], which aims to find an approximate answer by samples [28] as close as to the exact answer efficiently. While limited by response time and computing resources, the before-mentioned AQP systems only return a single approximate result. However, there is an increasing need for interactive human-driven exploratory analysis, whose desired accuracy cannot be known a priori and change dynamically based on unquantifiable factors [29]. For semi-structured and unstructured data, the state-of-the-art solutions are based on the content management system or the cube structure, such as ElasticSearch [1] and Text Cube [19]. ElasticSearch supports simple queries with key-value based filtering as well as full-text searching for fuzzy matching over the entire dataset. But it doesn't have good support for ad-hoc queries of term frequency on a subset. Text Cube uses techniques to pre-aggregate data and gives the user the possibility to make semantic navigation in the data dimension but requires extensive preprocessing and suffers from the curse of dimensionality.

6.2 Error Estimation

To make approximate answers useful, lots of error estimation techniques have been proposed—the earliest being closed-form estimates based on either the central limit theorem (CLT) [26] or large deviation inequalities such as Hoeffding bounds [12]. These techniques either compute an error bound much wider than the real which lost guidance to users or require data to follow the normal distribution while it's not suitable for natural languages. Another estimation technique, bootstrap [23, 30], can be applied to arbitrary queries. However, before bootstrap techniques have poor performance to apply in our progressive execution model due to lots of duplicate computation.

7 Conclusion and Future Work

In this paper, we propose a new query formulation by extending SQL grammar with UDF for term frequency calculation on text data. We apply AQP techniques to return an approximate result within a short time. We present a sample-based progressive processing model and progressive bootstrap to continuously refine the approximate result. We implement these methods in the system called Parrot. Experiment results show that Parrot is about 2.4×–19.7× faster than the blocking fashion for the first acceptable result and can provide a smooth trade-off between accuracy and latency. Meanwhile, the quantified error bound covers the accurate result well.

For future work, we will support more text analysis methods (e.g., LDA) and try to reduce the storage cost of the pre-computed samples. In addition, we will introduce the machine learning to Parrot. For example, we may train a machine learning model that represents the pre-computed samples to accelerate the query execution.

Acknowledgements This work is supported by the National Key R&D Program of China (Nos. 2018YFB1004404 and 2018YFB1402600), the NSFC (Nos. 61732004 and 61802066) and the Shanghai Sailing Program (No. 18YF1401300).

References

1. 7.4.2, E.S. (2019). https://www.elastic.co
2. Acharya S, Gibbons PB, Poosala V, Ramaswamy S (1999) The aqua approximate query answering system. In: Delis A, Faloutsos C, Ghandeharizadeh S (eds) SIGMOD 1999, Proceedings ACM SIGMOD international conference on management of data, June

1–3, Philadelphia, Pennsylvania, USA, ACM Press, pp 574–576 (1999). https://doi.org/10.1145/304182.304581

3. Agarwal S, Milner H, Kleiner A, Talwalkar A, Jordan MI, Madden S, Mozafari B, Stoica I (2014) Knowing when you're wrong: building fast and reliable approximate query processing systems. In: Dyreson CE, Li F, Özsu MT (eds) International conference on management of data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, ACM, pp 481–492 (2014). https://doi.org/10.1145/2588555.2593667
4. Agarwal S, Mozafari B, Panda A, Milner H, Madden S, Stoica I (2013) Blinkdb: queries with bounded errors and bounded response times on very large data. In: Hanzálek Z, Härtig H, Castro M, Kaashoek MF (eds) Eighth Eurosys conference 2013, EuroSys '13, Prague, Czech Republic, April 14–17, ACM, pp. 29–42 (2013). https://doi.org/10.1145/2465351.2465355
5. Bouakkaz M, Ouinten Y, Loudcher S, Strekalova Y (2017) Textual aggregation approaches in OLAP context: a survey. Int J Inf Manag 37(6):684–692. https://doi.org/10.1016/j.ijinfomgt.2017.06.005
6. Corral A, Boleda G, Ferrer-i-Cancho R (2014) Zipf's law for word frequencies: word forms versus lemmas in long texts. CoRR **abs/1407.8322** (2014). arXiv: org/abs/1407.8322
7. Dimitriadou K, Papaemmanouil O, Diao Y (2014) Interactive data exploration based on user relevance feedback. In: Workshops proceedings of the 30th international conference on data engineering workshops, ICDE 2014, Chicago, IL, USA, March 31–April 4, 2014, IEEE Computer Society, pp 292–295 (2014). https://doi.org/10.1109/ICDEW.2014.6818343
8. Efron B (1992) Bootstrap methods: another look at the jackknife. In: Breakthroughs in statistics, Springer, pp 569–593
9. Galakatos A, Crotty A, Zgraggen E, Binnig C, Kraska T (2017) Revisiting reuse for approximate query processing. PVLDB 10(10):1142–1153. https://doi.org/10.14778/3115404.3115418. http://www.vldb.org/pvldb/vol10/p1142-galakatos.pdf
10. Gray J, Chaudhuri S, Bosworth A, Layman A, Reichart D, Venkatrao M, Pellow F, Pirahesh H (2007) Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotals. CoRR **abs/cs/0701155**. arXiv:org/abs/cs/0701155
11. Griffin T, Libkin L (1995) Incremental maintenance of views with duplicates. In: Carey MJ, Schneider DA (eds) Proceedings of the 1995 ACM SIGMOD international conference on management of data, San Jose, California, USA, May 22–25, 1995, ACM Press, pp 328–339. https://doi.org/10.1145/223784.223849
12. Haas PJ, Haas PJ (1996) Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM
13. Hellerstein JM, Haas PJ, Wang HJ (1997) Online aggregation. In: Peckham J (ed) SIGMOD 1997, Proceedings ACM SIGMOD international conference on management of data, May 13–15, 1997, Tucson, Arizona, USA, ACM Press, pp. 171–182. https://doi.org/10.1145/253260.253291
14. Idreos S, Kersten ML, Manegold S (2007) Database cracking. In: CIDR 2007, Third biennial conference on innovative data systems research, Asilomar, CA, USA, January 7–10, 2007, Online Proceedings, pp 68–78. www.cidrdb.org. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf
15. Jain AK, Dubes RC, Chen C (1987) Bootstrap techniques for error estimation. IEEE Trans Pattern Anal Mach Intell 9(5):628–633. https://doi.org/10.1109/TPAMI.1987.4767957
16. Kleiner A, Talwalkar A, Sarkar P, Jordan MI (2012) The big data bootstrap. In: Proceedings of the 29th international conference on machine learning, ICML 2012, Edinburgh, Scotland, UK, June 26–July 1, 2012. icml.cc/Omnipress. http://icml.cc/2012/papers/861.pdf
17. Koch C, Ahmad Y, Kennedy O, Nikolic M, Nötzli A, Lupei D, Shaikhha A (2014) Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J 23(2):253–278. https://doi.org/10.1007/s00778-013-0348-4
18. Li K, Li G (2018) Approximate query processing: What is new and where to go? A survey on approximate query processing. Data Sci Eng 3(4):379–397. https://doi.org/10.1007/s41019-018-0074-4
19. Lin CX, Ding B, Han J, Zhu F, Zhao B (2008) Text cube: computing IR measures for multidimensional text database analysis. In: Proceedings of the 8th IEEE international conference on data mining (ICDM 2008), December 15–19, 2008, Pisa, Italy, IEEE Computer Society, pp 905–910 (2008). https://doi.org/10.1109/ICDM.2008.135
20. Lins LD, Klosowski JT, Scheidegger CE (2013) Nanocubes for real-time exploration of spatiotemporal datasets. IEEE Trans Vis Comput Graph 19(12):2456–2465. https://doi.org/10.1109/TVCG.2013.179
21. Liu Z, Jiang B, Heer J (2013) imMens: real-time visual querying of big data. Comput Graph Forum 32(3):421–430. https://doi.org/10.1111/cgf.12129
22. Palpanas T, Sidle R, Cochrane R, Pirahesh H (2002) Incremental maintenance for non-distributive aggregate functions. In: Proceedings of 28th international conference on very large data bases, VLDB 2002, Hong Kong, August 20–23, 2002, Morgan Kaufmann, pp 802–813. https://doi.org/10.1016/B978-155860869-6/50076-7. http://www.vldb.org/conf/2002/S22P04.pdf
23. Park Y, Mozafari B, Sorenson J, Wang J (2018) Verdictdb: universalizing approximate query processing. In: Das G, Jermaine CM, Bernstein PA (eds) Proceedings of the 2018 international conference on management of data, SIGMOD conference 2018, Houston, TX, USA, June 10–15, ACM, pp 1461–1476 (2018). https://doi.org/10.1145/3183713.3196905
24. Parr T, Fisher K (2011) Ll(*): the foundation of the ANTLR parser generator. In: Hall MW, Padua DA (eds) Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011, ACM, pp 425–436. https://doi.org/10.1145/1993498.1993548
25. Pol A, Jermaine C (2005) Relational confidence bounds are easy with the bootstrap. In: Özcan F (ed) Proceedings of the ACM SIGMOD international conference on management of data, Baltimore, Maryland, USA, June 14–16, 2005, ACM, pp 587–598. https://doi.org/10.1145/1066157.1066224
26. Rice JA (2006) Mathematical statistics and data analysis. Cengage Learning
27. Rossi RJ (2018) Mathematical statistics: an introduction to likelihood based inference. Wiley, New York
28. Wu Z, Jing Y, He Z, Guo C, Wang XS (2019) Polytope: a flexible sampling system for answering exploratory queries. World Wide Web, pp 1–22
29. Zeng K, Agarwal S, Stoica I (2016) iolap: managing uncertainty for efficient incremental OLAP. In: Özcan F, Koutrika G, Madden S (eds) Proceedings of the 2016 international conference on management of data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26–July 01, ACM, pp 1347–1361 (2016). https://doi.org/10.1145/2882903.2915240
30. Zeng K, Gao S, Mozafari B, Zaniolo C (2014) The analytical bootstrap: a new method for fast error estimation in approximate query processing. In: Dyreson CE, Li F, Özsu MT (eds) International conference on management of data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014, ACM, pp 277–288. https://doi.org/10.1145/2588555.2588579
31. Zgraggen E, Galakatos A, Crotty A, Fekete J, Kraska T (2017) How progressive visualizations affect exploratory analysis. IEEE Trans Vis Comput Graph 23(8):1977–1987. https://doi.org/10.1109/TVCG.2016.2607714

32. Zhang S, Sun C, He Z (2016) Listmerge: accelerating top-k aggregation queries over large number of lists. In: Navathe SB, Wu W, Shekhar S, Du X, Wang XS, Xiong S (eds) Database systems for advanced applications—21st international conference, DASFAA 2016, Dallas, TX, USA, April 16–19, 2016, Proceedings, Part II, lecture notes in computer science, vol 9643, Springer, pp 67–81.https://doi.org/10.1007/978-3-319-32049-6_5