Demonstrating the Voice-Based Exploration of Large Data Sets with CiceroDB-Zero

Immanuel Trummer
Cornell University
Ithaca, NY (USA)
it224@cornell.edu

ABSTRACT

This demonstration enables participants to explore large data sets via voice interfaces. The focus of the demonstration is on methods generating concise speech descriptions of query results, specified by users via voice input. The technical novelty of the demonstrated system lies in the fact that processing overheads are mostly moved into a pre-processing phase, generating speeches for batches of queries defined via templates. Visitors can access the demo via smart speakers on-site or via their own smart phones. They will be able to customize the generated voice descriptions and to tune voice output methods.

PVLDB Reference Format:
Immanuel Trummer. Demonstrating the Voice-Based Exploration of Large Data Sets via CiceroDB-Zero. *PVLDB*, 13(12): 2869-2872, 2020.
DOI: https://doi.org/10.14778/3415478.3415496

1. INTRODUCTION

Voice interfaces are becoming more and more popular. This is evidenced by the rise of devices and services (e.g., Google Home, Amazon Alexa, or Siri) that rely on voice as the primary medium of communication. Voice interfaces are most natural for many users. They are convenient in situations where hands or the visual attention are bound [1] and benefit user groups (e.g., visually impaired users) who cannot use traditional interfaces.

This demonstration will enable participants to explore large data sets via voice interfaces. Also, participants will be able to compare different methods for generating voice descriptions of query results in terms of efficiency and quality of the generated description. The demonstration setup will feature a smart speaker via which participants can issue voice queries and hear data summaries. Also, participants will have access via their phones to a publicly available online version of the demo.

EXAMPLE 1. *An early demo version is already publicly available. Say "OK Google, talk to Developer Facts" to your Google Assistant to explore the results of the 2019 Stack Overflow Developer Survey. Ask for instance "What's the job satisfaction in the US?". The system answers with "73 percent of employees are satisfied. For novice coders, even 76 percent of employees are satisfied. For a salary above 100K, 75 percent of employees are satisfied.".*

The focus of the demonstration differs from prior demonstrations on voice-based data access. Systems such as EchoQuery [1] demonstrate that voice input can be translated into SQL queries. However, they read out query results row by row (up to a certain row threshold) which works best for queries returning single rows or (non-grouped) aggregates. The focus of the demonstration is on exploratory analysis of large data sets instead. Here, we assume that user queries return potentially large result sets. The key problem is then to summarize the result via voice output.

Voice output needs to be very concise to avoid overwhelming the listener [6]. This means that only few facts can be transmitted. Hence, one must carefully select which pieces of information to share. The demonstrated system formalizes the generation of data summaries as an optimization problem. Given limits on the number of facts to describe, the goal is to select the combination of facts that gives users the best approximation of the associated data. I.e., when asking users to estimate values in specific rows and columns after listening to a speech description, the goal is to minimize their expected estimation error.

Of course, the number of facts that can be formed about a data set is large. The number of possible speeches grows exponentially in the speech length. Naively exploring all possible speeches to identify the optimal one is therefore prohibitive. Sampling has been proposed as a method to select speech descriptions [6]. However, sampling does not guarantee optimal speeches nor that those speeches use accurate facts. Incremental approaches [7], overlapping voice output with processing, reduce latency but not overall processing overheads. The goal of the demonstrated system is to reduce per-query overheads at run time to near zero, hence the name "CiceroDB-Zero". This is achieved by an (expensive) one-time pre-processing step, calculating near-optimal speech descriptions for results of queries that instantiate given templates. Low per-query overheads are particularly important in scenarios in which data is served to a large number of users via voice interfaces. Our current version runs as application in the Cloud and can be accessed via smart phones or smart speakers. Having negligible per-query overheads translates into negligible monetary fees for Cloud-based processing.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415496

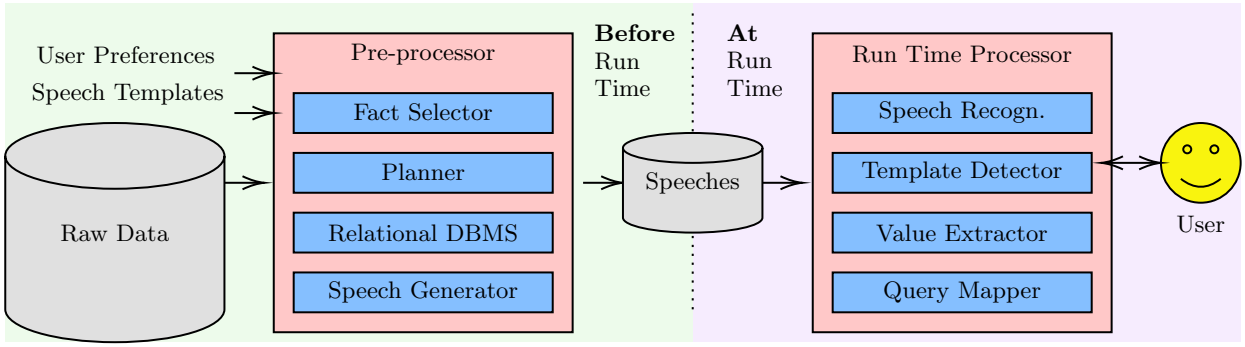

Figure 1: CiceroDB-Zero prepares speeches summarizing results for query templates in a pre-processing step. At run time, user queries are mapped to speeches of the closest available query.

2. PROBLEM MODEL

Our goal is to summarize (relational) query results via speech output. We consider results with one numerical column (the **Value Column**) and multiple (categorical) **Dimension Columns**. Such results are typical for queries in OLAP sessions.

DEFINITION 1. *We model a **Speech** as a collection of facts. Each **Fact** describes an aggregate (i.e., average, maximum, or minimum) of the value column for a data subset, defined by equality predicates on a subset of dimension columns. The **Speech Length** is the number of contained facts. **User Preferences** constrain generated speeches. We support constraints on maximal speech length and on dimensions that must be mentioned in facts. Furthermore, we can restrict the complexity of generated facts (measured by the number of restricted dimension columns).*

Our goal is to find the best speech summary, given constraints on speech length. We judge speeches in terms of how closely they approximate the actual data.

DEFINITION 2. *Given a result set R, we can model the effect of a speech $S \in \mathcal{S}$ (where $\mathcal{S}$ is the space of possible speeches) by a function $\mathcal{B}(S) : \pi_{dims}(R) \mapsto \mathbb{R}$. This function models the **Belief** of users after listening to S about the value column for specific rows, based on their dimension values (extracted via function π_{dims}). Our belief function is based on a simple user model following prior work [7]. We also assume a domain-specific prior that users apply in the absence of more specific information. We can measure **Error** as the distance between belief and actual result: $\mathcal{E}(S,R) = \sum_{r \in R} |\pi_{value}(r) - \mathcal{B}(S)[r]|$. Finally, we can quantify the **Utility** of a speech as $\mathcal{U}(S,R) = \mathcal{E}(\emptyset,R) - \mathcal{E}(S,R)$.*

Our goal is to find speeches with maximal utility within the entire search space.

EXAMPLE 2. *Imagine the following query result. Rows represent software developers and specify company size, level of experience, and the yearly salary: (large, novice, 70K), (small, novice, 60K), (small, expert, 100K), and (large, expert, 110K). Assume that at most two facts are allowed. Summarizing average salary as "Large companies pay 90K. Small companies pay 80K." would make listeners estimate a salary of 90 K for first and last row and a salary of 80K for second and third row. This means an accumulated error of four times 20K, i.e. 80K. Summarizing as "Novice coders earn 65K. Expert coders earn 105K." leads to per-row errors of 5K, therefore 20K in total. Hence, the second speech is preferable.*

3. APPROACH OVERVIEW

Figure 1 shows a high-level overview of CiceroDB-Zero. It features two primary components: a pre-processing component and a run time component. The goal of pre-processing is to generate text summaries for queries following given templates. At run time, user speech input is mapped to the closest query for which a summary is available. This summary is transmitted via voice output.

The input for pre-processing is a (potentially large) data set, stored as relational database, a set of query templates with associated text templates for generating summaries, and preferences determining for instance the length of the generated summaries. During pre-processing, we consider each query instantiating one of the given query templates. The output of pre-processing is a concise speech summarizing the result of each query on the input data. Each speech is a collection of facts. The Fact Selector selects for each query an optimal combination of facts, given constraints on speech length. To generate facts and to evaluate quality for fact sets, it issues SQL queries to a Relational DBMS in which the input data is stored. It exploits a cost-based Planner to choose the most efficient evaluation strategy. The Speech Generator generates text summaries, given fact combinations and text templates.

At run time, we map user speech input to one of the summaries generated during pre-processing. This summary is transmitted via voice output to the user. The run time component uses a Speech Recognition component to transform speech input into text. It uses a Template Detector to map input to one of the pre-defined query templates and a Value Extractor to substitute template placeholders by concrete values. The Query Mapper maps the user query to the closest query that was considered during pre-processing. The summary of that query's result is spoken out to the user. The following subsections provide more details.

3.1 Pre-Processing

During pre-processing, we process batches of queries efficiently, generating near-optimal speech summaries of query

```
1: // Expand speeches S_t for query template t on data D
2: // by best fact according to user preferences U.
3: function EXPAND(D, t, S_t, U)
4:     // Plan expansion via cost-based optimization
5:     ⟨F_1, P_1⟩, ..., ⟨F_n, P_n⟩ ←PLANNER(D, t, S_t, U)
6:     // Initialize remaining fact groups
7:     R ← ∪_{1≤i≤n} F_i
8:     // Iterate over expansion plan steps
9:     for i ← 1, ..., n do
10:        // Update speeches with new fact group
11:        S_t ←UPDATESPEECHES(D, t, S_t, U, F_i ∩ R)
12:        // Prune out dominated fact groups
13:        R ← R\PRUNE(D, t, S_t, U, P_i ∩ R)
14:    end for
15:    return S_t
16: end function
17: // Generate near-optimal speeches for templates T
18: // on data D with user preferences U.
19: function PREPROCESSING(D, T, U)
20:    R ← ∅
21:    // Iterate over query templates
22:    for t ∈ T do
23:        S_t ←EMPTYSPEECHES
24:        // Add facts until speech length limit
25:        for i ← 1, ..., U.length do
26:            // Expand speeches by optimal facts
27:            S_t ←EXPAND(D, t, S_t, U)
28:        end for
29:        R ← R ∪ {S_t}
30:    end for
31:    return R
32: end function
```

Algorithm 1: Generate near-optimal speeches summarizing results for all queries following query templates.

results. Algorithm 1 is the pre-processing algorithm, represented at a high level of abstraction. The input to pre-processing is a database, a set of query templates, and user preferences with regards to speech generation, including a bound on speech length.

We iterate over all given query templates. For each template, optimal speeches are generated in a batch operation for all associated queries. Speeches are defined as collections of facts. A naive approach iterates over all possible combination of facts (up to the given size limit) and evaluates quality. This guarantees an optimal speech, according to the quality model presented in Section 2, but is prohibitively expensive. Instead, we construct optimal speeches by repeatedly adding the most informative facts (loop starting in Line 25). This approach guarantees speeches whose utility is within a factor of 66% of the optimum (a consequence of the submodularity property of our speech utility function [2], i.e. adding more facts becomes less and less informative).

The EXPAND function expands each speech for the current query template by the most informative fact. A naive approach iterates over all facts and compares their utility. The number of available facts can however be large, rendering this approach inefficient. To increase efficiency, we try to prune fact groups early, without iterating over all contained facts. This becomes possible if we can establish that no fact in the corresponding group can optimally expand any speech. We establish upper bounds on the utility that can be achieved by adding facts from certain groups. We prune by comparing those upper bounds against utility achieved by the best facts considered so far.

Calculating bounds and trying to prune out fact groups creates overheads as well. Neither using no pruning nor using pruning whenever possible yields optimal performance. Hence, we use a cost-based optimizer that determines the most promising expansion strategy. It generates expansion plans that are characterized by a sequence of pruning steps. Each step is characterized by a pair $\langle F_i, P_i \rangle$ with the following semantics. Parameter F_i is a group of facts that is considered for speech expansions. Parameter P_i is a group of facts that is used as target for pruning. This means that we invest time to calculate upper utility bounds for P_i, potentially enabling us to prune out fact subsets. To maximize the effect of pruning, we also exploit relationships between different fact groups. E.g., an upper bound on the utility of a fact group also yields bounds for facts referring to more specific data subsets. Once Function EXPAND terminates, each fact has been either considered for expansion or pruned as part of a dominated fact group. Finally (not shown in Algorithm 1), we generate text summaries, based on the selected fact groups (by filling in given text templates).

3.2 Run Time

At run time, speech input by users is translated into corresponding queries on data. CiceroDB-Zero is based on the Google Assistant framework[1] which offers functions for speech recognition and natural language processing. The Google Assistant framework models interactions with users via so called "Intents", characterized by input examples and associated processing logic. In our scenario, each query template corresponds to an intent. We provide a few tens of example queries for each intent. Based on those samples and using machine learning, the Google Assistant framework is able to extract concrete values for placeholders in the query template.

Queries (templates with values for each parameter) are forwarded to our processing backend. This backend accesses speech summaries generated during pre-processing. If the input query matches one of the queries, treated during pre-processing, the associated speech summary is returned. Otherwise, the input query is mapped to the most similar query considered during pre-processing. For instance, for queries with equality predicates, we consider query generalizations obtained by removing equality predicates. Having the choice between multiple generalizations, we prefer the ones that require removing the smallest number of predicates.

Having selected one of the speech summaries generated during pre-processing, we forward the selected speech to the frontend where it is translated into speech output. In addition to the speech summary, we generate a short preamble informing users of the query that is summarized. Doing so is important due to the inherent challenges in speech recognition and allows users to verify whether the answer matches their query intent. Also, it addresses cases in which the given answer is more general than the user query.

[1] `https://assistant.google.com/`

4. DEMONSTRATION

The following subsections describe the demonstration setup and all the ways in which visitors may participate.

4.1 Demonstration Setup

In terms of hardware, the demonstration stand will feature a Google Nest Mini smart speaker, at least one Android tablet or smart phone with head phones, and a laptop (running the pre-processing software). Visitors may interact with the Google Nest smart speaker on which the voice interface is accessible. Voice interactions with that device typically work from a distance of a few meters. This means that the smart speaker can serve small groups of users gathering around it, answering questions from different visitors while the others are listening. If specific visitors want to have an uninterrupted dialog with the interface, they may borrow the Android device with headphones. Additionally, the stand will advertise instructions for users on how to access the publicly available Google Assistant service from their smart phones.

4.2 Modes of Interaction

Visitors can explore three data sets with a default configuration: results of the Stack Overflow Developer Survey 2019[2], a data set on flight cancellations and delays[3], and polling data provided by Fivethirtyeight[4]. During preprocessing, we consider query templates that restrict up to two dimension columns via equality predicates. Voice access to data is available via the smart speaker at the conference stand as well as via the smart phones of visitors.

Visitors will also be able to influence the generated voice descriptions. Interested visitors have access to a laptop running the pre-processing software. Pre-processing time for the three aforementioned data sets ranges from a few seconds (polling) to a few minutes (developer survey). Visitors may vary the following parameters. First, visitors may define priors as SQL expressions (describing assumed values for target rows), modeling a-priori assumptions of listeners about data. Changing prior leads to different voice descriptions as the system focuses on aspects that are assumed to be unknown. Second, visitors may change the set of dimension columns that are prioritized when formulating voice descriptions. This will highlight different trends in the data. Third, visitors may change per-row weights used to calculate approximation error when comparing speech descriptions. This will change the focus of voice output. Finally, visitors may change parameters related to the size and complexity of speech descriptions, as well as to the templates used for generating speeches (e.g., enabling or disabling addition of adjectives to highlight particularly remarkable trends).

Depending on the network conditions at the conference venue, participants will also be able to download their own data sets (preferably in .csv format) to make them accessible for voice output. According to the experiences of the author, preparing new data sets for voice interactions typically takes a few minutes only. This should be possible with interested visitors outside of periods of peak attendance.

Users who are interested in the internal details will also be able to change flags, determining how pre-processing is executed. For instance, visitors will be able to switch off fact pruning or select naive speech generation approaches (evaluating all possible speeches). Varying those parameters leads to significantly worse performance. For instance, disabling fact pruning increases pre-processing time by up to factor three for the aforementioned data sets on a MacBook Air with 8 GB of RAM and 1.8 GHz CPU. Switching to naive speech enumeration increases pre-processing time by up to factor 20 for the same platform. Visitors will be able to see the sequence of SQL queries generated during pre-processing for all variants.

[2] https://insights.stackoverflow.com/survey/2019

[3] https://www.bts.gov/topics/airlines-and-airports-0

[4] https://fivethirtyeight.com/

5. RELATED WORK

CiceroDB-Zero continues a sequence of recent publications [5–8], focused on the problem of "data vocalization" (i.e., how to summarize data best via speech descriptions). A technical report describing the demonstrated approach in more detail is available online[5]. None of the aforementioned predecessor systems has been demonstrated at any conference. The main technical novelty lies in the fact that expensive data processing is moved to a pre-processing step. This differs from prior work which either uses standard query processing [8], scenario-specific sampling methods [6], or sampling and incremental processing [5, 7]. CiceroDB-Zero avoids inaccurate results due to sampling. Also, it reduces run time overheads to close to zero (including monetary costs in a Cloud scenario). In contrast, overlapping speaking with processing [5, 7] only reduces latency. Other recent work on voice interfaces for data access [1,3,4] is complementary to CiceroDB-Zero as it does not focus on summarizing large data sets via voice output.

6. CONCLUSION

The proposed demonstration allows visitors to explore large data sets via voice interfaces on a multitude of devices. Also, visitors will be able to influence the methods by which voice output is generated, thereby gaining a better understanding of the internals.

7. REFERENCES

[1] G. Lyons, V. Tran, C. Binnig, U. Cetintemel, and T. Kraska. Making the case for Query-by-Voice with EchoQuery. In *SIGMOD*, pages 2129–2132, 2016.

[2] G. Nemhauser and L. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of Operations Research*, 3(3):177–188, 1978.

[3] V. Shah, S. Li, A. Kumar, and L. Saul. SpeakQL: towards speech-driven multimodal querying of structured data. Technical report, 2019.

[4] V. Shah, S. Li, K. Yang, A. Kumar, and L. Saul. Demonstration of SpeakQL: speech-driven multimodal querying of structured data. In *SIGMOD Demo Track*, pages 2001–2004, 2019.

[5] I. Trummer. Data Vocalization with CiceroDB. In *CIDR*, 2019.

[6] I. Trummer, M. Bryan, and R. Narasimha. Vocalizing large time series efficiently. *PVLDB*, 11(11):1563–1575, 2018.

[7] I. Trummer, Y. Wang, and S. Mahankali. A holistic approach for query evaluation and result vocalization in voice-based OLAP. In *SIGMOD*, pages 936–953, 2019.

[8] I. Trummer, J. Zhu, and M. Bryan. Data vocalization: optimizing voice output of relational data. *PVLDB*, 10(11):1574–1585, 2017.

[5] http://www.itrummer.org/drafts/cicerodbzero.pdf