BEAS: Bounded Evaluation of SQL Queries

Yang Cao\(^1\), Wenfei Fan\(^1\), Yanghao Wang\(^1\), Tengfei Yuan\(^1\) \quad Yanchao Li\(^2\) \quad Laura Yu Chen\(^3\)

\(^1\)University of Edinburgh \quad \(^2\)Nanjing University of Science and Technology \quad \(^3\)Huawei America Research Center

Abstract

We demonstrate BEAS, a prototype system for querying relations with bounded resources. BEAS advocates an unconventional query evaluation paradigm under an access schema \(A\), which is a combination of cardinality constraints and associated indices. Given an SQL query \(Q\) and a dataset \(D\), BEAS computes \(Q(D)\) by accessing a bounded fraction \(D_Q\) of \(D\), such that \(Q(D_Q) = Q(D)\) and \(D_Q\) is determined by \(A\) and \(Q\) only; no matter how big \(D\) grows. It identifies \(D_Q\) by reasoning about the cardinality constraints of \(A\), and fetches \(D_Q\) using the indices of \(A\). We demonstrate the feasibility of bounded evaluation by walking through each functional component of BEAS. As a proof of concept, we demonstrate how BEAS conducts CDR analyses in telecommunication industry, compared with commercial database systems.

1. Introduction

Querying big relations is often beyond reach for small companies. It may take hours to join tables of millions of tuples. Given a query \(Q\) and a dataset \(D\), it is NP-complete to decide whether a tuple is in \(Q(D)\) when \(Q\) is in SPC (selection, projection, Cartesian product). It is PSPACE-complete for \(Q\) in relational algebra [1]. One might think that parallelism would solve the problem by using more processors. However, small companies can often afford only limited resources.

Is it feasible to query big data with bounded resources?

One approach to addressing this challenge is based on bounded evaluation [5,8,9]. The idea is to use an access schema \(A\) over a database schema \(R\), which is a combination of cardinality constraints and associated indices. A query \(Q\) is boundedly evaluable under \(A\) if for each instance \(D\) of \(R\) that conforms to \(A\), there exists a small \(D_Q \subseteq D\) such that

\[ Q(D_Q) = Q(D), \]

\[ \text{the time for identifying } D_Q \text{ is decided by } Q \text{ and } A. \]

Intuitively, \(D_Q\) consists of only data needed for answering \(Q\). Its size \(|D_Q|\) is determined by \(Q\) and \(A\) only, not by \(|D|\).

BEAS. As a proof of concept of [5,8,9], we have developed BEAS [4], a prototype system for Bounded Evaluation of SQL. BEAS has the following unique features that differ from conventional query evaluation paradigm and DBMSs.

(1) **Quantified data access.** BEAS identifies \(D_Q\) by reasoning about the cardinality constraints in \(A\), and fetches \(D_Q\) by using the indices in \(A\). In the process it deduces a bound \(M\) on the amount of data to be accessed, and can hence decide whether \(Q\) is boundedly evaluable before \(Q\) is executed.

(2) **Reduced redundancy.** Using \(A\), BEAS fetches only (distinct) partial tuples needed for answering \(Q\). This reduces duplicated and unnecessary attributes in tuples fetched by traditional DBMS. It also reduces joins, in which the redundancies get inflated rapidly (see an example shortly).

(3) **Scalability.** Putting these together, BEAS computes \(Q(D)\) by accessing a bounded fraction \(D_Q\) of \(D\), no matter how big \(D\) grows. Hence to an extent, it makes big data analysis possible for small businesses with bounded resources.

(4) **Ease of use.** BEAS can be built on top of any conventional DBMS, and make seamless use of existing optimization techniques of DBMS. This makes it easy to extend DBMS with the functionality of bounded evaluation.

One of our industry collaborators has deployed and tested BEAS, and found that BEAS outperforms commercial DBMSs by orders of magnitude for more than 90% of their queries.

**Demo overview.** We demonstrate the bounded evaluation functionality of BEAS in two parts. (1) To illustrate how bounded evaluation works, we walk through each functional component of BEAS, from access schema discovery and maintenance to bounded query plan generation and execution. (2) To demonstrate the performance of BEAS, we adopt a real-life scenario from telecommunication industry for CDR (call detail record) analyses, and visualize how different query plans perform compared with commercial DBMSs.

Below we first present the foundation (Section 2) and the functional components (Section 3) of BEAS. We then propose a more detailed demonstration plan (Section 4).

2. Foundations of BEAS

We start with a review of access schema and bounded evaluability [5,8,9], which are the foundations of BEAS.

**Access schema.** Over a database schema \(R\), an access constraint \(\psi\) is of the form \(R(X \rightarrow Y, N)\), where \(R\) is a relation in \(R\), \(X, Y\) are sets of attributes of \(R\), and \(N\) is a natural number [5,8,9]. A relation instance \(D\) of \(R\) conforms to \(\psi\) if

\[ \text{o for any } X\text{-value } a \text{ in } D, |D_Y(X = a)| \leq N, \]

\[ \text{where } D_Y(X = a) = \{ t[Y] | t \in D, t[X] = a \}; \]

\[ \text{o there exists an index on } X \text{ for } Y \text{ that gives an } X\text{-value } a, \text{ retrieves } D_Y(X = a) \text{ by accessing at most } N \text{ tuples.} \]

That is, for any given \(X\)-value, there exist at most \(N\) distinct corresponding \(Y\) values in \(D\) (cardinality constraint), and the \(Y\) values can be fetched by using the index for \(\psi\) (index).

An access schema \(A\) over \(R\) is a set of access constraints over \(R\). A database instance \(D\) of \(R\) conforms to \(A\), denoted by \(D \models A\), if \(D\) conforms to each constraint in \(A\).

**Example 1:** Consider a commercial benchmark of schema \(\mathcal{R}_0\) from a telecommunication company (name withheld). It includes three (simplified) relations: (a) \(\text{call}(\text{pnum, recnum, date, region})\), recording that number \(\text{pnum}\) called \(\text{recnum}\) in region on date; (b) \(\text{package}(\text{pnum, pid, start, end, year})\), saying that \(\text{pnum}\) is in service package \(\text{pid}\) from month \(\text{start}\) to end in year; and (c) \(\text{business}(\text{pnum, type, region})\) says that business number \(\text{pnum}\) in region is of type, \(\text{e.g.}, \text{bank, hospital}\).

An access schema \(A_0\) over \(\mathcal{R}_0\) includes:

\[ \psi_1: \text{call}(\{\text{pnum, date}\}) \rightarrow \{\text{recnum, region}\}, 500, \]

\[ \psi_2: \text{package}(\{\text{pnum, year}\}) \rightarrow \{\text{pid, start, end}\}, 12, \]
Bounded evaluability. Underlying BEAS is the theory of bounded evaluability [5, 6, 8]. Given an access schema $A$ and a query $Q$, the key idea is to generate a bounded query plan that accesses data quantified by $A$, as illustrated below.

**Example 2:** Consider a benchmark query $Q$ to find regions in which there are numbers that were called by some business number $x$ in year $d_0$ in 2016, where $x$ was (a) of business type $t_0$ and (b) in region $r_0$ and (c) in service package $c_0$.

1. Select $\text{call.region}$ from $\text{call.packages, business}$ where $\text{business.type} = t_0$ and $\text{business.region} = r_0$ and $\text{business.pnum} = \text{call.pnum} = \text{and.call.date} = d_0$ and $\text{call.pnum} = \text{package.pnum} = \text{package.year} = 2016$ and $\text{package.start} \leq d_0$ and $\text{package.end} \geq d_0$ and $\text{package.pid} = c_0$.

Under access schema $A_0$ given in Example 1, $Q$ has a bounded query plan and can be answered as follows:

1. Fetch a set $T_1$ of at most 2000 $\text{pnum}'$s from relation $\text{business}$ by using the index for $\psi_3$ with key $(t_0, r_0)$.
2. For each $\text{pnum}$ in $T_1$, fetch at most 12 distinct $(\text{pid}, \text{start}, \text{end})$ triples using the index for $\psi_2$ with key $(\text{pnum}, 2016)$, yielding a set $T_2$ of at most $2000 \times 12$ partial $\text{package}$ tuples.
3. Select $\text{pnum}'$s from $T_2$ with $\text{start} \leq d_0 \leq \text{end}$ and $\text{pid} = c_0$.
4. For each $\text{pnum}$ selected in (3), fetch at most 500 $(\text{recnum}, \text{region})$ pairs from relation $\text{call}$ using the index for $\psi_1$ with key $(\text{pnum}, d_0)$. This yields a set $T_3$ of $\text{region}'$s as the query answer, by accessing at most $500 \times 2000 \times 12$ $\text{call}$ tuples.

The bounded plan accesses a set $D_Q$ of at most 2000 partial $\text{business}$ tuples, 24000 $\text{package}$ tuples and 12 million $\text{call}$ tuples in total, no matter how big the relations are.

As will be shown in the demo, on a 20GB database generated by the benchmark, BEAS can find exact answers to $Q$ in 96.13ms using the bounded plan, while a commercial DBMS takes 187.8s, i.e., BEAS is 1953 times faster, although it can find exact answers to $Q$ within bounded resources; otherwise, it generates a bounded query plan and computes exact answers to $Q$ within bounded resources; otherwise, it generates partially bounded plans and uses DBMS to compute exact answers (see Section 3).

**3. The Architecture of BEAS**

As shown in Fig. 1, BEAS consists of three major components: (1) offline service $\text{AS Catalog}$ to manage access schema for different applications; and (2) online service $\text{BE Query Planner}$ and $\text{BE Plan Executor}$ to process SQL queries. It can be built on top of any commercial DBMS. $\text{AS Catalog}$ includes three modules itself.

1. **Metadata** maintains (a) access schema, and (b) statistics including the index size in a system table $\text{asCatalog}$, for query plan generation and optimization.

![Figure 1: Architecture of BEAS](image-url)
(2) **Discovery module.** Given an application, it automatically discovers an access schema from its real-life datasets. It is a multi-criteria optimization problem that covers (a) the performance of bounded evaluation of the query load, (b) storage limit for indices, (c) historical query patterns, and (d) statistics of datasets in the application. We defer the details of the discovery algorithm to a later publication.

For each access constraint \( \psi = R(X \rightarrow Y, N) \) discovered, its index on a relation \( D \) of \( R \) is a modified hash index such that (a) it takes attributes \( X \) as the key; and (b) each key value \( \bar{a} \) points to a bucket \( D_Y(X = \bar{a}) \) (see Section 2), the set of at most \( N \) distinct \( Y \)-values in \( D \) corresponding to \( \bar{a} \).

(3) **Maintenance module** maintains access schema \( A \) in response to changes to query loads and datasets in each application. It (a) periodically adjusts constraints in \( A \) based on the changes to the historical queries, to optimize the performance of bounded evaluation; and (b) incrementally updates the indices of \( A \) in response to changes to the datasets, by employing an optimal incremental algorithms reported in [5].

**BE Query Planner.** It also has three modules.

(1) **BE Checker** checks whether an input SQL query \( Q \) is bounded evaluatable under the access schema discovered. A checking algorithm has been reported in [5] for RA, based on the effective syntax of the Feasibility Theorem. **BEAS** extends the algorithm to (possibly aggregate) SQL queries.

(2) **BE Plan Generator** generates (a) a bounded query plan for \( Q \) if \( Q \) is found bounded evaluated by **BE Checker**, by extending the bounded-plan generation algorithm reported in [5] from RA to SQL; and (b) if \( Q \) is not bounded, it picks a conventional query plan for \( Q \) generated by the underlying **DBMS**, and applies **BE Plan Optimizer** to it (see below).

As shown in Example 2, a bounded plan consists of relational algebra operators [10] (i.e., select, project, join, union and set difference), aggregates, group-by, and a new operator `fetch(X \in T, Y, R)` with access constraint \( R(X \rightarrow Y, N) \), which fetches all \( Y \)-values corresponding to the \( X \)-values in intermediate results \( T \). It accesses data only via `fetch` operations, and answers \( Q \) by using a bounded amount of data.

(3) **BE Plan Optimizer** improves the conventional plan of the **DBMS** for \( Q \) when \( Q \) is not bounded, to support partially bounded evaluation. It identifies sub-queries of \( Q \) that are boundedly evaluable under access schema \( A \), and speeds up the evaluation of \( Q \) by capitalizing on the indices of \( A \).

Alternatively, if users can afford only bounded resources and hence opt to take approximate query answers, **BEAS** offers resource bounded approximation. We defer the details of the approximation scheme to a later publication.

**BE Plan Executor.** It executes bounded query plans by extending the physical plan implementation of **DBMS** [10] to support the `fetch` operator. For each fetch \( (X \in T, Y, R) \) with access constraint \( R(X \rightarrow Y, N) \) in a bounded plan, where \( T \) is an intermediate relation, it fetches all associated \( Y \)-values for each \( X \)-value \( \bar{a} \) in \( T \) by using the modified hash index for \( \psi \) with key \( \bar{a} \), and returns their union (see Section 2).

Observe the following. (1) The design of **BE Query Planner** and **BE Plan Executor** allows us to implement **BEAS** on top of any **DBMS**. It is also easy to add other modules to **DBMS**, e.g., resource-bounded approximation. (2) There have been recent efforts to query big relations with limited resources, e.g., BlinkDB [2] and PiQL [3]. These systems, however, focus on approximate query answering, by sampling [2] or by restricting the fetched data with a user specified bound [3] in the flavor of anytime algorithms [11]. In contrast, **BEAS** introduces access schema and aims to provide exact query answers with bounded resources as much as possible.

### 4. Demonstration Overview

We next present our demonstration plan, including settings and scenarios, as well as an industry application. The target audience of the demo includes anyone who is interested in query answering with bounded resources.
We have implemented BEAS on top of PostgreSQL 9.4.6. We have also created a demo portal as shown in Fig. 2, via a telecommunication company (name withheld), and compare its performance with commercial DBMS using a telecommunication application as a testbed.

(1) A walk through. We visualize and demonstrate each major component of bounded evaluation underlying BEAS.

(a) Bounded evaluability checking. As shown in Fig. 2(A), the audience will be invited to enter an SQL query $Q$, select a dataset, pick an access schema $A$ discovered, and check whether $Q$ is boundedly evaluable under $A$ by using BE Checker. Users can also enter a budget on the amount of data to be accessed, and use BE Checker to find whether $Q$ can be answered within the budget under $A$, without executing $Q$.

(b) Bounded planning and optimization. As shown in Fig. 2(B), when $Q$ is boundedly evaluable under $A$, the users will see a bounded query plan suggested by BE Plan Generator, in which each fetch operation is annotated with an upper bound on the amount of data to be fetched. The upper bound is deduced by reasoning about $A$.

If $Q$ is not bounded, BEAS picks a query plan $\xi$ generated by PostgreSQL BE Plan Optimizer then makes $\xi$ partially bounded by identifying bounded sub-queries of $Q$ under $A$.

(c) Analysis. After a query plan is carried out by BE Plan Executor, a performance analysis is provided (Fig. 2(C)).

(d) Access schema management. As offline services, (i) the discovery module of BEAS takes as input a dataset $D$, a set $Q$ of query patterns, and a choice of the objective function; it discovers an access schema $A$ and registers it by AS_Catalog (Fig. 2(D)); For instance, Figure 2(E) shows part of an access schema discovered. (ii) The maintenance module automatically updates $A$ in response to changes to $D$ and queries. It also allows users to add or remove access constraints.

(2) Performance. We demonstrate how BEAS works in practice using a commercial benchmark from a telecommunication company (name withheld), and compare its performance with PostgreSQL, MySQL and MariaDB.

Efficiency. The users are invited to interact with BEAS, pick built-in queries or enter their own queries, and examine the effectiveness of bounded evaluation. For instance, for query $Q$ of Example 2 on a TLC dataset $D$ of 20GB, a snapshot of the BEAS performance analyzer is given in Fig. 3, which shows that BEAS is 1953, 6562 and 5135 times faster than PostgreSQL, MySQL and MariaDB, respectively. It details (a) the overall execution time, acceleration ratio compared to commercial DBMS, the total number of tuples fetched and the number of access constraints employed; and (b) a breakdown of the cost to each individual operation in the query plan, compared to its counterpart in plans generated by commercial DBMS. It illustrates why BEAS works better.

Scalability. The audience will also witness the scalability of BEAS by scaling up the datasets. Figure 4 shows the evaluation time of $Q$ of Example 2 with BEAS, PostgreSQL, MySQL and MariaDB when varying TLC from 1GB to 200GB. One can see that BEAS consistently takes about 1s when $D$ varies, and is hence "scale-independent". In contrast, PostgreSQL, MySQL and MariaDB grow to 1932s, 6187s and 5243s, respectively, if we allow them to run to completion.

5. References