S4: Top-k Spreadsheet-Style Search for Query Discovery

Fotis Psallidas[1] * Bolin Ding[2] Kaushik Chakrabarti[2] Surajit Chaudhuri[2]
[1]Columbia University, New York, NY
[2]Microsoft Research, Redmond, WA
fotis@cs.columbia.edu, {bolind, kaushik, surajitc}@microsoft.com

ABSTRACT

An enterprise information worker is often aware of a few example tuples that should be present in the output of the query. Query discovery systems have been developed to discover project-join queries that contain the given example tuples in their output. However, they require the output to exactly contain all the example tuples and do not perform any ranking. To address this limitation, we study the problem of efficiently discovering top-k project join queries which approximately contain the given example tuples in their output. We extend our algorithms to incrementally produce results as soon as the user finishes typing/modifying a cell. Our experiments on real-life and synthetic datasets show that our proposed solution is significantly more efficient compared with applying state-of-the-art algorithms.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, textual databases*

Keywords

SQL query discovery; example spreadsheet; relevance ranking

1. INTRODUCTION

Modern data warehouses usually have large and complex data schemas. A decision-support query on such a data warehouse typically touches a small portion of the schema. However, to express such a query, the enterprise information worker needs to comprehend the entire schema and locate the schema elements of interest. This is extremely burdensome for most users.

Query discovery has recently been proposed as a solution to this problem [18, 22]. An enterprise information worker is often aware of *a few example tuples* that should be present in the output of a query. These example tuples together form an *example spreadsheet*, one per each row. Previous systems discover *project-join queries* (*PJ queries*) that contain the given example tuples, or the example spreadsheet, in their output [18, 22]. This liberates users from understanding the entire schema. We illustrate this below.

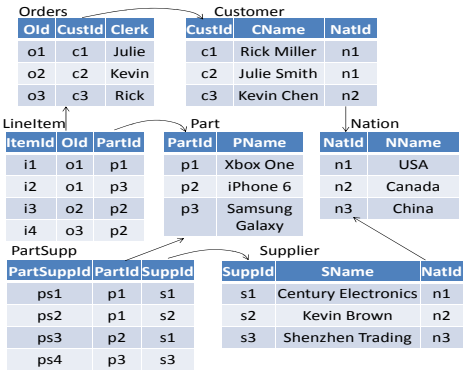


Figure 1: A sample database

EXAMPLE 1. *Consider a database instance of a TPC-H subschema in Figure 1. The database contains information about customers, the countries they live in, the orders they placed, the parts purchased in each order, the suppliers of those parts, and the countries the suppliers are based in. The arrows point in the direction of foreign-key to primary-key relationships between pairs of relations.*

*Suppose the user wants to discover the PJ query that outputs all customers and, for each customer, outputs her name, the name of the country she lives in, and the names of the parts she ordered. The PJ query and its output is shown in Figure 2(b)-(i). She is aware of an example spreadsheet of three example tuples that should be present in the query result: a customer named 'Rick' (does not know his full name) who lives in 'USA' and ordered an 'Xbox', a customer named 'Julie' (not sure where she lives) who ordered an 'iPhone' and a customer named 'Kevin' who lives in 'Canada' (not sure what he ordered). She can provide this information by typing these example tuples into an example spreadsheet (in forms of, e.g., Microsoft Excel and Google Sheets) as shown in Figure 2(a). Note that some cells in the example spreadsheet can be empty. The system returns the desired PJ query in Figure 2(b)-(i) as it contains all the example tuples in its output relation. The example tuples and corresponding tuples in the output are shaded with the same color. The system maps the columns of the example spreadsheet to the projected columns in the query for users to better understand the PJ query discovered; the latter are labeled by the name of the corresponding column in the example spreadsheet (*A, B *and* C*).*

One main limitation of these previous systems is that they require the output relations of PJ queries to *exactly* contain all the example tuples and do not perform any ranking. As a result, they cannot i) tolerate errors that the user might make while providing the example tuples and ii) perform IR-style relevance ranking.

*Work done while visiting Microsoft Research

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749452.

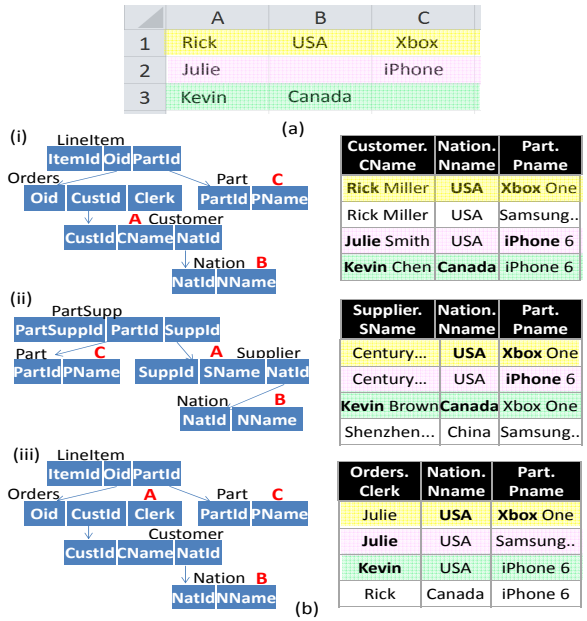


Figure 2: (a) Example spreadsheet (b) PJ queries and their outputs

• *Tolerating errors*: Suppose the user wants to discover the query that outputs all orders and, for each order, outputs the name of clerk who processed the order, the country of the customer who placed the order, and the parts in the order. She provides an example tuple: a clerk named 'Rick' processed an order from a customer in 'USA', and the order consisted of the part 'Xbox'. However, it is not 'Rick' but another clerk 'Julie' who processed that order. The desired PJ query and its output is shown in Figure 2(b)-(iii). We say that she has made a *relationship error* with respect to that PJ query as, although 'Rick' is a correct domain value (he is indeed a clerk), the provided relationship of 'Rick' with 'USA' and 'Xbox' is wrong.

The user can also make *domain errors*. Suppose the user wants to discover the query that outputs all suppliers and, for each supplier, outputs its name, the country it is based in and the parts it supplies. She then provides such an example tuple: a supplier named 'Rick' based in 'USA' who supplies the part 'Xbox'. The desired PJ query and its output is shown in Figure 2(b)-(ii). There is a domain error with respect to the PJ query as there is no supplier named 'Rick'.

• *Performing relevance ranking*: Suppose there is a supplier with name 'Welton USA' who supplies Xbox One. Consider the first example tuple in Figure 2(a). In addition to the the PJ query shown in Figure 2(b)-(i), some other PJ queries may also contain that example tuple in their outputs (e.g., a customer 'Rick Miller' who ordered 'Xbox One' which is supplied by 'Welton USA'). The former is more relevant, as country name 'USA' is a better match to 'USA' in the example tuple than supplier name 'Welton USA'.

To address the above issues, in this paper, we propose to *discover not only the PJ queries which exactly contain the given example tuples, or the example spreadsheet, in its output but also those that partially contain them*. We compute a relevance score for each PJ query that quantifies how well its output contain the example tuples and return *PJ queries with the top-k highest scores*.

Technical challenges. In a large real-world database, there are numerous ways of projecting and connecting tables and rows through foreign keys. So there could be millions of PJ queries that partially contain the user-specified example tuples in their output relations.

The first technical challenge is to develop a scoring model that allows us to tolerate relationship/domain errors and quantifies how well the user-given example spreadsheet is contained in the output of PJ queries, in order to perform a relevance ranking of them.

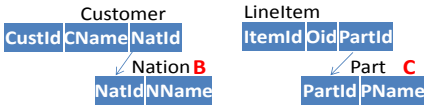


Figure 3: Common sub-expressions (sub-PJ queries) in Figure 2(b)

The second and the main technical challenge is to compute the top-k PJ queries *efficiently*. One important application of our system is to provide *online* data-search and discovery services in data processing tools, e.g., in Excel Online [3] and Google Sheets [1]. While a user may spend a significant amount of time in specifying the query, i.e., the example spreadsheet in our system, it has been shown that, in the context of online search, query latency is critical to user satisfaction. Increases in latency directly lead to lower utilization and higher rates of query abandonment [7, 16].

Overview of our solution and key insights. In this paper, we adapt a *candidate-enumeration and evaluation* framework, which is also used in keyword search systems for relational databases [5, 12, 15]. In the first step, called *PJ query enumeration*, we enumerate all candidate PJ queries that are potential answers for a user-specified *example spreadsheet*. The only requirement for these candidate PJ queries, called *minimality*, is that no table or projection column can be dropped without "losing" in relevance score. In the second step, called *PJ query evaluation*, we execute candidate PJ queries, and compare their output relations with the example spreadsheet to calculate their scores. A naive solution is to execute all the candidates and output the top-k with the highest scores.

It is important to note that the first step is very efficient, as it is pursued on the schema-level and no join is required. So it constitutes a negligible fraction of the overall query processing time. The second step, evaluating scores of PJ queries, is expensive (as it requires joins). As our scoring model quantifies how well the user-given example spreadsheet is contained in the output of join and projection, we need to at least examine rows in the join output, which may partially contain an example tuple. So this challenge translates to that of *evaluating as few PJ queries as possible*. The naive solution, evaluating all the candidates, is hence infeasible (we will compare it with the approaches we propose in Section 6).

Although calculating the exact relevance scores is expensive, we derive their upper bounds in a much more efficient way (without executing any join). Inspired by the work on multi-step kNN search [20], we evaluate PJ queries in decreasing order of their upper bound scores, and terminate with the top-k as soon as the max upper-bound score of un-evaluated queries is no higher than the current top-k score. This approach is referred to as BASELINE.

Our main insight to improve BASELINE is that there are many common sub-expressions, called *sub-PJ queries*, that are *shared* among the PJ queries. For example, the PJ queries (i) and (iii) in 2(b) share the two sub-PJ queries shown in Figure 3. If we can compute the output relations of these sub-PJ queries once, and cache (or memorize) them in memory, we can re-use them multiple times later when we evaluate queries containing these two sub-PJ queries. This reduces the overall evaluation cost significantly.

A novel component, called *caching-evaluation scheduler*, in our system determines, for the set of candidate PJ queries, i) the order following which these PJ queries will be evaluated; ii) output relations of which sub-PJ queries to be cached; and iii) when to put the output relations into the cache and when to remove them, as we have only a budgeted amount of memory. There are two aspects in the objective of this component: one is to evaluate *as few PJ queries as possible* to discover the top-k; and the other one is to utilize the cached output relations as much as possible to reduce the overall *evaluation cost*. We will formalize the task of this component and refer to it as *caching-evaluation scheduling problem*.

Contributions and organization. We have built a *spreadsheet-style search system* (called S4) to tackle the challenges based on the above insights. Our contributions are summarized as follows:

• We introduce a novel scoring model for a PJ query w.r.t. an example spreadsheet. It allows us to tolerate both types of errors and perform IR-style relevance ranking of PJ queries (Section 2).

• We introduce our system based on a candidate-enumeration and evaluation framework, and we enable a flexible caching-evaluation component in the system architecture (Section 3).

• To tackle the technical challenges of our task, we first introduce some basic operators in our system and propose BASELINE strategy. It aims to evaluate as few PJ queries as possible (Sections 4).

• We then introduce our cache-aware optimization techniques to improve BASELINE. We propose the *caching-evaluation scheduling problem* with the objective of minimizing the overall evaluation cost. We prove it is NP-complete. Several novel heuristics are proposed to solve this problem, and the resulting strategy is called FASTTOPK. We can prove that FASTTOPK has performance guarantee in the worst-case in two aspects: i) it does not evaluate too many PJ queries in addition to the necessary ones; and ii) the gap between the evaluation cost introduced by FASTTOPK strategy and the optimal evaluation cost is bounded in the worst case. We also introduce how to extend our system and strategies to handle incremental updates on the example spreadsheet (Section 5).

• We perform experimental study on real-life and synthetic datasets to evaluate the efficiency of our approaches, together with a user study to evaluate the effectiveness of our scoring model (Section 6).

2. SYSTEM TASK AND SCORING MODEL

We first present our data model, and formally define our system task of *discovering top-k project-join queries for a given example spreadsheet*. Then, we present the model to compute the relevance score of a project-join query w.r.t. an example spreadsheet.

2.1 Data Model

We consider a database $\mathcal{D}$ with m relations $R_1, R_2, \cdots, R_m$. For a relation R, let $R[i]$ denote its i^{th} column, and $\text{col}(R) = \{R[i]\}_{i=1,\ldots,|\text{col}(R)|}$ denote the set of columns of R. For a tuple r in R, denote $r \in R$ and let $r[i]$ be its *cell* value on the column $R[i]$.

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ denote the *directed schema graph* of $\mathcal{D}$ where the vertices in $\mathcal{V}$ represent the relations in $\mathcal{D}$, and the edges in $\mathcal{E}$ represent foreign key references between two relations: there is an edge from R_j to R_k in $\mathcal{E}$ iff the primary key defined on R_k is referenced by a foreign key defined in R_j. There can be multiple edges from R_j to R_k and we label each edge with the corresponding foreign key's attribute name. For simplicity, we omit edge labels in our examples and description if they are clear from the context.

In a relation R_i, we refer to a column as *text column* if its values are strings. Figure 1 shows an example database with a total of seven relations. There five text columns: Customer.CustName, Nation.NatName, Orders.Clerk, Part.PartName, and Supplier.SuppName. In the rest of this paper, we focus on only text columns and primary or foreign key columns of the relations.

2.2 Discovering Top-k PJ Queries by Example Spreadsheet

Example spreadsheet. An example spreadsheet is a multi-column table and serves as an interactive interface for PJ query discovery. Each cell of this spreadsheet is typed by the user, and could either be empty or contain some text. Figure 2(a) gives an example.

DEFINITION 1. (Example spreadsheet) *An example spreadsheet T is a table with multiple rows $\{t\}$ and columns $\text{col}(T)$. Each row $t \in T$ is called an example tuple, where each cell is either a string (i.e., one or more terms) or empty. Let $t[i]$ denote its cell value on the column $i \in \text{col}(S)$, and let $t[i] = \emptyset$ if $t[i]$ is empty. Each row t contains at least one term and so does each column $T[i]$.*

Project-Join (PJ) queries. We aim to discover queries in directed-tree shapes with projections and foreign key joins that generate a table from $\mathcal{D}$ to expand the user-given example spreadsheet T.

DEFINITION 2. (Project-Join Queries) *A* project-join query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ *for T is specified by:*

- *a* join tree $\mathcal{J} \subseteq \mathcal{G}$*, i.e., a directed subtree of the schema graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of the database $\mathcal{D}$ representing all the relations (vertices of $\mathcal{J}$) and joins (edges of $\mathcal{J}$) involved in the query – let $\text{col}(\mathcal{J})$ be the set of all columns of relations in $\mathcal{J}$,*
- *a set of* projection columns $\mathcal{C} \subseteq \text{col}(\mathcal{J})$ *from the relations in $\mathcal{J}$, which the join result is projected onto, and*
- *a* column mapping $\phi : \text{col}(T) \rightarrow \mathcal{C}$ *from columns of the example spreadsheet T to the projection columns in $\mathcal{C}$ – it is a surjective function, i.e., $\forall c \in \mathcal{C} : \exists i \in \text{col}(T)$ s.t. $\phi(i) = c$.*

It is important to ensure that there is no redundant table or projection column in the discovered PJ queries. Intuitively, a table or a projection column is redundant if, after it is dropped from the PJ query, the output relation matches the example spreadsheet equally well or even better. We formally define them as *minimal PJ queries* and only consider them as the candidates to be discovered.

DEFINITION 3. (Minimal Project-Join Queries) *A PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ w.r.t. an example spreadsheet T is* minimal *iff*

i) for any degree-1 vertex (relation) R in $\mathcal{J}$, there is a column $i \in \text{col}(T)$ s.t. $\phi(i) \in \text{col}(R)$, i.e., every degree-1 relation R has a column of the example spreadsheet mapped to it; and

ii) for every column i of T which is mapped to column $R[j]$ of a relation R in $\mathcal{J}$ through ϕ (i.e., $\phi(i) = R[j]$), there exists at least one term in column $T[i]$ appearing in column $R[j]$.

In the rest of this paper, when we refer to *PJ queries*, we refer to minimal project-join queries. For the example database in Figure 1, three PJ queries and their output relations are shown in Figure 2(b).

Let $\mathcal{A}(Q)$ be the output relation when Q is executed on database $\mathcal{D}$: joins in $\mathcal{J}$ are executed first, and then the results are projected on columns $\mathcal{C}$. Columns of the example spreadsheet T are mapped to columns $\mathcal{C}$ of the output relation $\mathcal{A}(Q)$ according to ϕ.

Property i) in Definition 3 is similar to the *minimality* of *candidate networks* in keyword search literatures like [5, 12, 15]. In our case, degree-1 relations not satisfying i) can be excluded from $\mathcal{J}$ s.t. we have no less distinct tuples in the output relation $\mathcal{A}(Q)$, because they have no column in the projection and the join tree is still valid after the removal of them.

Property ii) says that a column i in the example spreadsheet T should not be mapped to a column $R[j]$ in the projection $\mathcal{C}$ if none of the terms in the column $T[i]$ appears in $R[j]$ (and the corresponding column in $\mathcal{A}(Q)$). Intuitively, if the two columns $T[i]$ and $R[j]$ have no overlap in their vocabularies, they are likely from two different domains so it is meaningless to map $T[i]$ to $R[j]$. In fact, we can drop the column i from T to get a smaller example spreadsheet T', and drop the column $R[j]$ from the projection $\mathcal{C}$ and the mapping ϕ, denoting as $\mathcal{C}' = \mathcal{C} - \{R[j]\}$ and ϕ'; in our scoring model, we can prove that the relevance score of $Q' = (\mathcal{J}, \mathcal{C}', \phi')$ w.r.t. T' is no less than the score of $Q = (\mathcal{J}, \mathcal{C}, \phi)$ w.r.t. T.

Based on our scoring model introduced next in Section 2.3, we will show that we do not "lose" in score by looking only at the minimal PJ queries (Proposition 1). A bit more formally, for any

non-minimal PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$, we can find a minimal PJ query $Q' = (\mathcal{J}', \mathcal{C}', \phi')$ with $\mathcal{J}'$ as a subtree of $\mathcal{J}$ and/or ϕ' as a sub-mapping of ϕ such that the score of Q' is no less than the score of Q. For example, consider the example spreadsheet in Figure 2(a) *without* column C, Figure 2(b)-(i) is no longer a minimal PJ query, as the degree-1 relation Part violates property i) and removing it will not reduce the score. Another example is, in Figure 2, it does not make sense to map column A in the example spreadsheet to column Nation.NName – any PJ query with this mapping violates property ii), and we can remove this pair of columns from the example spreadsheet/PJ query without reducing the score.

End-to-end system task. For a user-given example spreadsheet T, the goal of our system is *to find minimal PJ queries with the top-k highest scores w.r.t. T*. The incremental version of our task is: suppose we have found the top-k PJ queries for a user-given example spreadsheet T, after one or more cells in T are updated by the user, how to find the updated top-k PJ queries efficiently.

2.3 Scoring Model for PJ Queries

The score of a PJ query Q w.r.t. an example spreadsheet T quantifies how well the Q's output $\mathcal{A}(Q)$ contains rows in T. We first present the scoring model and then show how it allows us to tolerate relationship and domain errors for performing relevance ranking.

An IR system computes a score of a document w.r.t. a keyword query, which quantifies how well the former contains the terms in the latter. A straightforward way to compute the score of Q w.r.t. T is to treat T as a "*query*" (by concatenating all text in T) and $\mathcal{A}(Q)$ as a "*document*" (again, by concatenating) and apply a traditional IR relevance scoring model [23]. We do not adopt this model as we need to quantify how well $\mathcal{A}(Q)$ contains each example tuple with their columns aligned according to the mapping ϕ; it is difficult to do so in this model as it removes the row/column boundaries.

Containment score w.r.t. single example tuple. We first define a score $\mathsf{score}(t \mid \mathcal{A}(Q))$ to quantify how well $\mathcal{A}(Q)$ contains *a single example tuple* $t \in T$. Let $\mathsf{score}(t \mid r)$ denote the similarity between an example tuple $t \in T$ and a row $r \in \mathcal{A}(Q)$ in the PJ query output (referred to as *row-row similarity*). By definition of containment, $\mathsf{score}(t \mid \mathcal{A}(Q))$ should be high as long as there is one row $r \in \mathcal{A}(Q)$ in the PJ query's output relation with a high row-row similarity $\mathsf{score}(t \mid r)$ with t; so we refer to the *most similar tuple* for t to define the containment score:

$$\mathsf{score}(t \mid Q) = \max_{r \in \mathcal{A}(Q)} \mathsf{score}(t \mid r). \tag{1}$$

Row-row similarity. One way to get row-row similarity $\mathsf{score}(t \mid r)$ between an example tuple $t \in T$ and a row $r \in \mathcal{A}(Q)$ in the PJ query output is to treat t as a "query" (by concatenating the terms in all cells in t) and r as a "document" (again, by concatenating). Again, this model is not suitable as we need to respect the mapping ϕ while computing the row-row similarity. We need to compare a cell $t[i]$ with the cell $r[\phi(i)]$ it is mapped to. Let $\mathsf{score}_{\text{cell}}(t[i] \mid r[j])$ denote the cell similarity between an example tuple cell $t[i]$ and a cell $r[j]$ in an output row. We compute the row-row similarity by summing up the cell similarities for all columns.

$$\mathsf{score}(t \mid r) = \sum_{i \in \text{col}(T)} \mathsf{score}_{\text{cell}}(t[i] \mid r[\phi(i)]). \tag{2}$$

We use a simple cell similarity $\mathsf{score}_{\text{cell}}(t[i] \mid r[j])$ as: how many terms in $t[i]$ appear in $r[j]$ if $t[i]$ is non-empty and 0 otherwise. We discuss how to adapt a more complicated IR-style cell similarity to perform relevance ranking in Appendix A.2.

Row containment score w.r.t. entire example spreadsheet. We are now ready to define the *row-wise containment score* to quantify how well $\mathcal{A}(Q)$ contains *all* the example tuples in T, denoted as $\mathsf{score}_{\text{row}}(T \mid Q)$. The more individual tuples in the example spreadsheet $\mathcal{A}(Q)$ contains, the higher should be the final score. So, a natural way is to sum up containment scores for all the example tuples in the example spreadsheet:

$$\mathsf{score}_{\text{row}}(T \mid Q) = \sum_{t \in T} \mathsf{score}(t \mid Q) = \sum_{t \in T} \max_{r \in \mathcal{A}(Q)} \mathsf{score}(t \mid r). \tag{3}$$

EXAMPLE 2. *We compute the score $\mathsf{score}_{\text{row}}(T \mid Q)$ of PJ query Q in Figure 2(b)-(iii) w.r.t. the example spreadsheet T in Figure 2(a). Recall that cell similarity $\mathsf{score}_{cell}(t[i] \mid r[j])$ is how many terms in $t[i]$ appear in $r[j]$ if non-empty, and 0 otherwise. For each row in T, the most similar row in $\mathcal{A}(Q)$ is shaded with the same color (yellow, pink and green for the three rows). The single tuple containment scores are 2, 1, and 1 respectively. So, $\mathsf{score}_{\text{row}}(T \mid Q) = 4$. Similarly, the score between the same example spreadsheet and PJ query in Figure 2(b)-(ii) is $2 + 1 + 2 = 5$.*

Tolerating errors. Naturally, the scoring function should have the following property: *higher the number of errors in the example spreadsheet with respect to the output of a PJ query Q, lower the score of Q*. The above score $\mathsf{score}_{\text{row}}$ satisfies this property. For example, the example spreadsheet in Figure 2(a) has 2 errors in the output of PJ query in Figure 2(b)-(ii) (Rick and Julie in column A do not appear in column Supplier.SName for the first two example tuples), while it has 3 errors in the output of PJ query (iii) (one term missed for each example tuple). From Example 2, we see that (ii) has a higher score $\mathsf{score}_{\text{row}}$ than (iii). However, it *penalizes relationship and domain errors equally*. Relationship errors are more common, so we want to penalize them less than domain errors. We next introduce column containment score for that purpose.

Column containment score. We define column containment score that penalizes *only domain errors*. Subsequently, we will put it together with the row containment score $\mathsf{score}_{\text{row}}$ to penalize the two classes of errors differently. A cell in column $i \in \text{col}(T)$ in an example spreadsheet T has a domain error in a PJ query Q iff it has one term not occurring in the mapped column $\phi(i)$ of the join tree $\mathcal{J}$ of Q. The column-wise containment score that quantifies how well the cells in each column $i \in \text{col}(T)$ are contained in the mapped column $\phi(i)$ will penalize only domain errors.

For each cell in the example spreadsheet T, we first find the *most similar cell* in the mapped column $\phi(i)$ of the join tree $\mathcal{J}$ of Q. We sum up the similarities between cells paired in this way to obtain the *column-wise containment score*:

$$\mathsf{score}_{\text{col}}(T \mid Q) = \sum_{i \in \text{col}(T)} \sum_{t \in T} \max_{r \in \mathcal{J}[\phi(i)]} \mathsf{score}_{\text{cell}}(t[i] \mid r[\phi(i)]), \tag{4}$$

where let $\mathcal{J}[\phi(i)]$ be the relation $T[i]$ is mapped to in database $\mathcal{D}$.

EXAMPLE 3. *We compute the column-wise containment score $\mathsf{score}_{\text{col}}$ of the PJ query Q in Figure 2(b)-(ii) w.r.t. the example spreadsheet T in Figure 2(a). Column A in T is mapped to column Supplier.SuppName (in the database in Figure 1) through ϕ. Only one (Rick) out of the three terms in T.A appears in Supplier.SuppName. For each of the other two columns in T, both terms can be found in the corresponding column in the database. So $\mathsf{score}_{\text{col}}(T \mid Q) = 5$. Similarly, the column-wise containment score of the PJ query Q in Figure 2(b)-(iii) is $3+2+2 = 7$. In contrast to Example 2, now (iii) has a higher score than (ii) because (iii) has no domain errors while (ii) has 2 domain errors.*

Putting it together. We obtain the final *relevance score* $\mathsf{score}(T \mid Q)$ of a PJ query Q w.r.t. an example spreadsheet T by taking a lin-

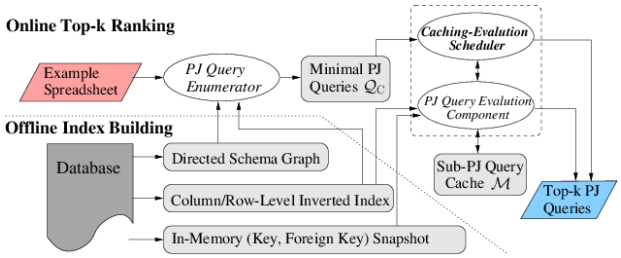


Figure 4: S4 System Architecture

ear combination of row-wise and column-wise containment scores. We introduce a parameter $0 \leq \alpha \leq 1$ to control the relative penalty of the two classes of errors. Similar to prior work, we also penalize PJ queries with larger join trees as the relationship among the mapped columns in $\mathcal{J}$ is looser. We penalize it with a factor of $1 + \ln(1 + \ln|\mathcal{J}|)$, where $|\mathcal{J}|$ is the number of relations in $\mathcal{J}$.

$$\mathsf{score}(T \mid Q) = \frac{\alpha \cdot \mathsf{score}_{\text{row}}(T \mid Q) + (1-\alpha) \cdot \mathsf{score}_{\text{col}}(T \mid Q)}{1 + \ln(1 + \ln|\mathcal{J}|)}. \tag{5}$$

Minimality and scores. In Proposition 1 below, we show why we are interested in finding only minimal PJ queries (Definition 3).

PROPOSITION 1. *Consider a user-given example spreadsheet T and a PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ in a database $\mathcal{D}$. If property i) in Definition 3 is not satisfied, let R be a degree-1 relation in $\mathcal{J}$ with no column in T mapped to it. Define a smaller $\mathcal{J}' = \mathcal{J} - R$ and $Q' = (\mathcal{J}', \mathcal{C}, \phi)$. We have $\mathsf{score}(T \mid Q) \leq \mathsf{score}(T \mid Q')$.*

If property ii) is violated, let i be a column in T s.t. no term in $T[i]$ appears in the database column $R[j]$ it is mapped to. We can remove the column i from T to get a smaller example spreadsheet T', and remove the column $R[j]$ from the projection $\mathcal{C}$ and the mapping ϕ: let $\mathcal{C}' = \mathcal{C} - \{R[j]\}$ and ϕ' be the range-restriction of ϕ by $\mathcal{C}'$, i.e., $\phi'(i) = \phi(i)$ iff $\phi(i) \in \mathcal{C}'$ and undefined otherwise. Let $Q'' = (\mathcal{J}, \mathcal{C}', \phi')$. We have $\mathsf{score}(T \mid Q) = \mathsf{score}(T' \mid Q'')$.

3. SYSTEM ARCHITECTURE

The S4 system architecture is depicted in Figure 4, with two major components: *offline index building* and *online top-k ranking*.

3.1 Offline Index Building

First, we have the *directed schema graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in memory, which keeps schema-level information about the database $\mathcal{D}$, including names of relations/columns (in $\mathcal{V}$), and foreign keys (in $\mathcal{E}$).

Secondly, for the purpose of PJ query enumeration and evaluation (computing their scores), we build two types of in-memory indexes that are extensions to *inverted indexes* in traditional IR.

• *Column-level inverted index.* Given a term w, index $\mathsf{inv}(\mathsf{w})$ returns all the database columns where w appears in at least one row.

• *Row-level inverted index.* For a term w and a column $R[i]$ in relation R, $\mathsf{inv}(\mathsf{w}, R[i])$ returns all rows in R, where w appears in column i, and its term frequency in each cell.

Finally, the PJ queries need to be executed and scored without accessing the database in disk. Attivio Active Intelligence Engine uses a similar index to perform "query-time join" [24]. We call it:

• *In memory (key, foreign key) snapshot of the database.* We have the primary-key column and foreign-key columns of each row in each relation of the database materialized in memory.

3.2 Online Top-k Ranking

We adopt a *PJ-query enumeration-ranking* framework for online top-k ranking. Our system processes a user-specified example spreadsheet in two steps as follows to output the top-k PJ queries.

PJ query enumeration. The set of minimal PJ queries, denoted as $\mathcal{Q}_C$, for the example spreadsheet could all be candidate answers for the top-k. The component, *PJ Query Enumerator*, generates this set of PJ queries. We adapt the *CN generation* algorithm described in [13] for this component. Since the generation of $\mathcal{Q}_C$ (without computing their scores) can be done by accessing only the schema graph and column-level inverted index, it is quite efficient. Moreover, we compute an *upper bound* of its score for each PJ query in $\mathcal{Q}_C$. This upper bound can be also computed efficiently without executing any join. More details about the generation of $\mathcal{Q}_C$ and upper-bound score computation will be given in Section 4.1.

PJ query ranking. Taking PJ queries in $\mathcal{Q}_C$ as the input, the main contribution of this paper is about how to identify those with the top-k highest scores from $\mathcal{Q}_C$. To this end, *PJ Query Evaluation Component* evaluates *some queries* in $\mathcal{Q}_C$ to get their scores. By *evaluating a PJ query*, we mean executing the query to compute its score w.r.t. the example spreadsheet. Because our scoring model quantifies how well the example spreadsheet is contained in the output of join and projection of a PJ query, to get the row containment score, we need to at least examine rows which either completely or partially contain the example tuples in the output. This process requires to execute joins and thus is the bottleneck in online top-k ranking. Details about the evaluation of PJ queries will be given in Section 4.1 (basic version) and Section 5.1 (cache-aware version).

Caching-Evaluation Scheduler finds a strategy that specifies: i) which queries to be evaluated to get the top-k, ii) the order of evaluations, and iii) how to use the in-memory *Sub-PJ Query Cache* to speedup the evaluations. We focus on this scheduler in the rest part. We present a baseline strategy in Section 4 without utilizing the cache and a more efficient cache-aware strategy in Section 5.

4. BASELINE EVALUATION STRATEGY

In Section 4.1, we first introduce the basic operators in our evaluation strategy: how to enumerate all candidate PJ queries $\mathcal{Q}_C$ for an example spreadsheet and compute their upper-bound scores and exact scores. We also analyze their costs. For interactive speed, it is affordable to enumerate $\mathcal{Q}_C$ first and compute the upper-bound scores but it would be too expensive to compute the exact scores for all queries in $\mathcal{Q}_C$. To design more efficient approaches, in Section 4.2, we study what is the minimal set of queries we have to evaluate to get the top-k given those upper bounds. We end this section with a worst-case optimal baseline strategy in Section 4.3.

4.1 Basic Operators in Evaluation Strategy

Before introducing our evaluation strategies, we give more details about the basic operators in our system and analyze their costs.

4.1.1 Enumerating Minimal PJ Queries $\mathcal{Q}_C$

For a user-specified example spreadsheet T, we utilize the directed schema graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and column-level inverted index to generate $\mathcal{Q}_C$. For each column $T[i]$ in T, we first find all the columns $\mathcal{C}_i$ in $\mathcal{D}$ which contain at least one term in $T[i]$, called *candidate projection columns*. $\mathcal{C}_i$ is essentially the union of $\mathsf{inv}(\mathsf{w})$'s for all terms w's in $T[i]$, i.e., $\mathcal{C}_i = \bigcup_{\mathsf{w} \in T[i]} \mathsf{inv}(\mathsf{w})$.

Consider the example table in Figure 2. The candidate projection columns for column A are: Customer.CustName, Orders.Clerk (containing all the 3 terms in A) and Supplier.SuppName (containing only the first term, Rick). For column B, there is only one, Nation.NatName (containing both terms in B). And for column C, Part.PartName (contains both terms) is the only one.

Given candidate projection columns $\mathcal{C}_i$'s generated in the above process, to enumerate $\mathcal{Q}_C$, we can pick one column from each $\mathcal{C}_i$ to form $\mathcal{C}$ and mapping ϕ, and generate directed Steiner trees to get

$\mathcal{J}$ that connect to relations in $\mathcal{C}$, using the *CN generation* algorithm described in [13]. It is important to note that all PJ queries violating i) and ii) in Definition 3 are pruned during the enumeration.

4.1.2 Computing Upper Bounds of Relevance Scores

It can be shown that the score of a PJ query Q w.r.t. T, $\mathsf{score}(T \mid Q)$, can be upper bounded by its column-wise score $\mathsf{score}_{\text{col}}$ for any value parameter α. This simple but effective upper bound can be computed in a light-weight way, without executing any join in Q. We will use this upper bound frequently in the rest of this paper.

PROPOSITION 2. (Upper Bound of Score) *For any* $0 \leq \alpha \leq 1$,

$$\mathsf{score}(T \mid Q) \leq \frac{\mathsf{score}_{\text{col}}(T \mid Q)}{1 + \ln(1 + \ln|\mathcal{J}|)} \triangleq \overline{\mathsf{score}}(T \mid Q). \quad (6)$$

It suffices to show that $\mathsf{score}_{\text{row}}(T \mid Q) \leq \mathsf{score}_{\text{col}}(T \mid Q)$ to prove Proposition 2. The intuition is that, in $\mathsf{score}_{\text{row}}(T \mid Q)$, each row in T is matched to the most similar row in the output relation $\mathcal{A}(Q)$, while in $\mathsf{score}_{\text{col}}(T \mid Q)$, each cell in T is matched to the most similar cell in the corresponding column of $\mathcal{A}(Q)$. The latter has a weaker constraint in matching so it produces a higher score.

Computing $\overline{\mathsf{score}}$ and column containment score $\mathsf{score}_{\text{col}}$. To compute the upper bound $\overline{\mathsf{score}}$, it suffices to compute the column containment score $\mathsf{score}_{\text{col}}$ in Equation (4). To compute $\mathsf{score}_{\text{col}}$, for each term w appearing in column i of T, suppose column i is mapped to column j of a relation R, we need to scan $\mathsf{inv}(\mathsf{w}, R[j])$ once to compute the cell similarity $\mathsf{score}(t[i] \mid r[j])$ for each cell $t[i]$ in T and each cell $r[j]$ in R. Then $\mathsf{score}_{\text{col}}$ can be computed directly as in (4). Refer to Algorithm 1 for more details.

1: Initialize cell similarities $\mathsf{score}_{\text{cell}}$ to be 0 for all entries.
2: For each column i of T: it is mapped to $R[j]$ of relation R
3: For each tuple $t \in T$ and each term w in $t[i]$
4: Retrive row-level inverted index $\mathsf{inv}(\mathsf{w}, R[j])$;
5: For each row $r \in \mathsf{inv}(\mathsf{w}, R[j])$: $\mathsf{score}_{\text{cell}}(t[i] \mid r[j])$++.
6: Compute $\mathsf{score}_{\text{col}}(T \mid Q)$ from $\mathsf{score}_{\text{cell}}$ as in (4).
7: Compute $\overline{\mathsf{score}}(T \mid Q)$ as in (6).

Algorithm 1: Computing $\mathsf{score}_{\text{col}}(T \mid Q)$ and $\overline{\mathsf{score}}(T \mid Q)$

PROPOSITION 3. *Consider an example spreadsheet T and a PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ in a database. For each column i in T and each term w in the column $T[i]$, suppose i is mapped to a column $R[j]$ in a relation R of $\mathcal{J}$, let $l_{\mathsf{w}} = |\mathsf{inv}(\mathsf{w}, R[j])|$ be the number of rows in R that contain w in the column $R[j]$, i.e., the size of row-level inverted index $\mathsf{inv}(\mathsf{w}, R[j])$ for term w in column $R[j]$. Algorithm 1 computes $\overline{\mathsf{score}}$ and $\mathsf{score}_{\text{col}}$ in $O(\sum_{\mathsf{w} \in T} l_{\mathsf{w}})$ time.*

4.1.3 Evaluating PJ Queries for Relevance Scores

We now introduce how to *evaluate* PJ queries, i.e., *to compute the exact relevance score* $\mathsf{score}(T \mid Q)$. According to (5), the only missing part is the row containment score $\mathsf{score}_{\text{row}}(T \mid Q)$.

Computing score and row containment score $\mathsf{score}_{\text{row}}$. As in (3), for each example tuple t in T, we need to find the most similar row in the output relation $\mathcal{A}(Q)$. Indeed, we can first execute the PJ query Q, for example, sending it as a SQL query to the database (as in [22]), and then examine each row in the output relation $\mathcal{A}(Q)$. To utilize our in-memory indexes and compute the scores more efficiently, we design an execution plan for Q using hash joins. More details are in Appendix B.1. Follwoing is its complexity.

PROPOSITION 4. *Consider an example spreadsheet T and a PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ in a database. For each term w in T, let l_{w} be defined as in Proposition 3. For each relation R in $\mathcal{J}$, let $|R|$ be the number of rows and $d_{\mathcal{J}}(R)$ be the degree of R in $\mathcal{J}$. We can compute $\mathsf{score}(T \mid Q)$ in $O(\sum_{R \in \mathcal{J}} |R| \cdot d_{\mathcal{J}}(R) + \sum_{\mathsf{w} \in T} l_{\mathsf{w}})$ time.*

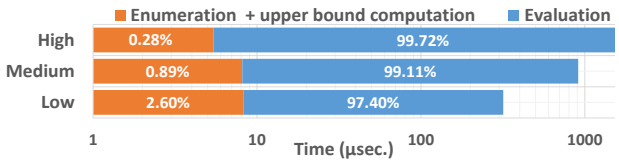


Figure 5: Average running time of "query enumeration + upper bound computation" v.s. "query evaluation" per PJ query

4.1.4 Cost Analysis

It is quite common to enumerate candidate database queries in previous keyword search literature [5, 13, 12, 15]. Similarly in our case, we only access the schema graph and column-level inverted index to enumerate $\mathcal{Q}_{\text{C}}$ (no need to execute any actual join).

Now, comparing the complexity of computing upper bounds of scores (Proposition 3) with computing exact scores (Proposition 4), we find that the additional cost, $O(\sum_{R \in \mathcal{J}} |R| \cdot d_{\mathcal{J}}(R))$, to compute the exact score is truly the bottleneck. In the worst case, this cost is proportional to the total number of rows in all relations involved in a PJ query. On the other hand, the cost, $O(\sum_{\mathsf{w} \in T} l_{\mathsf{w}})$, to compute the upper bound is only proportional to the number of rows that contain keyword terms in projection columns of the PJ query.

Figure 5 compares i) the time for query enumeration plus upper bound computation (orange bars), with ii) the time to compute exact scores via query evaluation (blue bars), on average per query. We generate 50 example spreadsheets for CSUPP dataset and divide them into three buckets (H, M, L) based on the frequency of terms in the dataset (from highly frequent to lowly). Please refer to Section 6.1 for more details about the setting. All PJ queries are generated for those example spreadsheet, and for each, we compute both the upper-bound score and the exact score. The experimental result also shows that query enumeration plus upper bound computation requires a negligible fraction of the overall processing time. So, in the rest part, we assume that $\mathcal{Q}_{\text{C}}$ *can be enumerated first for each given example spreadsheet* and $\overline{\mathsf{score}}$ *is associated with each query in* $\mathcal{Q}_{\text{C}}$ *generated in the query enumeration step.*

4.2 Minimal Evaluation Set

From both theoretical and experimental analysis in the last subsection, we find that the bottleneck in our PJ-query enumeration-ranking framework is to *evaluate candidate PJ queries in* $\mathcal{Q}_{\text{C}}$, i.e., to *execute PJ queries and compute their exact scores*; and on the other hand, we can enumerate all PJ queries in $\mathcal{Q}_{\text{C}}$ and compute the upper bounds of scores for all of them quite efficiently. Now the major challenge translates to that of *evaluating as few PJ queries in* $\mathcal{Q}_{\text{C}}$ *as possible to get the top-k with highest scores*, given that *an upper bound of score is associated with each one in* $\mathcal{Q}_{\text{C}}$.

In the rest of this paper, we will write $\overline{\mathsf{score}}(T \mid Q)$ as $\overline{\mathsf{score}}(Q)$ and $\mathsf{score}(T \mid Q)$ as $\mathsf{score}(Q)$ if T is clear from the context.

Given the set of queries in $\mathcal{Q}_{\text{C}}$ and their upper-bound scores, we now analyze what is the minimal (sub)set $\mathcal{Q}_{\min} \subseteq \mathcal{Q}_{\text{C}}$ of queries we have to evaluate to get the top-k with the highest scores in $\mathcal{Q}_{\text{C}}$. Let $\mathcal{Q}_{\text{C}} = \{Q_1, Q_2, \ldots, Q_N\}$, where Q_i's are ordered by their upper-bound scores $\overline{\mathsf{score}}(Q_1) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$. Intuitively, if we evaluate queries in $\mathcal{Q}_{\text{C}}$ in this order, we can *terminate* if

$$\text{top}_k\{\mathsf{score}(Q_1), \ldots, \mathsf{score}(Q_i)\} > \overline{\mathsf{score}}(Q_{i+1}), \quad (7)$$

where $\text{top}_k\{\ldots\}$ is the k-th largest number in the set, we can assert that the top-k queries are among $\{Q_1, \ldots, Q_i\}$. Let i^* be the minimal index i that satisfies the *termination condition* in (7), let

$$\mathcal{Q}_{\min} = \{Q_1, Q_2, \ldots, Q_{i^*}\} \subseteq \mathcal{Q}_{\text{C}}. \quad (8)$$

We can show that, informally, if based on only the upper-bound information $\overline{\mathsf{score}}$, $\mathcal{Q}_{\min}$, called *minimal evaluation set*, is the minimal subset of queries in $\mathcal{Q}_{\mathrm{C}}$ we have to evaluate to get the top-k.

PROPOSITION 5. (Optimality of $\mathcal{Q}_{\min}$) *Given a set of PJ queries $\mathcal{Q}_{\mathrm{C}}$ and upper bounds $\overline{\mathsf{score}}$ of their scores. Any multi-step ranking algorithm to find queries with the top-k scores in $\mathcal{Q}_{\mathrm{C}}$ has to evaluate all queries in the set $\mathcal{Q}_{\min}$ (in* (8)*) to compute their scores.*

The formalization of the class of *multi-step ranking algorithms* and the proof of this proposition are given in Appendix C.

4.3 Worst-Case Optimal Baseline Strategy

Based on the upper bounds of scores and the optimality of $\mathcal{Q}_{\min}$ introduced in the above two subsections, we have a simple but "worst-case optimal" strategy BASELINE described in Algorithm 2.

The idea is to evaluate queries $Q_1, Q_2, \ldots$ in $\mathcal{Q}_{\mathrm{C}}$ one by one (to get the exact scores) in the descending order of upper-bound scores. Recall that upper-bound scores (line 1) can be computed efficiently and are associated with all queries in $\mathcal{Q}_{\mathrm{C}}$, as discussed in Sections 4.1.2 and 4.1.4. For each Q_i, we use the operator introduced in Section 4.1.3 to compute its true score (line 4). If after we finish evaluating Q_i, (7) is satisfied, then we can terminate and output the current top-k among the evaluated PJ queries.

Input: queries in $\mathcal{Q}_{\mathrm{C}}$ and upper bounds of their scores
Output: top-k PJ queries in $\mathcal{Q}_{\mathrm{C}}$ with the highest scores

1: Sort queries in $\mathcal{Q}_{\mathrm{C}}$: $\overline{\mathsf{score}}(Q_1) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$.
2: Initialize $\mathcal{Q}_{\mathrm{topk}} \leftarrow \emptyset$.
3: For $i = 1$ to N do
4: Evaluate Q_i to compute $\mathsf{score}(Q_i)$.
5: Let $\mathcal{Q}_{\mathrm{topk}} \leftarrow \mathcal{Q}_{\mathrm{topk}} \cup \{Q_i\}$; if $|\mathcal{Q}_{\mathrm{topk}}| > k$, keep only queries in $\mathcal{Q}_{\mathrm{topk}}$ with the top-k highest scores.
6: If termination condition (7) is satisfied, exit the loop.
7: Output $\mathcal{Q}_{\mathrm{topk}}$.

Algorithm 2: Baseline Evaluation Strategy: BASELINE

It is not hard to show that BASELINE evaluates only queries in $\mathcal{Q}_{\min}$ and thus is worst-case optimal for finding top-k.

THEOREM 1. (Correctness and Optimality) BASELINE *correctly finds the PJ queries with the top-k highest scores among $\mathcal{Q}_{\mathrm{C}}$, and evaluates only queries in the minimal evaluation set $\mathcal{Q}_{\min}$.*

Disadvantages of the baseline. BASELINE strategy in Algorithm 2 is worst-case optimal in terms of the number of PJ queries it evaluates. But the evaluation cost (or, response time) can be potentially improved significantly. BASELINE has several disadvantages. First, this baseline algorithm does not utilize frequent common subexpressions in $\mathcal{Q}_{\mathrm{C}}$. Indeed, we can cache the output relations of some subexpressions so that they can be re-used for more than one PJ query $\mathcal{Q}_{\mathrm{C}}$. Secondly, as we do not have infinite memory, to maximize the benefit we get from caching, we need to make the decisions of which subexpressions to be cached and when, for a batch of PJ queries. BASELINE examines queries in $\mathcal{Q}_{\mathrm{C}}$ one-by-one, so such decisions cannot be made wisely. The strategy we introduce in Section 5 improves BASELINE from the above two angles.

5. OPTIMIZING CACHING-EVALUATION

We introduce our cache-aware optimization techniques in this section to overcome the disadvantages of BASELINE. The goal is still to find PJ queries with the top-k scores in $\mathcal{Q}_{\mathrm{C}}$ w.r.t. the user-given example spreadsheet T. We will first discuss how to evaluate a PJ query. i.e., compute its score, when output relations of some sub-parts of it are cached, together with a cost model which quantify the cost of such cache-aware evaluation, in Section 5.1. We then formulate the *cache-evaluation scheduling problem*, i.e., an abstract version of the task to be solved by caching-evaluation scheduler in Section 5.2. Core technical challenges we resolve here are: to determine the order of PJ queries in $\mathcal{Q}_{\mathrm{C}}$ to be evaluated, which sub-PJ queries to be cached, and for how long, with the goal of minimizing the evaluation cost, or maximizing the benefit we get from caching. We show that it is NP-hard. Section 5.3 introduces a near-optimal solution to this problem. We will discuss how to extend our approach for incremental computation in Section 5.4.

5.1 Cache-Aware Evaluation of PJ Queries

Recall that the evaluation of $\mathsf{score}_{\mathrm{col}}$ is light-weight without executing any join of relations as discussed in Section 4.1.2. So here, we focus on computing $\mathsf{score}_{\mathrm{row}}$ of a PJ query Q w.r.t. T.

Caching sub-PJ queries in evaluation. Let's formally define a *sub-PJ query* of Q, and discuss how to evaluate PJ query Q if the output relations of some sub-PJ queries of Q have been cached.

DEFINITION 4. (Sub-PJ Query) *$Q' = (\mathcal{J}', \mathcal{C}', \phi_{\mathcal{C}'})$ is said to be a sub-PJ query of a PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$, iff $\mathcal{J}'$ is a subtree of $\mathcal{J}$; $\mathcal{C}' \subseteq \mathcal{C}$ is a subset of columns from relations in $\mathcal{J}'$ which the columns in T are mapped to; and $\phi_{\mathcal{C}'}$ is a range-restriction of ϕ by $\mathcal{C}'$, i.e., $\phi_{\mathcal{C}'}(i) = \phi(i)$ iff $\phi(i) \in \mathcal{C}'$ and is undefined iff $\phi(i) \in \mathcal{C} - \mathcal{C}'$. We denote $Q' \preceq Q$ if Q' is a sub-PJ query of Q.*

Two types of subtrees are considered here: i) $\mathcal{J}'$ is a subtree of $\mathcal{J}$ rooted at some internal node, containing all leaves below; and ii) a type-i) subtree plus the parent of its root.

Figure 3 shows two sub-PJ queries Q'_1 (left) and Q'_2 (right) of the PJ query Q in Figure 2(b)-(i).

We have a *sub-PJ query cache* $\mathcal{M}$ in our system, which temporarily stores the output relations of some sub-PJ queries in a budgeted amount of memory. The execution plans of PJ-queries can be easily extended to take advantage of the cached output relations of sub-PJ queries: for a PJ query Q and a set of cached sub-PJ queries in $\mathcal{M}$, instead of starting from the leaves of Q, we start from the output relations of maximal sub-PJ queries of Q in $\mathcal{M}$ and follow the execution plan of Q afterward. Q' is said to be a *maximal sub-PJ query of Q in $\mathcal{M}$*, iff $Q' \preceq Q$ and there is no Q'' whose output relation is cached in $\mathcal{M}$ such that $Q' \prec Q'' \preceq Q$. Intuitively, we want to utilize the output relations cached in $\mathcal{M}$ as much as possible. More details are given in Appendix B.2.

Cost model of evaluation. We do not have unlimited memory to cache every single sub-PJ query. So our scheduling-evaluation scheduler needs to determine which sub-PJ queries to be cached and for how long, so as to maximize the benefit we obtain from caching, or equivalently, to minimize the total evaluation cost. To this end, a cost model needs to be introduced.

We define $\mathrm{cost}(Q)$ to be the cost of evaluating a (sub-)PJ query Q, without utilizing the cache, and $\mathrm{cost}(Q, \mathcal{M})$ be the cost of evaluating Q when a set of sub-PJ queries $\mathcal{M}$ have their output relations in the cache. We abuse the notation $\mathcal{M}$ a bit – we use it to denote both the set of sub-PJ queries as well as their cached output relations. Details about the cost model can be found in (12)-(13) in Appendix B.3, which is calibrated to the execution plans we use.

5.2 Caching-Evaluation Scheduling Problem

We are given a set of PJ queries $\mathcal{Q}_{\mathrm{C}} = \{Q_1, Q_2, \ldots, Q_N\}$, ordered by their upper-bound scores $\overline{\mathsf{score}}(Q_1) \geq \overline{\mathsf{score}}(Q_2) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$. Let $\mathcal{T}(Q_i)$ be the set of all *sub-PJ queries* of Q_i and $\mathcal{T}(\mathcal{Q}) = \bigcup_{Q_i \in \mathcal{Q}} \mathcal{T}(Q_i)$ for a set of PJ queries $\mathcal{Q} \subseteq \mathcal{Q}_{\mathrm{C}}$.

We first introduce a general framework of our caching-evaluation strategies to find the top-k answers from $\mathcal{Q}_{\rm C}$, and then define the *caching-evaluation scheduling problem* to find the best strategy.

Operators. To utilize output relations of sub-PJ queries shared by multiple PJ-queries in $\mathcal{Q}_{\rm C}$, we maintain a cache $\mathcal{M}$ of size at most B. Three types of operators are allowed:

a) $\underline{\rm Evaluate}(Q, \mathcal{M})$: to evaluate a PJ query or a sub-PJ query Q using output relations cached in $\mathcal{M}$, and get its relevance score $\mathsf{score}(T \mid Q)$ (for $Q \in \mathcal{Q}_{\rm C}$) and output relation $\mathcal{A}(Q)$.

b) $\underline{\rm Add}(Q, \mathcal{M})$: to store the output relation $\mathcal{A}(Q)$ in $\mathcal{M}$.

c) $\underline{\rm Delete}(Q, \mathcal{M})$: to delete $\mathcal{A}(Q)$ from $\mathcal{M}$.

The size of $\mathcal{M}$, denoted as $|\mathcal{M}|$, is the amount of memory we need to keep the output relations of (sub-)PJ queries in $\mathcal{M}$. We want to ensure that, at any time, $|\mathcal{M}|$ could be at most B.

Caching-evaluation schedule and termination condition. Initially, the cache $\mathcal{M}$ is empty, and for every PJ query $Q_i \in \mathcal{Q}_{\rm C}$ we only know its upper-bound score. We want to apply the above three types of operators in some order on PJ queries in $\mathcal{Q}_{\rm C}$ and their sub-PJ queries. The goal is that, at some point, the set of evaluated queries, denoted as $\mathcal{Q}_{\rm E}$, satisfies: $\mathcal{Q}_{\rm E} \supseteq \mathcal{Q}_{\min}$, i.e., the top-$k$ have been found. Note that $\mathcal{Q}_{\min}$ is not known in advance.

Objective. Each type-a operator $\mathrm{Evaluate}(Q, \mathcal{M})$ (Q is either a PJ query from $\mathcal{Q}_{\rm C}$ or a sub-PJ query) has a cost, $\mathrm{cost}(Q, \mathcal{M})$, as defined in (13) in Appendix B.3. The goal is to minimize the total evaluation cost of type-a operators. Each type-b/c operator also has a cost but it is negligible compared to type-a's cost.

Problem statement (CACHE-EVAL SCHEDULER) *Given a set of PJ queries $\mathcal{Q}_{\rm C} = \{Q_1, Q_2, \ldots, Q_N\}$, with their upper-bound scores $\overline{\mathsf{score}}(Q_1) \geq \overline{\mathsf{score}}(Q_2) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$, w.r.t. an example spreadsheet T, a sequence of operators in type-a,b,c are chosen to be executed and the objective is to:*

$$\text{minimize} \sum_{\text{Op. } \mathrm{Evaluate}(Q,\mathcal{M})\text{'s executed}} \mathrm{cost}(Q, \mathcal{M})$$
$$\text{s.t. } \mathcal{M} \text{ has size at most } B \text{ at any time, and}$$
$$\text{eventually } \mathcal{Q}_{\rm E} \supseteq \mathcal{Q}_{\min}.$$

THEOREM 2. (Hardness) *Even when the set $\mathcal{Q}_{\min}$ is known, the* CACHE-EVAL SCHEDULER *problem is NP-complete, with $|\mathcal{Q}_{\rm C}| + |\mathcal{T}(\mathcal{Q}_{\rm C})|$ as the input size, where $|\mathcal{T}(\mathcal{Q}_{\rm C})|$ is the total number of sub-PJ queries of queries in $\mathcal{Q}_{\rm C}$.*

5.3 A Near-Optimal Strategy

We will introduce a near-optimal strategy, FASTTOPK, for the CACHE-EVAL SCHEDULER problem. It is based on two heuristics: *guessing the minimal evaluation set* and *caching critical sub-PJ queries*. We will also analyze the theoretical guarantee on performance it provides, in terms of the evaluation cost.

5.3.1 Guessing The Minimal Evaluation Set

The first challenge is that the minimal evaluation set $\mathcal{Q}_{\min}$ is unknown to us. Our strategy BASELINE (Algorithm 2) examines only queries in $\mathcal{Q}_{\min}$ because it evaluates queries one by one, but $\mathcal{Q}_{\min}$ is known only after it terminates. Ideally, if we know $\mathcal{Q}_{\min}$ in advance, we can find *frequent* sub-PJ queries in $\mathcal{Q}_{\min}$ (the ones contained by many PJ queries), and cache their output relations so that they can be re-used multiple times in the evaluation. Indeed, we can consider all queries in $\mathcal{Q}_{\rm C}$ in one batch, but since $\mathcal{Q}_{\min}$ is usually a small subset of $\mathcal{Q}_{\rm C}$, sub-PJ queries that are frequent in $\mathcal{Q}_{\rm C}$ may not be frequent in $\mathcal{Q}_{\min}$ or even do not exist in $\mathcal{Q}_{\min}$ – so our decision of which sub-PJ queries to be cached based on their frequencies in $\mathcal{Q}_{\rm C}$ is likely to be sub-optimal.

We note that $\mathcal{Q}_{\min}$ is a "prefix" of $\mathcal{Q}_{\rm C}$, and can be uniquely specified by the index i^* as in (7)-(8). So our heuristic to resolve this challenge is to create a few batches $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \ldots$ of queries in the order of $Q_1, Q_2, \ldots, Q_i, \ldots$ to approximate $\mathcal{Q}_{\min}$. We will optimize the cache-evaluation schedule for queries in each batch. After we finish evaluating each batch of queries in $\mathcal{B}_j$, we check the termination condition (7), and eventually we stop at $\mathcal{B}_{j^*}$ after evaluating queries in $\mathcal{Q}_{\rm E} = \mathcal{B}_0 \cup \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_{j^*} \supseteq \mathcal{Q}_{\min}$. On one hand, we want to create as few batches as possible, i.e., j^* is small, so common sub-PJ queries across different queries can be found in one batch and we can cache and re-use their output relations; and on the other hand, we do not want to evaluate too many additional queries that are not in $\mathcal{Q}_{\min}$, i.e., $\mathcal{Q}_{\rm E} - \mathcal{Q}_{\min}$ is small.

The following batch-forming strategy balances the two concerns. The first batch of queries to be evaluated is $\mathcal{B}_0 = \{Q_1, \ldots, Q_k\}$, as Q_i's are ordered by the upper-bound scores and B_0 has the least number of queries we have to evaluate to get the top-k. After finishing evaluating this batch, if the termination condition in (7) is not satisfied, we will consider the next batch with a slightly larger number of queries: $\mathcal{B}_1 = \{Q_{k+1}, \ldots, Q_{k(1+\epsilon)}\}$. Again, if the termination condition is satisfied, we can stop and output the top-k; and otherwise, we consider the next batch. In general, the jth batch is $\mathcal{B}_j = \{Q_{k(1+\epsilon)^{j-1}+1}, \ldots, Q_{k(1+\epsilon)^j}\}$, for some constant $\epsilon > 0$.

This high-level procedure is outlined in Algorithm 3. Recall the discussion in Section 4.1, upper-bound scores can be computed efficiently and are associated with all queries in $\mathcal{Q}_{\rm C}$ (line 1). The loop (lines 4-9) forms batches one-by-one as described above. The value of i in line 7 of every loop is the index of the last PJ query evaluated up to now, so line 9 can check the termination condition in (7) to see whether the top-k have been found.

Input: queries in $\mathcal{Q}_{\rm C}$ and upper bounds of their scores
Output: top-k PJ queries in $\mathcal{Q}_{\rm C}$ with the highest scores

1: Sort queries in $\mathcal{Q}_{\rm C}$: $\overline{\mathsf{score}}(Q_1) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$.
2: Initialize $\mathcal{Q}_{\rm topk} \leftarrow \emptyset$.
3: Let the first batch be $\mathcal{B}_0 \leftarrow \{Q_1, \ldots, Q_k\}$, $i \leftarrow k$, and $j \leftarrow 0$.
4: Do
5: BatchEval($\mathcal{B}_j$) to get $\mathsf{score}(Q)$'s for all $Q \in \mathcal{B}_j$.
6: Let $\mathcal{Q}_{\rm topk} \leftarrow \mathcal{Q}_{\rm topk} \cup \mathcal{B}_j$;
keep only queries in $\mathcal{Q}_{\rm topk}$ with the top-k highest scores.
7: Let $i \leftarrow k(1+\epsilon)^j$ and then $j \leftarrow j + 1$.
8: Form next batch: $\mathcal{B}_j = \{Q_{i+1}, Q_{i+2}, \ldots, Q_{k(1+\epsilon)^j}\}$.
9: While $(\mathrm{top}_k\{\mathsf{score}(Q_1), \ldots, \mathsf{score}(Q_i)\} \leq \overline{\mathsf{score}}(Q_{i+1}))$
10: Output $\mathcal{Q}_{\rm topk}$.

Algorithm 3: Near-Optimal Strategy: FASTTOPK

Suppose the algorithm terminates with $i = i_{\rm end}$. In Section 5.3.3, we will utilize the fact that $i^* \leq i_{\rm end} \leq i^*(1+\epsilon)$, where $i^* = |\mathcal{Q}_{\min}|$ is the least number of PJ queries any strategy has to evaluate, to give a performance guarantee of our strategy. Intuitively, based on this fact, we have that the size of $\mathcal{Q}_{\rm E}$, the set of queries evaluated by FASTTOPK (Algorithm 3), is at most $(1+\epsilon)|\mathcal{Q}_{\min}|$. So FASTTOPK does not examine too many additional queries that are not in $\mathcal{Q}_{\min}$. To bound the total evaluation cost, we can also show that there are at most $O(\log_{1+\epsilon}(|\mathcal{Q}_{\min}|/k))$ batches.

The only missing building block in FASTTOPK (Algorithm 3) is now the subroutine $\mathrm{BatchEval}(\mathcal{B}_j)$ in line 5.

5.3.2 Caching Critical Sub-PJ Queries

Let's focus on the subroutine $\mathrm{BatchEval}(\mathcal{B}_j)$. We want to evaluate a batch $\mathcal{B}_j$ of queries using operators in type-a/b/c, so that the total cost is minimized. The basic idea of our approach for this

subroutine is to partition queries in a batch $\mathcal{B}_j$ into groups, such that each group of PJ queries share at least one heavy-cost sub-PJ query, called *critical sub-PJ query*. The output relation of this sub-PJ query is cached in $\mathcal{M}$ if its size is no more than the budget B. We will later show that this seemingly simple heuristic has strong theoretical/practical performance guarantee.

Critical sub-PJ query. Let $\mathcal{T}(Q_i)$ be the set of all sub-PJ queries of Q_i, and $\mathcal{T}(\mathcal{B}_j) = \bigcup_{Q_i \in \mathcal{B}_j} \mathcal{T}(Q_i)$. A sub-PJ query $Q^* \in \mathcal{T}(Q_i)$ is said to be *critical* to Q_i in $\mathcal{B}_j$, if i) Q^* is shared by more than one query: there exists $Q_{i'} \in \mathcal{B}_j$ s.t. $i' \neq i$ and $Q^* \in \mathcal{T}(Q_{i'}) \cap \mathcal{T}(Q_i)$; and ii) Q^* has the highest cost, $\text{cost}(Q^*)$, among those in $\mathcal{T}(Q_i)$ satisfying condition i).

It is intuitive that the output relation of a critical sub-PJ query is worth caching in $\mathcal{M}$, because it can benefit the evaluation of at least one PJ query in $\mathcal{B}_j$, as condition i); and it has the highest cost among all such sub-PJ queries of a PJ query, as condition ii), so that we can benefit the most from caching it. We describe this heuristic for $\text{BatchEval}(\mathcal{B}_j)$ more formally in Algorithm 4.

Input: queries in a batch $\mathcal{B}_j$ and cache budget B
Task: get $\mathsf{score}(Q)$ by evaluating every $Q \in \mathcal{B}_j$

1: Sort the M sub-PJ queries in $\mathcal{T}(\mathcal{B}_j)$ as:
$\text{cost}(Q'_1) \geq \text{cost}(Q'_1) \geq \ldots \geq \text{cost}(Q'_M)$.
2: While $\mathcal{B}_j$ is not empty
3: Clear $\mathcal{M}$ using type-c operators Delete.
4: Pick $Q^* = \text{argmax}_{Q' \in \mathcal{T}(\mathcal{B}_j)}\{\text{cost}(Q') \mid |\mathcal{A}(Q')| \leq B \wedge \exists i_1 \neq i_2 : Q' \in \mathcal{T}(Q_{i_1}) \cap \mathcal{T}(Q_{i_2})\}$.
5: If no such Q^* can be found, evaluate all the remaining queries in $\mathcal{B}_j$ and terminate.
6: Let $\text{Critical}^{-1}(Q^*) \leftarrow \{Q_i \in \mathcal{B}_j \mid Q^* \in \mathcal{T}(Q_i)\}$.
7: Execute Evaluate$(Q^*, \mathcal{M})$ and Add$(Q^*, \mathcal{M})$. (evaluate Q^* and store its output relation in $\mathcal{M}$)
8: For each $Q_i \in \text{Critical}^{-1}(Q^*)$ do
9: Evaluate Q_i using $\mathcal{M}$: call Evaluate$(Q_i, \mathcal{M})$.
10: Let $\mathcal{B}_j \leftarrow \mathcal{B}_j - \text{Critical}^{-1}(Q^*)$.

Algorithm 4: $\text{BatchEval}(\mathcal{B}_j)$: strategy for evaluating a batch of queries for their scores while minimizing the cost

In line 1, all sub-PJ queries of $Q = (\mathcal{J}, \mathcal{C}, \phi) \in \mathcal{B}_j$ can be enumerated efficiently by retrieving the rooted subtree at each node in $\mathcal{J}$. As discussed in Appendix B.3, the cost of each sub-PJ query Q'_i can be computed efficiently from (12) by summing up the sizes of relations in Q'_i and row-level inverted indexes for columns in the projection of Q'_i. For each iteration (lines 2-10), the critical sub-PJ query Q^* in $\mathcal{B}_j$ with the highest cost and output relation of size no more than B is picked (line 4). Since we are focusing on foreign-key joins, we will use the number of rows in the root relation of Q' multiplying the number of columns in the output relation as the size of the output relation, $|\mathcal{A}(Q')|$. The set of queries in $\mathcal{B}_j$ containing Q^* as a sub-PJ query is put into a set $\text{Critical}^{-1}(Q^*)$ is (line 6). We first evaluate and cache the output relation of Q^* in $\mathcal{M}$ (lines 7). Then all queries having Q^* as its sub-PJ query are evaluated to get their scores using $\mathcal{M}$ (lines 8-9), and removed form $\mathcal{B}_j$ (line 10). After that, the cache $\mathcal{M}$ is cleared up (line 3).

EXAMPLE 4. *Consider a batch of three PJ queries, $\mathcal{B}_j = \{Q_1, Q_2, Q_3\}$, where Q_1-Q_3 are depicted in Figure 2(b)-(i), (ii), (iii), respectively. Two sub-PJ queries Q'_1 and Q'_2 of them are shown in Figure 3 (left and right, respectively). Both are contained in two queries Q_1 and Q_3, so could potentially be critical sub-PJ queries to Q_1 and Q_3. If Q'_2 has the largest cost $\text{cost}(Q'_2)$ among all such sub-PJ queries, then Q'_2 is a critical to both Q_1 and Q_3, and line 4 will pick $Q^* = Q'_2$. We have $\text{Critical}^{-1}(Q^*) = \{Q_1, Q_3\} \subseteq B_j$ in line 6. Our strategy* FASTTOPK *would cache $\mathcal{A}(Q'_2)$ in $\mathcal{M}$ first (line 7), and then use it to evaluate Q_1 and Q_3 (lines 8-9). After that, only Q_2 is left in $\mathcal{B}_j$ and will be evaluated as in line 5.*

Although this subroutine is applied for each batch independently in Algorithm 3 to find the top-k in $\mathcal{Q}_\text{C}$, we will later show that it has an overall performance guarantee, using the fact that the total number of batches is small (the last value of j in Algorithm 3).

THEOREM 3. (Putting Together and Correctness) FASTTOPK *strategy (Algorithm 3 with subroutine* BatchEval *as Algorithm 4), correctly finds the queries in $\mathcal{Q}_\text{C}$ with the top-k highest scores.*

5.3.3 Performance Analysis

We will start with the time complexity of our strategy FASTTOPK, and then analyze its approximation ratio.

Time complexity. The online response time of our system is determined by i) time spent on generating PJ queries in $\mathcal{Q}_\text{C}$ and their upper bounds, ii) spent on evaluating PJ queries, and iii) spent optimizing the scheduling of caching and evaluations (FASTTOPK strategy). i) is negligible compared to ii) (analyzed in Section 4.1.4). ii) is modeled as the objective in our CACHE-EVAL SCHEDULER problem and the goal of FASTTOPK is to reduce it as much as possible. The purpose of the following theorem is to show that, excluding the time spent on executing operators in type-a/b/c, the time spent by FASTTOPK on iii) is negligible compared to ii).

THEOREM 4. (Time Complexity) *Given a set of PJ queries in $\mathcal{Q}_\text{C}$ and their sub-PJ queries $\mathcal{T}(\mathcal{Q}_\text{C})$ with costs and upper-bound scores associated, the time complexity of* FASTTOPK *is $O(N + M_\text{all}(s_\max + \log M_\text{all}))$ (excluding the running time of operators in type-a/b/c chosen by* FASTTOPK *to be run), where $N = |\mathcal{Q}_\text{C}|$ is the number of PJ queries in $\mathcal{Q}_\text{C}$, $M_\text{all} = \sum_{Q_i \in \mathcal{Q}_\text{C}} |\mathcal{T}(Q_i)|$ here is the total number of sub-PJ queries of queries in $\mathcal{Q}_\text{C}$, and $s_\max$ is the max size of a join tree $\mathcal{J}$ of a PJ query in $\mathcal{Q}_\text{C}$. Because of the definition of a sub-PJ query, we have $M_\text{all} = \Theta(s_\max \cdot N)$.*

Performance ratio. FASTTOPK strategy (Algorithms 3-4) provide a feasible solution (a scheduling of operators in type a/b/c) to the CACHE-EVAL SCHEDULER problem. We now compare the cost of this strategy with the cost of the optimal solution (which is hard to be found as shown in Theorem 2).

For a set of PJ queries $\mathcal{Q} = \{Q_i\}$, let $\text{cost}_\text{TOT}(\mathcal{Q})$ be the total cost of evaluating queries in $\mathcal{Q}$ one by one without caching:

$$\text{cost}_\text{TOT}(\mathcal{Q}) = \sum_{Q_i \in \mathcal{Q}} \text{cost}(Q_i).$$

Let $\text{cost}_\text{OPT}(\mathcal{Q})$ be the cost of evaluating all queries in $\mathcal{Q}$ using the optimal strategy in the CACHE-EVAL SCHEDULER problem.

Let $\text{cost}_\text{SOL}(\mathcal{Q})$ be the cost of evaluating all queries in $\mathcal{Q}$ using our FASTTOPK strategy in Algorithms 3-4.

THEOREM 5. (Performance Ratio) *Given PJ queries in $\mathcal{Q}_\text{C}$ with upper-bound scores associated, the strategy* FASTTOPK *in Algorithms 3-4 evaluates a set of PJ queries $\mathcal{Q}_\text{E}$ s.t.*

$$|\mathcal{Q}_\min| \leq |\mathcal{Q}_\text{E}| \leq (1+\epsilon)|\mathcal{Q}_\min|, \tag{9}$$

and the benefit from caching

$$\text{cost}_\text{TOT}(\mathcal{Q}_\text{E}) - \text{cost}_\text{SOL}(\mathcal{Q}_\text{E}) \tag{10}$$
$$\geq \frac{1}{2c^2}\left(\text{cost}_\text{TOT}(\mathcal{Q}_\text{E}) - \log_{1+\epsilon}\left(\frac{|\mathcal{Q}_\min|}{k}\right) \cdot \text{cost}_\text{OPT}(\mathcal{Q}_\text{E})\right)$$

where c is the number of columns in T.

Informally, the above theorem gives guarantees for our FASTTOPK strategy from two aspects: i) it does not evaluate too many additional PJ queries in additional to the necessary ones in $\mathcal{Q}_{\min}$, as in (9); and ii) $\text{cost}_{\text{TOT}}(\mathcal{Q}_{\text{E}}) - \text{cost}_{\text{SOL}}(\mathcal{Q}_{\text{E}})$ is the benefit we obtained from caching, and is lower bounded by the gap between the total cost and the cost of the optimal strategy (RHS of (10)).

5.3.4 Heuristics for Further Improvement

Our strategy can be further improved. Although not improving the performance ratio in Theorem 5, the following two heuristics are effective to improve its performance in practice.

The first heuristic is as follows. Consider each iteration of lines 3-10 in Algorithm 4, only one output relation (the one of Q^*) is cached in $\mathcal{M}$. Indeed, if there is still room in $\mathcal{M}$, it will always be beneficial to cache more sub-PJ queries to speed up the evaluation of queries in $\text{Critical}^{-1}(Q^*)$. So our heuristic here is to order queries in $\text{Critical}^{-1}(Q^*)$ in such a way that "similar" queries (sharing common sub-PJ queries) are consecutive. While we evaluate these queries one-by-one in this order, the standard LRU replacement algorithm is applied to insert and replace output relations of sub-PJ queries in $\mathcal{M}$ – but note that we never replace the output relation of Q^* until finishing evaluating all queries in $\text{Critical}^{-1}(Q^*)$. Such an order can be formed as follows. Starting with any query in $\text{Critical}^{-1}(Q^*)$, in each step, we pick the query, which shares the most sub-PJ queries with the last one but is not in the order yet, to be the next one in this order. Repeat until all queries are placed in the order.

The second heuristic is an extension to our termination condition in (7). We call it the *skipping condition*. During the execution of FASTTOPK, we maintain the current kth highest score for all the queries that have been evaluated. In line 9 of Algorithm 4, before we evaluate Q_i, we first check whether its upper-bound score is higher than the current top-k score – if not, we can safely skip the evaluation of Q_i. This heuristic is particularly powerful and necessary for the last batch of queries in Algorithm 3, as this batch is usually large and contains queries not in $\mathcal{Q}_{\min}$.

5.4 Incremental Computation

Our strategy can be easily extended for incremental computation. The incremental version of our end-to-end system task is: suppose we have found the top-k PJ queries for a user-given example spreadsheet T, after one or more cells in T are updated – the updated example spreadsheet is denoted as T' – how to find the top-k PJ queries for T' by re-using the evaluation results for T.

If the user adds/deletes a column in T, we re-start and generate a completely new caching-evaluation schedule using FASTTOPK (Algorithms 3-4), because in this case, the set of PJ queries, $\mathcal{Q}_{\text{C}}{}'$, to be evaluated for T' are different from $\mathcal{Q}_{\text{C}}$ for T.

We focus on speeding up the case when the set of columns in T are unchanged, but some rows are updated (with one or more cells). In this case, $\mathcal{Q}_{\text{C}}{}'$ may have large overlap with $\mathcal{Q}_{\text{C}}$ and thus evaluation results for $\mathcal{Q}_{\text{C}}$ can be re-used. The basic ideas are to derive a tighter upper bound of $\mathsf{score}(T' \mid Q)$ based on the unchanged part of T and to schedule the incremental part of evaluation for $\mathcal{Q}_{\text{C}}{}'$ carefully. Refer to Appendix A.1 for more details.

6. EXPERIMENTAL EVALUATION

We present an experimental study of the techniques proposed in this paper. We evaluate and compare three algorithms.

- NAIVE: evaluates all the enumerated PJ queries in $\mathcal{Q}_{\text{C}}$;
- BASELINE (Algorithm 2): as described in Section 4.3;
- FASTTOPK (Algorithms 3-4): as described in Section 5.

We compare the performance of the three algorithms, and evaluate their sensitivity with respect to various parameters (Section 6.2). We also conduct a user study to evaluate the effectiveness of our scoring model (Section 6.3). Additional experiments about incremental computation are deferred to the appendix.

6.1 Settings of Experiments

We have implemented all the algorithms using C++/CLI (Common Language Infrastructure) on a Windows 8.1 machine with an Intel i7-4770 CPU at 3.4GHz with 16GB RAM.

Datasets. We use two datasets to evaluate the system performance: CSUPP and AdventureWorks. Our primary dataset, CSUPP, is a real-life database containing information related to customer service and IT support from a Fortune 500 Company. It has a size of 95GB. AdventureWorks, ADVW for short, is a synthetic database with information related to sales, purchasing, product management and contact management with size 300MB [2]. Although ADVW has a small size, we use it for its realistic and complex schema (93 primary key-foreign key edges compared with 63 in CSUPP), and also scale up its dimension/fact tables by creating new rows.

	# Relations	# Columns	# Text Columns	# Edges
CSUPP	105	1721	821	63
ADVW	71	650	104	93

For the user study, we use the real database IMDB [4] with information about movies, because our judges are more familiar with the movie domain compared with CSUPP or ADVW.

Index building. To build the inverted indexes, we tokenize each cell in each text column in the database. We discard tokens containing non-alphanumeric characters and those with more than 15 characters. For each token, we construct a list consisting of column identifiers (which uniquely identifies a column across all columns in the database) of all columns containing it. This forms the *column-level inverted index*. For each token in each column, we construct a list consisting of the row identifiers (which identifies the row within the relation) of all cells containing it in the column. This forms *the row-level inverted index*. We also build an *in-memory (key, foreign key) snapshot* as discussed in Section 3.1. We store all indexes in memory. Table 1 shows the index sizes. For both databases, the total index size is about **7%** of the database size.

	Inv. index (MiB)	(key,fk) snap. (MiB)	Tokens
CSUPP	4759.7	1237.4	6434684
ADVW	6.86	12.57	125083

Table 1: Index sizes

Example spreadsheet (ES) generation. We manually choose 10 semantically meaningful join queries with 6 or more text columns. We execute them and project the results on all the text columns involved. We generate an ES with m rows and n columns by (i) randomly choosing one of the semantically meaningful join queries and (ii) randomly choosing m rows and n columns from its output. We keep only the first token of the cell and all cells of the ES are non-empty. We use $m = 3$ and $n = 3$ in all our experiments.

To simulate real-life inputs, we introduce relationship errors in the ESs (default is 2 errors). To introduce a relationship error, we randomly select a cell of an ES generated above and replace it with the value of another cell in the same column in the join query's output. As before, we keep only the first token of the chosen cell.

We generate 50 ESs for CSUPP and 450 ESs for ADVW. We divide the 50 ESs for CSUPP into 3 buckets, namely *low*, *medium*, and *high*, based on the sizes of row-level inverted indexes of terms in the ESs (from lowly frequent to highly frequent). There are 25, 15, and 10 ESs in the three buckets, respectively. This is to test how our approaches are sensitive about the frequency of terms.

Description	Symbol	Ranges and default values
Param. in scoring model	α	$0.5, 0.6, 0.7, \underline{0.8}, 0.9, 1.0$
k in top-k	k	$5, 10, 20, \underline{50}, 100$
Batch increase factor	ϵ	$0.2, \underline{0.4}, 0.6, 0.8, 1.0, 2.0$
Cache size (MiB)	B	$100, 200, 500, \underline{1000}, 2000$
# relationship errors	e	$0, 1, \underline{2}, \ldots, (m-1)*n$

Table 2: Parameters we vary in our experiments along with their description, value ranges, and default values (underlined)

6.2 System Performance

We compare the algorithms by: (i) execution time and (ii) number of PJ query-row evaluations (times PJ queries are evaluated on rows in the ES). (ii) indicates the benefit of *using upper bounds* $\overline{\text{score}}$ *of scores for early termination*. (i) indicates the combined benefit of *caching shared sub-PJ queries* and *early termination*.

In our experiments, we vary the 5 parameters in Table 2. Unless otherwise specified, we use the underlined default values. We use CSUPP as the dataset in Exp-I to IV, and ADVW in Exp-IIV.

Exp-I: Comparing FASTTOPK **with** NAIVE **and** BASELINE**.** Figure 6 shows the average execution times (in log scale) of the three algorithms for each of the three ES buckets (low, medium and high). The execution time here is partitioned into query "enumeration + upper bound computation" and "evaluation" (evaluating PJ queries to compute their scores). Figure 6 shows that the "enumeration + upper bound computation" part takes a tiny fraction of the overall execution time for all the three approaches.

We used default values of the 5 parameters shown in Table 2 for this experiment. FASTTOPK outperforms NAIVE *by factors of 11, 10 and 5* for the low, medium and high buckets respectively. NAIVE often takes several minutes to return answers, and it can be found from Figure 6 that the crucial bottleneck is query evaluation to compute scores. Such inefficiency motivates the problem addressed in this paper for interactive query discovery.

The improvement of execution time in BASELINE and FASTTOPK is the combined benefit gained from using the upper bounds $\overline{\text{score}}$ for early termination and caching shared sub-PJ queries.

Without using the upper bounds, NAIVE has to go through and evaluate all the PJ queries enumerated in $\mathcal{Q}_C$; but with the help of the upper bounds, BASELINE and FASTTOPK can terminate as soon as the condition in (7) is satisfied. Figure 7 plots the numbers of queries evaluated by the three approaches. The significantly smaller numbers of queries evaluated by BASELINE and FASTTOPK explain their faster execution time compared with NAIVE.

In Figure 6, FASTTOPK outperforms BASELINE *by factors of 5, 3 and 1.5* for the low, medium and high buckets respectively. This shows the benefit gained from solving the problem of *caching-evaluation scheduling* by our FASTTOPK strategy.

Exp-II: Vary cache size. Figure 8 shows execution time of the two algorithms for the low/high ES buckets. FASTTOPK outperforms the BASELINE for all cache sizes. Higher the cache size, more the sharing, larger the gap. With a cache size of B = 2GiB, FASTTOPK outperforms BASELINE *by a factor of 6X for the low-cost ESs* and *by 3.7X for medium-cost ESs*. The gap is smaller for high-cost ESs; FASTTOPK outperforms BASELINE *by a factor of 2.3X* for B = 2GiB. It is because the results of many of those common sub-PJ queries are too large to fit in the cache. We need a larger cache to obtain the full benefit of sharing for high-cost ESs and get speedups. The trend for the medium ES bucket is similar.

Exp-III: Vary parameter α**.** Figure 9(a) shows the execution times of the two algorithms with different values of α. Since $(1-\alpha)$ is the weight on the column containment score $\text{score}_{\text{col}}$ and the upper bound score is proportional to $\text{score}_{\text{col}}$, smaller values of α imply tighter upper bound scores and thus faster early termination.

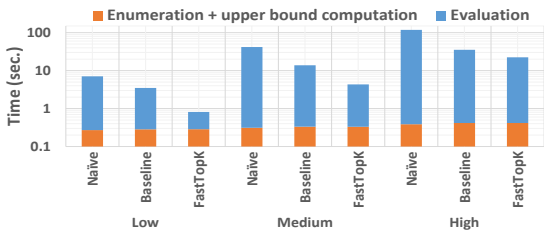


Figure 6: Comparison of FASTTOPK **with** NAIVE **and** BASELINE

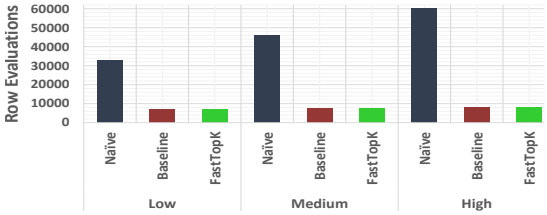


Figure 7: Amount of queries evaluated by NAIVE **(without using upper bound** $\overline{\text{score}}$**),** BASELINE **(using** $\overline{\text{score}}$**), and** FASTTOPK **(using** $\overline{\text{score}}$**)**

Hence, the number of PJ query-row evaluations and the execution time of both algorithms increase in α. FASTTOPK outperforms the BASELINE *by a factor of 3.5X* for all values of α. While they evaluate almost the same number of PJ queries, FASTTOPK performs better due to caching shared outputs of common sub-PJ queries.

Exp-IV: Vary k**.** With increase in k, both approaches evaluate more PJ queries before they terminate, and thus need more execution time. They perform almost the same number of PJ Query-row evaluations. But, due to shared evaluation, FASTTOPK outperforms BASELINE *by a factor of 3-4X* for k's as in Figure 9(b).

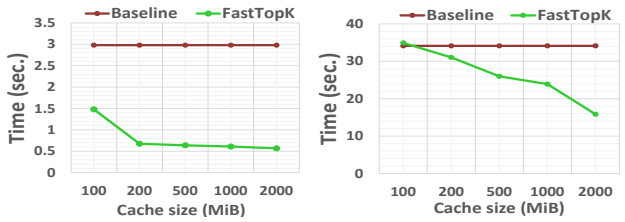
Exp-V: Vary number of relationship errors. We generate different sets of ESs with different numbers of relationship errors (varying from 0 to 5). Higher the number of errors, lower the final scores of the top-k PJ queries. A lower kth highest score delays the satisfaction of the termination condition. So the number of PJ query-row evaluations increase significantly with the number of errors. Overall, FASTTOPK outperforms BASELINE *by a factor of 2-6X*.

Exp-IV: Vary ϵ. We find that FASTTOPK is robust to ϵ. There is negligible change in execution time as we vary ϵ from 0.2 to 2.0, so we omit the plot in the paper. One would expect the performance to suffer when ϵ is high (say, 2.0) since FASTTOPK would evaluate many PJ queries outside the minimum evaluation set. However, due to both cache-evaluation scheduling and the skipping condition, it does not incur much more cost of evaluating such PJ queries.

Exp-IIV: Scale up dimension tables and fact tables. In this experiment, we start with the original ADVW database and scale it up to create databases with different statistical properties.

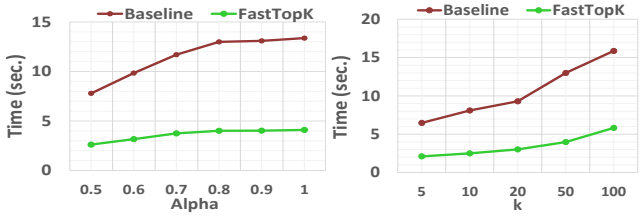
First, we scale up the dimension tables by creating new rows containing the same values as existing rows (but different row identifiers). We do not modify the fact tables, i.e., the new rows are not referenced by any fact rows. Figure 10(a) shows that the average execution time (over 450 ESs) of FASTTOPK increases slowly as we increase the scale factor (# new rows created for each existing row) from 1 to 2000. This is due to increase in cost of retrieving rows from the row-level inverted index. There is no increase in join cost (hash lookups) as the fact table is unchanged.

Next, we scale up the fact tables by creating new fact rows that reference the same dimension table rows as existing fact rows. We do not modify the dimension tables. This is to test the case that relations with a huge number of tuples and relatively few unique values in certain columns. Figure 10(b) shows the average execution time of FASTTOPK as we increase the scale factor (# new fact rows created for each existing fact row) from 1 to 50. The execution time



(a) Execution time for "low" bucket (b) Execution time for "high" bucket

Figure 8: Varying cache size B for ESs in "low"/"high" bucket



(a) Varying score weight α (b) Varying k

Figure 9: Varying α and k for ESs in "medium" bucket

increases at a much faster rate (superlinear) compared with the first case. This is due to increase in join cost (hash lookups), although the inverted index retrieval cost does not change. This also shows that the join cost dominates the overall cost of query processing.

Finally, if we scale up the dimension tables and the fact tables simultaneously, we have a combined effect of both on the performance. For the space limit, we do not include those results here.

6.3 User Study

We have conducted a user study in IMDB to evaluate the effectiveness of our scoring model. We use the IMDB database for this study as our judges are more familiar with the movie domain compared with CSUPP or ADVW. For the fairness of evaluation, we generate ESs from a source different from IMDB. We use HTML tables extracted from the web [8]. We select HTML tables about movies by creating a list of movies and checking for overlap of the subject column of the table with that list. We generate 52 ESs from randomly selected rows and columns of randomly selected movie HTML tables. For each ES, we compute the top-10 PJ queries and present it to the three judges via a web-based user interface.

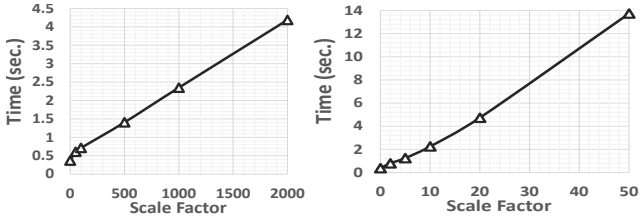
We have all the human judges get familiar with and agree on the organization of the IMDB database, and give them access to each original HTML table, so that they can mark each PJ query as relevant or non-relevant to an ES. Different subsets of ESs are assigned to different judges. On average, judges marked 2.3 results as relevant per ES. The overall mean reciprocal rank (MRR)[1] is 0.79; it shows that the relevant result(s) typically appear at the top.

To study the effectiveness for varying characteristics of the ES, we divide the 52 ESs into 3 buckets high (highly frequent terms), medium, and low, based on posting list sizes of the terms in the ESs. The MRRs for the high, medium and low buckets are 0.87, 0.78 and 0.71, respectively. The MRR is quite stable across all the buckets; the MRR values are slightly lower for the low and medium buckets due to presence of a few ESs containing foreign language movies (which are not well-covered by IMDB) in these buckets.

7. RELATED WORK

Our work is most related to query discovery using example tuples [25, 18, 27, 22]. These works can be divided into two categories: those that discover queries whose output is exactly the user-provided table [25, 27] and those that discover queries whose output contains the user-provided example tuples [18, 22]. None of the above works allow approximate containment and top-k ranking based on the degree of containment. This gives rise to unique challenges (exploiting common subexpressions while terminating early) which does not arise in these previous works.

[1]Defined as $\frac{1}{\text{number of inputs}} \sum_{\text{each input}} \frac{1}{\text{rank of the first right answer}}$



(a) Scaling up dimension tables (b) Scaling up fact tables

Figure 10: Varying scale factor in ADVW.

Our work is also related to the vast body of work on keyword search in databases [5, 12, 6, 15, 14]. Some of these works perform top-k ranking and early termination [12, 15]; some of them return answers incrementally [14]. However, the input and output is quite different: they take keyword search queries as input and return interconnected tuple structures (i.e., tuples connected via foreign key references) that contain some or all the keywords. Since the ranking objects are different, our problem requires a new scoring model, and thus new upper bounds of scores and early termination algorithms need to be invented. Specifically, none of those approaches exploit common subexpressions while still terminating early.

Top-k ranking and early termination techniques have been an area of intense research [20, 10]. These techniques compute a lightweight upper bound score for each object, and evaluate them (i.e., compute the exact score) in the descending order of upper bound scores to terminate early. They do not consider sharing computation among the evaluation of different objects.

While sharing some similarity with the line of works on multiple-query optimization (e.g., [19, 21] for traditional databases, and, recently, [9, 26] in MapReduce environment), our problem needs to be tackled with significantly different techniques. There are two major differences. First, multiple-query optimization problem is to evaluate all queries/jobs in a given set, while our problem is to find the top-k – one aspect of our objective is to evaluate as few queries as possible. Secondly, multi-query optimization techniques consider optimize only for cost and have no budget constraints. As we have only a budgeted amount of memory and the number of possible PJ queries is large, our system needs to determine when to insert output relations into the cache and when remove/replace them with new ones wisely. While our PJ queries have a more restrictive form (i.e., with only foreign-key join tree and projection), our caching scheme is much more flexible – this leads to a different technical problem (caching-evaluation scheduling).

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed and studied the problem of discovering top-k project join queries which approximately contain a user-given set of example tuples in their outputs. The main technical challenge is to share results of common subexpressions among the PJ queries and still terminate early. We formalize the problem as the caching-evaluation scheduling problem, show its hardness, and develop a near-optimal solution. Our experiments demonstrate that our solution is both efficient and effective in finding the top-k.

Our ranker captures some classes of errors; extending it to other types of errors (e.g., spelling errors and fuzzy matching) and evaluating its quality on real enterprise users are open challenges.

9. REFERENCES

[1] https://docs.google.com/spreadsheets/.
[2] https://msftdbprodsamples.codeplex.com/releases.
[3] https://www.office.com/start/default.aspx.
[4] http://www.imdb.com/interfaces.
[5] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.
[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.
[7] J. Brutlag. Speed matters for google web search (2009): http://services.google.com/fh/files/blogs/google_delayexp.pdf.
[8] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. Wu. Uncovering the relational web. In *WebDB*, 2008.
[9] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.
[10] R. Fagin, A. Lotem, and N. Naor. Optimal Aggregation Algorithms for Middleware. *J. Comput. Syst. Sci.*, 66(4), 2002.
[11] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified np-complete problems. In *STOC*, 1974.
[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.
[13] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB*, 2002.
[14] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: A tastier approach. In *SIGMOD*, 2009.
[15] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.
[16] M. Mayer. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.
[17] E. Millar, D. Shen, J. Liu, and C. K. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *J. Digit. Inf.*, 1(5), 2000.
[18] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.
[19] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
[20] T. Seidl and H. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.
[21] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
[22] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.
[23] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
[24] T. Smith, W. Johnson, R. Tamm-Daniels, and S. Probstein. Querying joined data within a search engine index. *US Patent No. 8073840*, 2011.
[25] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.
[26] G. Wang and C. Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013.
[27] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

APPENDIX

A. EXTENSION AND DISCUSSION

A.1 Incremental Computation

Let $T' = T^{\text{old}} \cup T^{\text{new}}$, where T^{old} is the set of unchanged rows that are identical to the ones in T, and T^{new} is the set of new rows or updated rows in T'. Recall that $\mathcal{Q}_{\text{E}} \subseteq \mathcal{Q}_{\text{C}}$ is the set of PJ queries that have been evaluated for T. For the new set of queries, $\mathcal{Q}_{\text{C}}{}'$, generated by the PJ query enumerator for T', can be partitioned into $\mathcal{Q}_{\text{C}}{}' = \mathcal{Q}_{\text{C}}{}^{\text{old}} \cup \mathcal{Q}_{\text{C}}{}^{\text{new}}$, where $\mathcal{Q}_{\text{C}}{}^{\text{old}} = \mathcal{Q}_{\text{C}}{}' \cap \mathcal{Q}_{\text{E}}$ and $\mathcal{Q}_{\text{C}}{}^{\text{new}} = \mathcal{Q}_{\text{C}}{}' - \mathcal{Q}_{\text{E}}$. For each query in $\mathcal{Q}_{\text{C}}{}^{\text{old}}$, since it has been evaluated for T, we keep and re-use its score w.r.t. rows in T^{old}; and for each query in $\mathcal{Q}_{\text{C}}{}^{\text{new}}$, since it is new or unevaluated before, we need to (re-)evaluate it for every row in T'. We will discuss how to utilize the "partial scores" of queries in $\mathcal{Q}_{\text{C}}{}^{\text{old}}$ to get better upper bounds of scores, and how to take the incremental changes into consideration when scheduling caching-evaluation.

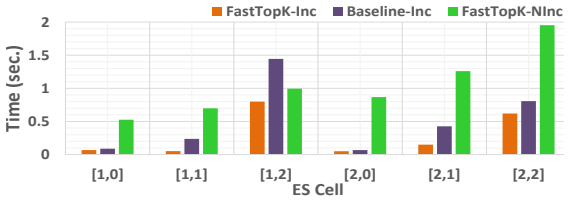


Figure 11: Execution time for incremental input [row, column]

Improved upper-bound scores for $\mathcal{Q}_{\text{C}}{}^{\text{old}}$. For a query $Q \in \mathcal{Q}_{\text{C}}{}^{\text{old}}$, a better upper bound of its score can be computed as a combination of its old score w.r.t. rows in T^{old} and column-wise score w.r.t. the rest part of T'. For any $0 \leq \alpha \leq 1$, we have

$$\begin{aligned}\mathsf{score}(T' \mid Q) &\leq \mathsf{score}(T^{\text{old}} \mid Q) + \frac{\mathsf{score}_{\text{col}}(T^{\text{new}} \mid Q)}{1 + \ln(1 + \ln|\mathcal{J}|)} \qquad (11)\\ &\leq \overline{\mathsf{score}}(T' \mid Q) \quad \text{(the one defined in Proposition 2).}\end{aligned}$$

Note that $\mathsf{score}(T^{\text{old}} \mid Q)$ in (11) is known, as Q has been evaluated for rows in T^{old}. $\mathsf{score}_{\text{col}}$ can be computed as in Section 4.1.2. Compared to $\overline{\mathsf{score}}$, RHS of (11) is a better upper bound of score, and thus implies a smaller minimal evaluation set $\mathcal{Q}'_{\text{min}}$ for T'. So our strategies terminate more quickly using this upper bound.

Incremental caching-evaluation scheduling. We only need to modify our cost model so that the incremental updates can be automatically considered in our strategy FASTTOPK (Algorithms 3-4). In our current cost model (12)-(13) in B.3, we need to evaluate each query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ for every row in T. The new cost model to handle updates will take into consideration the number of rows in T' we need to evaluate Q on. i) For a (sub-)PJ query $Q \in \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{old}}) - \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{new}})$, we only need to evaluate it for rows in T^{new}, so we define $\text{cost}'(Q) = |T^{\text{new}}| \cdot \text{cost}(Q)$ and $\text{cost}'(Q, \mathcal{M}) = |T^{\text{new}}| \cdot \text{cost}(Q, \mathcal{M})$; ii) for a (sub-)PJ query $Q \in \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{new}}) - \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{old}})$, we need to evaluate it for all rows in T', so we define $\text{cost}'(Q) = |T'| \cdot \text{cost}(Q)$ and $\text{cost}'(Q, \mathcal{M}) = |T'| \cdot \text{cost}(Q, \mathcal{M})$; and iii) for a (sub-)PJ query $Q \in \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{old}}) \cap \mathcal{T}(\mathcal{Q}_{\text{C}}{}^{\text{new}})$, we create two copies of it – the one belonging to queries in $\mathcal{Q}_{\text{C}}{}^{\text{old}}$ is associated with cost as i) and the one belong to queries in $\mathcal{Q}_{\text{C}}{}^{\text{new}}$ is associated with cost as ii). Such a weighted cost model is applied in FASTTOPK to get updated top-k PJ queries.

Experimental Results

We generate complete 3×3 example spreadsheets in CSUPP as described in Section 6.1. We simulate incremental input by starting with the completely filled-out first row and then adding one cell at a time from the complete example spreadsheet (i.e., 6 cell additions). Our simulator adds cells row-by-row, from left to right. We average our results over the 50 example spreadsheets.

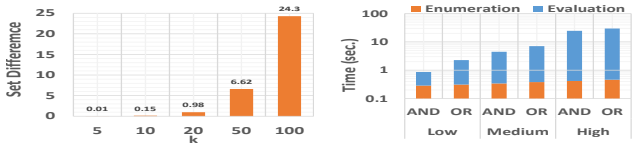
We evaluate three approaches for incremental input:

(i) FASTTOPK-NINC: always treating an example spreadsheet as a new one and applying FASTTOPK on it;

(ii) FASTTOPK-INC: described in Section 5.4 and above; and

(iii) BASELINE-INC: extending BASELINE using the same ideas as FASTTOPK-INC without caching-evaluation scheduling

Refer to Section 6.1 for the settings of experiments. Figure 11 shows the execution times of the three algorithms for the 6 cell additions via row-wise typing. FASTTOPK-INC significantly outperforms both BASELINE-INC and FASTTOPK-NINC. FASTTOPK-



(a) Difference in result sets (b) Execution time

Figure 12: AND-column mapping v.s. OR-column mapping

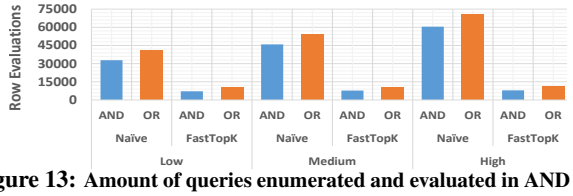


Figure 13: Amount of queries enumerated and evaluated in AND and OR semantics by NAIVE (not using $\overline{\text{score}}$) and FASTTOPK (using $\overline{\text{score}}$)

NINC performs poorly since it does not use previously computed scores for the unchanged example tuples. This is especially true for the first few cells in a new row (e.g., [2,0], [2,1]). In these cases, the incremental approaches evaluate the PJ queries only for the changed example tuple (only a few terms) while FASTTOPK-NINC evaluates them for both changed and unchanged example tuples. BASELINE-INC suffers as it does not share results of sub-PJ queries as in the non-incremental case.

A.2 Generalizing Cell Similarity

We can extend cell similarity to perform IR-style relevance ranking as in [23]. For example, we can define cell similarity to be higher if there is an exact match between an example tuple cell and output row cell (as opposed to the latter only containing some terms of the former). And we can incorporate terms weights (based on inverse document frequency and document length) in cell similarity.

We can also extend cell similarity to handle spelling errors, synonyms, and fuzzy matching. For example, to handle fuzzy matching, we can built the inverted indexes (both column-level and row-level) on n-grams as terms instead of words [17]. When processing an example spreadsheet, we split each cell into n-grams (instead of words) and retrieve the inverted indexes corresponding to those n-grams. For spelling errors, we can simply replace a term in the example spreadsheet with a list of similar terms (within certain edit distance), look up them in our inverted indexes, and take the union of posting lists. Synonyms can be handled similarly.

A.3 AND v.s. OR Semantics

Our definition of PJ queries (Definition 2) requires that every column in the example spreadsheet is mapped to some column in the database. We can relax this constraint by allowing that only a subset of the example spreadsheet columns is mapped to the set of database columns, i.e., simply ignoring some example spreadsheet columns. Consider a PJ query $Q = (\mathcal{J}, \mathcal{C}, \phi)$, this relaxation changes our mapping from "$\phi : \text{col}(T) \rightarrow \mathcal{C}$" to "$\phi : \text{col}(T) \rightarrow \mathcal{C} \cup \{\perp\}$". When a column i in the example spreadsheet T is mapped to $\perp$, this column does not correspond to any column in the output relation of Q. We call our old column mapping, *AND-column mapping*, and the new relaxed one, *OR-column mapping*.

Our approaches can be extended to handle OR-column mapping easily. A simple extension is as follows. For a user-given example spreadsheet T with c columns, our system can create 2^c example spreadsheets $T_0, T_1, \ldots, T_{2^c-1}$, each of which consists of a subset of columns of T. We then process them using our FASTTOPK strategy one by one. The 2^c top-k resulting lists are aggregated to generate the overall top-k. Since c is small in practice, the cost of this approach is affordable. A more direct way is to enumerate all PJ queries under this OR semantics. To this end, we can apply the *Candidate Network Generator* in [12] to generate an extended set of candidate PJ queries in $\mathcal{Q}_C{}^+$, each of which has a subset of example spreadsheet columns mapped to its projection. Then both of our strategies BASELINE and FASTTOPK still work on $\mathcal{Q}_C{}^+$.

Experimental Results

Refer to Section 6.1 for the settings of experiments. We use CSUPP dataset and corresponding example spreadsheets to compare FASTTOPK with AND-column mapping (the one described in the main body of this paper) with the extended version of FASTTOPK with OR-column mapping (the simple extension described above).

Figure 12(a) shows the average set difference between the result sets with AND and OR-column mapping for the 50 example spreadsheets for various values of k. For smaller values of k, there is almost no difference between the two result sets. For example, the top 10 results are identical for 49 out of the 50 ESs. It means that, even when we allow OR-column mapping in PJ queries, the top ones are likely to have all the columns in example spreadsheets mapped to their projections (AND semantics).

Figure 12(b) shows the average execution times of the two approaches. Note that "enumeration" here means "query enumeration + upper-bound score computation". The OR-semantics implementation is only a bit slower than the AND-semantics one since the execution time for the T_i created from the biggest subset of columns in T (i.e., the entire example spreadsheet) dominates the execution time. This shows that our system can be easily adapted to support OR-semantics whenever necessary, e.g., when the system returns an empty result for some user-specified example spreadsheet.

Figure 13 plots the number of PJ queries enumerated and evaluated in AND and OR semantics. NAIVE evaluates all the PJ queries enumerated in both semantics. We can find that, in the AND semantics, less PJ queries are enumerated than in OR because of stronger constraints in the column mapping ϕ. Using upper bounds $\overline{\text{score}}$, the number of PJ queries actually evaluated by FASTTOPK is much less than the total number of queries enumerated.

B. COMPUTING EXACT SCORES

B.1 Execution Plan for PJ Queries

We need to execute the PJ query Q and, for each tuple t in the example spreadsheet, examine every row in the output relation $\mathcal{A}(Q)$ to compute the terms $\max_{r \in \mathcal{A}(Q)} \text{score}(t \mid r)$ in (3). We utilize the *(key, foreign key)-snapshot* of the database (discussed in Section 3.1), and select a pre-optimized plan to execute Q in memory. Our execution plan for Q borrows ideas from hash joins:

Stage I *(scanning row-level inverted indexes to score cells)*: For each term w in each cell $t[i] \in T$, suppose column i is mapped to column j in a relation R in the database $\mathcal{D}$ through ϕ, we retrieve the row-level inverted index $\text{inv}(w, R[j])$ to compute cell similarities $\text{score}_{\text{cell}}(t[i] \mid r[j])$ (as lines 1-5 of Algorithm 1) for rows $r \in R$. $\text{score}_{\text{cell}}$ is then associated to primary keys of rows in R.

Stage II *(bottom-up hash joins)*: Starting from the leaf relations in $\mathcal{J}$, the primary key of each row, associated with cell similarities, is inserted into a hash table if cell similarity is non-zero in at least one cell – after that, a leaf relation is called *evaluated*.

Recursively, **Stage II-A *(scan/hash lookup)*:** for each relation above, if all of its children relations have been *evaluated*, we can start to *scan* its rows in the in-memory (key, foreign key)-snapshot of this relation, and for each row, *look up* all foreign keys (in different columns) in the corresponding hash tables popped up from the children relations to conduct the foreign-key joins. **Stage II-B *(building hash table)*:** Then, the primary key of each row in the

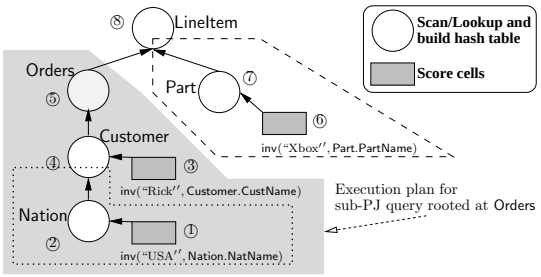


Figure 14: Execution plan for the PJ query in Figure 2(b)-(i) and its sub-PJ queries (operators executed in the order of 1, 3, 6, 2, 4, 5, 7, 8)

join output with nonzero cell similarities (in at least one cell) is put into a hash table. After that this relation is called *evaluated*.

Stage III *(computing scores)*: After the root relation is evaluated, we get the output relation $\mathcal{A}(Q)$, with cell similarities associated in each row of $\mathcal{A}(Q)$, and then we can compute the row containment score as in (1)-(3). Note at at each relation, we only need to keep primary/foreign key columns and columns in the projection $\mathcal{C}$. □

The above evaluation plan can be executed either for Q with one row of T, or with all rows of T together.

Figure 14 shows the execution plan for the PJ query in Figure 2(b)-(i) on the first row of the example table in Figure 2(a). A rectangle node represents the operation to retrieve row-level inverted indexes and compute cell similarities in Stage-I. A circle node represents the operations (scanning, hash lookups, and building hash table) we perform on a relation (labeled beside) in Stage-II, after all of its children are evaluated. For example, on the Orders node, we lookup foreign key Orders.CustId of each row in the hash table built by the node Custmer, and then build a hash table with Orders.OId (key in the hash table) and cell similarities on column Customer.CustName and column Nation.NatName. On the root node LineItem, we need to look up two foreign keys PartId and OId in the hash tables popped up by its two children Part and Orders, respectively. Operations are performed in the order of ①, ③, ⑥, ②, ④, ⑤, ⑦, ⑧ to compute the final score.

B.2 Speedup Execution using Cache

The execution plans of PJ-queries can be easily extended to take advantage the cached sub-PJ queries: for a PJ query Q and a set of cached sub-PJ queries in $\mathcal{M}$, instead of starting from the leaves of Q, we start from the output relations of maximal sub-PJ queries of Q in $\mathcal{M}$ and follow the execution plan of Q afterwards.

For example, Figure 3 shows two sub-PJ queries Q'_1 (left) and Q'_2 (right) of the PJ query Q in Figure 2(b)-(i). Intuitively, the execution plan of a sub-PJ query $Q' \preceq Q$ is a subtree of the execution plan of Q. For example, the dotted polygon and the dashed polygon in Figure 14 are the execution plans of Q'_1 and Q'_2, respectively.

In the execution plan of Q in Figure 14, if both Q'_1 and Q'_2 are materialized in $\mathcal{M}$, we can start from their output relations in $\mathcal{M}$ and execute only the operations ③, ④, ⑤, and ⑧ (only the join with Part is needed in ⑧). If a even larger one Q'_3 (rooted at Orders), whose plan is the shaded polygon in Figure 14, is cached, we can start from the output relations of Q'_1 and Q'_3 (Q'_2 is no more a maximal one) and execute only the operation ⑧.

B.3 Cost Model for Computing Exact Scores

For the PJ-query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ w.r.t. T, there are three major operators involved in our execution plan in B.1: i) retrieving row-level inverted index; ii) scanning a relation while doing hash lookups; and iii) building a hash table. The running time of each of these operators is constant. So a natural and light-weight cost model of the execution plan for Q is to count the number of operations ii) and iii) executed on tuples in the relations of $\mathcal{J}$ and the number of tuples retrieved from inverted indexes. For a (sub-)PJ query Q, define the cost of evaluating Q as:

$$\text{cost}(Q) = \sum_{R \in \mathcal{V}(\mathcal{J})} |R| \cdot d_{\mathcal{J}}(R) + \sum_{i \in \text{col}(T)} \sum_{\mathsf{w} \in T[i]} |\mathsf{inv}(\mathsf{w}, \mathcal{J}[\phi(i)])|. \quad (12)$$

The first component on the RHS of (12) quantifies the total number of hash lookups/inserts: $d_{\mathcal{J}}(R)$ is the degree of relation R in $\mathcal{J}$, as for each tuple in R, the number of hash lookups is equal to the number of children of R in $\mathcal{J}$, and for every relation except the root relation in R, we need to build a hash table by inserting tuples in this relation. The second component quantifies the number of tuples we need to retrieve from row-level inverted indexes: here let $\mathcal{J}[\phi(i)]$ be the column which i is mapped to in a relation of $\mathcal{J}$.

More generally, we have a set of sub-PJ queries cached in $\mathcal{M}$. The cost of the execution plan for Q when we reuse output relations of sub-PJ queries in $\mathcal{M}$ is defined to be:

$$\text{cost}(Q, \mathcal{M}) = \text{cost}(Q) - \sum_{\substack{\text{maximal } Q' \in \mathcal{M} \\ Q' \preceq Q}} \text{cost}(Q'), \quad (13)$$

as the output relations of maximal sub-PJ queries Q' of Q in $\mathcal{M}$ can be directly retrieved from $\mathcal{M}$ and reused.

Both $\text{cost}(Q)$ and $\text{cost}(Q, \mathcal{M})$ can be computed efficiently. In (12)-(13), $|R|$, the number of tuples in R, and $|\mathsf{inv}(\mathsf{w}, \mathcal{J}[\phi(i)])|$, the length of a row-level inverted index, can be gotten in constant time. So the total time is $O(\mathcal{V}(\mathcal{J}) + \#\text{ terms in } T)$.

C. PROOFS

Proof of Proposition 1. For the first part (property i)), since no column in T is mapped to R, by excluding R from Q, the column containment score is unchanged, i.e., $\mathsf{score}_{\text{col}}(T \mid Q) = \mathsf{score}_{\text{col}}(T \mid Q')$. So it suffices to prove $\mathsf{score}_{\text{row}}(T \mid Q) \leq \mathsf{score}_{\text{row}}(T \mid Q')$. Consdier the output relations $\mathcal{A}(Q)$ and $\mathcal{A}(Q')$, for any $t \in \mathcal{A}(Q)$, we have $t \in \mathcal{A}(Q')$, because Q' has less key-foreign key constraints in joins. So from (1), we have, for each $t \in T$, $\mathsf{score}(t \mid Q) \leq \mathsf{score}(t \mid Q')$. Then the conclusion follows from (3)

For the second part (property ii)), $\mathsf{score}_{\text{col}}(T \mid Q) = \mathsf{score}_{\text{col}}(T' \mid Q'')$ is also obvious as no term in the removed column $T[i]$ appears in the removed $R[j]$. It suffices to prove $\mathsf{score}_{\text{row}}(T \mid Q) = \mathsf{score}_{\text{row}}(T' \mid Q'')$. Comparing Q with Q'', their join trees are the same and the projection in Q'' is a subset of the projection in Q. So the output relation $\mathcal{A}(Q'')$ is essentially the projection of $\mathcal{A}(Q)$ on $\mathcal{C}'$. And since the column $R[j]$ excluded in Q'' contains no term in the spreadsheet column $T[i]$, the above claim follows. □

Proof of Proposition 2. From (5), it suffices to show $\mathsf{score}_{\text{row}}(T \mid Q) \leq \mathsf{score}_{\text{col}}(T \mid Q)$. Putting (2) into (3), and comparing it with (4), we can derive this relationship. □

Proof of Proposition 3. For each term w in a column i of T, if $T[i]$ is mapped to $R[j]$, we need to scan the row-level inverted index $\mathsf{inv}(\mathsf{w}, R[j])$ – the term in (4), $\mathsf{score}_{\text{cell}}(t[i] \mid r[\phi(i)])$, is obtained by aggregating the results for different terms. The total cost is dominated by $O(\sum_{\mathsf{w} \in T} l_{\mathsf{w}})$, i.e., lengths of inverted indexes. □

Proof of Proposition 4. Again, the row-level inverted indexes need to be scanned to compute the terms $\mathsf{score}_{\text{cell}}(t[i] \mid r[\phi(i)])$ in (2). To compute the terms $\max_{r \in \mathcal{A}(Q)} \mathsf{score}(t \mid r)$ in (3), we need to scan the output relation $\mathcal{A}(Q)$ at least once. The complexity of generating $\mathcal{A}(Q)$ using the hash-join execution plan introduced in Appendix B.1 is $O(\sum_{R \in \mathcal{J}} |R| \cdot d_{\mathcal{J}}(R))$. □

Proof of Proposition 5. Let's first define the class of algorithms, called *multi-step ranking* algorithms. A multi-step ranking algorithm takes i) a set of PJ queries $\mathcal{Q}_{\text{C}}$, and ii) upper bounds $\overline{\mathsf{score}}$

of their scores, as input. In each step, it picks one or more PJ queries in $\mathcal{Q}_{\rm C}$ with unknown scores and *evaluates* them, i.e., computes $\mathsf{score}(Q)$; based on the known scores, it continues to pick the next one or more PJ queries to evaluate, until the *top-k* of known scores is larger than the *max* of upper-bound scores of queries with unknown scores. The following proof follows from [20].

Recall $\mathcal{Q}_{\min} = \{Q_1, Q_2, \ldots, Q_{i^*}\} \subseteq \mathcal{Q}_{\rm C}$, and i^* is the minimal i s.t. $\text{top}_k\{\mathsf{score}(Q_1), \ldots, \mathsf{score}(Q_i)\} > \overline{\mathsf{score}}(Q_{i+1}) \geq \overline{\mathsf{score}}(Q_{i+2}) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$. We prove this proposition via contradiction. Consider any multi-step ranking algorithm that evalutes a set of PJ queries $\mathcal{Q}'$ and claims that all queries with the top-k scores in $\mathcal{Q}_{\rm C}$ have been found in $\mathcal{Q}'$. Pick any $Q_p \in \mathcal{Q}_{\min} - \mathcal{Q}'$. Let Q_q be the PJ query in $\mathcal{Q}'$ with the kth highest score, i.e., $\mathsf{score}(Q_q) = \text{top}_k\{\mathsf{score}(Q) \mid Q \in \mathcal{Q}'\}$.

i) If $\overline{\mathsf{score}}(Q_p) \geq \mathsf{score}(Q_q)$: Since the algorithm has not evaluated Q_p, the adversary can set $\mathsf{score}(Q_p) = \overline{\mathsf{score}}(Q_p)$. Then $\mathsf{score}(Q_p) \geq \mathsf{score}(Q_q) = \text{top}_k\{\mathsf{score}(Q) \mid Q \in \mathcal{Q}'\}$, so Q_p is missed from the top-k and the output of the algorithm is incorrect.

ii) If $\overline{\mathsf{score}}(Q_p) < \mathsf{score}(Q_q)$: Consider the set of top-k PJ queries in Q', $\mathcal{Q}'_{\text{topk}} = \{Q \mid \mathsf{score}(Q) \geq \mathsf{score}(Q_q)\} \cap \mathcal{Q}'$. We have $\mathcal{Q}'_{\text{topk}} \subseteq \{Q_1, Q_2, \ldots, Q_{p-1}\}$, because for any $Q \in \mathcal{Q}'_{\text{topk}}$, we have $\overline{\mathsf{score}}(Q) > \overline{\mathsf{score}}(Q_p)$. Then it follows that $\text{top}_k\{\mathsf{score}(Q_1), \ldots, \mathsf{score}(Q_{p-1})\} > \overline{\mathsf{score}}(Q_p)$. Note that $p \leq i^*$, and thus it contradicts with the minimality of i^*.

Both i) and ii) lead to contradiction, so we have $\mathcal{Q}_{\min} \subseteq \mathcal{Q}'$. □

Proof of Theorem 1. To prove the correctness, we only need to show that if (7) is satisfied, the top-k are among $Q_1, Q_2, \ldots, Q_i$. This is true, because, from the way how Q_i's are ordered in (7) and BASELINE, for any $j > i$, we have $\mathsf{score}(Q_j) \leq \overline{\mathsf{score}}(Q_{i+1})$.

The second part, it evaluates only queries in $\mathcal{Q}_{\min}$, is trivial. □

Proof of Theorem 2. Given a sequence of type-a,b,c operators, to check whether it is a feasible solution, we only need to check when it eventually evaluates all PJ queries in $\mathcal{Q}_{\min}$ and, at any time, the cache size it uses is no more than B. So when $\mathcal{Q}_{\min}$ is known, the problem is in NP. We use a reduction from the HAMILTONIAN PATH problem to show CACHE-EVAL SCHEDULER is NP-hard.

Consider a HAMILTONIAN PATH instance: given a undirected graph $G(V, E)$, whether there exist an ordering of all vertices v_1, $v_2, \ldots, v_n \in V$ ($|V| = n$) s.t. $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, n-1$. This problem is NP-complete even in a restricted class of graphs, where every vertex has degree equal to three [11]. Now let's construct an instance of our CACHE-EVAL SCHEDULER problem. For each vertex $v \in V$, create a PJ query Q_v in $\mathcal{Q}_{\min}$. For each edge $e = (v, u)$, create a sub-PJ query Q_e^{sub} and let Q_e^{sub} be a sub-PJ query of both Q_v and Q_u. So for each created PJ query Q_v, we have the set of all sub-PJ queries of Q_v to be $\mathcal{T}(Q_v) = \{Q_e^{\text{sub}} \mid e \text{ is incident on } v\}$. Finally, let $|\mathcal{A}(Q_e^{\text{sub}})| = B$ for every $e \in E$, i.e., at any time, we can only keep the output relation of one sub-PJ query in our cache; and let $\text{cost}(Q_e^{\text{sub}}) = C_1$ be equal for every e and $\text{cost}(Q_v) = C_2$ be equal for every $v \in V$. To prove the NP-hardness, it suffices to show such a claim: there is a Hamiltonian path in G if and only if there exists a sequence of operators to evaluate $\mathcal{Q}_{\min}$ with cost no more than $n \cdot C_2 - (n-1) \cdot C_1$.

To prove the claim, we need to transform a path into a sequence of operators and vice versa. A subpath $v_{i-1}v_iv_{i+1}$ in the Hamiltonian path corresponds to: when evaluting $Q_{v_{i-1}}$, we put $Q^{\text{sub}}_{(v_{i-1}, v_i)}$ in cache and reuse it to evaluate Q_{v_i}; and after that, we clear the cache and put $Q^{\text{sub}}_{(v_i, v_{i+1})}$ in cache. Details are omitted here. □

Proof of Theorem 3. The correctness follows directly from Theorem 1. The set of PJ queries FASTTOPK evaluate is always a superset of those evaluated by BASELINE (i.e., all queries in $\mathcal{Q}_{\min}$ have been evaluated when FASTTOPK terminates). □

Proof of Theorem 4. The second part, $M_{\text{all}} = \Theta(s_{\max} \cdot N)$, is directly from our definition of sub-PJ queries.

For the overall time complexity, we focus on Algorithm 4 first. For each sub-PJ query Q', we need $O(s_{\max})$ time to get $\text{cost}(Q')$ and $|\mathcal{A}(Q')|$, which are used in lines 1 and 4. So we need a total of $O(M \cdot s_{\max})$ time for all sub-PJ queries. Sorting all sub-PJ queries in line 1 needs $O(M \cdot \log M)$ time. The remaining question is how to get Q^* and $\text{Critical}^{-1}(Q^*)$ efficiently (lines 4, 6, and 8). Note that $\mathcal{B}_j$ is the set of unevaluated PJ queries. For each sub-PJ query Q', we keep a hash set $HS(Q')$ of *unevaluated* PJ queries it belongs to. For each PJ query Q, we also keep a list $LT(Q)$ of sub-PJ queries it contains. So after a PJ-queriy Q is evaluated, each $HS(Q')$ can be updated in constant time if Q' is a sub-PJ query of Q. To get Q^* in all iterations, we only need to scan all sub-PJ quries $Q'_1, \ldots, Q'_M$ in order and check whether $|HS(Q')| > 1$ for each. $\text{Critical}^{-1}(Q^*)$ can be directed retrived from $HS(Q^*)$. So the time complexity of Algorithm 4 is $O(M(s_{\max} + \log M))$.

In Algorithm 3, the additional time besides invoking Algorithm 4 is at most $O(N \cdot (\log N + \log k))$ (soring plus keeping the top-k).

So from $x \log x + y \log y \leq (x + y) \log(x + y)$, we have the overall time complexity is $O(N + M_{\text{all}}(s_{\max} + \log M_{\text{all}}))$. □

Proof of Theorem 5. The first part (9) is directly from the definition of $\mathcal{Q}_{\min}$ and the way we construct batches in Algorithm 3.

To prove the main result (10), we first prove, for each batch $\mathcal{B}$

$$\text{cost}_{\text{TOT}}(\mathcal{B}) - \text{cost}_{\text{SOL}}(\mathcal{B}) \geq \frac{1}{2c^2}\left(\text{cost}_{\text{TOT}}(\mathcal{B}) - \text{cost}_{\text{OPT}}(\mathcal{B})\right). \tag{14}$$

That is, the cost saved by FASTTOPK is no less than $1/2c^2$ of the optimal save. Summing up (14) for all batches $\mathcal{B} = \mathcal{B}_0, \mathcal{B}_1, \ldots$, since they are disjoint and $\cup\mathcal{B}_j = \mathcal{Q}_{\rm E}$, we have

$$\begin{aligned} &\text{cost}_{\text{TOT}}(\mathcal{Q}_{\rm E}) - \Sigma_{\mathcal{B}_j}\text{cost}_{\text{SOL}}(\mathcal{B}_j) \\ \geq &\frac{1}{2c^2}\left(\text{cost}_{\text{TOT}}(\mathcal{Q}_{\rm E}) - \Sigma_{\mathcal{B}_j}\text{cost}_{\text{OPT}}(\mathcal{B}_j)\right). \end{aligned} \tag{15}$$

FASTTOPK works batch-by-batch, so we have i) $\sum_{\mathcal{B}_j} \text{cost}_{\text{SOL}}(\mathcal{B}_j) = \text{cost}_{\text{SOL}}(\mathcal{Q}_{\rm E})$. Since $\mathcal{B}_j \subseteq \mathcal{Q}_{\rm E}$, we have ii) $\text{cost}_{\text{OPT}}(\mathcal{B}_j) \leq \text{cost}_{\text{OPT}}(\mathcal{Q}_{\rm E})$. And it is obvious that iii) the total number of batches processed by Algorithm 3 is at most $\log_{1+\epsilon}(|\mathcal{Q}_{\min}|/k)$. Our performance ratio (10) follows from (15) and i)-iii).

The only missing part is the proof for (14). Let the batch of PJ queries $\mathcal{B} = \{Q_1, \ldots, Q_n\}$. Suppose in the optimal solution,

$$\text{cost}_{\text{TOT}}(\mathcal{B}) - \text{cost}_{\text{OPT}}(\mathcal{B}) \leq \sum_{Q_i \in B} \text{cost}(Q_i) - \text{cost}(Q_i, \mathcal{M}_i^*), \tag{16}$$

where $\mathcal{M}_i^*$ is the status of cache before Q_i is evaluated. From (13),

$$\text{cost}(Q_i) - \text{cost}(Q_i, \mathcal{M}_i^*) = \sum_{\substack{\text{maximal } Q' \in \mathcal{M}_i^* \\ Q' \preceq Q_i}} \text{cost}(Q'). \tag{17}$$

On the other hand, in Algorithm 4, when Q_i is to be evaluated, we have cache $\mathcal{M}_i = \{Q^*\}$ (lines 8-9), where Q^* is the most costly sub-PJ query among those in $\{\text{maximal } Q' \in \mathcal{M}_i^* \mid Q' \preceq Q_i\}$ based on its selection (line 4). There are at most c maximal sub-PJ queries for Q_i (as there are at most c leaves in the join tree), so

$$\text{cost}(Q_i) - \text{cost}(Q_i, \mathcal{M}_i) \geq \frac{1}{c}\left(\text{cost}(Q_i) - \text{cost}(Q_i, \mathcal{M}_i^*)\right). \tag{18}$$

Among all PJ queries in $\mathcal{B}$, in the worst case, a fraction $\frac{1}{2c}$ of them can benefit as much as (18) from the cache. So putting (18) back to (16), we can obtain (14). The proof can be completed. □