Discovering Queries based on Example Tuples

Yanyan Shen[§*] Kaushik Chakrabarti[†] Surajit Chaudhuri[†] Bolin Ding[†] Lev Novik[‡]
[§]National University of Singapore [†]Microsoft Research [‡]Microsoft
[§]shenyanyan@comp.nus.edu.sg [†‡]{kaushik, surajitc, bolind, levn}@microsoft.com

ABSTRACT

An enterprise information worker is often aware of a few example tuples (but not the entire result) that should be present in the output of the query. We study the problem of discovering the minimal project join query that contains the given example tuples in its output. Efficient discovery of such queries is challenging. We propose novel algorithms to solve this problem. Our experiments on real-life datasets show that the proposed solution is significantly more efficient compared with naïve adaptations of known techniques.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query processing, Textual databases

Keywords

SQL query discovery, example tuples, project join query, filter selection, pruning

1. INTRODUCTION

Most real-life enterprise data warehouses have large and complex schemas [3, 11]. To pose queries, the user needs to understand the schema in detail and locate the schema elements of interest; this is a daunting task for most users. Consider a sales executive who needs to pose a query to obtain a set of sales tuples (say, containing which customers bought which products during a specific time period) so that she can create a report. This information might be split across multiple tables due to database normalization; locating those tables requires detailed understanding of the schema. Any aid we can provide in formulating the query or locating the relevant tables will immensely benefit such enterprise information workers.

An enterprise information worker is often aware of *a few example tuples* that should be present in the output of the query. They are typically not aware of all tuples in the output, otherwise they would not need to pose the query at all. The question we ask is: how can we use this to help the user formulate the query?

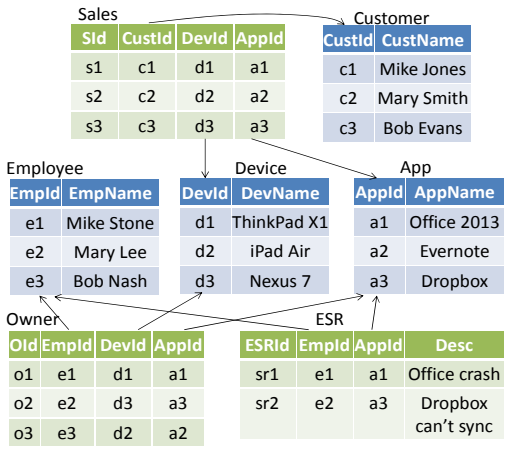


Figure 1: Example database

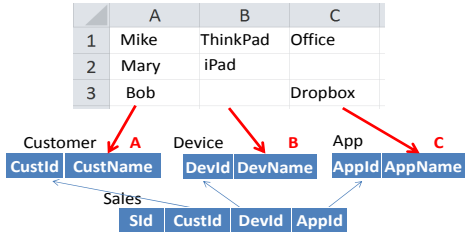


Figure 2: An example table and valid minimal query

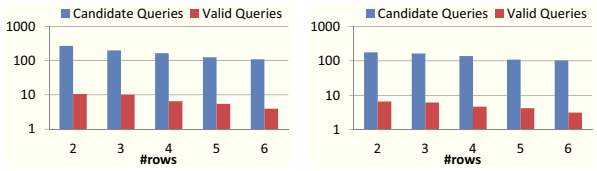
EXAMPLE 1. *Consider the simple sales and employee database at a computer retailer (e.g., Best Buy) shown in Figure 1. The database contains 3 fact tables (shown in green): 'Sales' where each row stores which customer bought which device with which applications (apps) installed, 'Owner' that stores which employee owns which device with which app installed and 'ESR' (Employee Service Requests) that stores which employee submitted service request for which app along with a textual description. There are 4 dimension tables: 'Customer', 'Device', 'App' and 'Employee' containing names of customers, devices, apps and employees respectively. Each dimension table has a primary key and the fact tables contain foreign key references to those primary keys (shown using directed edges).*

Consider the executive who needs to pose the query to obtain sales tuples. She obviously has the desired schema of the query result in her mind: say, it should contain the name of the customer, the name of device bought and the name of the app that was installed. She is also aware of a few tuples that should be present in the query result: say, she knows that a customer named 'Mike' (but does not know the full name) bought a 'Thinkpad' (does not know the full de-

[*]Work done while visiting Microsoft Research

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2593664.



(a) IMDB (b) CUST

Figure 3: #candidate queries vs #valid queries

vice name) with 'Office' installed, another customer named 'Mary' bought an 'iPad' (but not sure what app was installed on it) and another customer named 'Bob' bought a device with 'Dropbox' installed on it (but not sure which device). She provides the above information in the form of the "example table" (ET) shown in Figure 2; she can do so by typing it into a spreadsheet-style interface (e.g., Microsoft Excel, Google Spreadsheet). Note some cells are partially specified (e.g., customer, device, app names) and some are totally empty.

We propose to return a small set of queries so that the user can select one of them and easily modify it to obtain the final query. Each returned query should have the following desired characteristics:
(i) Its output should have the desired schema.
(ii) Its output should contain all the rows in ET.
(iii) Considering the class of SPJ queries where all joins are foreign key joins, there are numerous queries that satisfies (i) and (ii); we refer to them as "valid" queries. We should return the fewest valid queries in order to avoid overwhelming the user. At the same time, all the distinct join paths/projected columns should be covered. Hence, we do not consider selections, i.e., we consider only project join queries. Furthermore, we consider only the minimal project join queries, i.e., the ones where we cannot remove any relation or join condition so that it is still valid.

For the ET in Figure 2, there is only one valid minimal query as shown in the figure. The query is represented by the join tree. The red arrows show the projected column for each ET column. Each projected column is labeled by the name of corresponding ET column (A, B and C). Note that our problem is different from query by output which aims to find the query that produces the exact table [22, 23].

Technical Challenges: The key challenge is to compute the valid minimal queries *efficiently*. Existing approaches on keyword search in databases [1, 10, 2, 14, 9, 19, 4, 6, 15, 12] and sample driven schema mapping [18] focus on the *single input tuple* scenario. A naive adaptation to our problem is to first compute the set valid minimal queries for each row in ET and then intersect those sets.

Most valid minimal queries for the individual tuples are not valid for the overall ET; the latter set is *much more constrained*. The above adaptation does not push these constraints down. It executes join queries to compute valid queries for the individual tuples but subsequently throws most of them away as they are not valid for the overall ET. This is wasteful.

We identify a set of constraints that can be pushed below the joins. A valid minimal query must contain a projected column for each column C in ET; *all the values* in C must appear in it. We refer to it as the *column constraint* for C. A valid minimal query must satisfy the column constraints *for all the columns* in ET. We define the set of *candidate queries* as the set of minimal project join queries that satisfy all the column constraints. Figure 4 shows a few examples of candidate queries. We push the column constraints down by first computing the candidate set; this can be done very inexpensively (without performing any joins). We refer to this

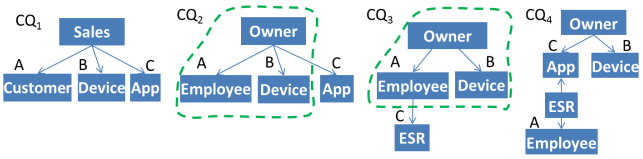


Figure 4: Candidate queries

step as the candidate generation step. *All the algorithms considered in the paper, including the baseline algorithms, first perform this candidate generation step.*

Not all the candidate queries are valid ones. Since the candidate generation step is relatively inexpensive, the main technical problem is: *given the set of candidate queries defined above, compute the valid ones efficiently*. We refer to it as *candidate verification problem*. One option is to individually verify whether the candidate query contains each row in ET in its output. We perform each candidate query-row verification by issuing a SQL join query with keyphrase containment predicates shown in Section 4.1; we refer to them as *CQ-row verification*. The algorithm (referred to as VERIFYALL) performs a high number of CQ-row verifications.

We observe that most candidate queries are invalid. Figure 3(a) and (b) shows the number of candidate and valid queries for two real-life databases and real-life ETs (we vary the # rows in the ETs). More than 90% candidate queries are invalid. The invalid candidate queries typically contain some of the example tuples in their output *but not all*. VERIFYALL performs many CQ-row verifications for such invalid queries; this is wasteful. *How can we quickly prune the invalid candidate queries without verifying them for any tuple?*

Our main insight is that there exists sub-join trees of the candidate queries which, when verified for carefully selected rows of ET, can prune many invalid candidate queries. We refer to these as *filters*. This happens when the filter fails and the sub-join tree is shared by many candidate queries. Judiciously selecting the filters and evaluating them can significantly reduce cost. It is important to consider sub-join trees because the original candidate queries are rarely shared by other candidate queries, no matter how judiciously we select. We illustrate this through an example.

EXAMPLE 2. *For the ET in Figure 2, suppose there are 4 candidate queries generated as shown in Figure 4. The baseline approach (VERIFYALL) performs 3 (CQ_1 contains all 3 rows in its output) $+2+2+2$ (CQ_2, CQ_3 and CQ_4 contain row 1 in their output but not row 2) $=9$ CQ-row verifications. There is a common sub-join tree that is shared by CQ_2 and CQ_3 (shown using green dashed line). If we can verify a common sub-join tree for a row that is likely to be not contained in its output* ***early on****, we can prune all the candidate queries that contain that sub-join tree. Suppose we evaluate the following filter first: the common sub-join tree between CQ_2 and CQ_3 for the second row. It fails. We can prune CQ_2 and CQ_3, thus reducing the number of CQ-row verifications to $3(CQ_1) + 2(CQ_4) + 1$(sub-join tree)$= 6$.*

Contributions: Our contributions are summarized as follows.
• We define the end-to-end system task, introduce the candidate generation-verification framework and define the main technical problem in the paper: candidate verification problem (Section 2).
• We develop efficient techniques to compute the valid queries from the set of candidate queries. All techniques considered in this paper produce the same output (the correct set of valid minimal queries); they differ only in efficiency. We introduce a set of filters which can be used to prune the invalid candidate queries. We then formalize the candidate verification problem as the following filter selection problem: find the set of filters that verifies the validity of

all the valid candidate queries and prunes all the invalid ones while incurring the least evaluation cost. We show the problem is NP-hard even if we have perfect knowledge about which candidates are valid/invalid and which filters succeed/fail. In reality, where we do not have this knowledge, we show that an algorithm can be much worse than the optimal solution. We propose a novel algorithm and show that it has provable performance guarantee with respect to the optimal solution (Section 4 and 5).

• We perform extensive experiments on two real-life datasets, a real customer support database as well as IMDB. Our experiments show that our proposed solution significantly outperforms the baseline approach (by a factor of 2-10) in terms of the number of CQ-row verifications as well as execution time (Section 6).

2. MODEL AND PROBLEM

We first introduce our data model and some notations. We then formally define the problem of *discovering queries by example table*. Finally, we introduce our candidate generation-verification framework to discover all the valid queries w.r.t. a given example table and discuss the challenges.

2.1 Notations and 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 $\text{col}(R) = \{R[i]\}_{i=1,2,...|\text{col}(R)|}$ denote the set of columns of R. For a tuple t in R, let $t[i]$ denote its 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. In general, 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 keyword search is allowed on that column. Example 1 illustrates an example database of seven relations, with arrowed lines denoting foreign key references (i.e., edges in the directed schema graph). There are five text columns: Customer.CustName, Employee.EmpName, Device.DevName, App.AppName, and ESR.Desc. In the rest of this paper, when we refer to a column, it is a text column by default.

2.2 Discovering Queries by Example Table

For a user-provided example table T, the goal of our system is to discover (project join) queries whose answers in the database $\mathcal{D}$ are consistent with T. We formally define an *example table*.

DEFINITION 1. *(Example Table) An example table T is a table with rows $\{r\}$ and columns $\text{col}(T)$, where each cell of a row is either a string (i.e., one or more tokens) or empty. For a row $r \in T$, let $r[i]$ denote its value on the column $i \in \text{col}(T)$, and let $r[i] = \emptyset$ if $r[i]$ is empty.*

Without loss of generality, we assume that T does not contain any empty row or column.

We focus on project join queries with foreign key joins. A *project join query* $(\mathcal{J}, \mathcal{C})$ can be specified by: i) a join graph $\mathcal{J} \subseteq \mathcal{G}$, i.e., a subgraph of the schema graph $\mathcal{G}$ of $\mathcal{D}$ representing all the relations involved (vertices of $\mathcal{J}$) and all the joins (edges of $\mathcal{J}$)–let $\text{col}(\mathcal{J})$ be the set of columns of all the relations in $\mathcal{J}$; and ii) a set of columns $\mathcal{C} \subseteq \text{col}(\mathcal{J})$ from the relations in $\mathcal{J}$, which the join result are projected onto. Let $\mathcal{A}(\mathcal{J}, \mathcal{C})$ be the resulting relation.

Informally, a project join query is said to be *valid* w.r.t. an example table T, if every row of T can be found in the join-project result in the database $\mathcal{D}$, with the columns of T properly *mapped* to the resulting relation.

DEFINITION 2. *(Valid Project Join Queries) Given an example table T in a database $\mathcal{D}$, consider a project join query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ with a mapping $\phi : \text{col}(T) \rightarrow \mathcal{C}$ from columns of T to projection $\mathcal{C}$. Let $\mathcal{A}(Q)$ be the resulting relation of the project join query. Q is* valid w.r.t. a row $r \in T$ *iff there exists a tuple $t \in \mathcal{A}(Q)$ s.t. for any column $i \in \text{col}(T)$,*

$$r[i] = \emptyset \vee r[i] \subseteq t[\phi(i)], \tag{1}$$

where "$x \subseteq y$" denotes that string x is contained in string y.

A project join query Q is valid w.r.t. an example table T, *iff Q is valid w.r.t. every row of T.*

Remarks. There are various ways to define "$x \subseteq y$". In our system, we define it as: tokens in x appear consecutively in y. Synonyms and approximate matching can be introduced to extend the definition of string containment. We can also enable users to specify that x is a number and thus needs to exactly match y. Our system and algorithms can be easily revised to accommodate such extensions.

Among all valid project join queries, we are interested in the minimal ones. A valid project join query is said to be *minimal* if none of relations and its foreign key join constraints can be removed so that it is still valid.

DEFINITION 3. *(Minimal Valid Project Join Queries) A valid project join query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ w.r.t. an example table T is* minimal *iff i) without directions on edges, $\mathcal{J}$ is an undirected tree; and ii) for any degree-1 vertex (relation) R in $\mathcal{J}$, there exists a column $i \in \text{col}(T)$ s.t. $\phi(i) \in \text{col}(R)$, i.e., a column of T is mapped to a column of R.*

EXAMPLE 3. *Figure 2 shows an example table with three rows and three columns* {A, B, C}. *Consider the database shown in Figure 1. Figure 2 shows the only minimal valid project join query for the example table. The mapping ϕ defined from* {A, B, C} *to projection $\mathcal{C}$ =* {Customer.CustName, Device.DeviceName, App.AppName} *is shown using red arrows in Figure 2. We also label each projected column with the ET column (*A, B *or* C*) it is mapped from. The query is valid w.r.t. the first row as there is a tuple (Mike Jones, ThinkPad X1, Office 2013) in the resulting relation that contains the three keywords in the first row. Similarly, it is valid w.r.t. the second and third rows. It is minimal because there is an example table column mapped to at least one column of each degree-1 relation (*Customer, Device, App*).*

End-to-end system task. For a user-provided *example table* T in a database $\mathcal{D}$, the goal of this paper is to find all the *minimal valid project join queries* w.r.t. T in $\mathcal{D}$.

2.3 Candidate Generation-Verification

Candidates of valid queries. In order to discover all the minimal valid project join queries w.r.t. a given example table, our system first generates a list of "candidates" of the final results. The list of candidates is relatively cheap to be generated, and should be a superset of all the valid queries.

DEFINITION 4. *(Candidate Project Join Queries) Given an example table T in a database $\mathcal{D}$, a project join query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ is said to be a* candidate *project join query w.r.t. T iff for any column $i \in \text{col}(T)$,*

$$\forall r \in T \; \exists t \in \mathcal{A}(Q) \text{ s.t. } r[i] \subseteq t[\phi(i)]. \tag{2}$$

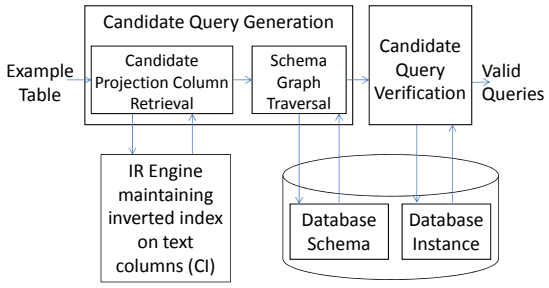


Figure 5: System architecture

Similarly, a candidate project join query is said to be minimal *if it satisfies the conditions i) and ii) in Definition 3.*

The minimal candidate project join queries are defined on the schema level, and no join is needed to generate them, so we can enumerate them efficiently. Details will be discussed in Section 3. We will call them *candidate queries* or *candidates* for short in the rest of this paper.

Intuitively, the strings in each column of the example table should appear in the corresponding (mapped) column of the resulting relation of the candidate query. Equation (2) states this intuition formally. We can easily prove the following result to ensure the set of candidates is complete.

COROLLARY 1. *Any (minimal) valid project join query must also be a (minimal) candidate project join query, for a given example table in any database.*

EXAMPLE 4. *Figure 4 shows the four candidate queries for the example table in Figure 2. The candidate query CQ_1 is the only valid query, and the rest of them are invalid.*

Candidate verification. Suppose the set of all the candidate queries $\mathcal{Q}_{\mathrm{C}}$ has been generated, in the rest of this paper, the task we focus on is to remove all the false positives.

3. SYSTEM ARCHITECTURE

The query discovery system has two components: offline preprocessing and query time processing.

3.1 Offline Pre-processing

There are 2 main steps in this component:

• **Build FTS and other indexes on relations**: As discussed in Section 1, we perform CQ-row verifications by executing SQL join queries with keyphrase containment predicates. Hence, we build a full text search (FTS) index on each text column of each relation in the database. All major commercial DBMSs support FTS indexes on text columns. We also assume appropriate physical design for efficient execution of the SQL join queries, specifically indexes on the primary and foreign key columns.

• **Build master inverted index on columns**: Our candidate generation step first identifies base table columns that satisfy the column constraints; we build a single master inverted index over all text columns in the database for that purpose. We refer to it as *column index* (CI). Given a string W, $CI(W)$ returns the cells (and their corresponding columns) containing W. Since commercial DBMSs do not support such a master index, we implement CI using an information retrieval (IR) engine. We scan each text column in the database. For each token in each cell in each column, we create a "posting" containing the column identifier (which uniquely identifies it across all columns in the database), the cell identifier (which uniquely identifies it within the column) and token position within the cell and store it in the posting list of the token. We can support the operation $CI(W)$ using these posting lists. Any IR engine (e.g., Lucene) that allows customization of the payload in a posting can be used to implement CI.

1	for each column $i \in \mathrm{col}(T)$
2	Compute the set C_i of candidate projection columns
3	$\mathcal{Q}_{\mathrm{C}} \leftarrow$ GenerateCandidateQueries$(\mathcal{G}, C_1, C_2, \cdots, C_{\lvert\mathrm{col}(T)\rvert})$
4	$\mathcal{Q}_V \leftarrow$ VerifyCandidateQueries$(\mathcal{Q}_{\mathrm{C}}, T, \mathcal{D})$

Table 1: Query discovery algorithm

3.2 Query Time Processing

Figure 5 shows the architecture of this component. The pseudocode of the overall query discovery is shown in Table 1. There are two main steps in this component:

• **Candidate query generation**. The input to this step is the ET T and the output is the set $\mathcal{Q}_{\mathrm{C}}$ of candidate queries of T (as defined in Definition 4). This step consists of two substeps:

(1) Candidate projection column retrieval: For each column $i \in \mathrm{col}(T)$ in the ET T, we first identify the set of base table columns that satisfies the column constraint (lines 1-2 in Table 1). These are the ones that contain all the values in column i. We refer to them as the *candidate projection columns* for column i. For example, the candidate projection columns for the ET in Figure 2 are Customer.CustName and Employee.EmpName (for column A), Device.DevName (for column B) and Apps.AppName and ESR.Desc (for column C).

We leverage the column index (CI) for this purpose. For each non-empty cell $T[i,j]$ in column j, we issue a query $CI(T[i,j])$ against the CI. Let $cols(T[i,j])$ denote the distinct columns in the output of $CI(T[i,j])$. Then the set C_j of candidate projection columns for column j of ET T is:

$$C_j = \bigcap_{i, T[i,j] \neq \text{ empty}} cols(T[i,j]) \tag{3}$$

(2) Candidate query enumeration: Given the candidate projection columns C_j of each column j in ET, we enumerate the candidate queries $\mathcal{Q}_{\mathrm{C}}$ by traversing the schema graph. We use a straightforward adaptation of candidate network generation algorithm proposed in [10] (line 3 in Table 1).

This step does not perform any joins and represents a negligible fraction of the overall query processing time.

• **Candidate query verification**. The input to this step is the ET T and the set $\mathcal{Q}_{\mathrm{C}}$ of candidate queries. The output is set $\mathcal{Q}_V$ of valid queries. This is the main contribution of the paper. The rest of the paper focuses on this step. Section 4 presents two baseline algorithms for candidate verification while Section 5 presents the proposed approach.

4. BASELINES FOR CANDIDATE VERIFICATION

4.1 Baseline Algorithm (VERIFYALL)

One way to verify whether a candidate query is valid is to execute it and check whether its output contains all the rows in the ET [23]. This is typically very expensive, hence we do not follow this approach.

Since ET typically contains a few rows (typically less than 10), a more efficient alternative is to *individually* verify whether the candidate query contains *each* of the rows in ET in its output. We henceforth refer to it as verifying the candidate query *"for the row"*.

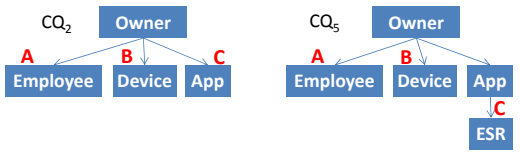


Figure 6: Failure dependency

Recall that a candidate query $(\mathcal{J}, \mathcal{C}, \phi)$ contains the row $r \in T$ in its output iff there exists a tuple t in its output such that every non-empty cell $r[i]$ in that row is textually contained in $t[\phi(i)]$ (Definition 2). Hence, we can verify the candidate query $(\mathcal{J}, \mathcal{C}, \phi)$ for row r by issuing the following SQL query and checking the results:
SELECT * TOP 1
FROM $V(\mathcal{J})$
WHERE $E(\mathcal{J}) \wedge \bigwedge_{i \in \text{col}(T), r[i] \neq \emptyset} CONTAINS(\phi(i), r[i])$
where $V(\mathcal{J})$ denotes the set of vertices (relations) of $\mathcal{J}$ and $E(\mathcal{J})$ the set of edges (join conditions) in $\mathcal{J}$. For example, the SQL query to verify the candidate query CQ_1 in Figure 4 for the second row in the ET in Figure 2 is:

```
SELECT * TOP 1 FROM Sales, Customer, Device, App
WHERE Sales.CustId=Customer.CustId AND
Sales.DevId=Device.DevId AND Sales.AppId=App.AppId AND
CONTAINS(CustName,'Mary') AND CONTAINS(DevName,'iPad')
```

If the result is non-empty, the candidate query $(\mathcal{J}, \mathcal{C}, \phi)$ contains row r, if it is empty, it does not. We also refer to it as candidate query *"satisfies"* the row r and *"fails"* for the row r respectively. Since we just need to check the existence of a row, we use $TOP\ 1$ in the SELECT clause to reduce the data transfer cost. We refer to these SQL queries as *"CQ-row verifications"*.

The baseline algorithm (referred to as VERIFYALL) iterates over the candidate queries $\mathcal{Q}_C$ in the outer loop and the rows in ET in the inner loop (or vice versa) and executes the SQL queries. If a candidate query fails for a row, we eliminates it right away, i.e., do not verify it for the remaining rows. Note that the order of the candidate queries does not affect the number of CQ-row verifications performed but the order of the rows does. In our experiments, we consider random ordering of rows. We also consider the rows in decreasing order of number of non-empty cells. This is because a candidate query is more likely to fail for densely populated rows; verifying them first might lead to early elimination of candidate queries.

EXAMPLE 5. *Figure 4 shows a subset of the candidate queries (4 of them) for the ET in Figure 2 on the example database in Figure 1.* VERIFYALL *performs* 3 *CQ-row verifications for* CQ_1 *as it satisfies all* 3 *rows and* 2 *each for the other* 3 *candidate queries as they all satisfy row 1 but fail for row 2. Hence, it performs a total of* 9 *CQ-row verifications.*

Although simple, VERIFYALL is wasteful as it performs many CQ-row verifications for invalid candidate queries. In the worst case, it performs $|T| \times |\mathcal{Q}_C|$ CQ-row verifications, where $|T|$ is the number of rows in ET. This is expensive as each CQ-row verification involves one or more joins.

4.2 Simple Pruning Algorithm

Since most candidate queries are invalid, we can speed up candidate verification by *quickly pruning the invalid candidate queries*. We observe that there exist dependencies among the results of CQ-row verifications. This can be used to prune candidate queries, i.e., eliminate them without verifying for any rows.

Consider the ET in Figure 2 and two candidate queries in Figure 6. CQ_2 fails for the second row. This implies that CQ_5 will also fail for that row. This is because: (1) the join tree in CQ_2 is a subtree of that in CQ_5 and (2) the non-empty columns in the second row (i.e., columns A and B) are mapped to the same base table columns; they both map A and B to Employee.EmpName and Device.DevName respectively. Since CQ_2 does not contain a tuple whose values on Employee.EmpName and Device.DevName contains 'Mary' and 'iPad' respectively in its output, any "supertree" (e.g., CQ_5) is more restrictive and hence cannot contain such a tuple in its output. We refer to it as the *"failure dependency"*:

LEMMA 1 (FAILURE DEPENDENCY). *For an ET T, the failure of candidate query $Q_1 = (\mathcal{J}_1, \mathcal{C}_1, \phi_1)$ on row r in T implies the failure of candidate query $Q_2 = (\mathcal{J}_2, \mathcal{C}_2, \phi_2)$ for row r in T if: i) $\mathcal{J}_1$ is a subtree of $\mathcal{J}_2$ and ii) for any column $i \in \text{col}(T)$,*

$$r[i] = \emptyset \vee \phi(i) = \phi'(i)$$

We denote $(Q_1, r) \succ_- (Q_2, r)$ iff Q_1 and Q_2 satisfies i) and ii) in Lemma 1.

If we evaluate CQ_2 on row 2 *earlier*, we can prune CQ_5 right away. We present a simple algorithm SIMPLEPRUNE that leverages the above observation to save CQ-row verifications. Like VERIFYALL, SIMPLEPRUNE iterates over the candidate queries in the outer loop and the rows in the inner loop. It maintains the list of CQ-row verifications performed so far that failed. When we encounter a new candidate query Q during the outer loop iteration, we first check whether there exists a CQ-row evaluation (Q', r) in that list such that $(Q', r) \succ_- (Q, r)$. If so, we can prune Q immediately as it is guaranteed to fail for row r and hence is not valid. Otherwise, we verify it for all the rows.

Since the join trees corresponding to the candidate queries are small, the overhead of checking failure dependencies is negligible; the cost is dominated by that of executing the CQ-row verifications.

Unlike VERIFYALL, the order of candidate queries affects the number of CQ-row verifications performed by SIMPLEPRUNE. Intuitively, smaller join trees are more likely to have supertrees and verifying them first will result in more pruning. Hence, we order the candidate queries in increasing join tree size.

EXAMPLE 6. *Consider the ET in Figure 2 and two candidate queries in Figure 6.* VERIFYALL *performs* 4 *CQ-row verifications (both candidate queries satisfy row* 1 *and fail for row* 2*).* SIMPLEPRUNE *verifies* CQ_2 *first and then* CQ_5*. It performs* 2 *CQ-row verifications for* CQ_2*. When it encounters* CQ_5*, it finds a CQ-row verification* $(CQ_2, 2)$ *in the failed list. Since* $(CQ_2, 2) \succ_- (CQ_5, 2)$*, it prunes* CQ_5*. The total number of CQ-row verifications reduces from* 4 *to* 2*.*

5. FAST VERIFICATION VIA FILTERS

We focus on how to speedup candidate verification, which is the bottleneck of our system. We first introduce the concept of *filters* together with its properties in Section 5.1 and give intuitions on why it is the key to the speedup. Intuitively, a filter is a substructure of a candidate query. A candidate is valid only if all of its filters are valid, and a single filter is invalid implies that the candidate is invalid. Different candidates may share common filters and evaluating filters is usually cheaper than evaluating the candidate itself, so evaluating those shared filters first is usually beneficial. On the other hand, the number of filters is usually much larger than the number of candidates, so we need to select the filters to be evaluated judiciously, for which we define the *filter selection problem* in Section 5.2. This problem is shown to be hard in theory but we propose an effective adaptive solution in Section 5.3, which works well in practice and has provable performance guarantee.

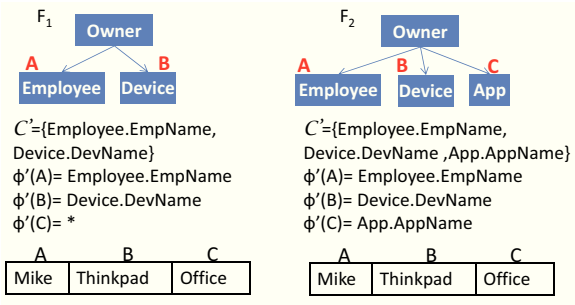


Figure 7: Two filters of candidate query CQ_2 (in Figure 4) on the first row of the example table (in Figure 2)

5.1 Utilizing Filters of Candidate Queries

SIMPLEPRUNE can achieve significant pruning only if there are many subtree-supertree relationship among the candidate queries. Most pairs of candidates do not have this relationship. Figure 4 shows four candidates; and no pair has subtree-supertree relationship. So SIMPLEPRUNE achieves limited pruning with little reduction in execution cost.

Inspired by SIMPLEPRUNE, we find that *there is a high degree of overlap of common subtrees among the candidate queries*. A subtree in a candidate's join tree is more likely to be a subtree in multiple other candidates. Failure of such a subtree for a single row will imply the invalidness of all the candidates containing it. In Figure 4, failure of the common subtree shown using green dashed line implies that the two candidate queries CQ_2 and CQ_3 are invalid. Hence, considering subtree-row verifications can result in fast verification/pruning of candidates. Such subtree-row verifications are formally defined as *filters* below.

DEFINITION 5. *(Filter) Given project join query $Q = (\mathcal{J}, \mathcal{C}, \phi)$ w.r.t. an example table T in a database $\mathcal{D}$, a filter is defined for each connected sub-join tree $\mathcal{J}' \subseteq \mathcal{J}$ and each row $r \in T$ as $Q(\mathcal{J}', r) = (\mathcal{J}', \mathcal{C}', \phi', r)$, where:*
(i) $\mathcal{C}' = \text{col}(\mathcal{J}') \cap \mathcal{C}$ is the set of common columns in relations of $\mathcal{J}'$ and projection $\mathcal{C}$, and;
(ii) $\phi' = \phi \triangleright \mathcal{C}'$ is the range restriction of ϕ on the set of columns $\mathcal{C}'$: for any column $i \in \text{col}(T)$, $\phi'(i) = \phi(i)$ if $\phi(i) \in \mathcal{C}'$ and is undefined ($\phi'(i) = *$) otherwise.
If $\mathcal{J}' = \mathcal{J}$, $Q(\mathcal{J}', r)$ is called a basic *filter. We denote the set of all filters of Q as $\mathcal{F}(Q)$, and the set of all basic filters as $\mathcal{F}_B(Q)$, if T and $\mathcal{D}$ are clear from the context.*

Remarks. Two filters $F_1 = (\mathcal{J}_1, \mathcal{C}_1, \phi_1, r_1)$ and $F_2 = (\mathcal{J}_2, \mathcal{C}_2, \phi_2, r_2)$ are *identical* iff the four parameters $\mathcal{J}_i$'s, $\mathcal{C}_i$'s, ϕ_i's, and r_i's are identical. So two candidate queries Q_1 and Q_2 may share the same filter $Q_1(\mathcal{J}', r) = Q_2(\mathcal{J}', r)$ as long as they are the same within the sub-join tree $\mathcal{J}'$.

EXAMPLE 7. *Figure 7 shows two filters of candidate query CQ_2 in Figure 4 on the first row of the example table in Figure 2. Each of them has the sub-join tree specified on the top, and $\mathcal{C}', \phi'$ specified below. The right one is a basic filter.*

Finally, we discuss the evaluation of a filter and its result. A filter *succeeds* if the row r is found to be consistent to the join result according to the mapping ϕ', and *fails* otherwise. Formally, we have

DEFINITION 6. *(Filter Success and Failure) Consider an example table T in a database $\mathcal{D}$. A filter $Q(\mathcal{J}', r) = (\mathcal{J}', \mathcal{C}', \phi', r)$ of a candidate query Q* succeeds *iff there exists a tuple $t \in \mathcal{A}(\mathcal{J}', \mathcal{C}')$ s.t. for any column $i \in \text{col}(T)$ with $\phi'(i) \in \mathcal{C}'$, we have $r[i] \neq \emptyset \Rightarrow r[i] \subseteq t[\phi(i)]$, and* fails *otherwise.*

Filters can be easily verified using a SQL query. For example, the SQL to evaluate the left filter in Figure 7 is:

```
SELECT * TOP 1 FROM Owner, Employee, Device
WHERE Owner.EmpId=Employee.EmpId AND
Owner.DevId=Device.DevId AND
CONTAINS(EmpName,'Mike') AND CONTAINS(DevName,'ThinkPad')
```

5.1.1 Dependency Properties of Filters

The goal of the filters is to quickly prune the invalid candidate queries. We do so by leveraging the dependencies among the evaluation results of filters as well as between results of filters and the candidate queries.

Filter-query dependency. We first consider dependencies between the evaluation result of filters and the candidate queries. Directly from the definition, we have:

LEMMA 2. *(Filter-Query Dependency) The failure of a filter $F \in \mathcal{F}$ implies a candidate Q is invalid iff $F \in \mathcal{F}(Q)$.*

We denote $F \succ_- Q$ iff $F \in \mathcal{F}(Q)$, and let $\mathcal{Q}_{\rightarrow -}(F) = \{Q \mid F \in \mathcal{F}(Q)\}$ be the set of candidates whose invalidness can be implied by the failure of filter F.

Inter-filter dependency. Failure of a filter may imply failure of other filters. Lemma 3 presents such failure dependencies among the basic filters; this result generalizes to all filters.

LEMMA 3. *(Inter-Filter Failure Dependency) the failure of a filter $F_1 = (\mathcal{J}_1, \mathcal{C}_1, \phi_1, r)$ implies the failure of a filter $F_2 = (\mathcal{J}_2, \mathcal{C}_2, \phi_2, r)$ if i) $\mathcal{J}_1$ is a sub-join tree of $\mathcal{J}_2$ and ii) for any column $i \in \text{col}(T)$, $r[i] = \emptyset \vee \phi_1(i) = * \vee \phi_1(i) = \phi_2(i)$.*

We denote $F_1 \succ_- F_2$ iff F_1 and F_2 satisfies i) and ii) in Lemma 3. We also write $\mathcal{F}_{\rightarrow -}(F_1) = \{F_2 \mid F_1 \succ_- F_2\}$ as the set of filters whose failure is implied by F_1's failure.

Success of a filter may also imply success of other filters.

LEMMA 4. *(Inter-Filter Success Dependency) The success of a filter $F_1 = (\mathcal{J}_1, \mathcal{C}_1, \phi_1, r)$ implies the success of a filter $F_2 = (\mathcal{J}_2, \mathcal{C}_2, \phi_2, r)$ if i) $\mathcal{J}_2$ is a sub-join tree of $\mathcal{J}_1$ and ii) for any column $i \in \text{col}(T)$, $r[i] = \emptyset \vee \phi_2(i) = * \vee \phi_1(i) = \phi_2(i)$.*

We denote $F_1 \succ_+ F_2$ iff F_1 and F_2 satisfies i) and ii) in Lemma 4. We also write $\mathcal{F}_{\rightarrow +}(F_1) = \{F_2 \mid F_1 \succ_+ F_2\}$ as the set of filters whose success is implied by F_1's success.

EXAMPLE 8. *Again consider the two filters F_1 and F_2 in Figure 7 of the candidate query CQ_2 in Figure 4 on the same row of the example table. It is not hard to verify the conditions in Lemmas 3-4 and thus $F_1 \succ_- F_2$ and $F_2 \succ_+ F_1$.*

5.1.2 Adaptive Filter Selection

We use a running example to show intuitions on why and how we can benefit from filters when verifying candidates.

Consider the example table in Figure 2 over the database in Figure 1. Let's focus on three candidates queries CQ_2, CQ_3, and CQ_4 in Figure 4. Figure 8 shows four sub-join trees $\mathcal{J}_1, \ldots, \mathcal{J}_4$ of these candidates, with the projection columns in the underlined relations. Columns {A, B, C} in the example table are mapped to the projection columns as labeled in the figure. We use $(\mathcal{J}_i, j)$ to denote a filter for the sub-join tree $\mathcal{J}_i$ on the j^{th} row of the example table–projection column and ϕ are omitted for simplicity as they have been specified with the sub-join trees in the figure.

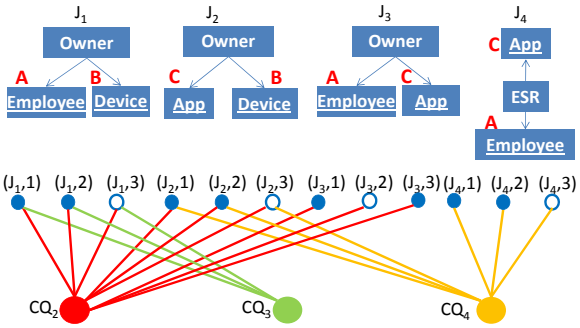


Figure 8: Utilize filters for candidate verification

The lower part of Figure 8 shows twelve filters (four sub-join trees on three rows). Each filter $F = (\mathcal{J}_i, j)$ is connected to a candidate CQ_i if $F \in \mathcal{F}(CQ_i)$. The node corresponding to a filter is filled if the filter successes, and is not filled otherwise, which is unknown before evaluating.

Judicious. The three candidates CQ_2, CQ_3, and CQ_4 are all invalid. To verify this, we only need to evaluate two filters $(\mathcal{J}_1, 3)$ and $(\mathcal{J}_2, 3)$. However, even if we know the evaluation results beforehand, how to choose the minimum set of failed filters to cover those invalid candidates is a hard problem (consider the Set Cover problem). So our algorithm needs to be judicious in choosing filters to evaluate.

Information of failure likelihood. Without any additional information about the likelihood to fail for each filter, it is possible for our algorithm to ends up with evaluating all the successful filters before evaluating the failed ones. So we need at least a rough estimation of how likely a filter fails.

Adaptive. We need to choose the next filter to evaluate adaptively. For example, suppose we evaluated $(\mathcal{J}_2, 3)$ at some point and found it failed. Then the next filter we want to choose should be among $\{(\mathcal{J}_1, 1), (\mathcal{J}_1, 2), (\mathcal{J}_1, 3)\}$, because now, only the validness of CQ_3 is unknown and we can focus on only the filters associated with CQ_3.

We will take into account these three intuitions in the following problem formulation and algorithm design.

5.2 Filter Selection Problem

We now introduce the *filter selection problem*. For all the candidate queries $\mathcal{Q}_\mathrm{C}$ of a given example table T, we want to verify the validness of these queries by evaluating filters. Recall $\mathcal{F} = \cup_{Q \in \mathcal{Q}_\mathrm{C}} \mathcal{F}(Q)$ is the set of all filters derived from the candidates, and $\mathcal{F}_\mathrm{B} = \cup_{Q \in \mathcal{Q}_\mathrm{C}} \mathcal{F}_\mathrm{B}(Q) \subseteq \mathcal{F}$ be the basic ones. Indeed, it is not necessary to evaluate all the filters in $\mathcal{F}$, and our goal is to evaluate the set of filters with the least cost so that all the candidates are verified as valid/invalid. More formally, when the following *termination condition* is satisfied, all the candidates have been verified.

Termination condition. Let $\hat{\mathcal{F}} \subseteq \mathcal{F}$ be the set of evaluated filters. $\hat{\mathcal{F}}$ satisfies the termination condition if: i) for each valid query Q in $\mathcal{Q}_\mathrm{C}$, $\mathcal{F}_\mathrm{B}(Q) \subseteq \hat{\mathcal{F}}$: we have evaluated all the basic filters in $\mathcal{F}_\mathrm{B}(Q)$ to verify this query is valid w.r.t. every row in the example table T; and ii) for each invalid query Q, there exists a failed filter $Q(\mathcal{J}, r) \in \hat{\mathcal{F}}$: we have evaluated at least one failed filter $Q(\mathcal{J}, r)$ in $\mathcal{F}(Q)$.

Problem statement (FILTERSELECTION)
Given the set of filters $\mathcal{F}$ for all the candidate queries $\mathcal{Q}_\mathrm{C}$ of a user-specified example table T in a database $\mathcal{D}$, and suppose each filter $F \in \mathcal{F}$ is associated with some cost, denoted by $\mathrm{cost}(F) \in \Re$, to find the set of filters $\hat{\mathcal{F}} \subseteq \mathcal{F}$ that satisfies the termination condition defined above, with the objective to minimize total evaluation cost for filters in $\hat{\mathcal{F}}$:

$$\text{minimize} \quad \sum_{F \in \hat{\mathcal{F}}} \mathrm{cost}(F) \tag{4}$$

$$\text{s.t.} \quad \hat{\mathcal{F}} \subseteq \mathcal{F} \quad \text{satisfies the termination condition.}$$

Remarks. We want to point out that the cost of evaluating a filter is hard to be estimated in general. In our system, we use the number of joins in a filter F to approximate the cost. Although this approximation is not perfect, it is sufficient as a guideline for our filter selection algorithm. Furthermore, the overhead of the filter selection algorithm is negligible compared with the cost of evaluating the filters. So, we ignore the former in our cost model.

Hardness results. The hardness of FILTERSELECTION lies in two aspects. First, even if our algorithm has *perfect knowledge* about which candidates are valid or invalid and which filters success or fail (but still need to perform the evaluation action), FILTERSELECTION is hard as a combinatorial problem. Second, as the validness of each candidate and the evaluation result for each filter are unknown before the evaluation starts, without the perfect knowledge, an algorithm can be much worse than the optimal solution. The two aspects of hardness are formulated as follows.

THEOREM 1. *The* FILTERSELECTION *problem is NP-hard, with $|\mathcal{F}| + |\mathcal{Q}_\mathrm{C}|$ as the input size of the problem.*

THEOREM 2. *Let* MIN *be the minimum number of filters that need to be evaluated in* FILTERSELECTION. *For any algorithm, there exists an instance of* FILTERSELECTION *s.t. the algorithm choses to evaluate $\Omega(|T|) \cdot$ MIN *filters, where $|T|$ is the number of rows in the example table T.*

In the following part of this subsection, we will introduce an algorithm that performs much better in practice, and provide theoretical analysis for its performance.

5.3 Adaptive Verification Algorithm

The algorithm we propose for FILTERSELECTION is adaptive, in the sense that it chooses to evaluate a sequence of filters $F_1, F_2, \ldots$, in $\mathcal{F}$ in a way that the choice of F_i depends on the evaluation results of $F_1, \ldots, F_{i-1}$. We use $\mathrm{ex}(F_i) \in \{+, -\}$ to denote the evaluation result of F_i, with "+" representing success and "−" representing failure. Similarly, we use $\mathrm{ex}(Q) \in \{+, -\}$ to denote the validness of a candidate query Q, with "+" representing valid and "−" representing invalid. Initially, we have $\mathcal{F}_\mathrm{X} = \mathcal{F}$ as the set of filters whose evaluation result is unknown, and $\mathcal{Q}_\mathrm{X} = \mathcal{Q}_\mathrm{C}$ as the set of candidate queries whose validness is unknown.

In each iteration, when we choose to evaluate filter F_i, if the result is success, the evaluation result of all queries that have success dependency can be also determined to be success, and thus we remove F_i together with $\{F \mid F_i \succ_+ F\}$ from $\mathcal{F}_\mathrm{X}$; and if the result is failure, we know every query Q with $F_i \in \mathcal{F}(Q)$ is invalid, so we remove those queries from $\mathcal{Q}_\mathrm{X}$, and also remove F_i and filters in $\{F \mid F_i \succ_- F\}$ that are known to be failed from $\mathcal{F}_\mathrm{X}$. At the same time, if at some point, all the basic filters of a candidate query Q are evaluated to be success, we know Q is valid and thus Q can be also removed from $\mathcal{Q}_\mathrm{X}$. The algorithm can terminate when $\mathcal{Q}_\mathrm{X}$ is empty, i.e., the validness of all candidate queries is known. The framework is formalized in Algorithm 1.

Based on the properties of failure and success dependencies (Lemmas 2-4), we can prove the above algorithm correctly verifies the validness of all candidate queries. As long as SelectNext (in line 3) picks a filter $F_i \in \mathcal{F}_\mathrm{X}$ each time, the final sequence of queries

Algorithm 1: AdaptiveVerify($\mathcal{F}$, $\mathcal{Q}_{\mathrm{C}}$)

Input : all filters $\mathcal{F}$ and all candidate queries $\mathcal{Q}_{\mathrm{C}}$
Output: validness $\mathrm{ex}(Q)$ for each $Q \in \mathcal{Q}_{\mathrm{C}}$

1 $\mathcal{F}_{\mathrm{X}} \leftarrow \mathcal{F}$, $\mathcal{Q}_{\mathrm{X}} \leftarrow \mathcal{Q}_{\mathrm{C}}$, $i \leftarrow 1$;
2 **while** $\mathcal{Q}_{\mathrm{X}} \neq \emptyset$ **do**
3 $F_i \leftarrow$ $\mathrm{SelectNext}(\mathcal{F}, \mathcal{Q}_{\mathrm{C}}, F_1, \ldots, F_{i-1}, \mathrm{ex}(F_1), \ldots, \mathrm{ex}(F_{i-1}))$, evaluate F_i, and let $\mathcal{F}_{\mathrm{X}} \leftarrow \mathcal{F}_{\mathrm{X}} - \{F_i\}$;
4 **if** F_i succeeds **then**
5 $\mathcal{F}_{\mathrm{X}} \leftarrow \mathcal{F}_{\mathrm{X}} - \{F \mid F_i \succ_+ F\}$;
6 **for each** $Q \in \mathcal{Q}_{\mathrm{X}}$, if $\mathcal{F}_{\mathrm{B}}(Q) \subseteq \mathcal{F} - \mathcal{F}_{\mathrm{X}}$:
7 $\mathrm{ex}(Q) \leftarrow$ "+" and $\mathcal{Q}_{\mathrm{X}} \leftarrow \mathcal{Q}_{\mathrm{X}} - \{Q\}$;
8 **else**
9 $\mathcal{F}_{\mathrm{X}} \leftarrow \mathcal{F}_{\mathrm{X}} - \{F \mid F_i \succ_- F\}$;
10 **for each** $Q \in \mathcal{Q}_{\mathrm{X}}$, if $F_i \in \mathcal{F}(Q)$:
11 $\mathrm{ex}(Q) \leftarrow$ "−" and $\mathcal{Q}_{\mathrm{X}} \leftarrow \mathcal{Q}_{\mathrm{X}} - \{Q\}$;
12 Return $\mathrm{ex}(Q)$ for all candidate queries;

$\{F_1, \ldots, F_n\}$ satisfies the termination condition defined earlier for FILTERSELECTION problem.

THEOREM 3. *Algorithm 1 can correctly verify the validness of all candidate queries in $\mathcal{Q}_{\mathrm{C}}$.*

The only missing block in Algorithm 1 is how to choose the next filter to evaluate based on the current evaluation results, i.e., function SelectNext in line 3. The choice of F_i in each iteration is critical to the overall performance (measured by the total cost of filters needed to be evaluated). We will introduce our solution and analyze why it performs well using a probabilistic model in the following part.

5.3.1 Greedy Filter Selection

Intuitively, the selection of F_i depends on i) its evaluation cost; and ii) how much it can benefit the following verification process. i) has been modeled as $\mathrm{cost}(F_i)$ when we define the FILTERSELECTION problem. However, ii) is hard to be modeled as the algorithm has no knowledge about whether a filter successes or fails before evaluating it, which leads to the hardness result in Theorem 2. We first introduce a probabilistic model which provides the algorithm clue on how likely a filter will succeed to make the decision.

Probabilistic model on filter failure. Consider the set of all candidate queries $\mathcal{Q}_{\mathrm{C}}$ and all the filters $\mathcal{F}$. All the valid candidates $\mathcal{Q}_+$ and invalid candidates $\mathcal{Q}_-$ ($\mathcal{Q}_+ \cup \mathcal{Q}_- = \mathcal{Q}_{\mathrm{C}}$) are fixed. Let $\mathcal{F}(\mathcal{Q}_+)$ be the set of filters associated with some query in $\mathcal{Q}_+$, and $\mathcal{F}(\mathcal{Q}_-)$ be the ones associated with some in $\mathcal{Q}_-$. Indeed, all filters in $\mathcal{F}(\mathcal{Q}_+)$ must success. The probabilistic model is about filters in $\mathcal{F}(\mathcal{Q}_-) - \mathcal{F}(\mathcal{Q}_+)$, i.e., the ones that are possible to fail. For each filter $F \in \mathcal{F}(\mathcal{Q}_-) - \mathcal{F}(\mathcal{Q}_+)$, we assume it will fail with certain probability $p(F) = \mathbf{Pr}[F \text{ fails}]$, because the user's input example table T is "random". For each filter $F \in \mathcal{F}(\mathcal{Q}_+)$, define $p(F) = 0$ (i.e., F must succeed). It is hard to estimate $p(F)$ in general. And in our system, the estimation should not be computationally expensive. Therefore, we employ the following simple model to approximate $p(F)$.

Consider filter $F = (\mathcal{J}, \mathcal{C}, \phi, r)$ for a row r in T. Let n_F be the number of non-empty cells in r that are mapped to a column in $\mathcal{C} \subseteq \mathrm{col}(\mathcal{J})$, i.e., $n_F = |\{i \mid \phi(i) \neq * \wedge r[i] \neq \emptyset\}|$. Recall $|\mathrm{col}(T)|$ is the total number of columns in T. And we assume that the average failure probability is a constant $\bar{p}$. Intuitively, $p(F)$ is proportional to n_F and inversely proportional to $\mathrm{col}(T)$ (more constraints imply higher probability to fail). It should also be proportion to $\bar{p}$. Putting them together, we let $p(F) = \bar{p} \cdot \frac{n_F}{|\mathrm{col}(T)|}$. Again, it is a very rough estimation (similar to $\mathrm{cost}(F)$), but it suffices for our theoretical performance analysis, as a guideline for choosing filters, and it is shown to be effective in experiments.

Verification workload. Now we quantify the workload that is done by evaluating a filter F_i. We use $\mathrm{W}(F_i \mid F_1, \ldots, F_{i-1})$ to denote the *workload* done, or *benefit*, by evaluating F_i given that $F_1, \ldots, F_{i-1}$ have been evaluated. Let $\mathcal{Q}_{\mathrm{X}}$ and $\mathcal{F}_{\mathrm{X}}$ be as before line 3 in Algorithm 1. Two cases are considered separately (F_i succeeds or fails):

i) F_i succeeds: For each candidate $Q \in \mathcal{Q}_{\mathrm{X}}$, if a filter $F \in \mathcal{F}(Q)$ is not verified yet, i.e., $F \in \mathcal{F}_{\mathrm{X}}$, the success of F_i implies the success of F if $F_i \succ_+ F$. Recall $\mathcal{F}_{\rightarrow +}(F_i) = \{F \mid F_i \succ_+ F\}$. The workload done by F_i is defined to be all such (query, filter) pairs:

$$\mathrm{W}_+(F_i | F_1 \ldots, F_{i-1}) = \sum_{Q \in \mathcal{Q}_{\mathrm{X}}} |\mathcal{F}(Q) \cap \mathcal{F}_{\mathrm{X}} \cap \mathcal{F}_{\rightarrow +}(F_i)|. \quad (5)$$

ii) F_i fails: In this case, every candidate query $Q \in \mathcal{Q}_{\mathrm{X}}$ with $F_i \in \mathcal{F}(Q)$ can be verified as invalid. So all the remaining workload on Q can be considered done, which is the number of unevaluated filters in $\mathcal{F}(Q) \cap \mathcal{F}_{\mathrm{X}}$:

$$\mathrm{W}_-(F_i | F_1 \ldots, F_{i-1}) = \sum_{Q \in \mathcal{Q}_{\mathrm{X}}:\ F_i \in \mathcal{F}(Q)} |\mathcal{F}(Q) \cap \mathcal{F}_{\mathrm{X}}|. \quad (6)$$

Within our probabilistic model described above, the workload $\mathrm{W}(F_i | F_1 \ldots, F_{i-1})$ done by F_i is a random variable:

$$\mathrm{W}(F_i | \ldots) = \begin{cases} \mathrm{W}_+(F_i | \ldots) & \text{with probability } 1 - p(F_i) \\ \mathrm{W}_-(F_i | \ldots) & \text{with probability } p(F_i) \end{cases}. \quad (7)$$

But no matter which sequence $F_1, F_2, \ldots, F_n$ Algorithm 1 takes from the beginning to the termination and whether each filter F_i succeeds or fails, we have the total workload to be done fixed by input, as stated in the following lemma.

LEMMA 5. *For any sequence of filters $F_1, F_2, \ldots, F_n$ such that the termination condition is achieved after they are evaluated in order, we have:*

$$\sum_{i=1}^{n} \mathrm{W}(F_i \mid F_1, \ldots, F_{i-1}) = \sum_{Q \in \mathcal{Q}_{\mathrm{C}}} |\mathcal{F}(Q)|. \quad (8)$$

Filter selection criterion. Now we are ready to present our criterion for selecting the next filter to evaluate in Algorithm 1, i.e., function SelectNext:

In line 3 of each iteration, we choose F_i from the set of all filters that are not evaluated yet (and not implied by the previously evaluated ones), i.e., the set $\mathcal{F}_{\mathrm{X}}$. We choose the one that maximizes expected workload per unit cost:

$$F_i \leftarrow \arg\max_{F \in \mathcal{F}_{\mathrm{X}}} \frac{\mathbf{E}[\mathrm{W}(F | F_1, \ldots, F_{i-1})]}{\mathrm{cost}(F)}, \quad (9)$$

where the expectation done by F is $\mathbf{E}[\mathrm{W}(F|\ldots)] = (1 - p(F)) \cdot \mathrm{W}_+(F|\ldots) + p(F) \cdot \mathrm{W}_-(F|\ldots)$.

We want to emphasize that Algorithm 1 can be executed without any probabilistic model/assumption. All the quantities in (9) can be computed from the data. The probabilistic model is only

used for the analytical purpose. We will demonstrate that Algorithm 1 with the SelectNext function defined in (9) is effective in practice in Section 6. In the rest of this section, we explain why it performs well from a theoretical point of view based on the probabilistic model.

5.3.2 Performance Analysis

The toolbox we use to analyze the performance of Algorithm 1 is called *adaptive submodularity* introduced by [7]. It is an extension to *submodularity* on probabilistic data to capture diminishing return in expectation. In our case, the definition in [7] can be revised as follows.

DEFINITION 7. *(Adaptive Diminishing Marginal) The function* $\mathrm{W}(F|\ldots)$ *is said to be* adaptive diminishing *on marginal, iff for any two sets of filters* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *with* $\mathcal{F}_1 \subseteq \mathcal{F}_2$ *and the same evaluation results on the common filters (in* $\mathcal{F}_1$*), we have* $\mathrm{W}(F|\mathcal{F}_1) \geq \mathrm{W}(F|\mathcal{F}_2)$ *for any filter* $F \notin \mathcal{F}_2$.

We can prove that the workload function $\mathrm{W}(F|\ldots)$ defined by Equations (5)-(7) is indeed diminishing on marginal. The proof is omitted because of space limit.

LEMMA 6. *The workload function* $\mathrm{W}(F|\ldots)$ *defined by Equations* (5)-(7) *is adaptive diminishing on marginal.*

With Theorem 7.1 in [8] and Lemmas 5-6 here, we can prove the following performance ratio of Algorithm 1.

THEOREM 4. *Let* SOL *be the total cost of filters evaluated by Algorithm 1. Let* OPT *be the minimum expected total cost of filters chosen by* any algorithm with no knowledge about which filters succeed or fail. *We have*

$$\mathbf{E}\left[\mathrm{SOL}\right] \leq (1 + \ln|\mathcal{F}| + \ln|\mathcal{Q}_{\mathrm{C}}| + 2^{|\mathrm{col}(T)|}) \cdot \mathrm{OPT}.$$

Remark. Because of the space limit, we omit the proof of the above theorem. But we want to mention several important points. i) The performance guarantee holds under the probabilistic model described in Section 5.3.1 with a reasonable independence assumption: when two filters have no success/failure dependency between them, they are independent. But *the algorithm itself does not rely on any probabilistic model or assumption and can be applied for any dataset*. ii) The factor $2^{|\mathrm{col}(T)|}$ seems to be large, but in fact, in our analysis, it only applies to the portion of cost for verifying the set of valid candidate queries, which is usually a small subset of all candidates $\mathcal{Q}_{\mathrm{C}}$. The form of bound we present in Theorem 4 is not tight, and Algorithm 1 performs well in practice as shown in the experiments.

6. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of the techniques proposed in this paper. The goals of our study are:

- To compare the performance of our FILTER approach with VERIFYALL and SIMPLEPRUNE
- To evaluate the sensitivity of VERIFYALL, SIMPLEPRUNE and FILTER on various example tables
- To compare our FILTER approach with WEAVE algorithm proposed in [18]

6.1 Experimental Setting

Datasets. We perform experiments on two real-world datasets: IMDB and CUST. IMDB is a database of information related to movies, directors and actors, with size of 10GB. CUST is a large data collection about customer service and IT support from a Fortune 500 company, with size of 90GB. Table 2 provides the details for these two datasets.

Table 2: Datasets

	Relations	Edges	Columns	Text Columns
IMDB	21	22	101	42
CUST	100	63	1263	614

Table 3: Parameter ranges and default values(underlined)

Parameter	Description	Range
m	row number	2, 3, <u>4</u>, 5, 6
n	column number	2, 3, <u>4</u>, 5, 6
s	sparsity	0, 0.2, 0.3, <u>0.5</u>, 0.7
v	cell value length	1, <u>2</u>, 3
l	maximal join length	3, <u>4</u>, 5

Example table generation. We generate example tables for the experiments as follows. First, we choose 10 meaningful join graphs based on database schema graph. Each join graph contains more than 6 text columns. We execute the join query for each graph and project the result on all the text columns involved. The project join result forms a matrix. Thus, we have 10 matrices.

We then generate example tables based on the above matrices. We control the example tables generated via 4 parameters:

- m: number of rows
- n: number of columns
- s: sparsity of the table, i.e., fraction of empty cells
- v: value length (i.e., number of tokens) in non-empty cells

Given a matrix and the values for the four parameters, we generate an example table in three steps:

1) we generate a $m \times n$ grid by selecting m random complete rows (i.e., without empty cells) from the matrix and projecting them onto n random columns

2) we randomly choose $\lfloor m \times n \times s \rfloor$ cells from the grid to be empty.

3) if there is no completely empty row or column in the grid, we keep the first v tokens for each non-empty cell and get an example table. Otherwise, go to Step 2.

Algorithms. We implement four algorithms for the experiments:

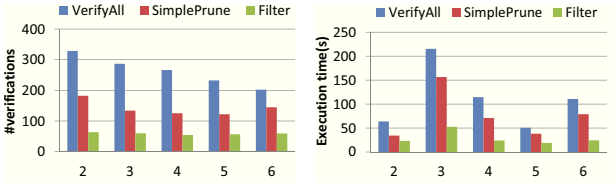
- VERIFYALL as described in Section 4.1
- SIMPLEPRUNE as described in Section 4.2
- FILTER: our proposed algorithm
- WEAVE: the approach proposed in [18]

Measure. We measure the performance of the algorithms via three metrics: the number of verifications performed to find the valid queries from the candidate queries, the total estimated cost of these verifications (i.e., the sum of join tree sizes) and the execution time. For the FILTER algorithm, the number of verifications is the number of filters evaluated. We observe the total estimated cost is consistent with the number of verifications throughout the experiments; we do not include those charts to avoid duplication.

We implement all the algorithms in C# and run experiments on a 64-bit Windows 7 machine with 3.4GHz Intel CPU and 8GB of RAM. We use SQL Server 2012 Enterprise Edition as the DBMS.

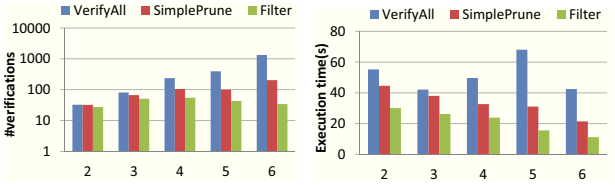
6.2 Comparison of Various Approaches

We compare our FILTER approach with VERIFYALL and SIMPLEPRUNE for ETs with different numbers of rows, columns, sparsities and cell value lengths. We also measure the effect of maximal join length l (i.e., the allowed maximal join tree size in a valid query) on the performance of three algorithms. Table 3 shows the ranges for the five parameters. Unless otherwise specified, we use the underlined default values. For a given choice of the parameter values, we generate 5 ETs from each of the 10 matrices, resulting in 50 ETs. All results are averaged over the 50 ETs.



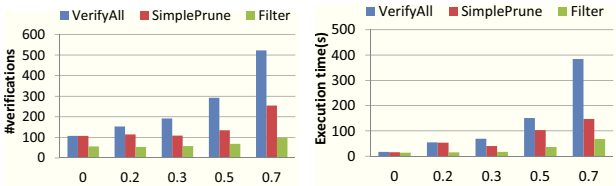
(a) #verifications (b) Execution time

Figure 9: Vary the number of rows (IMDB)



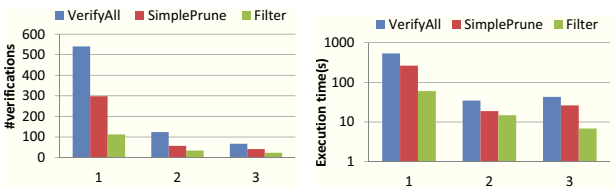
(a) #verifications (b) Execution time

Figure 10: Vary the number of columns (IMDB)



(a) #verifications (b) Execution time

Figure 11: Vary sparsity (IMDB)



(a) #verifications (b) Execution time

Figure 12: Vary cell value length (IMDB)

6.2.1 *Results on IMDB*

We first present the results on IMDB dataset.

Varying the number of rows. Figure 9(a) shows the number of verifications for ETs with different numbers of rows (m). On average, FILTER outperforms VERIFYALL and SIMPLEPRUNE by performing *5X and 2X fewer verifications* respectively. Furthermore, FILTER is robust to m, i.e., it requires similar number of verifications as m varies. On the other hand, VERIFYALL and SIMPLEPRUNE is sensitive to m. VERIFYALL performs more verifications for smaller m as there are more candidate queries. SIMPLEPRUNE has a U-shaped curve: it performs worse for very low and high values of m ($m = 2, 6$) and better for medium values ($m = 3, 4, 5$). For very low values of m, it performs too many verifications as there are many more candidate queries; for very high values of m, there are fewer candidate queries but they need to be verified for more rows, thus leading to higher number of verifications.

Figure 9(b) shows the execution time on IMDB. FILTER runs *4X and 3X faster* than VERIFYALL and SIMPLEPRUNE respectively for all the values of m. The execution time is not always consistent with the number of verifications as it is affected by various factors such as number of joins in the candidate queries as well as their execution plans determined by the database query optimizer.

Varying the number of columns. Figure 10(a) shows the number of verifications performed for ETs with different numbers of columns (n). FILTER requires fewer verifications than VERIFYALL and SIMPLEPRUNE over all the column values. The difference becomes larger when n increases. For $n = 6$, FILTER saves nearly 50% and 30% of verifications for VERIFYALL and SIMPLEPRUNE, respectively. Furthermore, FILTER is robust to n while the other two approaches perform more verifications as n becomes larger.

Figure 10(b) provides the execution time. FILTER is *2X and 1.5X faster* faster than VERIFYALL and SIMPLEPRUNE respectively. For both FILTER and SIMPLEPRUNE, the execution time decreases with the increase of n. This is because more input columns lead to candidate queries with different join tree sizes; this provides more opportunities to leverage failure dependencies (for SIMPLEPRUNE) and shared filters (for FILTER) for pruning.

Varying sparsity. Figure 11(a) shows the average number of verifications for different values of sparsity (s). FILTER outperforms SIMPLEPRUNE by performing 2X fewer verifications over all the sparsity values, and is significantly better than VERIFYALL. FILTER is robust to different values of s while VERIFYALL suffers from larger s, e.g., it performs 5X more verifications than FILTER for $s = 0.7$. This is because more sparse the ETs, less restrictive the column constraints, higher the number of candidate queries.

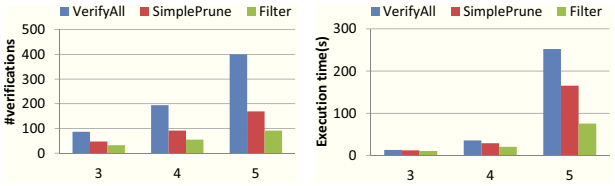
Figure 11(b) provides the average execution time on IMDB for various sparsity values. For all values of s except $s = 0$ (i.e., when ET has at least a few empty cells), FILTER is *3X and 2.25X faster* than VERIFYALL and SIMPLEPRUNE. For $s = 0$ (i.e., when ET has no empty cell at all), FILTER requires similar time as VERIFYALL and SIMPLEPRUNE, which is within 20 seconds. For $s = 0$, the improvement in the number of verifications does not translate to improvement in execution time because SQL Server could execute the CQ-row verifications for the invalid candidates efficiently in this special case. This is because there are more keyphrase containment predicates on the text columns for all such queries; the joins are efficient due to low intermediate result sizes. This is not the case for other sparsity values. The latter is more common in practice as it is unlikely that the information worker can provide all the cell values in the ET.

Varying cell value length. Figure 12(a) provides the number of verifications for different cell value lengths (v). FILTER requires the smallest number of verifications over all the values of v. When v equals to 1, FILTER *performs 5X and 3X fewer verifications* than VERIFYALL and SIMPLEPRUNE respectively. As v increases, the number of verifications decreases for all the three algorithms. This is because the input ETs are become more constrained and hence produce fewer candidate queries.

Figure 12(b) shows the average execution time. FILTER is nearly *10X faster* than VERIFYALL for $v = 1$ to 3, and outperforms SIMPLEPRUNE by a *factor of* 2.

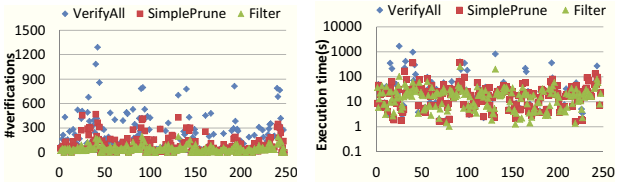
Varying maximal join length. Figure 13(a) shows the number of verifications on different maximal join lengths (l). When l increases, all the three algorithms perform more verifications. This is because a higher value of l allows more queries (with larger join trees) to become candidates. We observe FILTER requires the smallest number of verifications over all the values of l, and it outperforms VERIFYALL and SIMPLEPRUNE by saving more than 60% and 40% verifications.

Figure 13(b) provides the execution time on IMDB. The result is consistent with that for the number of verifications. Namely, all the algorithms have longer running time as l becomes larger. FILTER outperforms the other two algorithms over all the values of l. When l equals to 5, FILTER is nearly *3X and 2X faster* than VERIFYALL and SIMPLEPRUNE, respectively.



(a) #verifications (b) Execution time

Figure 13: Vary maximal join length (IMDB)



(a) #verifications (b) Execution time

Figure 15: Case-by-case performance

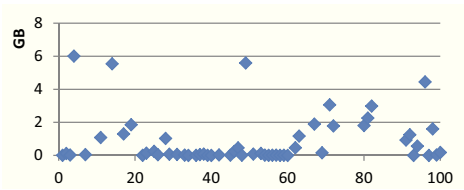


Figure 16: Memory usage for WEAVE

Table 4: Comparison between WEAVE and FILTER

$s = 0$	Avg. query#	Avg. cost	Avg. time(s)
WEAVE	611	1982.9	76.1
FILTER	56	143.5	13.2
$s = 0.2$	Avg. query#	Avg. cost	Avg. time(s)
WEAVE	585	1893.1	79.5
FILTER	54	141.3	14.3
$s = 0.5$	Avg. query#	Avg. cost	Avg. time(s)
WEAVE	583	1892.8	81.2
FILTER	64	179.3	21.9

6.2.2 *Results on CUST*

Figure 14 shows the results on CUST dataset. We find the results on CUST share similar trends as IMDB and hence only plot the number of verifications by varying three most important parameters m, n and s. Figure 14(a) shows the number of verifications for ETs with different numbers of rows. FILTER performs the fewest number of verifications over all the values of m, which outperforms VERIFYALL and SIMPLEPRUNE by saving nearly 70% and 60% of verifications respectively. Further, FILTER is robust as m varies from 2 to 6, while the numbers of verifications performed VERIFYALL and SIMPLEPRUNE vary for different m.

Figure 14(b) shows the number of verifications for ETs with different number of columns. FILTER outperforms VERIFYALL and SIMPLEPRUNE over all the column values, and the advantage of FILTER becomes more significant when n increases. For $n = 6$, FILTER performs one and two orders of magnitude fewer verifications compared with VERIFYALL and SIMPLEPRUNE respectively.

Figure 14(c) shows the number of verifications for ETs with different sparsities. FILTER requires the fewest verifications over all the sparsities. Its advantage over VERIFYALL and SIMPLEPRUNE becomes more significant as s increases. For $s = 0.7$, FILTER performs nearly 5X and 2X fewer verifications compared with VERIFYALL and SIMPLEPRUNE respectively. Further, FILTER is more robust to different sparsities than the other two algorithms.

Henceforth, we only report the results for IMDB as we observe very similar results for the CUST dataset.

6.2.3 *Case-by-case Comparison*

In order to study their worst-case behavior, we now present the performance of the three algorithms on individual cases. We generated 250 ETs with default values of the five parameters as shown in Table 3. Figure 15(a) shows the number of verifications for the 250 ETs. FILTER outperforms VERIFYALL and SIMPLEPRUNE by performing significantly fewer verifications in most cases. We observe FILTER requires less than 200 verifications in all the 250 cases, while VERIFYALL and SIMPLEPRUNE require more than 200 verifications in 169 and 223 cases respectively. The number of verifications in the worst case for VERIFYALL and SIMPLEPRUNE are over 1200 and 400 respectively.

Figure 15(b) reports the corresponding execution time. FILTER is more efficient than VERIFYALL and SIMPLEPRUNE in most cases. For the ETs that can be answered quickly (i.e., within 20 seconds), the performance of the 3 algorithms is similar. But in the worst cases, VERIFYALL and SIMPLEPRUNE run for nearly one thousand seconds, while FILTER is 10*X faster*. The results indicate that FILTER is more robust to handle bad cases which is a desirable property in real systems.

6.3 Comparison with WEAVE

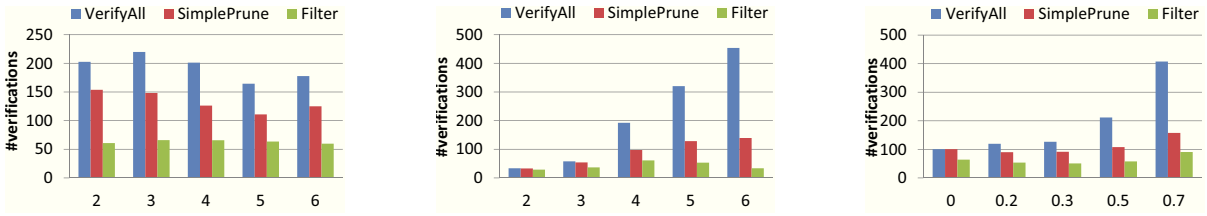
We now compare FILTER with the WEAVE approach proposed in [18]. WEAVE generates candidates for first row and verifies for remaining rows. WEAVE keeps interconnected tuple trees in memory as they progressively "weave" these tuple trees to obtain the final set of tuple trees. In our implementation, we store the tuple trees in temporary tables in SQL Server, thus delegating the memory management to the DBMS. We observe that the performance of WEAVE suffers due to the high memory usage and thrashing of data between memory and disk; We tested WEAVE on 100 ETs with default settings, and find only 56 ETs finished within 10 minutes. Figure 16 shows the size of in-memory tuple trees for the finished cases; WEAVE requires a large amount of memory in many cases.

To reduce the memory usage, we studied an alternate implementation of WEAVE that weaves the join trees instead of tuple trees. To make the comparison fair, we push column constraints down as in our approaches. Table 4 presents the comparison results of WEAVE and FILTER for ETs with different values of sparsity ($s = 0$, 0.2 and 0.5). The results are averaged over 100 ETs for each sparsity value. FILTER significantly outperforms WEAVE for all the three metrics over all the sparsity values. On average, FILTER performs 10*X fewer verifications* and it runs 4*X faster* than WEAVE. Our filter-based approach judiciously selects filters based on their cost and benefit; this can prune candidate queries much more effectively compared with the weaving approach which does not consider such cost/benefit.

Summary: In summary, our experiments show that the filter-based solution can prune invalid candidate queries more quickly and hence significantly outperforms the baseline algorithms. Furthermore, it is robust to different characteristics of the ETs. Finally, the filter-based solution significantly outperforms WEAVE due to the better pruning achieved by judicious cost-benefit based filter selection.

7. RELATED WORK

Our work is related to works on keyword search in databases [1, 10, 2, 14, 9, 19, 4, 6, 15, 12, 5]. In keyword search, the user can provide only a single tuple as input and the system returns all join trees that contain at least one tuple containing all the keywords.



(a) Vary the number of rows (b) Vary the number of columns (c) Vary sparsity

Figure 14: Results on CUST dataset

An enterprise information worker often has knowledge of multiple rows and that can help constrain the valid queries; this is not expressible in keyword search. Straightforward adaption of the solutions for keyword search would compute the set of valid minimal queries for each tuple and then intersect those sets; this leads to poor performance. The sample driven schema mapping system proposed by Qian et. al. allows multiple tuples as input (but needs to be completely filled) and produces the valid minimal project join queries [18]. However, like keyword search in databases, the algorithm focuses on computing the valid minimal queries for a single row and then prunes the queries based on the subsequent rows. This solution has much less pruning power than our filter based approach as shown in Section 6. Furthermore, this approach has a high memory footprint as it needs to keep the current set of tuple trees in memory; this is validated by our experiments as well.

Our work is also related to approaches for query by output [22, 23]. These approaches take a table as input (needs to be completely filled) and return the query that produces that exact table. Their focus is not on finding the minimal project join query; they compute the minimal project join queries using naive techniques and try to find selection conditions or additional joins that yield exact equivalence. Hence, these approaches are orthogonal to our problem.

A related problem is that of expanding a few, user-provided example rows with more rows [17]. Gupta and Sarawagi propose to harness unstructured lists on the web for this task. This is quite different from our goal of discovering project join queries. The challenges they address (e.g., record segmentation, row resolution) are also quite different from those that arise in our problem.

The filter-query verification problem we map our system task to is similar to the works on shared filter evaluation [16, 13]. There are two important differences. First, the items in their problem are streaming, so the query result is not fixed and follows some distribution. In our case, the validness of candidates is fixed. This leads to a different probabilistic model. Second, previous works assume all filters are independent while we consider dependencies among the filters.

Finally, our technique for filter selection is related to identifying common subexpressions in multi-query optimization [20, 21]. The main difference is that filter selection is motivated by pruning invalid candidate queries.

8. CONCLUSION AND FUTURE WORK

In this paper, we studied the problem of discovering minimal project join queries based on an example table. The main technical challenge is to efficiently verify which queries, among a candidate set of queries, are valid answers. We formalize the problem as the filter selection and develop a novel solution. Our experiments demonstrate that our filter-based approach is much more efficient than straightforward adaptation of known techniques.

Our work can be extended in multiple directions. In this paper, we require the valid query to contain all the tuples in its output; this might sometimes lead to empty answers. How to relax this requirement is an item of future work. Typically, the user would like the valid queries to be ranked, especially in the above relaxed setting. How to rank the valid queries is also an open challenge.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[3] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *PVLDB*, 2012.

[4] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[5] J. Fan, G. Li, and L. Zhou. Interactive sql query suggestion: Making databases user-friendly. In *ICDE*, 2011.

[6] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.

[7] D. Golovin and A. Krause. Adaptive submodularity: A new approach to active learning and stochastic optimization. In *COLT*, 2010.

[8] D. Golovin and A. Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *JAIR*, 2011.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[10] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[11] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[12] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[13] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *SIGMOD*, 2008.

[14] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[15] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD*, 2007.

[16] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. In *PODS*, 2007.

[17] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 2012.

[18] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[19] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of rdbms. In *SIGMOD*, 2009.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[21] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 1988.

[22] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.

[23] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.