Efficient Rank Join with Aggregation Constraints

Min Xie
Dept. of Computer Science, Univ. of British Columbia
minxie@cs.ubc.ca

Laks V.S. Lakshmanan
Dept. of Computer Science, Univ. of British Columbia
laks@cs.ubc.ca

Peter T. Wood
Dept. of CS and Inf. Syst., Birkbeck, U. of London
ptw@dcs.bbk.ac.uk

ABSTRACT

We show aggregation constraints that naturally arise in several applications can enrich the semantics of rank join queries, by allowing users to impose their application-specific preferences in a declarative way. By analyzing the properties of aggregation constraints, we develop efficient deterministic and probabilistic algorithms which can push the aggregation constraints inside the rank join framework. Through extensive experiments on various datasets, we show that in many cases our proposed algorithms can significantly outperform the naive approach of applying the state-of-the-art rank join algorithm followed by post-filtering to discard results violating the constraints.

1. INTRODUCTION

In the last several years, there has been tremendous interest in rank join queries and their efficient processing [6, 20, 4]. In a rank join query, you are given a number of relations, each containing one or more value attributes, a monotone score aggregation function that combines the individual values, and a number \( k \). The objective is to find the top-\( k \) join results, i.e., the join results with the \( k \) highest overall scores. Rank join can be seen as a generalization of classic top-\( k \) queries where one searches for the top-\( k \) objects w.r.t. a number of criteria or features [3]. For classic top-\( k \) queries, assuming that objects are stored in score-sorted inverted lists for each feature, the top-\( k \) objects w.r.t. a monotone score aggregation function can be computed efficiently using algorithms such as TA, NRA and their variants [3]. These algorithms satisfy a property called instance optimality, which intuitively says that no algorithm in a reasonable class can perform more than a constant time better, for some fixed constant.

Ilyas et al. [6] were the first to develop an instance-optimal algorithm for rank join queries involving the join of two relations. Their algorithm employs the so-called corner-bounding scheme. Polyzotis et al. [20] showed that whenever more than two relations are joined or relations are allowed to contain multiple value-attributes, the corner bounding scheme is no longer instance optimal. They proposed a tight bounding scheme based on maintaining a “cover set” for each relation, and using this bounding scheme results in instance optimal algorithms [20, 4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.

Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

In many applications such as database queries, data mining, and optimization, constraints can add considerable value [10, 18, 14, 17] in two ways. First, they allow the relevant application semantics to be abstracted and allow users to impose their application-specific preferences on the query (or mining task) at hand. Second, constraints can often be leveraged in optimizing the query or mining task at hand. In this paper, we argue that aggregation constraints can enrich the framework of rank join queries by including such application semantics. We next illustrate this with examples.

![Figure 1: Example for rank join with aggregation constraints.](image)

Example 1. [Travel Packages] A tourist is looking for a weekend travel package comprising a visit to one museum and one restaurant. A local information website such as Yelp.com can provide us with lists of museums and restaurants along with information about their location, cost and rating, sorted by the rating (see Fig. 1). The tourist may wish to impose some constraints on the package she is looking for, e.g., the overall cost should be at most $20, the maximum cost of any visit should be less than $15, etc.

Example 2. [Course Recommendations] A university student wishes to choose courses that are both highly rated by past students and satisfy certain degree requirements. Assume each course is assigned a level, a category, and a number of credits. In order to obtain an MSc degree, students must take 8 modules, subject to the following further constraints: (i) at least 75 credits must come from courses in the “database” category; (ii) the minimum level of any course taken is 6; and (iii) the maximum number of credits taken at level 6 is 30. The requirements above can be expressed as a conjunction of aggregation constraints on a rank join.

In both examples above, we saw the utility of aggregation constraints in letting a user specify her requirements as part of a rank join query. Notice that as discussed in previous work [18, 14, 17], these hard constraints based on aggregation naturally arise in many situations. We should highlight the fact that, in our constraints, aggregation is applied to values appearing in each tuple resulting from a join, rather than in the traditional sense where aggregation is over sets of tuples. In this sense, aggregation constraints exhibit some similarity to selections applied to a join.

A natural question is how to process rank joins with aggregation constraints efficiently. A naive approach is to perform the
rank join, then apply post-filtering, dropping all results that violate the constraints, and finally report the top-k among the remaining results. We show that rank joins with aggregate constraints can be processed much faster than this post-filtering approach. First, we develop techniques for pushing constraint processing within the rank join framework, allowing irrelevant and "unpromising" tuples to be pruned as early as possible. As a result, we show that tuples that will not contribute to the top-k answers can be detected and avoided. Second, based on the observation that such an optimized algorithm still needs to access many tuples, we propose a probabilistic algorithm which accesses far fewer tuples while guaranteeing the quality of the results returned.

Specifically, we make the following contributions in this work: (1) we introduce the problem of efficient processing of rank join queries with aggregation constraints (Sec. 2), showing the limitations of the post-filtering approach (Sec. 3); (2) we analyze the properties of aggregation constraints and develop an efficient algorithm for processing rank joins with aggregation constraints, based on two strategies for pruning tuples (Sec. 4); (3) we also develop a probabilistic algorithm that terminates processing in a more aggressive manner than the deterministic approach while guaranteeing high quality answers (Sec. 5); (4) we report on a detailed set of experiments which show that the execution times of our algorithms can be orders of magnitude better than those of the post-filtering approach (Sec. 6).

2. PROBLEM DEFINITION

Consider a set \( R \) of \( n \) relations \( \{R_1, R_2, \ldots, R_m\} \), with \( R_i \) having the schema \( schema(R_i) \), \( 1 \leq i \leq n \). For each tuple \( t \in R_i \), the set of attributes over which \( t_i \) is defined is \( schema(t_i) = schema(R_i) \). We assume each relation has a single value attribute \( V \), and (for simplicity) a single join attribute \( J \). Given a tuple \( t \in R \), and an attribute \( A \subseteq schema(t) \), \( t.A \) denotes \( t.i \)'s value on \( A \). We typically consider join conditions \(jc\) corresponding to equi-joins, i.e., \( J = J \).

Let \( R' = \{R_1, R_2, \ldots, R_m\} \subseteq R \). Given a join condition \(jc\), we define \( s = \{t_1, \ldots, t_m\} \) to be a joinable set (JS) if \( t_i \in R_{j_i} \), \( i = 1, \ldots, m \), and \( j_i \neq j_j \) whenever \( i \neq j \). If \( m = n \), we call \( s \) a full joinable set (FJS), while if \( m < n \) we call \( s \) a partial joinable set (PJS). We denote by JS the set of all possible (partial) joinable sets. Furthermore, for a JS which comes from \( R' \), we define \( Rel(s) = R' \).

2.1 Language for Aggregation Constraints

Aggregation Constraints can be defined over joinable sets. Let \( AGG \in \{\text{MIN, MAX, SUM, COUNTER, AVG}\} \) be the aggregation function, and let the binary operator \( \theta \) be \( \leq, \geq \) or \( = \).

Let \( p := A \theta \lambda \) be an attribute value predicate, where \( A \) is an attribute of some relation, \( \theta \) is as above, and \( \lambda \) is a constant. We say tuple \( t \) satisfies \( p \), \( t \models p \), if \( A \subseteq schema(t) \) and \( t.A \theta \lambda \) is true. An attribute value predicate \( p \) can be the constant \( true \) in which case every tuple satisfies it. A set of tuples satisfies \( p \), \( s \models p \), if \( \forall t \in s \models p \).

We now consider aggregation constraints which are applied to tuples resulting from a join. A **primitive aggregation constraint** (PAC) is of the form \( pc := AGG(A, p) \theta \lambda \), where \( AGG \) is an aggregation function, \( A \) is an attribute (called the aggregated attribute), \( p \) is an attribute value predicate (called the selection predicate) as defined above, and \( \theta \) and \( \lambda \) are defined above. Given a joinable set \( s \), we define

\[
Eval_{pc}(s) = AGG(\{t.A \mid t \in s \land t \models p\})
\]

where we use \([\cdots]\) to denote a multiset. Then we say \( s \) satisfies the primitive aggregation constraint \( pc \), \( s \models pc \), if \( Eval_{pc}(s) \theta \lambda \) holds.

The language for (full) aggregation constraints can now be defined as follows:

- **Predicates:** \( p := true \mid A \theta \lambda \mid p \land p \)
- **Aggregation Constraints:** \( ac := pc \mid pc \land ac \mid ac := AGG(A, p) \theta \lambda \)

The meaning of a full aggregation constraint \( ac \) is defined in the obvious way, as are the notions of joinable sets satisfying \( ac \) and the satisfying subset \( R^a \) of a relation \( R \) resulting from a join.

Let \( R \) be a relation resulting from a (multi-way) join \( R_1 \bowtie \cdots \bowtie R_m \). Each tuple \( t \in R \) can also be viewed as a joinable set \( s_t \) of tuples from the relations \( R_i \). Given an aggregation constraint \( ac \), we define \( R^a \) as \( \{t \mid t \in R \land s_t \models ac\} \).

We now illustrate how the examples in the introduction can be expressed in our framework. Example 1 can be expressed by imposing the constraint \( SUM(Cost \land true) \leq 20 \land MAX(Cost \land true) \leq 15 \) on the rank join between relations Museum and Restaurant.

Example 2 can be expressed by imposing, on the rank join of 8 copies of the relation Course, the conjunction of the constraints:

\( i \) \( SUM(Credits, Category = \text{"database")} \geq 75 \),

\( ii \) \( MIN(Level, true) \geq 6 \),

\( iii \) \( SUM(Credits, Level = 6) \leq 30 \).

2.2 Problem Studied

We assume the domain of each attribute is normalized to \([0, 1]\). Let \( R \) denote the set of reals and \( S : \mathbb{R}^n \rightarrow \mathbb{R} \) be the score function, defined over the value attributes of the joined relations. Following common practice, we assume \( S \) is monotone, which means \( S(x_1, \ldots, x_n) \leq S(y_1, \ldots, y_n) \) whenever \( V \) for \( x_i \leq y_i \). To simplify the presentation, we will mostly focus on \( S \) being SUM, so given a joinable set \( s \), the overall value of \( s \), denoted as \( v(s) \), can be calculated as \( v(s) = \sum t \subseteq s \).

Furthermore, in this paper we assume that the join condition \(jc\) is equi-join, which means that given two tuples \( t_1\) and \( t_2\) from two relations, \( t_1 \bowtie t_2 \) \( \iff t_1, t_2 \in J \). For brevity we will omit the join condition \(jc\) from the join operator when there is no ambiguity.

Let \( ac \) be a user-specified aggregation constraint (which may be a conjunction of PACs) and \( jc\) be the join condition. We study the problem of **Rank Join with Aggregation Constraints (RJAC):**

**Definition 1. Rank Join with Aggregation Constraints:** Given a set of relations \( R = \{R_1, \ldots, R_m\} \) and a join condition \(jc\), let \( RS \) denote \( RS \bowtie R_m \). Now given a score function \( S \) and an aggregation constraint \( ac \), find the top-k join results \( RS^a \subseteq RS^e \), that is, \( \forall s \in RS^a \land \forall s' \in RS^e \), we have \( v(s) \geq v(s') \).

We denote an instance of the RJAC problem by a 5-tuple \( R = \{R_1, \ldots, R_m\}, jc, ac,k\). Because we are usually only interested in exactly \( k \) join results, we will discard potential join results which have the same value as the \( k^{th} \) join result in \( RS^a \); however, the proposed technique can be easily modified to return these as well if needed. Our goal is to devise algorithms for finding the top-k answers to RJAC as efficiently as possible.

We will discuss in Appendix C.3 about extending rank joins to rank outer-joins, which might be useful for some applications.
3. RELATED WORK

3.1 Rank Join and Post-Filtering

The standard rank join algorithm with no aggregation constraints works as follows [6, 20]. Given a set of relations \( R = \{R_1, \ldots, R_n\} \), assume the tuples of each relation are sorted in the non-increasing order of their value. The algorithm iteratively picks some relation \( R_i \in R \) and retrieves the next tuple \( t \) from \( R_i \). Each seen tuple \( t \) from \( R_i \) is stored in a corresponding buffer \( HR_i \), and \( t \) is joined with tuples seen from \( HR_j, j \neq i \). The join result is placed in an output buffer \( O \) which is organized as a priority queue. To allow the algorithm to stop early, the value of \( t \) is used to update a stopping threshold \( \tau \), which is an upperbound on the value that can be achieved using any unseen tuple. It can be shown that if there are at least \( k \) join results in the output buffer \( O \) which have value no less than \( \tau \), the algorithm can stop, and the first \( k \) join results in \( O \) are guaranteed to be the top-\( k \) results. We give the pseudo-code of this standard rank join algorithm in Appendix A.1.

To characterize the efficiency of a rank join algorithm, previous work has used the notion of instance optimality, proposed by Fagin et al. [3]. The basic idea is that, given a cost function \( cost \) (which is a monotone function of the total number of tuples retrieved), with respect to a class \( \mathcal{A} \) of algorithms and a class \( \mathcal{D} \) of data instances, a top-\( k \) algorithm \( A \) is instance optimal if, for some constants \( c_0 \) and \( c_1 \), for all algorithms \( B \in \mathcal{A} \) and data instances \( D \in \mathcal{D} \), we have \( cost(A, D) \leq c_0 \times cost(B, D) + c_1 \).

Instance optimality of a rank join algorithm is closely related to the bounding scheme of the algorithm, which derives the stopping threshold at each iteration. It has been shown in [20] that an algorithm using the corner-bounding scheme [6] is instance optimal if and only if the underlying join is a binary join and each relation contains one value attribute. To ensure instance optimality in the case of multiple value attributes per relation and multi-way rank join, Schnaitter et al. [20] proposed the feasible region (FR) bounding scheme. This FR bound was later improved by Finger and Polyzotis [4] using the fast feasible region (FR*) bounding scheme.

Suppose each relation has \( m \) value attributes, then the basic idea of FR/FR* bounding scheme is to maintain a cover set \( CR \) for each relation \( R \). \( CR \) stores a set of points that represents the \( m \)-dimensional boundary of the values of all unseen tuples in \( R \). Given an \( n \)-way rank join over \( R = \{R_1, \ldots, R_n\} \), to derive the stopping threshold \( \tau \), we first enumerate all possible subsets of \( R \). Then for each subset \( R' \), we derive the maximum possible join result value by joining the \( HR_i \) of relations in \( R' \) with the \( CR \) of relations in \( R - R' \). The threshold \( \tau \) is the maximum of all such values. We note that although FR/FR* bounding scheme is tight, its complexity grows exponentially with the number of relations involved [20]. Indeed, following Finger and Polyzotis [4], we mainly consider rank joins with a small number of relations.

In addition to the bounding scheme, the accessing strategy (which determines which relation to explore next) may also affect the performance of the rank join algorithm. For example, a simple accessing strategy such as round-robin often results in accessing more tuples than necessary. More efficient accessing strategies include the corner-bound-adaptive strategy [6] for binary, single value-attribute rank join and the potential adaptive strategy [4] for multi-way, multiple value-attribute rank join.

As shown in the introduction, there are many situations where it is very natural to have aggregation constraints along with rank join. While previous work on rank join algorithms has devoted much effort to optimizing the bounding scheme and accessing strategy, little work has been done on opportunities for improving runtime efficiency by using constraints that may be present in a query.

One way to handle aggregation constraints in the standard rank join algorithm is by post-filtering each join result using the aggregation constraints. It can be shown that an algorithm based on post-filtering remains instance optimal (see Appendix A.2). However, as we will demonstrate in the next section, this naive algorithm misses many optimization opportunities by not taking full advantage of the properties of the aggregation constraints, and, as we will show in Sec. 6, can have poor empirical performance as a result. This observation coincides with recent findings that instance optimal algorithms are not always computationally the most efficient [4].

3.2 Other Related Work

As described in the introduction, rank join can be seen as a generalization of classic top-\( k \) querying where one searches for the top-\( k \) objects w.r.t. a number of criteria or features [3]. Ilyas et al. [8] discussed how to incorporate binary rank join operator into relational query engines. The query optimization framework used in [8] follows System R’s dynamic programming-based approach, and in order to estimate the cost of the rank join operator, a novel probabilistic model is proposed. In [11], Li et al. extended [8] by providing a systematic algebraic support for relational ranking queries. Tsaparas et al. proposed in [22] a novel indexing structure for answering rank join queries. In this work, various tuple pruning techniques are studied to reduce the size of the index structure. In [13], Martinenghi et al. proposed a novel proximity rank join operator in which the join condition can be based on a nontrivial proximity score between different tuples. A more detailed survey of top-\( k \) query processing and rank join can be found in [7]. We note that no previous work on rank join has considered aggregation constraints.

Our work is also closely related to recent efforts on package recommendation [1, 16, 2, 19, 23]. Though some of these works [2, 23] discuss finding high-quality packages under certain aggregation constraints such as budgetary constraints, none of them provide a systematic study of aggregation constraints. A detailed comparison with these works can be found in Appendix C.3.1.

4. DETERMINISTIC ALGORITHM

We begin by illustrating rank joins with aggregation constraints.

Example 3. [Rank Join with Aggregation Constraints] Consider two relations, Museum and Restaurant, each with three attributes, Location, Cost and Rating, where Rating is the value attribute and Location is the join attribute (see Fig. 2). Assume we are looking for the top-2 results subject to the aggregation constraint \( SUM(Cost, true) \leq 20 \). Under the corner bounding scheme and round-robin accessing strategy, the algorithm will stop after accessing 5 tuples in Museum and 4 tuples in Restaurant. Note that even though the joinable set \( \{t_5, t_7\} \) has a high value, it is not a top-2 result because it does not satisfy the constraint.

<table>
<thead>
<tr>
<th>Location</th>
<th>Cost</th>
<th>Rating</th>
<th>Location</th>
<th>Cost</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Museum</td>
<td>5</td>
<td>4.5</td>
<td>Restaurant</td>
<td>10</td>
<td>4.5</td>
</tr>
<tr>
<td>Restaurant</td>
<td>15</td>
<td>5</td>
<td>Restaurant</td>
<td>20</td>
<td>4.5</td>
</tr>
<tr>
<td>Restaurant</td>
<td>15</td>
<td>4.5</td>
<td>Restaurant</td>
<td>10</td>
<td>4.5</td>
</tr>
<tr>
<td>Museum</td>
<td>5</td>
<td>3</td>
<td>Museum</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 2:** Post-filtering rank join with aggregation constraints.

Our motivation in this section is to develop efficient pruning techniques for computing rank joins with aggregation constraints fast. Thereeto, we first present a number of properties of aggregation constraints and show how these properties can be leveraged to prune seen tuples from the in-memory buffers. We then propose an efficient rank join algorithm supporting aggregation constraints that minimizes the number of tuples that are kept in the in-memory buffers, which in turn helps cut down on useless joins.
4.1 Properties of Aggregation Constraints

Let \( pc := \text{AGG}(A, p) \) \( \theta \lambda \) be a primitive aggregation constraint (PAC). In order to use \( pc \) to prune seen tuples, we first study properties of the various forms of \( pc \), i.e., for AGG \( \in \{ \text{MIN}, \text{MAX}, \text{SUM} \} \) and \( \theta \in \{ \leq, \geq \} \).

First consider the cases when AGG is \( \text{MIN} \) and \( \theta \) is \( \geq \), or AGG is \( \text{MAX} \) and \( \theta \) is \( \leq \). These cases are the simplest because \( pc \) need only be evaluated on each seen tuple individually rather than on a full joinable set. When accessing a new tuple \( t \), if \( A \in \text{schema}(t) \) and \( t \) satisfies \( p \), we can simply check whether \( t.A \lambda \theta t \) holds. If not, we can prune \( t \) from future consideration as \( pc \) will not be satisfied by any join result including \( t \). After this filtering process, all join results obtained by the algorithm must satisfy the constraint \( pc \). We name this property the direct-pruning property.

When AGG is \( \{ \text{MAX}, \text{SUM} \} \) and \( \theta \) is \( \geq \), or AGG is \( \text{MIN} \) and \( \theta \) is \( \leq \), the corresponding aggregation constraint \( pc \) is monotone.

**Definition 2. (Monotone Aggregation Constraint)** A PAC \( pc \) is monotone if \( \forall t \in R, \forall s \in JS \), where \( R \notin \text{Rel}(s) \): if \( \{ t \} \models p \) and \( \{ t \} \not\models s \neq \emptyset \), then \( \{ t \} \not\models s \models pc \).

For the case when AGG is \( \text{SUM} \) and \( \theta \) is \( \leq \), the PAC is anti-monotone. This means that if a tuple \( t \) does not satisfy \( pc \), no join result of \( t \) with any partial joinable set will satisfy \( pc \) either.

**Definition 3. (Anti-Monotone Aggregation Constraint)** A PAC \( pc \) is anti-monotone if \( \forall t \in R, \forall s \in JS \), where \( R \notin \text{Rel}(s) \): if \( \{ t \} \not\models pc \), then either \( \{ t \} \not\models s \equiv \emptyset \) or \( \{ t \} \not\models s \models pc \).

As a special case, when AGG \( \in \{ \text{MIN}, \text{MAX} \} \) and \( \theta = \equiv \), we can efficiently check whether all the joinable sets considered satisfy \( \text{AGG}(A, p) \geq \lambda \) and \( \text{AGG}(A, p) \leq \lambda \), using a combination of direct pruning and anti-monotonicity pruning.

Finally, for the case when AGG is \( \text{SUM} \) and \( \theta = \equiv \), it is easy to see that \( pc \) is neither monotone nor anti-monotone. However, as discussed in [14], \( pc \) can be treated as a special constraint in which the evaluation value of a tuple \( t \) on \( pc \), \( \text{Eval}_p(t) \), determines whether or not the anti-monotonic property holds. For example, let \( pc := \text{SUM}(A, p) = \lambda \) and \( t \) be a tuple. If \( t \models p \) and \( \{ t \} \not\models \), either \( \text{Eval}_p(t) > \lambda \) or \( \text{Eval}_p(t) < \lambda \). In the first case, the anti-monotonic property still holds. We call this conditional anti-monotonicity property \( c\text{-anti-monotone} \). Table 1 summarizes these properties.

<table>
<thead>
<tr>
<th>AGG(( \theta ))</th>
<th>( \leq )</th>
<th>( \geq )</th>
<th>( = )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>monotone</td>
<td>direct-pruning</td>
<td>monotone after pruning</td>
</tr>
<tr>
<td>MAX</td>
<td>direct-pruning</td>
<td>monotone</td>
<td>monotone after pruning</td>
</tr>
<tr>
<td>SUM</td>
<td>anti-monotone</td>
<td>monotone</td>
<td>monotone</td>
</tr>
</tbody>
</table>

Table 1: Properties of primitive aggregation constraints.

Properties like direct-pruning, anti-monotonicity and \( c\text{-anti-monotonicity} \) can be used to filter out tuples that do not need to be maintained in buffers. However, this pruning considers each tuple individually. In the next subsection, we develop techniques for determining when tuples are “dominated” by other tuples. This helps in pruning even more tuples.

4.2 Subsumption-based Pruning

Consider Example 3 again. After accessing four tuples from \( \text{Museum} \) and three tuples from \( \text{Restaurant} \) (see Figure 3), the algorithm cannot stop as it has found only one join result. Furthermore, we cannot prune any seen \( \text{Museum} \) tuple since each satisfies the constraint. However, it turns out that we can safely prune \( t_4 \) (from \( \text{Museum} \)) because, for any unseen tuple \( t' \) from \( \text{Restaurant} \), if \( t' \) could join with \( t_4 \) to become a top-2 result, \( t' \) could also join with \( t_1 \) and \( t_2 \) without violating the constraint and giving a larger score.

![Figure 3: Tuple pruning using aggregation constraints.](image-url)

The above example shows that, in addition to the pruning that is directly induced by the properties of the aggregation constraints, we can also prune a tuple by comparing it to other seen tuples from the same relation. As we discuss in the next section, this pruning can help to reduce the number of in-memory join operations. The key intuition behind pruning a tuple \( t \in R \) in this way is the following. Call a join result feasible if it satisfies all applicable aggregation constraints. To prune a seen tuple \( t \in R \), we should establish that whenever \( t \) joins with tuples (joinable set) \( s \) from other relations to produce a feasible join result \( p \), there is another seen tuple \( t' \in R \) that joins with \( s \) and produces a feasible result whose overall value is more than that of \( p \). Whenever this condition holds for a seen tuple \( t \in R \), we say \( t \) beats \( t' \). If there are \( k \) distinct seen tuples \( t'_1, \ldots, t'_k \in R \) such that each of them beats \( t \), then we call \( t \) beaten. Clearly, a seen tuple that is beaten is useless and can be safely pruned. In the rest of this section, we establish necessary and sufficient conditions for detecting (and pruning) beaten tuples among those seen. Thereto, we need the following notion of tuple domination.

**Definition 4. (pc-Dominance Relationship)** Given two tuples \( t_1, t_2 \in R \), \( pc \)-dominates \( t_1 \), denoted \( t_1 \geq_{pc} t_2 \), if for all \( s \in JS \), \( s.t. R \notin \text{Rel}(s), \{ t_1 \} \not\models s \neq \emptyset \) and \( \{ t_2 \} \not\models s \models pc \).

Intuitively, a tuple \( t_1 \) dominates another tuple \( t_2 \) from the same relation (for some given PAC \( pc \)) if for any possible partial joinable set \( s \) which can join with \( t_2 \) and satisfy \( pc \), \( s \) can also join with \( t_1 \) without violating \( pc \).

Note that the \( pc \)-dominance relationship defines a quasi-order over tuples from the same relation since it is reflexive and transitive but not anti-symmetric: there may exist two tuples \( t_1 \) and \( t_2 \), such that \( t_1 \geq_{pc} t_2 \) and \( t_2 \geq_{pc} t_1 \), but \( t_1 \neq t_2 \).

For the various PACs studied in this paper, we can characterize precisely when the \( pc \)-dominance relationship holds between tuples. The conditions depend on the type of the PAC.

First of all, consider a monotone PAC \( pc \). Because \( pc \) is monotone, given a tuple \( t_1 \), if \( t_1 \models pc \), then the join result of \( t \) with any other joinable set will also satisfy \( pc \), as long as \( t \) is joinable with \( s \). So we have the following lemma\(^4\) in the case where \( pc := \text{SUM}(A, p) \geq \lambda \).

**Lemma 1.** Let \( pc := \text{SUM}(A, p) \geq \lambda \) be a primitive aggregation constraint and \( t_1, t_2 \) be tuples in \( R \). Then \( t_1 \geq_{pc} t_2 \) iff \( t_1.J = t_2.J \) and either \( t_1 \models pc \) or \( t_1.A \geq_{pc} t_2.A \).

We can prove a similar lemma for the other monotonic aggregation constraints, where AGG is \( \text{MIN} \) and \( \theta \in \{ \leq, \equiv \} \), or AGG is \( \text{MAX} \) and \( \theta \in \{ \geq, \equiv \} \).

**Lemma 2.** Let \( pc \) be a primitive aggregation constraint in which AGG is \( \text{MIN} \) and \( \theta \in \{ \leq, \equiv \} \), or AGG is \( \text{MAX} \) and \( \theta \in \{ \geq, \equiv \} \). Given two tuples \( t_1 \) and \( t_2 \), if \( t_1 \geq_{pc} t_2 \) iff \( t_1.J = t_2.J \) and either \( t_1 \models pc \) or \( t_2 \not\models pc \).

\(^4\)Proofs are given in Appendix B.
For the anti-monotone constraint \( pc : = SUM(A, p) \leq \lambda \), we can directly prune any tuple \( t \) such that \( t \not\geq pc \); however, for tuples that do satisfy \( pc \), we have the following lemma.

**Lemma 3.** Let \( pc : = SUM(A, p) \leq \lambda \) be a primitive aggregation constraint and \( t_1, t_2 \) be two tuples such that \( t_1 \geq pc \) and \( t_2 \not\geq pc \). Then \( t_1 \geq pc \) iff \( t_1.J = t_2.J \) and \( t_1.A \leq t_2.A \).

Similarly, for the \( c \)-anti-monotone constraint \( SUM(A, p) = \lambda \), we have the following lemma.

**Lemma 4.** Let \( pc : = SUM(A, p) = \lambda \) be a primitive aggregation constraint and \( t_1, t_2 \) be two tuples such that \( t_1 \geq SUM(A, p) \leq \lambda \) and \( t_2 \not\geq SUM(A, p) \leq \lambda \). Then \( t_1 \geq pc \) iff \( t_1.J = t_2.J \) and \( t_1.A = t_2.A \).

Given the \( pc \)-dominance relationship for each individual aggregation constraint, we can now define an overall subsumption relationship between two tuples.

**Definition 5. (Tuple Subsumption)** Let \( t_1, t_2 \) be seen tuples in \( R \) and \( ac : = pc_1 \land \cdots \land pc_m \) be an aggregation constraint. We say that \( t_1 \) subsumes \( t_2 \), denoted \( t_1 \geq t_2 \), if \( t_1.J = t_2.J, t_1.V \geq t_2.V \) and, for all \( pc \in \{ pc_1, \ldots, pc_m \} \), \( t_1 \geq pc \).

Recall, the main goal of this section is to recognize and prune beaten tuples. The next theorem says how this can be done.

**Theorem 1.** Given an RJAC problem instance \( I = \{ R, S, jc, ac, k \} \), let \( T \) be the set of seen tuples from relation \( R \). Tuple \( t \in T \) is beaten iff \( t \) is subsumed by at least \( k \) other tuples in \( T \).

### 4.3 Efficient Algorithm for Top-k RJAC

Given an instance of RJAC, \( I = \{ R, S, jc, ac, k \} \), our algorithm kRJAC (see Algorithm 1) follows the standard rank join template [6, 20] as described in Section 3.1. However, it utilizes the pruning techniques developed in Sec. 4.1 and 4.2 to leverage the power of aggregation constraints.

**Algorithm 1:** kRJAC\((R, S, jc, ac, k)\)

1 \( \tau \leftarrow \infty; \)
2 \( O \leftarrow \) Join result buffer; \n3 while \( |O| < k \lor \forall (O.kthResult) \not< \tau \) do
4 \( i \leftarrow \) ChooseInput(); \n5 \( t_i \leftarrow R, next(); \)
6 if Promising\((t_i, ac) \) /* (c-)Anti-monotone pruning */
7 if \( \neg \)Prune\((t_i, HR, ac, k)\) /* Subsumption pruning */
8 \( \)ConstrainedJoin\((t_i, HR, ac, O)\);
9 \( \tau \leftarrow \) UpdateBound\((t_i, HR, ac)\);

Below, we first explore the pruning opportunities in the kRJAC algorithm using aggregation constraints (lines 6–8), and then discuss how the presence of aggregation constraints can affect the accessing strategy (line 4) and the stopping criterion (line 9).

#### 4.3.1 Optimizing In-Memory Join Processing

First of all, to leverage the (c)-anti-monotonicity property of the aggregation constraints, in line 6 of Algorithm 1, whenever a new tuple \( t_i \) is retrieved from relation \( R \), we invoke the procedure Promising (see Algorithm 2) which prunes tuples that do not satisfy the corresponding aggregation constraint. In Appendix C we describe how we can potentially filter further tuples if additional information, such as a histogram, is available for each attribute.

Let \( R_t \) be the \( k^{th} \) join result in \( RS_t \). To handle the case where all join results which have the same score as \( R_t \) need to be returned, we can change the condition \( t_i.V \geq t_2.V \) in Definition 5 to \( t_i.V > t_2.V \) and report all such results.

Let \( HR = \{ HR_1, \ldots, HR_n \} \) be the in-memory buffers for all seen tuples from each relation. Similar to previous work [6], in line 8 of Algorithm 1, when a new tuple \( t \) is seen from \( R \), we perform an in-memory hash join of \( t \) with seen tuples from all \( HR_j, j \neq i \). The idea of this hash join process is that we break each \( HR \) into hash buckets based on the join attribute value. Note that for an RJAC problem instance in which no join condition is present or \( j = 1 \) true, all seen tuples from the same relation will be put into the same hash bucket.

Algorithm 3 shows the pseudo-code for the aggregate-constrained hash join process. We first locate all relevant hash buckets from each relation (lines 1–2), then join these buckets together and finally check, for each join result found, whether it satisfies the aggregation constraints or not (lines 3–5).

**Algorithm 2:** Promising\((t_i, ac)\)

1 foreach \( pc \) in \( ac \) do
2 \( \)if \( pc \) is \( MIN(A, p) \geq = \) \( \lambda \) return \( Eval_{pc}(\{ t_i \}) \geq \lambda ; \)
3 \( \)else if \( pc \) is \( MAX(A, p) \leq = \) \( \lambda \) return \( Eval_{pc}(\{ t_i \}) \leq \lambda ; \)
4 \( \)else if \( pc \) is \( SUM(A, p) = \lambda \) \( \lor \) \( pc \) satisfies \( \text{aggregation constraints} \)
5 \( \text{return } Eval_{pc}(\{ t_i \}) \not> \lambda ; \)
6 return true

**Algorithm 3:** ConstrainedJoin\((t_i, HR, ac, O)\)

1 for \( j = 1, \ldots, i, i + 1, \ldots, n \) do
2 \( B_j \leftarrow \) LocateHashBuckets\((t_i, J, HR\);
3 foreach \( s \in B_1 \multimap \cdots \multimap B_i \multimap \{ t_i \} \multimap B_{i+1} \multimap \cdots \multimap B_n \) do
4 \( \)if \( s \) is \( ac \) and \( v(s) < (O.kthResult) \)
5 \( \)Replace \( O.kthResult \) with \( s \).

One important observation about this hash join process is that the worst case complexity for each iteration is \( O(|HR_1| \cdots |HR_m|) \times (HR_{ni} \cdots |HR_{0i}|) \), which can result in a huge performance penalty if we leave all seen tuples in the corresponding buffers. As a result, it is crucial to minimize the number of tuples retained in the \( HR \)’s. Next we will show how our subsumption-based pruning, as discussed in Section 4.2, can be used to remove tuples safely from \( HR \).

Consider a hash bucket \( B \) in \( HR \) and a newly seen tuple \( t_i \). We assume every tuple in a bucket \( B \) has the same join attribute value, so according to Theorem 1, if we find that there are at least \( k \) tuples in \( B \) which subsume \( t_i \), we no longer need to place \( t_i \) in \( HR \). This is because we already have at least \( k \) tuples in \( B \) that are at least as good as \( t_i \). We call this pruning subsumption-based pruning (SP). Furthermore, \( t_i \) does not need to be joined with \( HR_j, j \neq i \), as shown in line 7 of Algorithm 1. We will show in Sec.6 that this subsumption-based pruning can significantly improve the performance of the kRJAC algorithm.

Algorithms 4 and 5 give the pseudo-code for the subsumption-based pruning process. We maintain for each tuple \( t \) a count \( t.xcount \) of the number of seen tuples that subsume \( t \). Note that, although in the pseudo-code we invoke the Subsume procedure twice for each tuple \( t \) in the current hash bucket \( B \), the two invocations can in fact be merged into one in the implementation.

So given the basic subsumption-based pruning algorithm as presented in Algorithm 4, a natural question to ask is whether can we prune more tuples from the buffer? The answer is “yes”. Assume we are looking for the top-\( k \) join results. As we consume more tuples from the underlying relations, the value of the stopping threshold \( \tau \) may continue to decrease, which means some join results in the output buffer \( O \) may have a value larger than \( \tau \). These join results are guaranteed to be among the top-\( k \) and can be output.
Algorithm 4: Prune$(t_i, HR_i, ac, k)$
1 $B \leftarrow \text{LocateBucket}(t_i, HR_i)$;
2 foreach $t \in B$ do
3     if Subsume$(t_i, t, ac)$ then $scount \leftarrow scount + 1$;
4     if Subsume$(t_i, t, ac)$ then $scount \leftarrow scount + 1$;
5     if $scount \geq k$ Remove $t$ from $B$;
6 return $t_i, scount \geq k$;

Algorithm 5: Subsume$(t_1, t_2, ac)$
1 if $t_1, V < t_2, V$ return false;
2 $\text{Dominate} \leftarrow$ true;
3 foreach $pc$ in $ac$ do
4     switch $pc$ do
5         case $\text{MIN}(A, p) \leq (=) \lambda$ and $\text{MAX}(A, p) \geq (=) \lambda$ do
6             $\text{Dominate} \leftarrow \text{Dominate} \wedge (t_1 \models pc \text{ or } t_2 \not\models pc)$;
7         case $\text{SUM}(A, p) \geq \lambda$ do
8             $\text{Dominate} \leftarrow \text{Dominate} \wedge (t_1 \models pc \text{ or } t_1.A \geq t_2.A)$;
9         case $\text{SUM}(A, p) \leq \lambda$ do
10            $\text{Dominate} \leftarrow \text{Dominate} \wedge (t_1.A \geq t_2.A)$;
11         case $\text{SUM}(A, p) = \lambda$ do
12             $\text{Dominate} \leftarrow \text{Dominate} \wedge (t_1.A = t_2.A)$;
13 return $\text{Dominate}$;

Now suppose that the top $k$ results, for some $k' < k$, have been found so far. Then it is clear that we need only look for the next top $k - k'$ results among the remaining tuples. So when applying our subsumption-based pruning, we could revise $k$ to $k - k'$, i.e., in a hash bucket $B$ of a buffer HR, if a new tuple $t_i$ is subsumed by $k - k'$ other tuples in HR, we can safely prune $t_i$ from that buffer. We call this optimization adaptive subsumption-based pruning (ASP).

Consider the example of Figure 4. After retrieving four tuples in the Museum relation and three tuples in the Restaurant relation, we find one joinable set $\{t_1, t_5\}$ which is guaranteed to be the top-1 result, and we have pruned $t_4$. Using adaptive subsumption-based pruning, we can now also prune $t_2$ as it is subsumed by $t_1$.

<table>
<thead>
<tr>
<th>Restaurant</th>
<th>Location</th>
<th>Cost</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restaurant</td>
<td>Location</td>
<td>Cost</td>
<td>Rating</td>
</tr>
<tr>
<td>$t_1$</td>
<td>a</td>
<td>13.5</td>
<td>5</td>
</tr>
<tr>
<td>$t_2$</td>
<td>b</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>$t_3$</td>
<td>c</td>
<td>10</td>
<td>4.5</td>
</tr>
<tr>
<td>$t_4$</td>
<td>d</td>
<td>15</td>
<td>3.5</td>
</tr>
<tr>
<td>$t_5$</td>
<td>d</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Museum</th>
<th>Location</th>
<th>Cost</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Museum</td>
<td>Location</td>
<td>Cost</td>
<td>Rating</td>
</tr>
<tr>
<td>$t_1$</td>
<td>a</td>
<td>13.5</td>
<td>5</td>
</tr>
<tr>
<td>$t_2$</td>
<td>a</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>$t_3$</td>
<td>b</td>
<td>10</td>
<td>4.5</td>
</tr>
<tr>
<td>$t_4$</td>
<td>a</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>$t_5$</td>
<td>a</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4: Adaptive subsumption-based pruning.

If adaptive subsumption-based pruning is utilized, from the correctness and completeness proof of Theorem 1, we can derive the following corollary.

**Corollary 1.** At the end of each iteration of the kRJAC algorithm, the number of accessed tuples retained in memory for each relation is minimal.

In the worst case, the overhead of the rank join algorithm using subsumption-based pruning compared to one which does not perform any pruning (both algorithms will stop at the same depth $d$) will be $O(d^2 \cdot c_{dom})$, where $c_{dom}$ is the time for one subsumption test. This worst case situation will happen when no tuples seen from a relation subsume any other tuples. However, as we show in Section 6, this seldom happens, and often $d$ is very small after our pruning process.

### 4.3.2 Bounding Scheme and Accessing Strategy

When rank join involves more than two relations, the corner-bounding strategy should be replaced by a bounding strategy based on cover sets [4, 20]. As described in Section 3, for the optimal bounding scheme, to derive the stopping threshold $\tau$, we need to consider each subset $R'$ of $R$ and join the $HRs$ of relations in $R'$ with the $CRs$ of relations in $R - R'$. Because the cover set $CR_i$ of each relation $R_i$ considers only the value of an unseen item, data points in $CR_i$ can be joined with any other tuple from a tuple buffer $HR_j$, where $i \neq j$. So the presence of aggregation constraints does not affect the operations in the bounding scheme that are related to the cover set, which means when joining $CRs$ of $R - R'$, and when joining the join results of $CRs$ of $R - R'$ and join results of $HRs$ of $R'$, we don’t need to consider aggregation constraints. However, when joining $HRs$ of $R'$, in order for the derived bound to be tight, we need to make sure that each partial join result satisfies the aggregation constraints.

Similarly, for the accessing strategy that decides which relation to access a tuple from next, because the potential value of each relation is determined by the bounding scheme as discussed in [4, 6, 20], the existing accessing strategy can be directly used by taking the modified bounding scheme as described above into account.

5. **PROBABILISTIC ALGORITHM**

Our kRJAC algorithm in Section 4 returns the exact top-$k$ results. However, similar to the standard NRA [3] algorithm, this deterministic approach may be conservative in terms of its stopping criterion, which means that it still needs to access many tuples even though many of them will be eventually pruned. Theobald et al. [21] first investigated this problem and proposed a probabilistic NRA algorithm; however, their algorithm and analysis cannot be directly used to handle rank join (with aggregation constraints). In the rest of this section, we will describe a probabilistic algorithm, based on the framework of [21], which accesses far fewer tuples while guaranteeing the quality of the results returned.

Let $I = (R, S, jc, ac, k)$ be a RAC problem instance, where $R = \{R_1, \ldots, R_n\}$. The main problem we need to solve is, at any stage of the algorithm, to estimate the probability that an unseen tuple can achieve a value better than the value of the $k$th tuple in the top-$k$ buffer. This probability will clearly depend on the selectivity of the join condition $jc$ and on the aggregation constraint $ac$.

We assume the join selectivity of $jc$ over $R$ can be estimated using some existing techniques such as adaptive sampling [12]. We denote the resulting join selectivity by $\delta_{jc}(R)$, which is defined as the estimated number of join results divided by the size of the cartesian product of all relations in $R$. Given a set $s = \{t_1, \ldots, t_n\}$ of $n$ tuples, where $t_i \in R_i$, by making the uniform distribution assumption, we set the probability $P_{jc}$ of a tuple satisfying $jc$ as $\delta_{jc}(R)$. Similarly, considering each primitive aggregation constraint $ac$ in $pc$, we can also estimate the probability $P_{ac}$ of a tuple satisfying $ac$ as the selectivity of $pc$ over $R$, denoted as $\delta_{ac}(R)$. We discuss in Sec. 5.1 how $\delta_{ac}(R)$ can be estimated under common data distribution assumptions. The probability $P_{wc}$ of a tuple satisfying $ac$ can then be estimated as $P_{wc} = \prod_{ac \in ac} P_{ac}$.

Given a set of tuples $s = \{t_1, \ldots, t_n\}$, $t_i \in R_i$, assuming the join condition and the aggregation constraints are independent, we can estimate the probability $P_{wc}$ of a tuple satisfying the join condition $jc$ and the aggregation constraints $ac$ as $P_{wc} = P_{jc} \times P_{ac}$.

After some fixed number of iterations of the kRJAC algorithm, let the value of the $k$th best join result in the output buffer $O$ be $min_k$. We can estimate the probability $P_{\text{min}_k}(R)$ that an unseen tuple $t_i$ from $R_i$ can achieve a better result than $min_k$. Suppose the
current maximum value for an unseen item in $R_i$ is $\overline{v}_i$. To estimate $P_{\min}(R_i)$, similarly to [21], we assume a histogram $H_i^\theta$ for the value attribute $V$ of each relation $R_i \in R$ is available. Then using the histograms we can estimate the number $N_i$ of tuple sets \( \{t_1, \ldots, t_i, t_{i+1}, \ldots, t_n\}, t_i \in R_i \) s.t. $\overline{v}_i + \sum_{j \in \{i+1, \ldots, n\}} v(t_j) > \min_j$. We omit the obvious detail. Then the probability that $t_i$ can join with any of these $N_i$ tuple sets to become one of the top-$k$ results can be estimated as $P_{\min}(R_i) = 1 - (1 - P_{\min}(R_i))^N_i$.

Given a user specified threshold $\epsilon$, we can stop our kRJAC algorithm when $\forall \epsilon_i \in [1, \ldots, n], P_{\min}(R_i) \leq \epsilon$.

5.1 Estimating Constraint Selectivity

Given a PAC $pc ::= \text{AGG}(A, p) \theta \lambda$, and $n$ relations $R_1, \ldots, R_n$, to simplify the analysis, we assume $p = \text{true}$ and that attribute values of different relations are independent.

Consider an example of the binary RJAC problem: given a set $s = \{t_1, t_2\}$, with $t_1 \in R_1$, $t_2 \in R_2$. For the aggregation constraint $pc ::= \text{SUM}(A, \text{true}) \leq \lambda$, it is clear from Figure 5(a) that $s$ can satisfy $pc$ only when $t_1.A \leq \lambda$ and $t_2.A \leq \lambda$ fall into the gray region. We call this gray region the valid region for $pc$, denoted $VR_{pc}$. Similarly Figure 5(b) illustrates the valid region for the constraint $pc ::= \text{MIN}(A, \text{true}) \leq \lambda$.

![Figure 5: Selectivity of aggregation constraints.](image)

Based on the valid region $VR_{pc}$ for $pc$, we can estimate the selectivity of $pc$ by calculating the probability of a tuple set $s$ falling inside $VR_{pc}$.

Given a set $s = \{t_1, \ldots, t_n\}$ of $n$ tuples, $t_i \in R_i$, and given a PAC $pc ::= \text{AGG}(A, \text{true}) \theta \lambda$, if we assume $t_1.A, \ldots, t_n.A$ are independent random variables following a uniform distribution, we can calculate the closed formula for the probability $P(\text{VR}_{pc})$ of $t_1.A, \ldots, t_n.A$ falling inside $VR_{pc}$ as follows:

- If $pc ::= \text{SUM}(A, \text{true}) \leq \lambda$: $P(\text{VR}_{pc}) = \frac{\epsilon^\lambda}{\gamma}$.
- If $pc ::= \text{MIN}(A, \text{true}) \leq \lambda$: $P(\text{VR}_{pc}) = 1 - (1 - \lambda)^n$.

These facts are easily verified. Because of symmetry, for $pc ::= \text{SUM}(A, \text{true}) \geq \lambda$ and $pc ::= \text{MAX}(A, \text{true}) \geq \lambda$, the corresponding probabilities are very similar to $pc ::= \text{SUM}(A, \text{true}) \leq \lambda$ and $pc ::= \text{MIN}(A, \text{true}) \leq \lambda$ respectively: we only need to replace $\lambda$ by $1 - \lambda$ in the corresponding formulas. And for a PAC $pc$ where $\theta$ is $\epsilon$, note that the probability is 0 under continuous distributions, so in practice, we will set these probabilities to a small constant which is estimated by sampling the database.

Similarly, if we assume that each $A$ follows other distributions such as exponential distribution, similar formulas can be derived (see Appendix D).

6. EXPERIMENTS

In this section, we study the performance of our proposed algorithms based on two synthetic datasets. The goals of our experiments are to study: (i) the performance of various pruning techniques, (ii) the performance of the probabilistic method, and (iii) the result quality of probabilistic method. All experiments were done on a Intel Core 2 Duo machine with 4GB RAM and 250GB SCSI hard disk. All code is in C++ and compiled using GCC 4.2.

![Figure 6: Uniform dataset: (a), (b) $\text{SUM}(A, \text{true}) \geq \lambda$, selectivity $10^{-5}$; (c), (d) $\text{MIN}(A, \text{true}) \leq \lambda$, selectivity $10^{-5}$.](image)

We call the synthetic datasets we generated the uniform dataset and the exponential dataset. For both datasets, the join selectivity between two relations is fixed at 0.01 by randomly selecting the join attribute value from a set of 100 predefined values. The value and other attributes are set as follows. For the uniform dataset, the value of each attribute follows a uniform distribution within the range $[0,1]$; for the exponential dataset, the value of each attribute follows an exponential distribution with mean 0.5. Note that in order to ensure values from the exponential distribution fall inside the range $[0,1]$, we first uniformly pick 1000000 values from $[0,1]$, and then resample these values following the exponential distribution. Values of each attribute are independently selected. Results on the exponential dataset can be found in Appendix E.

We implemented four algorithms: (a) the post-filtering based rank join algorithm (Post Filtering); (b) the probabilistic algorithm with subsumption based pruning (SubS-Pruning); (c) the deterministic algorithm with adaptive subsumption based pruning (Adaptive SubS-Pruning); (d) the probabilistic algorithm with subsumption based pruning.

6.1 Efficiency Study

We first compare the algorithms in a binary RJAC setting. As can be seen from Figure 6, subsumption based pruning works very well for monotonic constraints. One interesting observation from Figure 6(d) is that, adaptive subsumption based pruning does not prune significantly more tuples than non-adaptive subsumption based pruning. By inspecting the dataset, we found out this is because there are $k$ tuples which subsume every other tuple, so the adaptive pruning strategy has no effect in this case.
Figure 7 shows another example of one aggregation constraint, $\text{SUM}(A, \text{true}) \leq \lambda$, under the selectivity of $10^{-5}$. As discussed in previous sections, such a constraint can result in both anti-monotonicity based pruning and subsumption based pruning. However, as can be seen from Figure 7, the anti-monotonicity based pruning can be very powerful which, in turn, renders the subsumption based pruning less effective.

We also tested our algorithms in settings where we have binary RJAC and multiple aggregation constraints (see Figure 8). For the case of $\text{SUM}(A, \text{true}) \geq \lambda$ and $\text{SUM}(B, \text{true}) \leq \lambda$ and overall selectivity is $10^{-5}$ ((a) and (b)), because of the presence of an anti-monotone constraint, many tuples can be pruned so the subsumption based algorithm outperforms the post-filtering algorithm. However, as can be seen from Figure 8(c) and (d), when the selectivity of aggregation constraints is very high and no anti-monotonic or direct-pruning aggregation constraint is present, the overhead of subsumption testing causes the execution time of the subsumption based algorithms almost to match that of the post-filtering based algorithm. As future work, we would like to study cost-based optimization techniques which can be used to help decide which strategy should be used.

![Figure 8: Uniform dataset: (a), (b) $\text{SUM}(A, \text{true}) \geq \lambda$, $\text{SUM}(B, \text{true}) \leq \lambda$, overall selectivity $10^{-5}$; (c), (d) $\text{SUM}(A, \text{true}) \geq \lambda$, $\text{SUM}(B, \text{true}) \geq \lambda$, overall selectivity $10^{-5}$.](image)

### 6.2 Probabilistic Algorithm

Similar to previous work on a probabilistic NRA algorithm, Figures 6, 7 and 8 show that our probabilistic algorithm will stop earlier than the deterministic and post-filtering based algorithms. In most experiments, the probabilistic algorithm accesses far fewer tuples from the underlying database than the other algorithms. We note that this property can be very important for scenarios where tuples are retrieved using web services [3], for example, as a monetary cost might be associated with each access and the latency of retrieving the next tuple might be very high.

In terms of the quality of results returned, as Figure 9 shows for binary RJAC with several different aggregation constraints, the value of the join results returned by the probabilistic algorithm at each position $k$ is very close to the exact solution. The percentage of value difference at each position $k$ is calculated as $\frac{\text{Difference}_k}{\text{Exact}_k}$, where $x_k$ is the exact $k^{th}$ result and $x'_k$ is the $k^{th}$ result returned by the probabilistic algorithm.

![Figure 9: Quality of the probabilistic algorithm.](image)

### 7. CONCLUSION

In this paper, we motivated the use of aggregation constraints in rank join queries. By analyzing their properties, we developed deterministic and probabilistic algorithms for their efficient processing. In addition to showing that the deterministic algorithm retains the minimum number of accessed tuples in memory at each iteration, we empirically showed both our deterministic and probabilistic algorithms significantly outperform the obvious alternative of rank join followed by post-filtering in many cases and that the probabilistic algorithm produces results of high quality.

### 8. REFERENCES

APPENDIX

A. RANK JOIN ALGORITHM

A.1 Algorithm Template for Rank Join

Algorithm 6: RJ(R, S, jc, k)
1 \( \tau \leftarrow \infty; \)
2 \( O \leftarrow \text{Join result buffer}; \)
3 while \( |O| < k \lor v(O, \text{JoinResult}) < \tau \) do
4 \( i \leftarrow \text{ChooseInput}(); \)
5 \( t_i \leftarrow R_{\text{next}}(); \)
6 \( R \leftarrow HR_i \cup \ldots \cup HR_{i1} \leftarrow t_i \leftarrow HR_{i1} \leftarrow \ldots \leftarrow HR_{i}; \)
7 \( \text{Add } R \text{ to } O, \text{ retain top-}k \text{ join results}; \)
8 \( HR.\text{insert}(t_i); \)
9 \( \tau \leftarrow \text{UpdateBound}(t_i, HR); \)

A.2 Post-Filtering based Algorithm

Given an instance optimal algorithm RJ for the rank join problem, we will show in this section that some simple modifications based on post-filtering can make the resulting algorithm, denoted RJPF, instance optimal for the RJAC problem.

RJPF differs from RJ in the following two respects: (1) Each newly generated candidate join result will be filtered using the aggregation constraints; (2) When deriving the upperbound \( \tau \) on the value that can be achieved using any unseen tuple, in order to make the upperbound \( \tau \) tight, each “potential” join result which involves an unseen item also needs to be filtered using the aggregation constraints; this can be done by assuming that each attribute of an unseen item can take an arbitrary value in the range of \([0,1]\). Similar to RJ, RJPF will stop once there are at least \( k \) join results in the output buffer \( O \) which satisfy the aggregation constraints and have value no less than \( \tau \).

Now consider a class \( \mathcal{A} \) of algorithms which access tuples in non-increasing order of their value and a class \( \mathcal{D} \) of data instances for rank join with aggregation constraints. We will show that RJPF with a round-robin accessing strategy \(^6\) is instance optimal.

Theorem 2. RJPF with round-robin accessing strategy is instance optimal within the class of algorithms \( \mathcal{A} \) and instances \( \mathcal{D} \) with an optimality ratio of \( n \), where \( n \) is the number of relations involved.

Proof. Given an RJAC instance \( I = (R, S, jc, ac, k) \), \( I \in \mathcal{D} \), let \( A \) be an arbitrary algorithm from \( \mathcal{A} \). Assume \( A \) needs to access \( p_i \) tuples from each relation \( R_i \in R \) and let \( p_{\text{max}} = \max(p_1, \ldots, p_{|R|}) \).

We claim that the algorithm RJPF will not access more than \( p_{\text{max}} \) tuples from each relation \( R_i \in R \). We prove this claim by contradiction. W.l.o.g., assume there are at least \( k \) join results satisfying \( ac \).

At the time when \( A \) stops, because RJPF cannot stop, it must be true that the \( k^{\text{th}} \) join result which satisfies \( ac \) has value smaller than the upperbound \( \tau \). However, because the bounding scheme is tight, we can find a configuration of the unseen items in \( I \) such that the current top-\( k \) join results are not the actual top-\( k \) join results, which contradicts the assumption that \( A \) can return the correct top-\( k \) join results when it stops. So RJPF need not access more than \( p_{\text{max}} \) tuples from each relation, and thus, \( \text{cost}(\text{RJPF}, I) \leq n \times \text{cost}(A, I), n = |R| \). \( \square \)

B. PROOFS FOR SECTION 4.2

B.1 Proof of Lemma 1

Proof. (If) Consider \( t_1, J = t_2, J \), if \( t_1 \parallel pc \), because \( pc \) is monotone, for any joinable set \( s \) s.t. \( R \cap \text{Rel}(s) = \emptyset \) and \( t_1 \parallel s \neq \emptyset \) we will have \( t_1 \parallel s \neq \emptyset \). Then \( \forall s \in JS, \text{ if } t_2 \parallel s \neq \emptyset \), because \( t_1, J = t_2, J \), we know \( t_2 \parallel s \neq \emptyset \), so \( t_1 \parallel s \neq \emptyset \), and \( t_1 \parallel pc \), and \( t_1 \geq pc \). On the other hand, if \( t_1, A \geq t_2, A \), for all \( s \in JS \), if \( t_1 \parallel s \neq \emptyset \) and \( t_2 \parallel s \neq \emptyset \), then \( t_1, A \geq t_2, A \), and \( t_1 \geq pc \), so again we have \( t_1 \geq pc \).

(Only if) Because \( t_1 \geq pc \), for all \( s \in JS \) s.t. \( R \cap \text{Rel}(s) = \emptyset \), \( t_1 \geq s \neq \emptyset \) and \( t_2 \parallel s \neq \emptyset \), according to the definition of pc-dominance: first, we have \( t_1 \parallel s \neq \emptyset \); second, \( t_1 \parallel s \neq \emptyset \), so \( t_1 \parallel s \neq \emptyset \).

Consider the second point where \( t_1, A < t_2, A \), consider a tuple \( t' \in R \) s.t. \( R \cap R' \neq \emptyset \) and \( t' \parallel s \neq \emptyset \), because \( t_1 \parallel s \neq \emptyset \). Because \( t_1 \parallel pc \), it is clear \( t_1 \parallel s \neq \emptyset \), however \( t_1 \parallel s \neq \emptyset \), so \( t_1 \neq pc \) and \( t_1 \neq pc \), contradiction with the assumption that \( t_1 \geq pc \). \( \square \)

B.2 Proof of Lemma 2

The proof is similar to that of Lemma 1, so omitted here for brevity.

B.3 Proof of Lemma 3

Proof. (If) Consider \( t_1, J = t_2, J, \forall s \in JS, \text{ if } t_1 \parallel s \neq \emptyset \) and \( t_2 \parallel s \neq \emptyset \), first, we know \( t_1 \parallel s \neq \emptyset \); second, we know \( t_2, A \cap \sum_{t_1} t_1 \parallel s \neq \emptyset \), because \( t_1, A \cap \sum_{t_1} t_1 \parallel s \neq \emptyset \), so \( t_1 \parallel s \neq \emptyset \), so \( t_2 \parallel s \neq \emptyset \), and \( t_1, A \geq t_2, A \).

(Only if) Because \( t_1 \geq pc \), for all \( s \in JS \) s.t. \( R \cap \text{Rel}(s) = \emptyset \), \( t_1 \parallel s \neq \emptyset \), according to the definition of pc-dominance: first, we have \( t_1 \parallel s \neq \emptyset \); second, \( t_1 \parallel s \neq \emptyset \). Consider the second point where \( t_1, A < t_2, A \), consider a tuple \( t' \in R \) s.t. \( R \cap R' \neq \emptyset \) and \( t' \parallel s \neq \emptyset \), it is clear \( t_1 \parallel s \neq \emptyset \), however \( t_1 \parallel s \neq \emptyset \), so \( t_1 \neq pc \) and \( t_1 \neq pc \), contradiction with the assumption that \( t_1 \geq pc \). \( \square \)

B.4 Proof of Lemma 4

The proof is similar to that of Lemma 3, so is omitted here for brevity.

B.5 Proof of Theorem 1

Proof. (If) Let \( t \) be subsumed by \( k \) tuples \( t_1, \ldots, t_k \), let "best" full joinable set which contains \( i \), i.e. \( t \in s, s \parallel ac \) and \( s' \parallel ac \) if \( s \in JS \), if \( t \in s' \parallel ac \), we have \( v(s) \geq v(s') \). Now assume we cannot find \( k \) other full joinable sets \( s_1, \ldots, s_k \), s.t. \( s_i \parallel ac, t \not\in s_i \), and \( v(s_i) \geq v(s) \). Because \( t \parallel s \), \( s = \ldots = s_k \parallel ac \) and furthermore \( t, V \geq t, V \), which means \( v(s) = v(s) \). So this contradicts with the assumption that we cannot find \( k \) other full joinable sets that don't contain \( t \).

(Only) If \( t \) cannot be subsumed by \( k \) other seen tuples, there must be a configuration of the remain unseen tuples such that \( t \) can join with some unseen tuples to become the top-\( k \) results. Without loss of generality, assume \( t \) cannot be subsumed by any other tuples, and we will prove that \( t \) may become the top-1 result.7 And to simplify the presentation, we consider the

\(^7\)For the general case where \( t \) is subsumed by \( k' \) tuples, where \( k' < k \), we can first remove the \( k' \) tuples that dominate \( t \), and prove \( t \) can become the top-1 result among the remaining ones.
join performed here is a binary join, but note that the result here can be easily extended the case of multi-way rank join.

Assume to the contrary that for all possible unseen tuple $t'$ from another relation $R'$ s.t. $\{t\} \Rightarrow \{t\} \neq \emptyset$, there exists a tuple $t_i \in T$, s.t. $t_i \neq t$, $\{t_i\} \Rightarrow \{t\} \neq \emptyset$, $\{t_i\} \Rightarrow \{t\} \Rightarrow ac$, and either $\{t\} \Rightarrow \{t'\} \neq ac$ or $\nu(\{t\} \Rightarrow \{t'\}) \geq \nu(\{t\} \Rightarrow \{t\})$.

First of all there must exist $t_i \in T$, $t_i \neq t$ and $t_i \neq t_i$, $J = J$ is true, because otherwise, we can assume there is only one unseen tuple $t'' \in R'$, s.t. $t'' \neq J$ and $\{t''\} \Rightarrow \{t\} \Rightarrow ac$. Then obviously, $t$ will join with $t'$ to become top-1 result which contradicts with the assumption. So we assume that there exist unseen tuples in $T$ which have the same join attribute value as $t$.

Because no tuple in $T$ can subsume $t$, for each $t_i \in T$, $t_i \neq t$ and $t_i \neq t_i$, $J = J$, it must be true that either $t_i \Rightarrow V < t.V$ or there exists $pc$ in ac s.t. $t_i \not\prec pc t$, $t$. We first consider each tuple $t_j \in T$ s.t. $t_i \Rightarrow V \geq t.V$ and exists $pc$ in ac s.t. $t_i \not\prec pc t_i$.

Let $ac'$ be the set of primitive aggregation constraints in ac s.t. if $pc \in ac'$, then $\exists t_j \in T$, $t_j \neq t$, $J = J$, $t_i \Rightarrow V \geq t.V$ and $t_i \not\prec pc t$. And for each $pc \in ac'$, we set $T^{ac} = \{t_j \in T \mid t_j \not\prec pc t\}$. Thus we have $pc \in ac s.t. t_i \not\prec pc t_i$.

In the following, we consider each $pc \in ac'$ separately and try to find a configuration of an unseen tuple $t' \in R'$ s.t. $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.

1. $pc := SUM(A, p) \geq A$.
   a. If $t \Rightarrow pc$, because $\forall t_j \in T^{ac}, t_j \not\prec pc t$, from Lemma 1, it must be true that $t_i \Rightarrow pc$. Assume $\forall t_j \in T^{ac}, t_i \not\prec A$, then we set $t.A > \lambda - A$. So $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.
   b. If $t \Rightarrow \neg pc$, because $\forall t_j \in T^{ac}, t_j \not\prec pc t$, from Lemma 1, it must be true that $t.A > \lambda$. We can set $t.A = \lambda - t.A$, then $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.

2. $pc := MIN(A, p) \leq \lambda$ or $pc := MAX(A, p) = \lambda$.
   a. Because $\forall t_j \in T^{ac}, t_j \not\prec pc t$, from Lemma 2, we know that it must be true that $t \Rightarrow pc$ and $t_i \Rightarrow \neg pc$, because otherwise $t_i$ will dominate $t$. So we simply set the value of $t'$ on attribute $A$ s.t. $t.A > \lambda$, then $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.

3. $pc := MAX(A, p) \geq \lambda$ and $pc := MAX(A, p) = \lambda$. Similar to case 1, except that we set $t.A < \lambda$.

4. $pc := SUM(A, p) \leq \lambda$.
   a. Because $\forall t_j \in T^{ac}, t_j \not\prec pc t$, from Lemma 3, we know that $t.A < \lambda$, so we can set $t.A = \lambda - t.A$, then $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.

5. $pc := SUM(A, p) = \lambda$.
   We only need to consider the case where $\forall t_j \in T^{ac}, p < \lambda$ and $t_i \not\prec SUM(A, p) \leq \lambda$, as otherwise, $t$ and/or $t_i$ will be filtered using the c-anti-monotone property. Then from Lemma 4, we know $t_i \not\prec A$, so we can set $t.A = \lambda - t.A$, $\forall t_j \in T^{ac}, \{t\} \Rightarrow \{t'\} \Rightarrow pc$ whereas $\{t_i\} \Rightarrow \{t'\} \Rightarrow pc$.

Consider the unseen item $t' \in R'$ is configured as in the above process, now for each tuple $t_i \in T$ s.t. $t_i \neq t$ and $t_i \neq t_i$, $J = J$, $V < t.V$, it is clear, either $\{t\} \Rightarrow \{t'\} \Rightarrow ac$ or $\nu(\{t\} \Rightarrow \{t'\}) < \nu(\{t\} \Rightarrow \{t\})$, so $\{t\} \Rightarrow \{t'\}$ is still a better join result.

So to sum, we find an unseen item $t'$ from $R'$ s.t. $\forall t_j \in T$, $t_i \neq t$, $t_i \neq t_i$, we will have $\{t\} \Rightarrow \{t'\} = \emptyset$, or $\nu(\{t\} \Rightarrow \{t'\}) < \nu(\{t\} \Rightarrow \{t\})$, or exists $pc$ in ac, $\{t\} \Rightarrow \{t'\} \Rightarrow ac$ and $\{t\} \Rightarrow \{t'\} \Rightarrow pc$.

So we find a configuration of unseen items such that $t$ will become top-1 result, this contradicts the assumption.

Note that we don’t need to consider direct-pruning aggregation constraints as they will always hold after the filtering process.

C. DISCUSSION

In this section, we discuss several extensions to this work and briefly sketch the main ideas for realizing the extensions.

C.1 Multi-Constraints per Attribute

Given a set of relations $R = \{R_1, \ldots, R_n\}$, consider an aggregation constraint $ac = pc_1 \land \ldots \land pc_n$ imposed on the same attribute $A$. Then before initiating any algorithm, we need first to ensure that the $n$ PACs in ac are satisfiable, which means there exists a joinable set $s$ from $R$, such that all the $n$ PACs are satisfied by $s$ from $R$. E.g., if $pc_1 := MIN(A, true) \geq 20$ and $pc_2 := MAX(A, true) \leq 10$, we should identify that the two aggregation constraints are not satisfiable by any joinable set. Some previous work [18] on satisfiability of aggregation constraints can be leveraged to handle this problem.

In addition, the presence of multiple constraints on one attribute can also affect our necessary condition of determining whether a tuple $t$ can be pruned. Recall in Theorem 1, a tuple $t$ is beaten only if $t$ is subsumed by at least $k$ other tuples in the corresponding seen tuple buffer. Now consider a binary RJAC problem and assume we are looking for the top-1 result. Let $A \in schema(t)$ be an attribute and assume we have two PACs, $pc_1 := MIN(A, true) \leq 100$ and $pc_2 := MAX(A, true) \geq 10$. Assume further that $t_1$ and $t_2$ are tuples from the same relation s.t. $t = t_1, V = t_2, V, A = 50, t_1.A = 120$ and $t_2.A = 5$. It is clear that $t \Rightarrow pc_1, t \Rightarrow pc_2, t_1 \not\prec pc_1$ and $t_2 \not\prec pc_2$. So we can derive from our pc-dominate relationship that $t$ cannot be subsumed by either $t_1$ or $t_2$. However, consider an arbitrary tuple $t'$ from the other relation which can join with $t$ and so can also join with $t_1$ and $t_2$. Because either $t.A \leq 100$ or $t.A \geq 10$ must be true, it means either $t.A, t_i$ or $t'.A$ can satisfy both aggregation constraints. So for all possible $t'$, we will always have a joinable set $s$ which satisfy all aggregation constraints and has at least the same value as the the join result of $t$ and $t'$, which means we can prune $t$.

C.2 Further Optimization using Background Information

The properties of aggregation constraints discussed in Section 4 are general rules which mean they can be applied to all possible data instances. However, this generality also limits their pruning ability as the information about the data itself is ignored.

In practice, because data are often stored in a database, simple statistics like boundary values or a histogram of each attribute often come for free, and this information can be used to reason whether a tuple $t$ can potentially satisfy an aggregation constraint or not.

Given a relation $R$ and an attribute $A$, assume statistical information about the values of $A$ is stored in a single dimensional histogram [9], denoted $R.A_h$, or $R.A_h$ if there is no ambiguity. An equi-depth histogram [9], in which tuples are divided into a number of buckets and each bucket holds statistical information for the same number of tuples, is a popular choice among single dimensional histograms.

For a histogram $H_A$, we denote the number of buckets in $H_A$ as $|H_A|$, and let the set of buckets in $H_A$ be $\{B_1, \ldots, B_{|H_A|}\}$. For each bucket $B$, the number of tuples in $B$ is $|B|$, and attribute values of these tuples are bounded by $B.lb$ and $B.ab$. We assume a copy of $H_A$ can be obtained when performing top-k query processing, so the value of $B_i$ can be easily adjusted as we access new items.

Given the histogram information for each relation, we can easily derive rules for determining whether a tuple $t$ can potentially satisfy the aggregation constraint ac.

Considering a PAC $pc := AGG(A, p) \theta \lambda$ in which $AGG \in [MIN, MAX]$ and a tuple $t$, then $t$ can potentially satisfy $pc$ if there exists a relation $R$ s.t. for the corresponding histogram $H_R$ of $R$, we
can find a non-empty bucket which contains a value that satisfies \( pc \).

Similarly, for a PAC \( pc \) in which \( AGG = SUM, \theta \in \{ \leq, \geq \} \) and a tuple \( t \), we can check the histograms of the remaining relations \( R' = R - Rel(t) \) and verify whether the maximum/minimum sum of the tuples from \( R' \) can satisfy \( pc \) along with \( t \).

For a PAC \( pc := AGG(A, p) = \lambda \), note that the problem of determining whether a subset of values from the histograms can add up to a constant \( \lambda \) resembles the classical Subset Sum problem [5], which means that under query complexity, the problem is NP-Complete. However, in practice, the total number of relations to be joined is often bounded by a small number, and under data complexity, this problem is polynomial-time solvable. Let \( R' = R - Rel(t) \) be the set of remaining relations for a tuple \( t \), we can check whether \( t \) may potentially satisfy \( pc \) by enumerating all possible combinations of entries in the corresponding histograms of relations in \( R' \).

So in Algorithm2 of Section 4, in addition to pruning tuples using the basic aggregation constraint properties, if histogram information is given, we can also check for each PAC \( pc \in ac \) whether \( t \) may potentially satisfy \( pc \) or not.

C.3 Rank Outer-joins with Aggregation Constraints

For some applications like package recommendation [23, 2], users may benefit from further flexibility in the schema of the rank join result, e.g., we may want to join as many tuples together as possible as long as the overall cost of all tuples does not exceed the cost budget.

For such requests, we need to extend the current semantics of the rank join operator to rank outer-join. We note that some of the pruning techniques proposed in this work, such as direct pruning, can be simply applied to outer rank-join with aggregation constraints. Other pruning techniques, such as subsumption-based pruning, can also be adapted to prune tuples which come from the same relation. However, a systematic study of aggregation constraints over rank outer-joins is beyond the scope of the current paper, so we will leave it to be explored as future work.

Next we will give a summary of previous work on package recommendation.

C.3.1 Related Work on Package Recommendation

Angel et al. [1] proposed an interesting way of finding top-\( k \) tuples of entities. Examples of entities include cities, hotels and airlines. In this work, they query documents using keywords in order to determine entity scores. This work leverages the existing rank join algorithm, but it does not consider aggregation constraints that a user might impose on the join results.

In CourseRank [16], items need to satisfy complex constraints. The problem is motivated by a course planning application for students, where constraints are of the form “take \( k_i \) courses from \( S_j \)”, where \( k_i \) is a non-negative integer and \( S_j \) is a set of courses. Each course in this system is associated with a score which is calculated using an underlying recommendation engine. Given a number of constraints of the form above (and others), the system finds a minimal set of courses that satisfies the requirements and has the highest score, where one course can satisfy one or more requirements. Later work [15] extends CourseRank with prerequisite constraints, and proposes several approximation algorithms that return high-quality course recommendations which satisfy all the prerequisites. However, [15, 16] are orthogonal to the problem studied here and do not consider aggregation constraints on join results. Furthermore, they do not consider the scenario where data can only be accessed in non-increasing value order.

Motivated by online shopping applications, [19] studies the problem of recommending “satellite items” related to a given “central item” subject to a cost budget. The aggregation constraints considered in this work are quite restricted, and item values are not taken into account.

Budgetary constraints in package recommendation are studied in [2] and [23]. In [2], a novel framework is proposed to automatically generate travel itineraries from online user-generated data like picture uploads, and the authors formulate the problem of recommending travel itineraries of high quality where the travel time is under a given time budget. The algorithm studied in [2] can find high quality packages, but it needs to access all items in the system. In [23], an alternative framework is proposed to find high quality packages under budgetary constraints. The algorithms proposed in [23] can optimize the number of items accessed while guaranteeing the quality of the top-\( k \) packages returned.

D. Selectivity Estimation under Exponential Distribution

If we assume that each \( \tau_i A \) follows an exponential distribution with mean \( \frac{1}{\mu} \), which is relevant when the values of attribute \( A \) come from a Poisson process, we can derive the following formulas. Selectivity for other aggregation functions can be derived similarly.

- If \( pc := SUM(A, true) \leq \lambda \): \( P(\text{VR}_{pc}) = 1 - e^{-\mu \lambda} - \sum_{i=2}^{n} \frac{(\mu \lambda)^{i-1}}{(i-1)!} e^{-\mu \lambda} \).
- If \( pc := MIN(A, true) \leq \lambda \): \( P(\text{VR}_{pc}) = (1 - e^{-\mu \lambda})^n - (e^{-\mu \lambda})^n \).

To prove the correctness of the above two formulas, consider \( x_1, \ldots, x_n \) as \( n \) independent random variables, where each \( x_j \) follows the exponential distribution with the probability density function of \( p(x) = \mu e^{-\mu x} \) and value range of \([0, 1]\).

**Lemma 5.** \( P(\sum_{i=1}^{n} x_i \leq \lambda) = 1 - e^{-\mu \lambda} - \sum_{i=2}^{n} \frac{(\mu \lambda)^{i-1}}{(i-1)!} e^{-\mu \lambda} \).

**Proof.** We prove the lemma by induction. For the base case, consider \( n = 1 \), then:

\[
P(x_1 \leq \lambda) = \int_{0}^{\lambda} \mu e^{-\mu x_1} dx_1 = -e^{-\mu x_1}|_{0}^{\lambda} = 1 - e^{-\mu \lambda}
\]

Now assume when \( n = i \), \( P(\sum_{j=1}^{i} x_j \leq \lambda) = 1 - e^{-\mu \lambda} - \sum_{j=2}^{i} \frac{(\mu \lambda)^{j-1}}{(j-1)!} e^{-\mu \lambda} \), then for \( n = i + 1 \), we have \( P(\sum_{j=1}^{i+1} x_j \leq \lambda) = P(\sum_{j=1}^{i} x_j \leq \lambda - x_{i+1}) \), so we have:

\[
P(\sum_{j=1}^{i+1} x_j \leq \lambda) = P(\sum_{j=1}^{i} x_j \leq \lambda - x_{i+1})
\]

\[
= \int_{0}^{\lambda} (\mu e^{-\mu x_{i+1}})(1 - e^{-\mu (\lambda - x_{i+1})}) - \sum_{j=2}^{i} \frac{(\mu \lambda)^{j-1}}{(j-1)!} e^{-\mu (\lambda - x_{i+1})} dx_{i+1}
\]

\[
= \int_{0}^{\lambda} \mu e^{-\mu x_{i+1}} - \mu e^{-\mu x_{i+1}} - \sum_{j=2}^{i} \frac{(\mu \lambda)^{j-1}}{(j-1)!} e^{-\mu x_{i+1}} dx_{i+1}
\]

\[
= 1 - e^{-\mu \lambda} - \mu e^{-\mu \lambda} - \sum_{j=2}^{i} \frac{(\mu \lambda)^{j-1}}{(j-1)!} e^{-\mu \lambda}
\]

\[
= 1 - e^{-\mu \lambda} - \sum_{j=2}^{i} \frac{(\mu \lambda)^{j-1}}{(j-1)!} e^{-\mu \lambda}
\]

**Lemma 6.** \( P(\text{min}, x_i \leq \lambda) = (1 - e^{-\mu \lambda})^n - (e^{-\mu \lambda})^n \).
E. EXPONENTIAL DATASET EXPERIMENT

For the exponential dataset, first consider the binary RJAC setting. From Figure 10, similar to the uniform dataset, subsumption based pruning works very well for monotonic constraints. And for Figure 10(d), the adaptive subsumption based pruning prunes the same number of tuples compared with the non-adaptive subsumption based pruning. The reason is the same as for the uniform dataset, there are \( k \) tuples which subsume every other tuple, so the adaptive pruning strategy has no effect in this case.

Consider the settings where we have binary RJAC and multiple aggregation constraints (see Figure 12). For the case of \( SUM(A, true) \geq \lambda \) and \( SUM(B, true) \leq \lambda \) and overall selectivity \( 10^{-5} \) ((c) and (d)), because of the presence of an anti-monotone constraint, many tuples can be pruned so the subsumption based algorithm outperforms the post-filtering algorithm. However, the performance gain is not great for this case simply because the presence of \( SUM(A, true) \geq \lambda \) makes all algorithms stop early with the estimated \( \lambda \). In Figure 12(a) and (b), consider two aggregation constraints \( SUM(A, true) \geq \lambda \) and \( SUM(B, true) \geq \lambda \). Similar to the above case with one \( SUM(A, true) \geq \lambda \), because the estimated \( \lambda \) value for the given selectivity is not very tight, all algorithms stop very soon after accessing a few hundred tuples.

Figure 11 shows the experimental results for \( SUM(A, true) \leq \lambda \), with selectivity of \( 10^{-5} \). Similar to the uniform dataset, the antimonotonicity based pruning can still be very powerful; however, from Figure 11, it can be observed that our proposed algorithm for this case may incur slightly more time cost compared with the post-filtering based algorithm. The reason for this is because the value \( \lambda \) derived from the selectivity using our estimation is not very tight for the exponential dataset, and all algorithms stop after accessing a few hundred tuples in total, which means there are not many tuples left in each hash bucket for the algorithm to prune, and our algorithms still need to pay the penalty of doing some extra subsumption checking.

![Figure 10: Exponential dataset](image)

![Figure 11: Exponential dataset](image)

![Figure 12: Exponential dataset](image)