Ad-hoc Holistic Ranking Aggregation

by

Mina Saleeb

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

Data exploration is considered one of the major processes that enables the user to analyze massive amount of data in order to find the most important and relevant information needed. Aggregation and Ranking are two of the most frequently used tools in data exploration. The interaction between ranking and aggregation has been studied widely from different perspectives. In this thesis, a comprehensive survey about this interaction is studied. Holistic Ranking Aggregation which is a new interaction is introduced. Finally, various algorithms are proposed to efficiently process ad-hoc holistic ranking aggregation for both monotone and generic scoring functions.

Acknowledgements

I would like to extend my gratitude to my advisor, Ihab Ilyas, for his support, guidance, ideas, and expertise in helping me complete this thesis. Ihab gave me the opportunity to come to University of Waterloo to pursue a graduate degree. He also directed me developing a career.

I am also grateful to my mother, Seham, who always sacrifices to make me able to achieve all my dreams. She is the main reason for who I am now.

My lovely wife, Nardine, supports me to the max and I could not do it without her.

Special thanks to my colleagues; George Beskales and Mohamed Soliman for their help.

Dedication

This is dedicated to my mother, my late father, and my wife.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data exploration is considered one of the major processes that enables the user to analyze massive amount of data. It guides users through the large bulks of data to find the most important and relevant information. Aggregation and Ranking are two well known tools in data exploration. Aggregation (grouping) divides data objects into groups based on a grouping criterion. The most used grouping criterion is the equality where the members of a group have exactly the same values of the grouping attributes. *Group By* construct is the main mechanism used in traditional relational database management systems to support equality aggregation.

Consider a real estate company that maintains a database of houses and lands. The database contains information about the premises, their locations, and their prices. The company is interested in performing certain queries over this database.

Example 1.1 [Equality Aggregation Example]. *The company is interested in presenting the average renting of the apartments in all neighborhoods. The information about the apartments, their locations, and their recent known rent are maintained in the table ApartmentForRent. Figure 1.1 shows a SQL-like formulation to this problem.*

```
SELECT Apt.Neighborhood, AVG(Apt.RentPerMonth)
FROM ApartmentForRent Apt
GROUP BY Apt.Neighborhood
```

Figure 1.1: Equality Grouping SQL-like query

In the previous example, the *GROUP BY* construct is used to divide the apartments into groups. The apartments in the same neighborhood belong to the same group. Then, the aggregating function *AVG* is used to calculate the average of the rent per month for each group.

A natural extension to equality aggregation is the similarity aggregation. In this type, the grouping criterion is based on similarities such that the members of a group are similar to each other in a certain aspect but different from other groups members.

Example 1.2 [Similarity Grouping Example]. *A group of investors hired the real estate company to find some candidate lands to build a large shopping center. They prefer the land with the maximum population within radius 50 miles. The information about the neighborhoods is maintained in the database in the table Neighborhood, while the information about the available lands for sale is maintained in the table LandForSale. Consider CLUSTER BY construct that performs similarity grouping based on a given condition. Also, the function DISTANCE that returns the distnce between two locations. Figure 1.2 formulates this problem in SQL-like query.*

```
SELECT L.Location, SUM(N.Population)
FROM Land L, Neighborhood N
CLUSTER BY N.CenterLocation
ON DISTANCE(L.Location, N.CenterLocation) < 50
```

Figure 1.2: Similarity Grouping SQL-like query

In the previous example, the *CLUSTER BY* construct is used to divide the neighborhoods into groups based on the given condition. The neighborhoods in the same groups are all far from the location by less than 50 miles. Then, the aggregating function *SUM* is used to calculate the summation of the neighborhoods population for each group.

Any equality aggregation can be represented as a similarity aggregation where the clustering condition is an equality condition. Figure 1.3 rewrites example 1 in terms of similarity aggregation.

```
SELECT Apt.Neighborhood, AVG(Apt.RentPerMonth)
FROM ApartmentForRent Apt
CLUSTER BY Apt.Neighborhood
ON EQUALITY(Apt.Neighborhood)
```

Figure 1.3: Similarity Grouping SQL-like query

A natural extension to similarity aggregation is the holistic aggregation. In this type, the grouping criterion may not exist or exist in any form.

Example 1.3 [Holistic Grouping Example]. *Another group of investors hired the real estate company to find candidate lands to build three resident buildings. Their primary* *criterion is the locations that have the minimum inter-distance. Consider HOLISTIC BY construct that perform holistic grouping. Figure 1.4 shows a SQL-like formulation to the problem.*

```
SELECT DISTANCE(L.Location) AS TotalDistance
FROM LandsForSale L
HOLISTIC BY L.Location SIZE 3
```

Figure 1.4: Holistic Grouping SQL-like query

In the previous example, the *HOLISTIC BY* construct is used to divide the locations into all possible groups of size three. Then, the aggregating function *DISTANCE* is used to calculate the inter-distance of the locations for each group.

Any similarity aggregation can be represented as a holistic aggregation. Figure 1.5 rewrites example 2 in terms of similarity aggregation.

```
SELECT L.Location, SUM(N.Population)
FROM Land L, Neighborhood N
HOLISTIC BY L.Location, N.Population SIZE MAX
CONSTRAINT EQUALITY(Apt.Neighborhood)
AND DISTANCE(L.Location, N.CenterLocation) < 50
```

Figure 1.5: Holistic Grouping SQL-like query

In the previous example, the *HOLISTIC BY* construct is used to create groups of pairs of the location and neighborhood such that two constraints are satisfied. The first one is that all the pairs in the same group have the same location, while the second one is that the distance between the location and the center of the neighborhood in the same pair is less than 50 miles.

As stated above the equality aggregation is a special type of the similarity aggregation which is considered a special type of the holistic aggregation. Figure 1.6 shows the venn diagram for the aggregation types. The equality aggregation is widely used in current database management systems. The similarity aggregation is widely studied in machine learning, data mining, pattern recognition, image processing, and bio-informatics. The holistic aggregation is the main focus of this thesis.

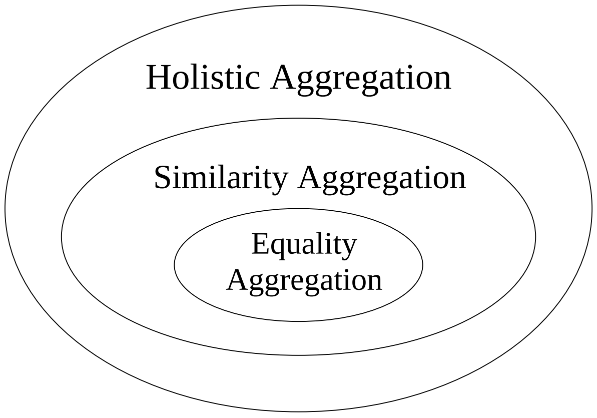


Figure 1.6: Aggregation types venn diagram

Holistic aggregation enriched the aggregation semantics which added more complexity to the aggregation computation. Thus solving it using the current techniques will not be efficient.

On the other hand, ranking presents the most important results in an efficient way based on a ranking criterion. It adds to the complexity of the problem. Not only groups are needed to be computed, but also they should be ranked according to the ranking criterion.

The interaction between ranking and aggregation has been studied widely from different perspectives. In this thesis, Holistic Ranking Aggregation is studied.

The naive approach to solve the holistic ranking aggregation problem is materializing all possible groups then sorting the results using the required ranking criterion. However, that is not efficient since there are exponential number of groups, out of which, only small k required.

1.1 Problem Statement

Holistic Ranking Aggregation *HRA* finds the top-k groups where these groups are created based on the scoring function over the group members satisfying all the given constraints. Without loss of generality, the scores of the members are assumed to be between 0 and 1. The higher score values are assumed to be preferred. Also, the data and the queries are assumed to be ad-hoc; they are only known at the query time. First, formal definitions for the main terms are introduced. Then, the *HRA* is formally defined.

Definition 1.1. **[Group** g**]** *A group is a set of distinct tuples* $g = \{t_1, t_2, ..., t_m\}$.

Definition 1.2. **[Group-based Scoring Function** $\mathcal{F}$**]** *Let* $R = \{t_1, t_2, ..., t_n\}$ *be a set of tuples,* $g = \{t_1^{'}, t_2^{'}, ..., t_m^{'}\}$ *be a group of* m *tuples where* $g \subseteq R$*, and* $\mathcal{F}$ *be the scoring function defined on* R*.* $\mathcal{F}$ *is a Group-based scoring function if it is evaluated over all the group members and does not depend on other tuples* $\mathcal{F}(g) = F(t_1^{'}, t_2^{'}, ..., t_m^{'})$.

This class of scoring function includes many typical aggregation functions. For example: summation, average, weighted average, standard deviation, summation of pairs difference, and many others. The following is an example for a function that does not belong to this class.

Example 1.4. *Proxy server intercepts the requests from the clients and serves the clients with the requested object. To reduce the user-perceived latency of the object requests, the proxy server fetches the objects and stores them in advance, hoping that the prefetched objects are likely to be accessed in the near future. Wu et al [19] formalizes this problem as finding a subset* S' *of size* n *from the entire collection of objects* S*, such that*

$$S' = \underset{S' \subset S, |S'| = n}{argmax} \left[\frac{\sum\limits_{i \in S} p_i f(i) + \sum\limits_{j \in S'} p_j (1 - f(j))}{\sum\limits_{i \in S} \frac{s_i}{l_i} f(i) + \sum\limits_{j \in S'} \frac{s_i}{l_i} (1 - f(j))}\right]$$

Definition 1.3. **[Group Constraint** c**]** *A group constraint can be either partial or holistic. A partial constraint is a constraint that can be used to prune out the group by knowing only a subset of the groups' members. A holistic constraint is a constraint that can be used to prune out the group only if all the group members are known.*

For example, a constraint of the existence of certain elements is a partial constraint, while a constraint on the inter-distance between all the members is a holistic constraint.

Equality aggregation can be defined as a holistic aggregation with a partial constraint where all the group members have the same values of the given attributes. Also, similarity aggregation can be defined as a holistic aggregation with partial constraint where all the group members are similar in given aspect.

Definition 1.4. **[Holistic Ranking Aggregation Query (**$HRAm$**)]** *Let* $R = \{t_1, t_2, ..., t_n\}$ *be a set of tuples,* $\mathcal{F}$ *be a group-based scoring function, and* C *be a set of group constraints. The query* $HRAm(R, \mathcal{F}, m, k, C)$ *computes the* k *groups of size* m *with the highest* $\mathcal{F}$ *values and satisfying constraints in* C.

The definition can be extended to support variable size groups.

Definition 1.5. **[Holistic Ranking Aggregation Query (**$HRAM$**)]** *Let* $R = \{t_1, t_2, ..., t_n\}$ *be a set of tuples,* $\mathcal{F}$ *be a group-based scoring function,* M *be a set of group sizes, and* C *be a set of group constraints. The query* $HRAM(R, \mathcal{F}, M, k, C)$ *computes the* k *groups of any size in* M *with the highest* $\mathcal{F}$ *values and satisfying constraints in* C.

1.2 Technical Challenges

- Due to the exponential state space generated from the holistic ranking aggregation problem, it is inefficient to be completely generated. Thus, rules should be defined to prune the generation of unqualified groups
- Since multiple techniques can be used to generate groups, a sound rule should be defined to guarantee their correct global ranks

1.3 Contribution

The main contributions of this dissertation are as follows:

- The Holistic Ranking Aggregation *HRA* problem is formalized.
- A comprehensive survey about the interaction between ranking and aggregation is introduced.
- Algorithms are proposed to efficiently process ad-hoc holistic ranking aggregation for both monotone and generic scoring functions.

1.4 Summary and Outline

The rest of this dissertation is organized as follows. Chapter 2 introduces the classification of ranking aggregation processing techniques. It studies the different interactions between ranking and aggregation. Chapter 3 studies the holistic ranking aggregation problem when the scoring function is monotone. Chapter 4 studies the holistic ranking aggregation problem when the scoring function is generic. Appendix A applies the idea of dominance studied in section 3.2.2 to the rank-join problem.

Chapter 2

Background and Related Work

Aggregation or Grouping means the division of tuples or data objects into groups based on a certain grouping criteria. Ranking (Top-k) gives the most important k results. The interaction between ranking and grouping has been studied from different perspectives. Figure 2.1 depicts the classifications of Ranking Aggregation processing techniques. In the next sections, each dimension is studied.

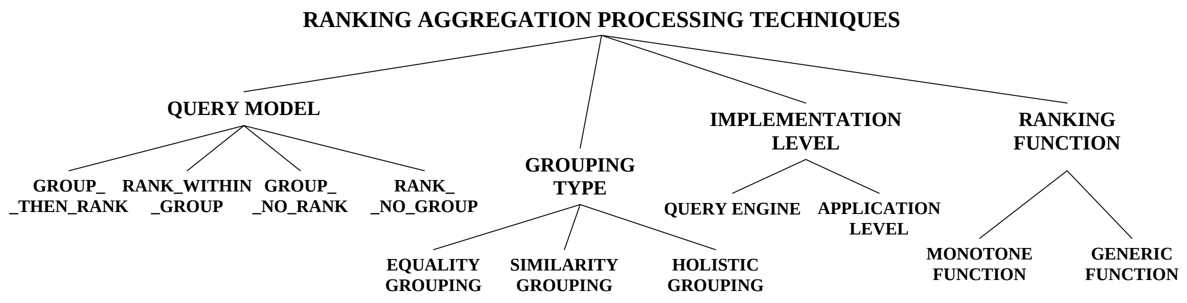


Figure 2.1: Classification of Ranking Aggregation processing techniques

Other classifications have been studied but are not included in this chapter. These classifications can be according to the result types or the query types. The result type can be *exact*, *approximate*, or *probabilistic* while query type can be *offline* or *ad-hoc*. Offline queries are pre-known but ad-hoc queries are known only during query time.

2.1 Query Model Dimension

The ranking Aggregation processing techniques can be classified according to the query model. Several research studies were held on different interactions between ranking and grouping. Some focus on applying ranking over the groups, while others focus on applying ranking within each group. Other research studies focus on one and ignore the other.

- ### Group_then_Rank

In this model, there is a grouping criterion to construct groups and a ranking criterion to provide the total order among groups. This model appears in scenarios where the top groups are required.

Example 2.1 [Group_then_Rank Example]. *A researcher needs to know the best 5 neighborhoods in the city with respect to the number of restaurants and schools each with different weight. Figure 2.2 shows a SQL-like formulation to the problem.*

```
SELECT R.Neighborhood, AVG(w1*R.count + w2*S.count) AS Weight
FROM Restaurant R, Schools S
WHERE R.Neighborhood = S.Neighborhood
GROUP BY R.Neighborhood
ORDER BY Weight
LIMIT 5
```

Figure 2.2: Group_then_Rank SQL-like query

Gang Li *et al* [4] proposed a ranking aggregation framework where queries are computed over a specified range on some dimensions other than the grouping dimensions. This is done by storing precomputed partial aggregation information in the data cube. The proposed technique supports only predetermined aggregation function and aggregation expression lacking the ability to support ad-hoc queries.

Li *et al* [10] gave a framework for efficient processing of ad-hoc ranking grouping queries. A special class of aggregation functions is defined named the max-bounded functions class. An upper-bound of the aggregation over a group can be obtained by applying the maximum values of the member tuples. This can be done if the cardinality of each group can easily be obtained. Three principles were proposed; Upper-Bound, Group-Ranking, and Tuple-Ranking.

First, the upper-Bound principle dictates the requirements of early pruning and the groups with the highest upper-bound scores are prioritized. Then the Group-Ranking principle dictates the order in which groups are probed. The incomplete groups within the current top-k are prioritized. On the other hand, the Tuple-Ranking dictates the order in which tuples should be accessed from each group and the tuples with the highest scores are prioritized.

Yiu *et al* [23] solves the Group_then_Rank problem under the following assumptions: 1- no multi-dimensional indexes exist on the grouping tuples; 2- the memory of the system is assumed to be not large enough to hold a counter for each distinct group in the input. Three scenarios were studied and appropriate algorithms were designed. For the scenario with the data physically ordered by the tuple score, write-optimized multi-pass sorted access algorithm ($WMSA$) is designed. The algorithm exploits available memory for efficient top-k groups computation. For the scenario with the unsorted data, recursive hash algorithm (RHA) is designed. The algorithm applies hashing with early aggregation coupled with branch-and-bound techniques and derivation heuristics for tight score bounds of hash partitions. For scenarios with data clustered according to a subset of group by attributes, clustered groups algorithm (CGA) is designed.

Wang *et al* [18] surveys techniques used to group related search results and provide them ranked to the user.

• Rank_within_Group

In this model, there is a grouping criterion to construct groups and a ranking criterion to provide total order between members of each group. This model appears in scenarios where each group or cluster should be represented by their top candidates. These scenarios appear much in guided navigation tools and websites.

Example 2.2 [Rank_within_Group Example]. *Consider a warehouse website that wants to present the top five products in each category based on user rates. Figure 2.3 formulates this problem in SQL-like query.*

```
SELECT PR.Category, PR.Product, AVG(PR.UserRate) AvgRate,
RANK() OVER (PARTITION BY Category ORDER BY AvgRate) AS Rank
FROM ProductReviews PR
GROUP BY PR.Category, PR.Product
HAVING Rank <= 5
```

Figure 2.3: Rank_within_Group SQL-like query

Li *et al* [11] proposed framework to support clustering and ranking together with the order-within-groups semantics, as a generalization of group by and order by to support fuzzy data retrieval applications. Their solutions were built upon summary-based clustering and ranking using dynamically constructed data summary at query time. They implemented this framework by utilizing a bitmap index to construct such summary on-the-fly and to integrate Boolean filtering, clustering and ranking.

• Group_no_Rank

In this model, there is a grouping criterion to construct groups but there is no ranking criterion; all the groups are required. This model appears in scenarios where all groups are independent.

Example 2.3 [Group_no_Rank Example]. *The government needs some statistics about the unemployed people, persons with disabilities, and seniors (above 65) over all the states. The result will be analyzed and no state is going to be prioritized. Figure 2.4 formulates this problem in SQL-like query.*

```
SELECT State(C.Address), COUNTIF(C.Unemployed),
COUNT(C.WithDisability), COUNT(C.Senior)
FROM Citizens C
GROUP BY State
```

Figure 2.4: Group_no_Rank SQL-like query

Antony *et al* [1] studied skyline queries over aggregated data to identify the most interesting groups. The developed algorithm uses best-first search and a Sequential Index Bounding strategy to find the skyline.

• Rank_no_Group

In this model, there is a ranking criterion without any grouping criterion. This model is a special case of Group_then_Rank where each candidate represents a group (i.e. group size is one). Also, it is considered a special case of Rank_within_Group where number of groups is one and its size is equal to the number of candidates.

Example 2.4 [Rank_no_Group Example]. *A journalist needs to get the top ten cities with the highest population density. Figure 2.5 formulates this problem in SQL-like query.*

```
SELECT C.Name, C.Population/C.Area AS PopulationDensity
FROM City C
ORDER BY PopulationDensity
LIMIT 10
```

Figure 2.5: Rank_no_Group SQL-like query

Ilyas *et al* [7] gave a comprehensive survey about the Top-k processing techniques. Different design dimensions had been studied including query models, data access methods, implementation levels, data and query certainty, and supported scoring functions.

2.2 Grouping Type Dimension

Ranking Aggregation processing techniques can be classified according to the grouping technique. Three categories can be used: equality grouping, similarity grouping (clustering), and holistic grouping.

• Equality Grouping

Grouping is performed based on equality when the members of a group have exactly the same values of the grouping attributes. Grouping capabilities of this category are widely studied in the data management systems. It is also a core operation in OLAP and decision support systems.

In the traditional relational database, *Group By* clause [12] is the main aggregation mechanism to provide equality aggregation.

Example 2.5 [Equality Grouping Example]. *A CEO of an auto company is interested in analyzing the car sales across some cities in 2012. Figure 2.6 formulates this problem in SQL-like query.*

```
SELECT CS.Model, CS.CarYear, CS.CityName, COUNT(*)
FROM CarSales CS
WHERE EXTRACT(CS.PurchaseDate, YEAR) = 2012
GROUP BY CS.Model, CS.CarYear, CS.CityName
```

Figure 2.6: Equality Grouping SQL-like query

• Similarity Grouping

Grouping is performed based on similarities such that the members of a group are similar among themselves and dissimilar to the members of other groups. Clustering is studied widely in machine learning, data mining, pattern recognition, image processing, and bio-informatics.

Example 2.6 [Similarity Grouping Example]. *A group of investors hired the real estate company to find some candidate lands to build a large shopping center. They prefer the land with the maximum population within radius* 50 *miles. The information about the neighborhoods is maintained in the database in the table Neighborhood, while the information about the available lands for sale is maintained in the table LandForSale. Figure 2.7 formulates this problem in SQL-like query.*

```
SELECT L.Location , SUM(N.Population)
FROM LandForSale L, Neighborhood N
HOLISTIC BY L.Location , N.Population SIZE MAX
CONSTRAINT EQUALITY(Apt.Neighborhood)
AND DISTANCE(L.Location , N.CenterLocation) < 50
```

Figure 2.7: Similarity Grouping SQL-like query

Berkhin [2] surveys clustering techniques used in data mining. Xu *et al* [21] surveys clustering techniques used in statistics, computer science, and machine learning. Jain *et al* [8] presents an overview of pattern clustering methods from a statistical pattern recognition perspective.

Zhang *et al* [24] introduced *Cluster By* clause to provide some clustering facilities using *SQL* in spatial databases. The functionality provided is only a wrapper of clustering algorithms and no integration with the query engine.

Silva *et al* [15, 16, 14] proposed a new SQL construct that supports similarity-based Group-by (SGB). SGB provides enhancement over the previous clustering algorithms. The execution times of *SGB* are very close to that of the regular *GROUP BY*. *SGB* are fully integrated with the query engine allowing the direct use of their results in complex query pipelines. The computation of aggregation functions is integrated in the grouping process and considers all the tuples in each group, not a summary or a subset based on sampling.

- **Holistic Grouping**

In this category no grouping criteria is defined. Instead, a scoring function defined over the group members is used. Group scores are used as a metric for their order. This category is the main focus of this thesis.

Example 2.7 [Social Network]. *Consider a social network where the administrator likes to analyze the number of mutual friends between a group of friends. He is interested in finding 100 groups of size 50 where the number of mutual friends is maximum. Figure 2.8 shows a SQL-like formulation to the problem.*

```
SELECT INTERSECTION(F.FriendList) AS MutualFriends
FROM Friends F
HOLISTIC BY F.FriendList SIZE 50
ORDER BY MutualFriends
Limit 100
```

Figure 2.8: Holistic Grouping SQL-like query

2.3 Implementation Level Dimension

Ranking Aggregation processing techniques can be classified according to the level of integration with the database systems. Some techniques are implemented as query operators while others are implemented as an application level over the database system.

- **Query Engine**

The operator is added to the database engine. This allows the engine to apply optimization techniques to the new operators. Examples are the group by [12] clause and SGB [15, 16, 14].

The main Eager and lazy aggregation presented in [22] was extended to support SGB operator to give the new operator flexibility to be moved up and down the query tree.

- **Application Level**

The operator is implemented as an outer layer over the database engine. This technique allows the extensibility of the operators. *CLUSTER BY* [24] is an example.

2.4 Ranking Function Dimension

Ranking Aggregation processing techniques can be also classified according to the type of the ranking function. Some techniques assume monotone functions while others assume no restrictions on the ranking function.

• Monotone Function

A function is monotone if $\mathcal{F}(x_1, ..., x_m) \geq \mathcal{F}(x_1^{'}, ..., x_m^{'})$ whenever $x_i \geq x_i^{'}$ for every i. Thus if for every attribute, the value of x_i is at least as high as that of $x_i^{'}$, then the overall value of $\mathcal{F}(x_1, ..., x_m)$ is at least as high as that of $\mathcal{F}(x_1^{'}, ..., x_m^{'})$.

Example 2.8 [Monotone Function Example]. *A real estate wants to present the average price of the houses in each neighborhood. Figure 2.9 formulates this problem in SQL-like query.*

```
SELECT H.Neighborhood, AVG(H.Price)
FROM House H
GROUP BY H.Neighborhood
```

Figure 2.9: Monotone Function SQL-like query

• Generic Function

This category includes all types of functions including monotone functions.

Example 2.9 [Generic Function Example]. *Another group of investors hired the real estate company to find candidate lands to build three resident buildings. Their primary criterion is the locations that have the minimum inter-distance. Consider HOLISTIC BY construct that perform holistic grouping. Figure 2.10 shows a SQL-like formulation to the problem.*

```
SELECT DISTANCE(L.Location) AS TotalDistance
FROM LandsForSale L
HOLISTIC BY L.Location SIZE 3
```

Figure 2.10: Generic Function SQL-like query

2.5 Summary

Table 2.1 shows some of ranking aggregation processing techniques and how the thesis fits.

Table 2.1: Ranking Aggregation Processing Techniques

	Query Model				Aggregation Type			Implem. Level		Ranking Function	
	Group_then_Rank	Rank_within_Group	Group_no_Rank	Rank_no_Group	Equality Aggregation	Similarity Aggregation	Holistic Aggregation	Query Engine	Application Level	Monotone	Generic
RankAggr [10]	√				√			√		√	
ClusterRank [11]		√				√		√		N/A	
SGB [15]			√			√		√		N/A	
Cluster by [24]			√			√			√	N/A	
Group by [12]			√		√			√		N/A	
Thesis	√						√		√	√	

Chapter 3

HRA - Monotone Functions

In this chapter, the Holistic Ranking Aggregation Query (*HRA*) with a group-based monotone scoring function is studied. A function is said to be monotone if $\mathcal{F}(x_1, ..., x_m) \geq \mathcal{F}(x_1^{'}, ..., x_m^{'})$ whenever $x_i \geq x_i^{'}$ for every i. The group size is assumed to be fixed. First, an algorithm based on the Rank-Join problem is studied in section 3.1. Then, algorithms based on A^* are introduced in section 3.2. Finally, the assumption of fixed group size is relaxed in section 3.3.

The trivial solution of the *HRA* problem is to compute all the groups of size m, rank them, and report only the top k groups. If there are no constraints and all the states will be considered as candidates, only groups created from the first $k+m-1$ tuples have to be considered. This is stated and proved in the following theorem.

Theorem 3.1. *In case of no constraints, the Top-K groups of size m are a subset of the groups of size m that can be created from the largest $k+m-1$ elements (i.e. A group containing the $k+m$ element is dominated by k groups that contain elements from the largest $k+m-1$ elements).*

Proof: Let $\mathcal{G}_\mathcal{A}$ be a set of groups of size m created form the $k+m-1$ elements ($|\mathcal{G}_\mathcal{A}| = \binom{k+m-1}{m}$). Let n be the $(k+m)^{th}$ element and $\mathcal{G}_\mathcal{B}$ be all the new groups created by introducing n ($|\mathcal{G}_\mathcal{B}| = \binom{k+m-1}{m-1}$).

$$\forall b \in \mathcal{G}_\mathcal{B}, |a \in \mathcal{G}_\mathcal{A} : b - n \subset a| = k + m - 1 - (m-1) = k$$

Since n is smaller than or equal to the k elements, k groups dominate b. Thus b will never be chosen as one of the top-k groups.

3.1 Reusing Rank-Join Algorithms

Holistic ranking aggregation can be defined as a special type of the rank-join problem.

Definition 3.1. [**Rank-Join Query (RJ)**] *Let $\mathcal{R} = \{R_1, R_2, ..., R_m\}$ be a set of m relations, $\mathcal{F}$ be a scoring function defined over the relations, and C be a set of constraints. The query $RJ(\mathcal{R}, \mathcal{F}, k, C)$ computes the k join results over $\mathcal{R}$ with the highest $\mathcal{F}$ values and satisfying constraints in C.*

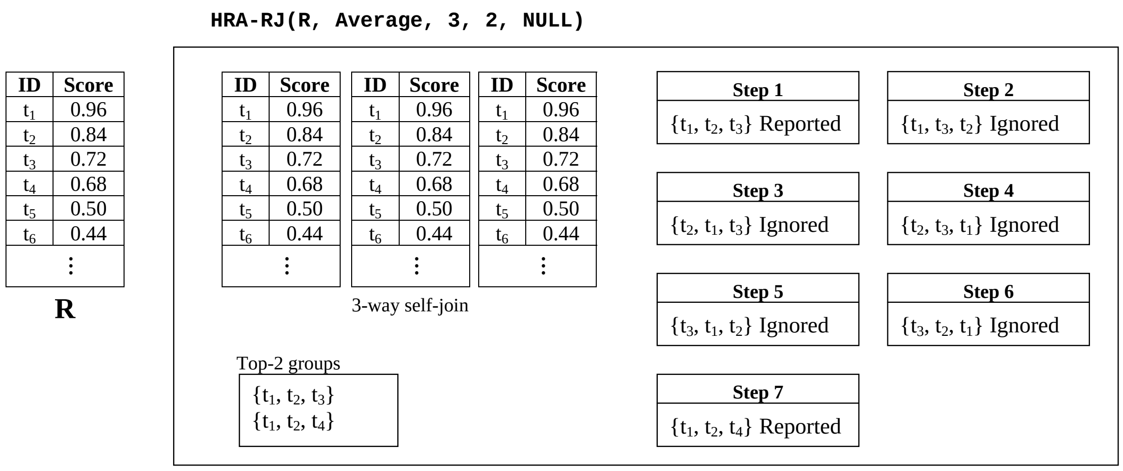
Theorem 3.2. *Let $R = \{t_1, t_2, ..., t_n\}$ be a set of tuples, $\mathcal{F}$ be a group-based scoring function, and $\mathcal{R} = \{R, R, ..., R\}$ be a set of identical relations where $|\mathcal{R}| = m$. The result of a holistic ranking aggregation query $HRA(R, \mathcal{F}, m, k, C)$ is equivalent to the result of an m-way self-join ranking query $RJ(\mathcal{R}, \mathcal{F}, k, C)$ assuming the following two conditions.*
1. The joining condition is the not-equal operator.
2. All the result instances that are computed from the same original tuples, ignoring the order, are equivalent. Only one of the equivalent groups is reported.

Algorithm 1 outlines the *HRAm-RJ* algorithm. First, a hash table and a rank-join instance are defined. The hash table is used to carry the IDs used in building the join results. The rank-join instance can be any rank-join algorithm that supports non-equality joining condition. Then, the algorithm iterates until returning the top k groups. In each iteration, a new result tuple is retrieved and the IDs of tuples participated in the result are extracted. If the IDs combination is not in the hash table, it constructs the corresponding group, reports it, and adds the IDs combination to the hash table.

Algorithm 1 *HRAm-RJ* (R, $\mathcal{F}$, m, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
1: Build hash table $H$ to hash groups generated, the key is the IDs of the original tuples
2: $\mathcal{R} \leftarrow \{R, R, ..., R\}$ where $|\mathcal{R}| = m$
3: Initialize a Rank-Join instance $RJ(\mathcal{R}, \mathcal{F}, k, C)$
4: while |reported groups| $\leq k$ do
5:     $j \leftarrow RJ$.getNextResult()
6:     $\{ID_1, ID_2, ..., ID_m\} \leftarrow j$.getIDs()
7:     if $\{ID_1, ID_2, ..., ID_m\}$ does not exist in $H$ then
8:         $g \leftarrow j.constructGroup()$
9:         Report $g$
10:        Add $\{ID_1, ID_2, ..., ID_m\}$ to $H$
11:    end if
12: end while
```

Example 3.1 [*HRAm-RJ* **Example**]. *Consider a relation R ranked on the score value (score $\in [0, 1]$). The group size is three, the number of groups is two, the group-based* *scoring function is Average, and there are no group constraints. Figure 3.1 depicts the first seven steps of the HRAm-RJ algorithm. The group $\{t_1, t_2, t_3\}$ first shows up in step one, so it is reported. Then, all its permutations show up in the next steps (two - six) and they are ignored. Finally, a new group shows up in step seven $\{t_1, t_2, t_4\}$, so it is reported and the algorithm stops.*



ID	Score
t_1	0.96
t_2	0.84
t_3	0.72
t_4	0.68
t_5	0.50
t_6	0.44
⋮	

ID	Score	ID	Score	ID	Score
t_1	0.96	t_1	0.96	t_1	0.96
t_2	0.84	t_2	0.84	t_2	0.84
t_3	0.72	t_3	0.72	t_3	0.72
t_4	0.68	t_4	0.68	t_4	0.68
t_5	0.50	t_5	0.50	t_5	0.50
t_6	0.44	t_6	0.44	t_6	0.44
⋮		⋮		⋮	

Figure 3.1: *HRAm-RJ* Example

The main shortcoming of this algorithm is that the results of *RJ* are based on ordered pairs while the results of *HRA* are based on set theory. This will affect the out-coming of the rank-join algorithm in the following points:
(1) If a group is reported, all the equivalent groups will also be reported
(2) An extra memory space is needed to check the group equivalence.

Lemma 3.1. *If a tuple is materialized, it will be materialized m times.*

Rank-join algorithms treat each relation as a distinct one and there is no special treatment for self-joins. Thus, if a tuple is materialized, it will be a good candidate to be materialized from other relations. Thus, it will be materialized m times.

Lemma 3.2. *If a group is reported, all the equivalent groups will also be reported.*

If a group is reported, this means that its score is the current highest score. Since all the equivalent groups have the same score, all the remaining groups will be candidates to be chosen by the algorithm.

3.2 New A^* Algorithms

3.2.1 Bottom-Up Algorithm

As discussed in the previous section, using the existing rank-join algorithm to solve the *HRA* problem faces shortcomings in the execution time and memory. More shortcomings will show up when dealing with the case of the variable group size in section 3.3. In this section, an algorithm *HRAm-B* that is based on A^* [5] class of search algorithms is studied. *HRAm-B* considers the shortcomings of *HRAm-RJ* and tries to avoid them.

The idea is to maintain a priority queue of partial and complete groups ordered on the upper bounds estimates of their total scores. At each step, the algorithm tries to complete the group by adding the next tuple to it. It reports the next top group as soon as a complete group is ready on the top of the queue.

A state is defined as a binary string with a variable length. Each state represents a group where the position of each bit refers to the corresponding tuple. A bit is set if the corresponding tuple belongs to the group,otherwise it will be reset. For example, 110 represents a group having the first and second tuples. A state is complete if the number of ones is equal to the size of the group. The problem of finding a group with maximum score is reduced to finding the binary string with number of ones equal to the group size that maximizes the score. The scores of the complete states are computed from the scores of the group members. While the scores of the incomplete states are computed from the scores of the materialized members and the score estimates of the non-materialized members. A tie breaking rule that prioritizes the group with the most materialized elements is used.

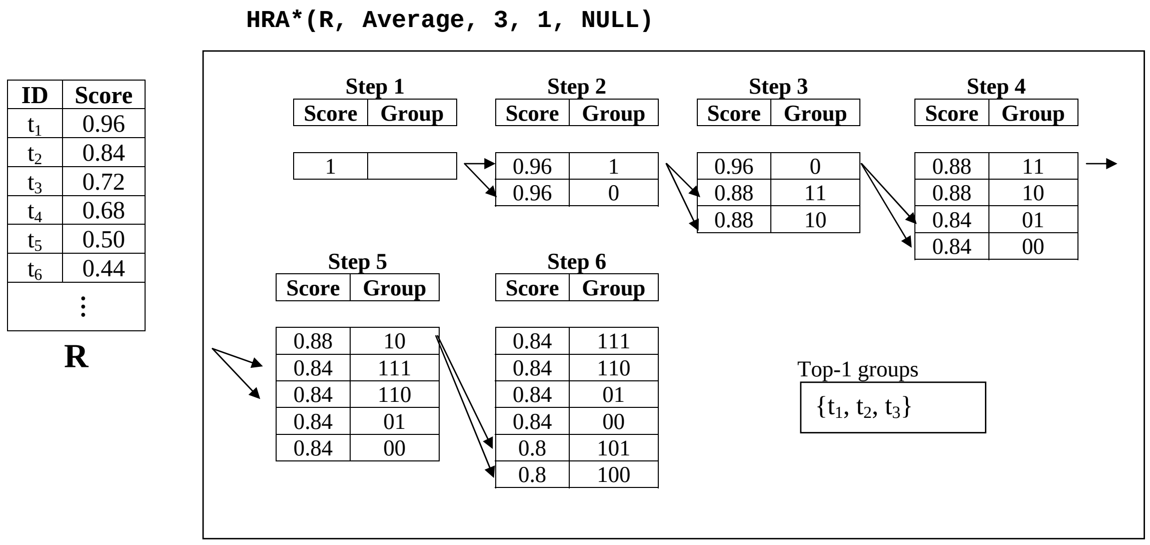
HRAm-B adopts the state extension technique used in $U-Topk$ query [17]. A state is expanded into two states; one considers the next tuple and the other ignores it. However a straight forward adoption of $U - TopK$ is not applicable because of the different data model and the score definition.

The details of the *HRAm-B* are illustrated in algorithm 2. First, a priority queue and a hash table are created. The priority queue holds the states based on the scoring function provided and initializes it by adding an empty state. The hash table is used to hash the tuples retrieved using the order of retrieval as a key. Then, the algorithm iterates until returning the top k groups. In each iteration, the state on the top of the priority queue is retrieved. If the state is complete and satisfying the holistic constraints, the corresponding group is constructed and reported. If the state is partial, the corresponding tuple is retrieved either from the hash or by calling the $getNextTuple$ function. Then two states are constructed; one considering the next tuple and the other ignoring it. It adjusts their scores using the new tuple score. The state with the new one uses the score as the score of one of the members. Both states use the score as the upper bound for the remaining un-materialized group's members. If each one is satisfying the partial constraints, it will be added to the priority queue.

Algorithm 2 *HRAm-B* (R, $\mathcal{F}$, m, k, C)

```
Require: R: ranked relation and F: group-based scoring function
 1: Create a priority queue $Q$ to order states based on $\mathcal{F}$
 2: Create a hash table $H$ to hash tuples retrieved, the key is the order of the tuple retrieval
 3: $s \leftarrow$ ""
 4: while |reported groups| $\leq k$ do
 5:     $s \leftarrow$ Q.dequeue()
 6:     if $s$ contains $m$ ones and $s$ satisfy $C$.getHolisticConstraints() then
 7:         $g \leftarrow s$.constructGroup()
 8:         report g
 9:     else
10:         if $(|s|+1)$ exists in H then
11:             $t \leftarrow H$.get$(|s|+1)$
12:         else
13:             $t \leftarrow R$.getNextTuple()
14:             $H$.add$(|s|+1, t)$
15:         end if
16:         $s^{'} \leftarrow s +$ "0"
17:         $s^{'}$.adjustScore(t)
18:         if $s^{'}$ satisfy $C$.getPartialConstraints() then
19:             Q.enqueue($s^{'}$)
20:         end if
21:         $s^{''} \leftarrow s +$ "1"
22:         $s^{''}$.adjustScore($t$)
23:         if $s^{''}$ satisfy $C$.getPartialConstraints() then
24:             Q.enqueue($s^{''}$)
25:         end if
26:     end if
27: end while
```

Example 3.2 [*HRAm-B* **Example**]. *Consider a relation R ranked on the score value (score $\in [0,1]$). The group size required is three, the number of groups is one, the group-based scoring function is Average, and no constraints. Figure 3.2 depicts the first six steps of the HRAm-B algorithm. The priority queue is first initialized with an empty bit string. The score of this state can be calculated by assuming the upper bound of the tuples score that will belong to the group ($\frac{1+1+1}{3} = 1$). Thus the score is one. Then this state is removed from the queue and two new states are enqueued, one represents groups* *that contain the first tuple and the other represents groups that will not contain the first tuple. The scores are calculated in the same way. According to the tie breaking rule, state* (1) *is prioritized over state* (0)*. The process continues until a complete state appears on the top of the queue.*



```
HRA*(R, Average, 3, 1, NULL)
```

ID	Score
t_1	0.96
t_2	0.84
t_3	0.72
t_4	0.68
t_5	0.50
t_6	0.44
⋮	

R

Step 1

Score	Group
1	

Step 2

Score	Group
0.96	1
0.96	0

Step 3

Score	Group
0.96	0
0.88	11
0.88	10

Step 4

Score	Group
0.88	11
0.88	10
0.84	01
0.84	00

Step 5

Score	Group
0.88	10
0.84	111
0.84	110
0.84	01
0.84	00

Step 6

Score	Group
0.84	111
0.84	110
0.84	01
0.84	00
0.8	101
0.8	100

Top-1 groups

$\{t_1, t_2, t_3\}$

Figure 3.2: *HRAm-B* Example

Lemma 3.3. *In the worst case, HRAm-B Algorithm generates an exponentail state space.*

HRAm-B inherits all the properties of A^* search, thus *HRAm-B* is guranteed to find the optimal solution since the heuristic score is always upper bounded. Also *HRAm-B* suffers from exponential large space requirements.

HRAm-B can perform early pruning; as soon as an invalid state violates any partial constraint, the state will be pruned and consequently all its children will not be generated. However *HRAm-B* has to wait till the groups are fully materialized to check holistic constraints.

3.2.2 Top-Down Algorithm

As discussed in the previous section, *HRAm-B* algorithm suffers from exponential space requirements in some scenarios. In this section, a linear space algorithm *HRAm-T* is described. This algorithm loses early pruning property in the favour of fast complete states generation.

As in *HRAm-B* , a state is defined as a binary string. For simplicity, no constraints are assumed. Thus all complete states are valid and the length of a state is fixed ($k+m-1$ according to theorem 3.1.1). This assumption is relaxed later.

Definition 2. [**State Dominance**] *Let* $R = \{t_1, t_2, ..., t_n\}$ *be a set of tuples and* s *and* $s^{'}$ *be complete states with* m *ones. Let* P*,* $P^{'}$ *be the positions of ones in* s *and* $s^{'}$ *respectively. State* s *dominates state* $s^{'}$ *(denoted* $s \succ s^{'}$*) iff* $\forall i \in [1, m] : t_{P_i}.score \geq t_{P_i^{'}}.score$*. Which is equivalent to say* $\forall i \in [1, m] : P_i \geq P_i^{'}$

This dominance relation results in the following important property

Property 3.1. *HRA problem constructs a partially ordered set (poset) in which states are the elements and the binary relation is the state dominance (*$\succ$*).*

Figure 3.3 shows the hasse diagram of the state dominance where the group size is three.

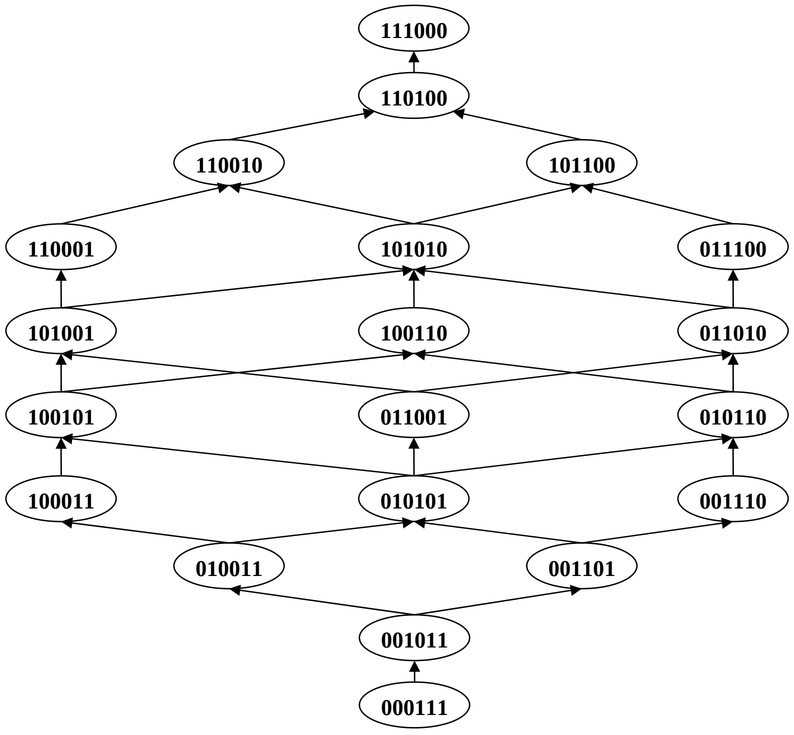


Figure 3.3: The Hasse diagram of the state dominance

This property suggests a state generation technique in which a state is responsible for the generation of the next dominated states only.

Definition 3. [**Next Dominated States**] *A set of states $\mathcal{S}$ is the next dominated states of state s if there are no other state $s^{'} \notin \mathcal{S}$ dominated by s and not dominated by any member in $\mathcal{S}$.*

For example $s^{'} = 101010$ is a nearest domainted state to $s = 110010$ where $P^{'} = \{1, 3, 5\}$ and $P = \{1, 2, 5\}$

Definition 4. [**States Incomparability**] *Two states s and $s^{'}$ are incomparable iff $s \not\succ s^{'}$ and $s^{'} \not\succ s$*

For example 110010 and 011100 are incomparable.

The first group is assumed to have the first m tuples which means that the early materialized tuples are prioritized. This assumption is adopted by most of the ranking algorithms. For example $HRJN$ [6] and TA [3] algorithms.

As seen in the hasse diagram, a state can generate multiple states and be generated from multiple states. These results are studied in the next theorems. To resolve the duplicate problem; a hash can be used or a state can be assigned to one parent state only. Figure 3.4 shows the hasse diagram of the state dominance after assigning each node to the parent with the higest significant bit.

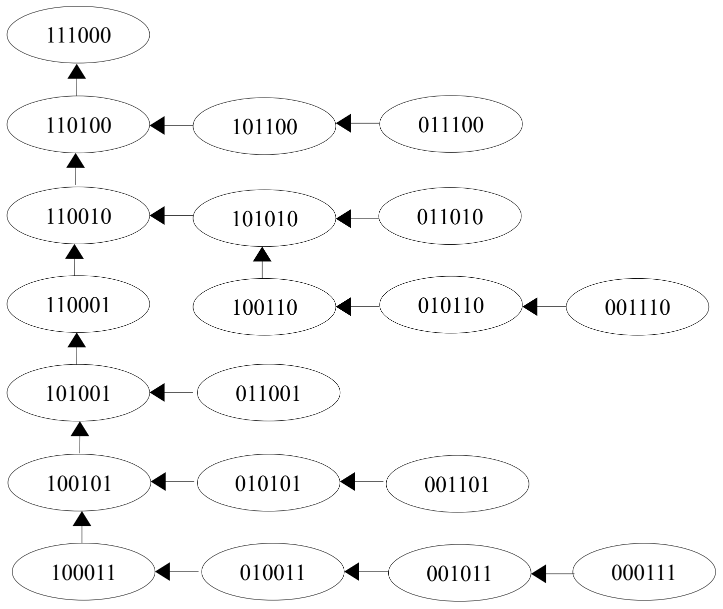


Figure 3.4: The Hasse diagram of the state dominance with one parent restriction

Theorem 3.3. *A state can generate zero states up to the minimum of the group size and the number of groups required minus one. $|s.children| = min(m, k - 1)$*

Proof: Given a state s, the number of states that can be generated from s is equal to the number of '10'-patterns in s. The number of ones in s is equal to the group size (m). The number of zeros is the total size of s minus the number of ones (i.e. $|s| - |s.ones| = k + m - 1 - m = k - 1$). Thus the minimum number of patterns is zero in which no zero comes after one. The maximum number of patterns is the minimum of m and $k - 1$.

Theorem 3.4. *A state can have zero parents up to the minimum of the group size and the number of groups required minus one.* $|s.parent| = min(m, k - 1)$

Proof: The proof is similar to the previous one. Given a state s, the number of its parents is equal to the number of '01' patterns in s. The number of ones in s is equal to the group size (m). The number of zeros is the total size of s minus the number of ones (i.e. $|s| - |s.ones| = k + m - 1 - m = k - 1$). Thus the minimum number of patterns is zero in which no one comes after zero. The maximum number of patterns is the minimum of m and $k - 1$.

The algorithm is based on Uniform-cost search [9] which is a special case of the A^* search where the heuristic is a constant function. In *HRAm-T* , the constant is zero. The idea is to maintain a priority queue for the complete groups only ordered on their total scores. At each step, the algorithm reports the group on the queue top and generate the next groups that can be candidates to be on the queue top.

Algorithm 3 *HRAm-T* (R, $\mathcal{F}$, m, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
1: Create a priority queue $Q$ to order states based on $\mathcal{F}$
2: Create a hash table $H$ to hash tuples retrieved, the key is the order of the tuple retrieval
3: $s \leftarrow 11...100...0$ where $|s| = k + m - 1$ and $|11...1| = m$
4: $Q$.enqueue($s$)
5: while |reported groups| $\leq k$ do
6:     $s \leftarrow$ Q.dequeue()
7:     if $s$ satisfy $C$ then
8:         $g \leftarrow s$.constructGroup()
9:         report g
10:    end if
11:    for all next dominated state $s^{'}$ do
12:        if $s^{'}$ is a new state then
13:            Q.enqueue($s^{'}$)
14:        end if
15:    end for
16: end while
```

The details of the *HRAm-T* algorithm are illustrated in algorithm 3. First, a priority queue and a hash table are created. The priority queue holds the states based on the scoring function provided and it is initialized by a state of length $k + m - 1$ with m ones. The hash table is used to hash tuples retrieved using the order of retrieval as a key. Then, the algorithm iterates until returning the top k groups. In each iteration, it gets the state on the top of the priority queue. If the state is satisfying all constraints, it constructs the corresponding group and reports it. Then it iterates over all the next dominated states. If it is a new state, it is added to the priority queue.

Example 3.3 [***HRAm-T*** **Example**]. *Consider a relation R ranked on the score value (score $\in [0, 1]$). The group size required is three, the number of groups is four, the group-based scoring function is Average, and there are no constraints. Figure 3.5 depicts the first four steps of the HRAm-T algorithm. The priority queue is first initialized with a bit string of length six with three ones. The score of this state can be calculated exactly from the members of the group represented by the state.* ($\frac{0.96+0.84+0.72}{3} = 0.84$)*. Then this state is removed from the queue and reported. Since the state has one next dominated states thus a new state is generated and enqueued. The score is calculated in the same way. The process continues until four states are reported.*

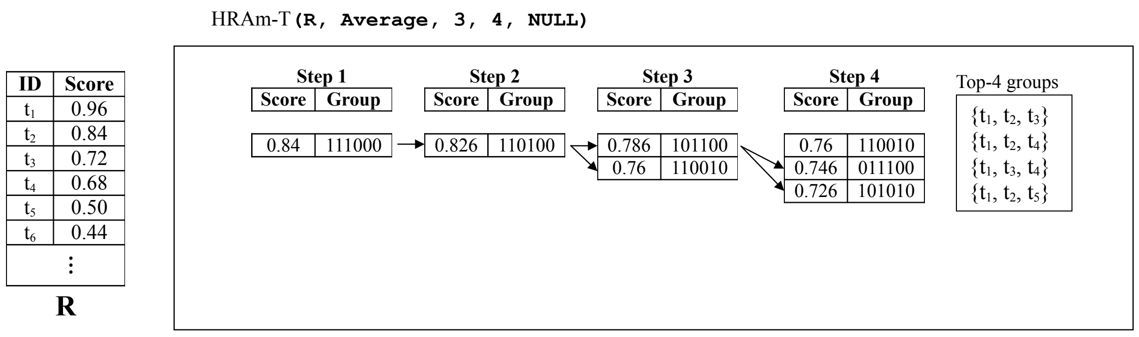


Figure 3.5: *HRAm-T* Example

The following two theorems demonstrate two important properties of the proposed algorithm: soundness and completeness.

Theorem 3.5. *HRAm-T is sound. i.e. All groups dominate g_i will be generated and chosen before g_i.*

Proof: This theorem is proved by contradiction. For the first part, assume a state $s^{'}$ dominates state s and is generated after it. $s^{'}$ dominates s thus $\forall i \in [1, m], P_i^{'} \leq P_i$. Also $s^{'}$ is generated after s thus $\forall i \in [1, m], P_i^{'} \geq P_i$. This is a contradiction.

For the second part, assume a state $s^{'}$ dominates state s and is chosen after it. $s^{'}$

dominates s thus $s^{'}$.score $>$ s.score. Also $s^{'}$ is chosen after s thus s.score $<$ $s^{'}$.score. This is also a contradiction.

Theorem 3.6. *HRAm-T is Complete. i.e. All groups are generated.*

Proof: This theorem is proved by contradiction. Assume a state s that is not the initial one and is not generated from another state. This means that there is no state with positions less than s positions. A state does not have any state with position less than its position if it has all ones in the most significant bit. This case is satisfied only in the initial state. Thus it is a contradiction so this state can not exist.

Relaxing the assumption of no constraints will change the state definition. A state is a binary string with variable size having m ones. An imaginary zero will be assumed in the least significant position to perform the transpose operation.

Table 3.1 shows comparison between *HRAm-B* search and *HRAm-T* search.

	HRAm-B Search	*HRAm-T* Search
Types of States	Partial and complete	Complete only
Generator	Partial states	Complete states
Result States	Complete states	Complete states
Searching Technique	A^* search	UCS search
State Space Size	Exponential	Exponential
$g(x)$	Materialized members	Materialized members
$h(x)$	Estimation of unmaterialized members	Zero

Table 3.1: *HRAm-B* Search versus *HRAm-T* Search

3.3 Variable Group-size

The algorithms provided in this chapter assumes that the size of the group is fixed. However, in many cases the size may be a range or multiple sizes are allowed. In the next subsections, the three algorithms proposed are modified to support the variable group size.

Modified *HRAM-RJ*

There are two modifications; (1) Several instances of the rank-join algorithm are used. One for each possible group size. (2) A priority queue is used to hold the results from all the instances.

These modifications add to the shortcomings discussed in section 3.1. Using multiple instances of the rank-join algorithm will increase the execution time dramatically. Also, the priority queue used increases the memory demands of the algorithm.

Algorithm 4 *HRAM-RJ* (R, $\mathcal{F}$, M, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
 1: Build hash table $H$ to hash groups generated, the key is the IDs of the original tuples
 2: Create a priority queue $Q$ to order states based on $\mathcal{F}$
 3: for all group size $m$ in $M$ do
 4:     $\mathcal{R} \leftarrow \{R, R, ..., R\}$ where $|\mathcal{R}| = m$
 5:     Initialize a Rank-Join instance $RJ(\mathcal{R}, \mathcal{F}, k, C)$
 6:     $j \leftarrow RJ$.getNextResult()
 7:     $Q$.enqueue($j$)
 8:     $\{ID_1, ID_2, ..., ID_m\} \leftarrow$ j.getIDs()
 9:     Add $\{ID_1, ID_2, ..., ID_m\}$ to $H$
10: end for
11: while |reported groups| $\leq k$ do
12:     $j \leftarrow Q$.dequeue()
13:     $g \leftarrow j.constructGroup()$
14:     Report $g$
15:     $RJ \leftarrow j$.getRJinstance()
16:     $m \leftarrow RJ$.getNumberOfRelations()
17:     repeat
18:         $j' \leftarrow RJ$.getNextResult()
19:         $\{ID_1, ID_2, ..., ID_m\} \leftarrow j'$.getIDs()
20:     until $\{ID_1, ID_2, ..., ID_m\}$ does not exist in $H$
21:     $Q$.enqueue($j'$)
22:     Add $\{ID_1, ID_2, ..., ID_m\}$ to $H$
23: end while
```

Modified ***HRAM-B***

There is only one modification; if the state on the top of the priority queue is a complete state, it will generate two new states. One of the new state considers the next tuple and the other ignores it.

Algorithm 5 *HRAM-B* (R, $\mathcal{F}$, M, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
 1: Create a priority queue $Q$ to order states based on $\mathcal{F}$
 2: Create a hash table $H$ to hash tuples retrieved, the key is the order of the tuple retrieval
 3: $s \leftarrow$ ””
 4: while |reported groups| $\leq k$ do
 5:     $s \leftarrow$ Q.dequeue()
 6:     if Least significant bit is one then
 7:         if $s$.numberOfOnes() $\in M$ and and $s$ satisfy $C$.getHolisticConstraints() then
 8:             $g \leftarrow s$.constructGroup()
 9:             report g
10:         end if
11:     end if
12:     if $(|s| + 1)$ exists in H then
13:         $t \leftarrow H$.get($|s| + 1$)
14:     else
15:         $t \leftarrow R$.getNextTuple()
16:         $H$.add($|s| + 1$, $t$)
17:     end if
18:     $s^{'} \leftarrow s +$ ”0”
19:     $s^{'}$.adjustScore(t)
20:     if $s^{'}$ satisfy $C$.getPartialConstraints() then
21:         Q.enqueue($s^{'}$)
22:     end if
23:     $s^{''} \leftarrow s +$ ”1”
24:     $s^{''}$.adjustScore($t$)
25:     if $s^{''}$ satisfy $C$.getPartialConstraints() then
26:         Q.enqueue($s^{''}$)
27:     end if
28: end while
```

Modified *HRAM-T*

There is only one modification; the priority queue is initialized by a set of states. One for each possible group size.

Algorithm 6 *HRAM-T* (R, $\mathcal{F}$, M, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
Create a priority queue $Q$ to order states based on $\mathcal{F}$
Create a hash table $H$ to hash tuples retrieved, the key is the order of the tuple retrieval
for all group size $m$ in $M$ do
    $s \leftarrow$ 11...100...0 where $|s| = k + m - 1$ and $|11...1| = m$
    $Q$.enqueue($s$)
end for
while |reported groups| $\leq k$ do
    $s \leftarrow$ Q.dequeue()
    if $s$ satisfy $C$ then
        $g \leftarrow s$.constructGroup()
        report g
    end if
    for all next dominated state $s^{'}$ do
        if $s^{'}$ is a new state then
            Q.enqueue($s^{'}$)
        end if
    end for
end while
```

Example 3.4 [Modified *HRAM-T* Example]. *Consider a relation R ranked on the score value (score $\in [0, 1]$). The group size required is three, the number of groups is either three or four, the group-based scoring function is Average, and no constraints. Figure 3.6 depicts the first four steps of the HRAM-T algorithm. The priority queue is first initialized with two bit strings of length six; one with three ones and the other with four ones. The scores are calculated from the members of the group represented by the state. The state '111000' has the higher score, thus it is removed from the queue and reported. Since the state has one '10' pattern thus a new state is generated by transposing the pattern and then enqueued. The score is also calculated in the same way. In step three, the state '111100' becomes on the queue top, thus it is removed and reported. The process continues until four states are reported.*

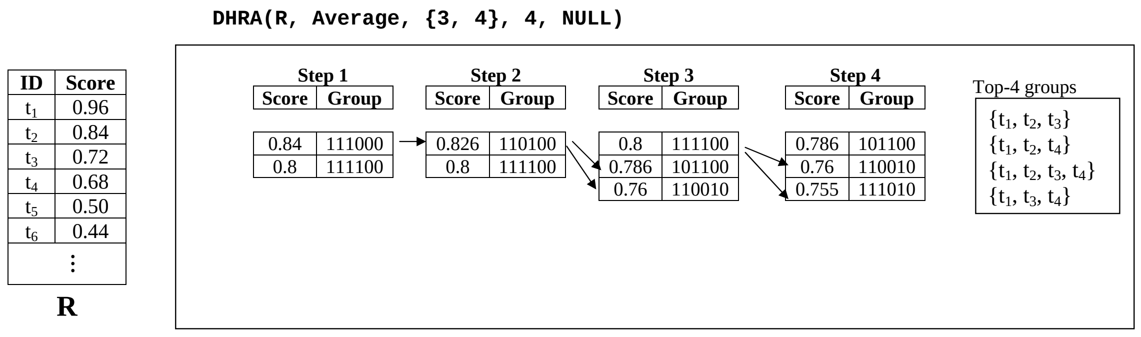


Figure 3.6: Modified *HRAM-T* Example

3.4 Experiments

All experiments were conducted on a SunFire X4100 server with Dual Core 2.2GHz processor, and 8GB of RAM. The parameters of the experiments are the group size m and the number of the groups required k. Two metrics are reported, the overall running time and the scan depth.

Experiment I: All algorithms

In these experiments, the three proposed algorithms were compared. In the first experiment, the group size is fixed to 4 and the number of required groups changed from 1 to 60. Figure 3.7 compares the running time of the three algorithms. Algorithms based on A^* shows better running time over the algorithm based on Rank-Join problem. Figure 3.8 compares the total scan depth of the three algorithms which are comparable.

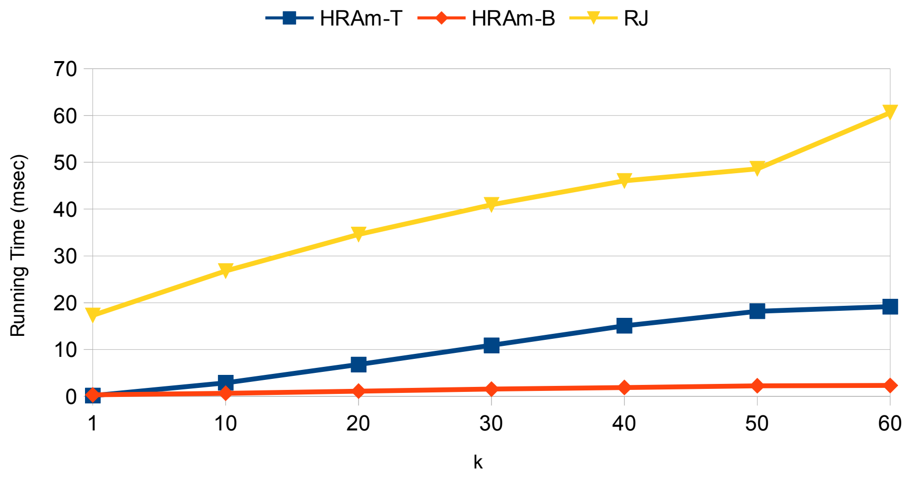


Figure 3.7: Change k - Running Time

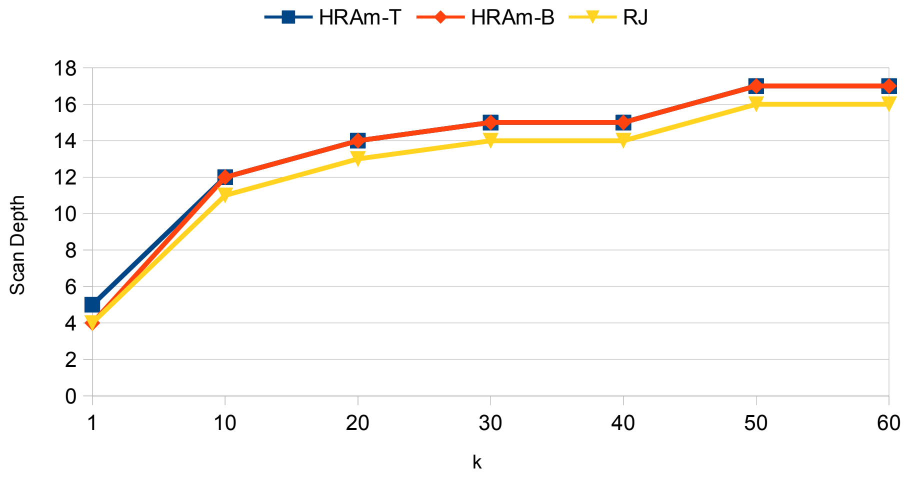


Figure 3.8: Change k - Scan Depth

In the second experiment, the number of required groups is fixed to 50 and the group size changed from 1 to 6. Figure 3.9 compares the running time of the three algorithms. Algorithms based on A^* shows better running time over the algorithm based on Rank-Join problem. Figure 3.10 compares the total scan depth of the three algorithms which are comparable.

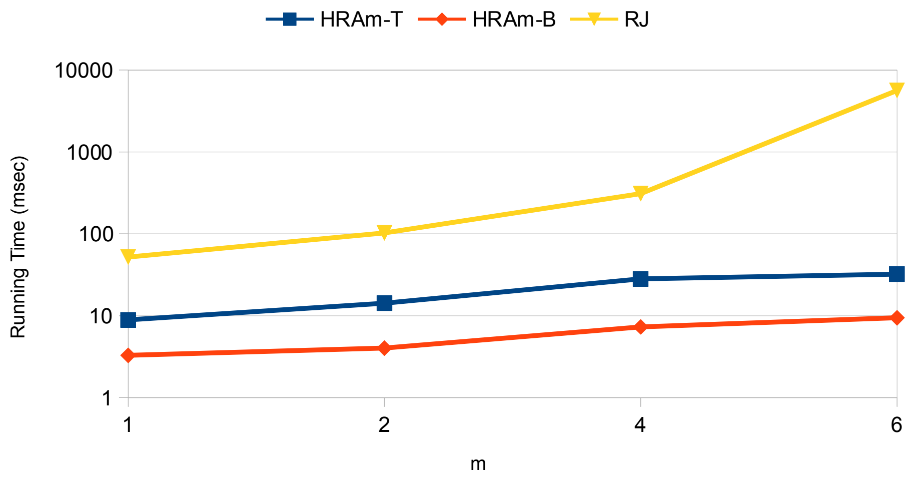


Figure 3.9: Change m - Running Time

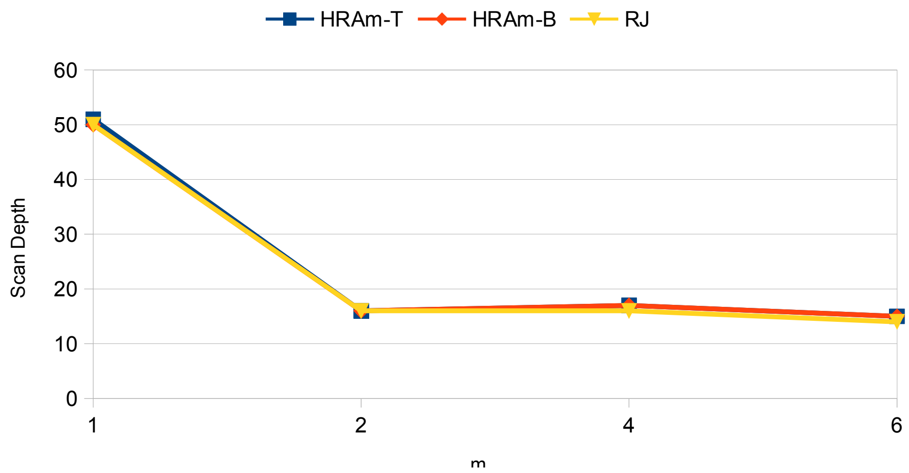


Figure 3.10: Change m - Scan Depth

As a conclusion, algorithms based on A^* show better performance over the algorithm based on Rank-Join problem.

Experiment II: A^* algorithms

In these experiments, the two proposed algorithms based on A^* algoriths were compared. In the first experiment, the group size is fixed to 8 and the number of required groups changed from 1 to 100. Figure 3.11 compares the running time of the two algorithms. Bottom-Up algorithm shows better running time over Top-Down algorithm. Figure 3.12 compares the total scan depth of the two algorithms which are comparable.

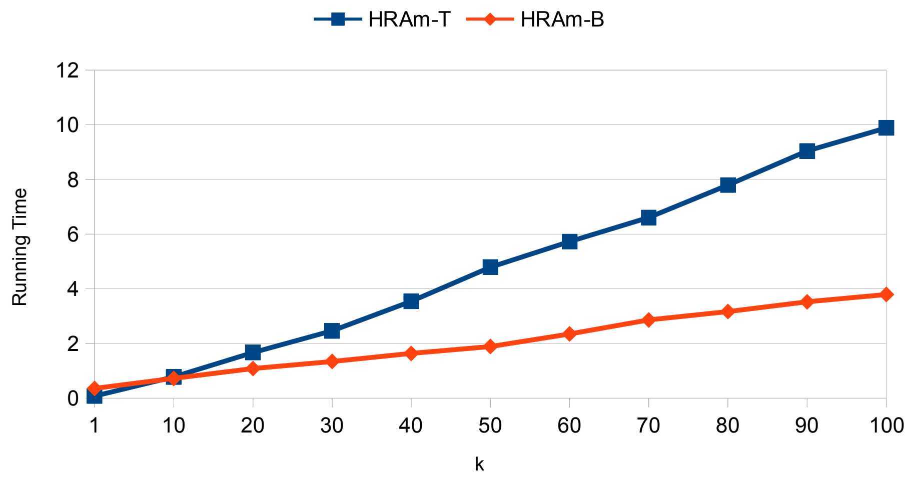


Figure 3.11: Change k (m = 8) - Running Time

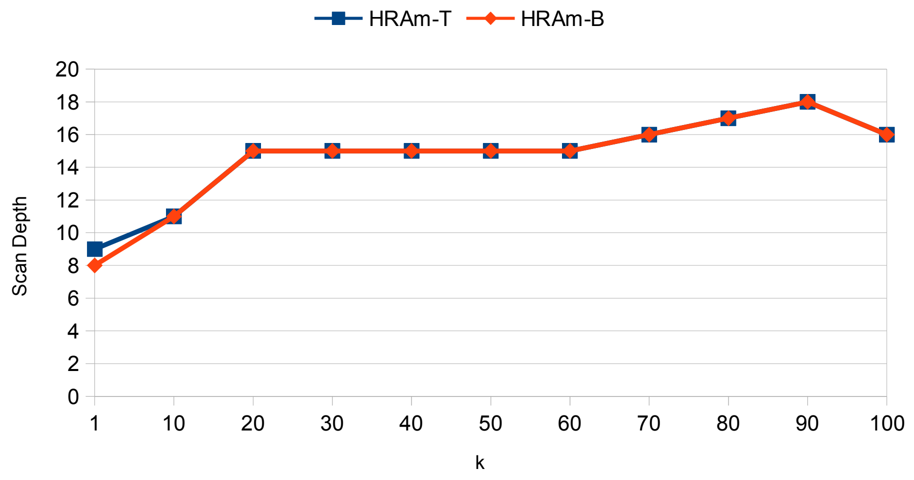


Figure 3.12: Change k (m = 8) - Scan Depth

In the second experiment, the group size is fixed to 32 and the number of required groups changed from 1 to 100. Figure 3.13 compares the running time of the two algorithms. The

Top-Down algorithm shows better running time over the Bottom-Up algorithm. Figure 3.14 compares the total scan depth of the two algorithms which are comparable.

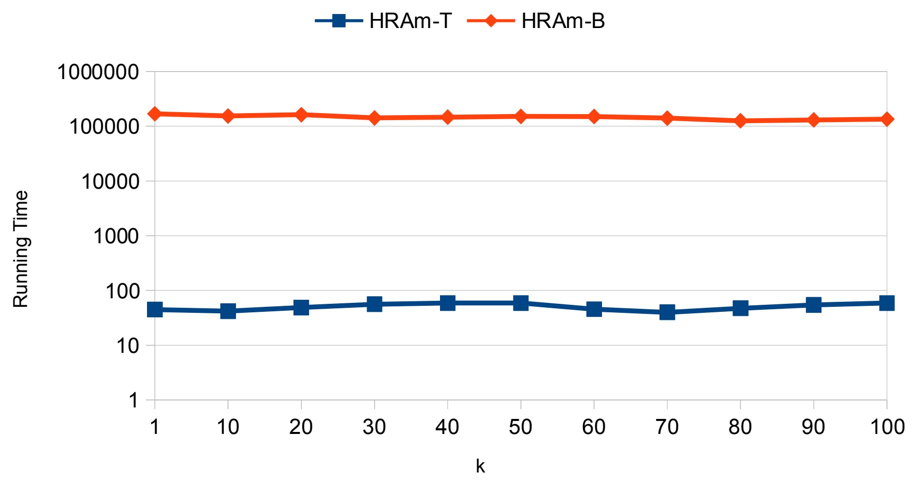


Figure 3.13: Change k (m = 32) - Running Time

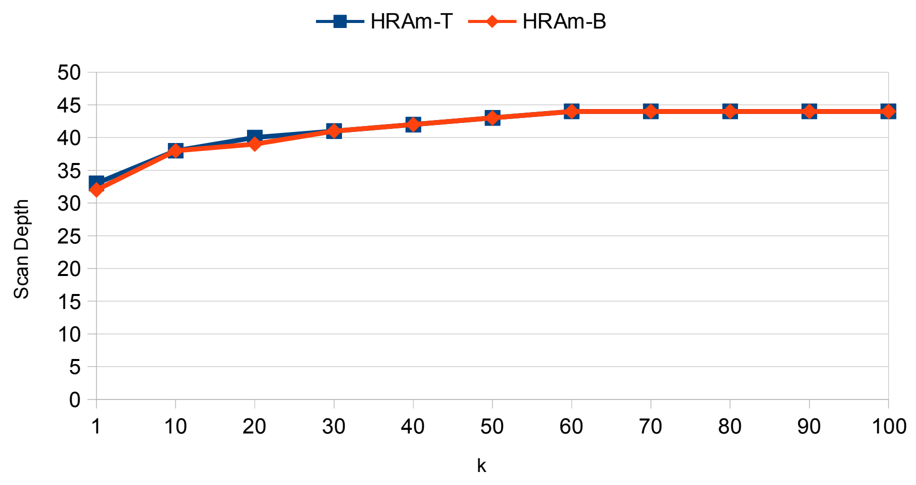


Figure 3.14: Change k (m = 32) - Scan Depth

As a conclusion, each algorithm wins in different configuration. If the group size is small,the Bottom-Up algorithm is used. Otherwise, the Top-Down algorithm is used.

3.5 Summary

In this chapter, the Holistic Ranking Aggregation Query (HRA) with a group-based monotone scoring function was studied. An algorithm based on the Rank-Join problem was introduced and its shortcomings was discussed. Then, two algorithms based on A^* were introduced. One was Bottom-up algorithm and the other was Top-down algorithm. Also, relaxing the assumption of fixed group size was discussed. Finally, experiments comparing the different algorithms were presented.

Chapter 4

HRA - Generic Functions

In this chapter, the Holistic Ranking Aggregation Query (*HRA*) with group-based generic scoring function is studied. The group size is assumed to be fixed. A new algorithm is introduced based on splitting the problem into multiple rank-join problems with monotone function.

Consider the following two examples that will be used for illustration.

Example 4.1. *Consider a relation R, group size required is only one where the scoring function is represented by the following equation.*

$$f(x) = x^4 - \frac{9}{5}x^3 + \frac{103}{100}x^2 - \frac{99}{500}x + \frac{7}{625} \tag{4.1}$$

Figure 4.1 shows the graphical representation of the function and figure 4.2 shows the input relation, the group score and their rank. Tuple t_1 with score $27.19 * 10^{-3}$ is ranked first while tuple t_4 with score $-0.89 * 10^{-3}$ is ranked tenth.

Example 4.2. *Consider a relation R, the group size required is two where the scoring function is represented by the following equation.*

$$f(x, y) = (4x - 2)e^{-((4x-2)^2+(4y-2)^2)} \tag{4.2}$$

Example 4.3 shows graphical representation of the function.

4.1 *HRAG* Algorithm

The idea is to split the problem of holistic ranking using generic function into multiple problems of holistic ranking using monotone function. This can be achieved by determining

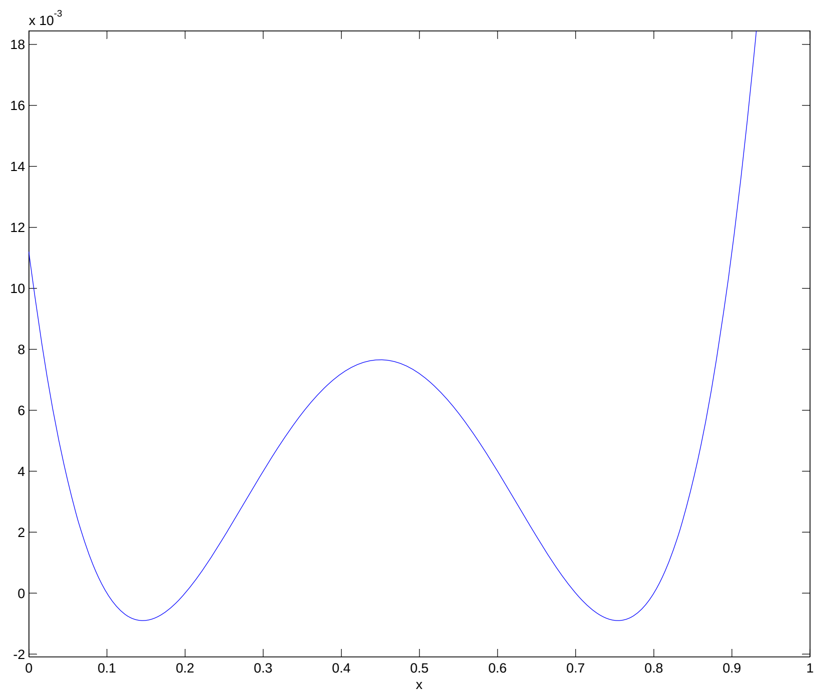
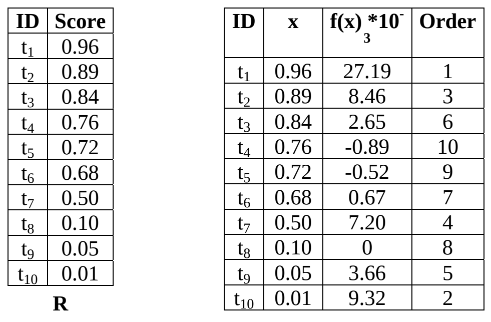


Figure 4.1: Function Graphical Representation



ID	Score
t_1	0.96
t_2	0.89
t_3	0.84
t_4	0.76
t_5	0.72
t_6	0.68
t_7	0.50
t_8	0.10
t_9	0.05
t_{10}	0.01

R

ID	x	f(x)*10^{-3}	Order
t_1	0.96	27.19	1
t_2	0.89	8.46	3
t_3	0.84	2.65	6
t_4	0.76	-0.89	10
t_5	0.72	-0.52	9
t_6	0.68	0.67	7
t_7	0.50	7.20	4
t_8	0.10	0	8
t_9	0.05	3.66	5
t_{10}	0.01	9.32	2

Figure 4.2: Input Relation, Group Score, and Group Rank

the local maxima and minima points of the function. There are multiple ways to get these points, however analyzing such methods is out of the thesis scope. Points are assumed to be distinct where duplicate points are filtered out. Concept of Intervals and Regions are

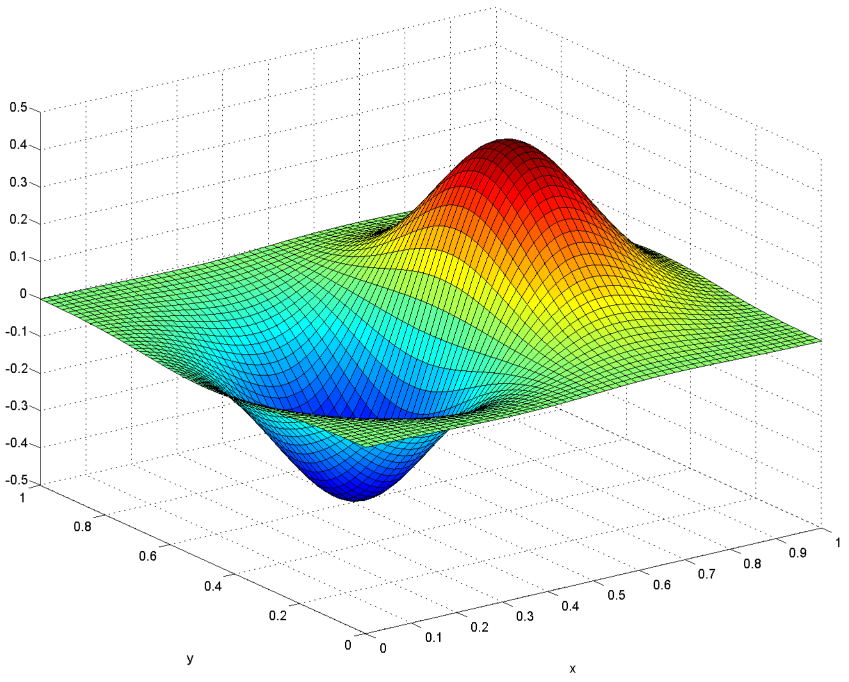


Figure 4.3: Function Graphical Representation

introduced in the following definitions.

Definition 4.1. **[Interval** I**]** *An interval I is a continuous range of points with starting point x_s and ending point x_e. It is written as $I = [x_s, x_e]$.*

Definition 4.2. **[Region** REG**]** *A region REG is an ordered list of intervals I such that there is one interval for each corresponding dimension of the function. The function is either monotonic increasing or monotonic decreasing in the region.*

The construction of intervals for a given set of maxima and minima points is defined in algorithm 7. First, a multidimensional list is created to hold a list of intervals for each dimension. Then, the algorithm iterates over all the dimensions. For each dimension, the values of the points at that dimension are extracted and sorted. Then, intervals are constructed from the values where the first point in the first interval is zero and the last point in the last interval is one.

Theorem 4.1. *The number of intervals in each dimension is less than or equal to the number of given points plus one i.e. $|\mathcal{I}(d)| \leq |P| + 1$. This is valid over all dimensions. Thus, the total number of intervals in all dimensions is less than or equal to the multiplication of the number of dimensions and the number of intervals. i.e. $|\mathcal{I}| \leq d * (|P| + 1)$.*

Algorithm 7 GetIntervals (P)

```
Require: $P$: List of local maxima and minima points
1: Create a list $\mathcal{I}$ to hold a list of intervals
2: $D \leftarrow$ Dimensions of the given points
3: for all dimension $d$ in $D$ do
4:     $\mathcal{I}(d) \leftarrow \{\}$
5:     $pre \leftarrow 0$
6:     $P_d \leftarrow P$ values of dimension $d$
7:     Remove duplicates from $P_d$
8:     Sort $P_d$
9:     for all point $p$ in $P_d$ do
10:        $\mathcal{I}(d) \leftarrow \mathcal{I}(d) \cup \{[pre, p]\}$
11:        $pre \leftarrow p$
12:    end for
13:    $\mathcal{I}(d) \leftarrow \mathcal{I}(d) \cup \{[pre, 1]\}$
14: end for
15: Return $\mathcal{I}$
```

The construction of the regions for a given list of Intervals is defined in algorithm 8. First, a list is created to hold the computed regions. Then, regions are constructed by performing cartesian product of intervals over all dimensions.

Algorithm 8 GetRegions (I)

```
Require: $I$: List of list of intervals
1: Create a list $REG$ to hold regions
2: $d \leftarrow$ Number of dimensions in $\mathcal{I}$
3: $REG \leftarrow \mathcal{I}(1) \times \mathcal{I}(2) \times ... \times \mathcal{I}(d)$
4: Return $REG$
```

Theorem 4.2. *The number of regions is equal to the multiplcation of the number of intervals in each dimension i.e.* $|REG| = |\mathcal{I}(1)| * |\mathcal{I}(2)| * ... * |\mathcal{I}(d)|$.

Figure 4.4 shows example 4.1 after constructing the regions. There are four regions: $R_1 = ([0, 0.1459]), R_2 = ([0.1459, 0.45]), R_3 = ([0.45, 0.7541]), R_4 = ([0.7541, 1])$

Figure 4.5 shows example 4.2 after constructing the regions. There are six regions: $R_1 = ([0, 0.3232], [0, 0.5])$, $R_2 = ([0, 0.3232], [0.5, 1])$, $R_3 = ([0.3232, 0.6768], [0, 0.5])$, $R_4 = ([0.3232, 0.6768], [0.5, 1])$, $R_5 = ([0.6768, 1], [0, 0.5])$, $R_6 = ([0.6768, 1], [0.5, 1])$

The details of the *HRAG* algorithm are illustrated in algorithm 9. First, a priority queue is defined. It is used to hold the groups based on the scoring function provided.

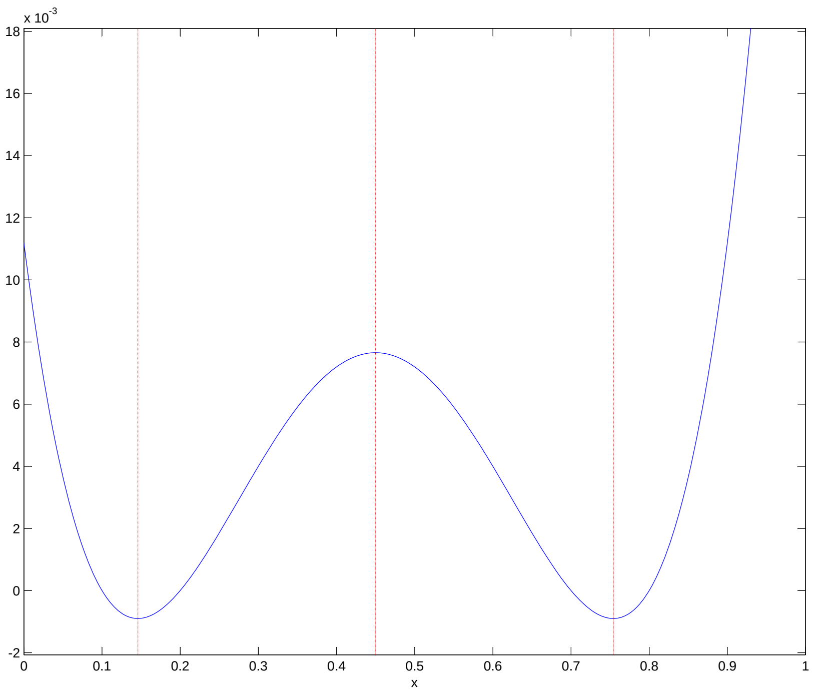


Figure 4.4: Regions

Then, the maxima and minima points P of the function are calculated. Intervals I and regions REG are constructed. Then, the tuples are assigned to the appropriate regions over all the dimensions where each tuple is assigned to one region over each dimension. In the next step, the algorithm iterates over all the regions. For each region, tuples assigned to the intervals are returned as a set of relations. Then, an instance of rank join algorithm is assigned to each region and it is used to get the next group from each region. The algorithm iterates until returning the top k groups. In each iteration, it gets the group on the top of the priority queue and reports it. Then another group is added to the queue from the same region.

Figure 4.6 depicts the first four steps of the $HRAG$ algorithm on example 4.1. After regions construction, tuples are assigned to the corresponding region. $\{t_8, t_9, t_{10}\}$ to $R_1 = ([0, 0.1459])$. No tuple to $R_2 = ([0.1459, 0.45])$. $\{t_5, t_6, t_7\}$ to $R_3 = ([0.45, 0.7541])$. $\{t_1, t_2, t_3, t_4\}$ $R_4 = ([0.7541, 1])$. Then the algorithm starts by getting one group from each region and enqueuing them to the priority queue. The top group t_1 is reported and a new group is constructed from the same region. The process continues until the required number of groups is reported.

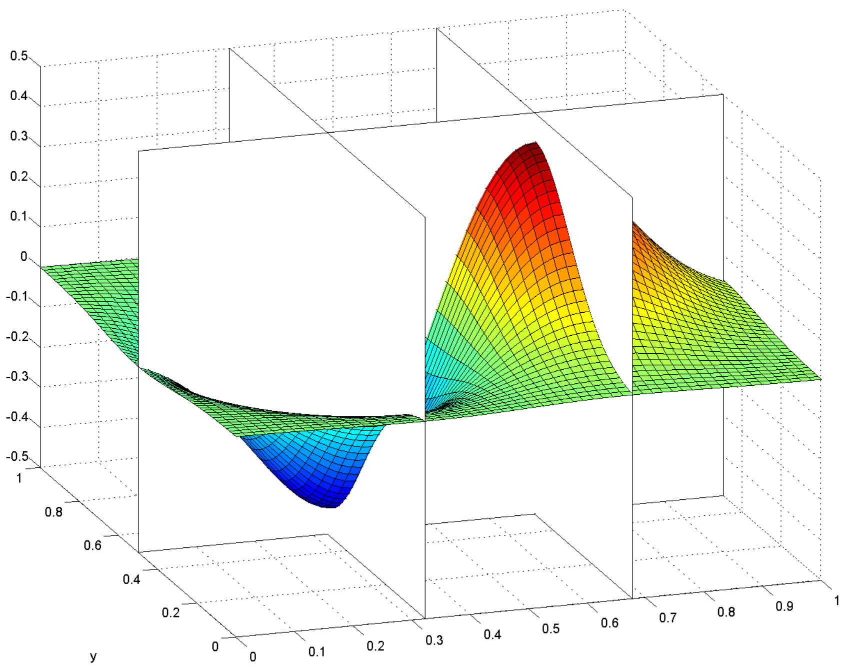
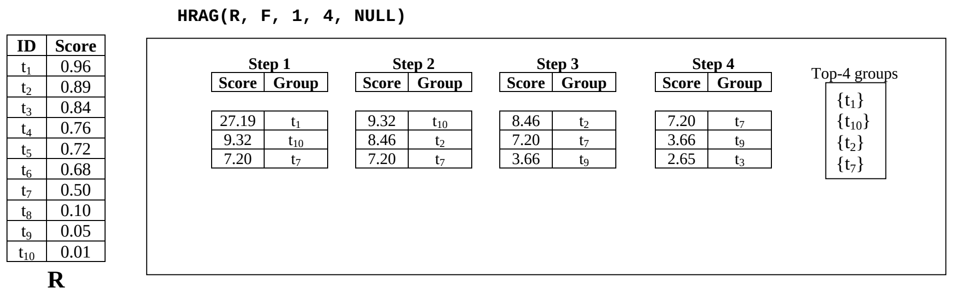


Figure 4.5: Regions



HRAG(R, F, 1, 4, NULL)

ID	Score
t_1	0.96
t_2	0.89
t_3	0.84
t_4	0.76
t_5	0.72
t_6	0.68
t_7	0.50
t_8	0.10
t_9	0.05
t_{10}	0.01

R

Step 1

Score	Group
27.19	t_1
9.32	t_{10}
7.20	t_7

Step 2

Score	Group
9.32	t_{10}
8.46	t_2
7.20	t_7

Step 3

Score	Group
8.46	t_2
7.20	t_7
3.66	t_9

Step 4

Score	Group
7.20	t_7
3.66	t_9
2.65	t_3

Top-4 groups

$\{t_1\}$
$\{t_{10}\}$
$\{t_2\}$
$\{t_7\}$

Figure 4.6: Applying $HRAG$ on example 4.1

4.2 $HRAG - L$ Algorithm

Analyzing figure 4.4, the number of comparable regions can be reduced by considering the maximum and minimum value of each region and splitting regions into smaller ones. First,

Algorithm 9 *HRAG* (R, $\mathcal{F}$, m, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
 1: Create a priority queue $Q$ to order groups based on $\mathcal{F}$
 2: $P \leftarrow$ GetMaximaAndMinima($\mathcal{F}$)
 3: $\mathcal{I} \leftarrow$ GetIntervals($P$)
 4: $REG \leftarrow$ GetRegions ($\mathcal{I}$)
 5: Assign tuples in $R$ to intervals
 6: for all region $reg$ in $REG$ do
 7:     $\mathcal{R} \leftarrow reg$.GetRelations()
 8:     $reg$.assign($RJ(\mathcal{R}, \mathcal{F}, k, C)$)
 9:     $Q$.add($reg$.getNextGroup())
10: end for
11: while |reported groups| $\leq k$ do
12:     $g \leftarrow$ Q.dequeue()
13:     report $g$
14:     $Q$.add($g$.getRegion().getNextGroup())
15: end while
```

the definition of level is introduced.

Definition 4.3. [Level L**]** *A level L is a set of comparable regions.*

The construction of levels for a given function is defined in algorithm 10. First, a list is created to hold the computed levels. Then, the function is applied to all the points. These points are sorted after duplicates removal. Then, levels are constructed from these values where the first point of the first level is $-\infty$ and the last point of the last level is ∞. Finally the order of levels in L is reversed.

Figure 4.7 shows example 4.1 after constructing the levels. There are three levels; the first one has no region, the second one has four regions, and the last one has two regions.
$L_1[0.0076, \infty] : \{([0, 0.0199]), ([0.8801, 1])\}$
$L_2[-0.0009, 0.0076] : \{([0.0199, 0.1459]), ([0.1459, 0.45]), ([0.45, 0.7541]), ([0.7541, 0.8801])\}$
$L_3[-\infty, -0.0009] : \{\}$

For example 4.2, there are three levels; the first and the third levels has no region and the last one has six regions.
$L_1[0.4288, \infty] : \{\}$
$L_2[-0.4288, 0.4288] : \{([0, 0.3232], [0, 0.5]), ([0, 0.3232], [0.5, 1]),$
$([0.3232, 0.6768], [0, 0.5]), ([0.3232, 0.6768], [0.5, 1]),$
$([0.6768, 1], [0, 0.5]), ([0.6768, 1], [0.5, 1])\}$ $L_3[-\infty, -0.4288] : \{\}$

The details of the $HRAG - L$ are illustrated in algorithm 11. First, a priority queue is defined. It is used to hold the groups based on the scoring function provided. Then,

Algorithm 10 GetLevels (P, $\mathcal{F}$)

```
Require: $P$: List of local maxima and minima and $\mathcal{F}$: group-based scoring function
 1: Create a list $\mathcal{L}$ to hold the constructed levels
 2: $\mathcal{L} \leftarrow \{\}$
 3: $pre \leftarrow -\infty$
 4: $P_{\mathcal{F}} \leftarrow$ Apply $\mathcal{F}$ on $P$
 5: Remove duplicates from $P_{\mathcal{F}}$
 6: Sort $P_{\mathcal{F}}$
 7: for all point $p$ in $P_{\mathcal{F}}$ do
 8:     $\mathcal{L} \leftarrow \mathcal{L} \cup \{[pre, p]\}$
 9:     $pre \leftarrow p$
10: end for
11: $\mathcal{L} \leftarrow \mathcal{L} \cup \{[pre, c]\}$
12: Reverse the order of of $\mathcal{L}$
13: Return $\mathcal{L}$
```

Algorithm 11 $HRAG - L$ (R, $\mathcal{F}$, m, k, C)

```
Require: $R$: ranked relation and $\mathcal{F}$: group-based scoring function
 1: Create a priority queue $Q$ to order groups based on $\mathcal{F}$
 2: $LI \leftarrow$ GetLevels($P$, $\mathcal{F}$)
 3: $P \leftarrow$ GetIntersection($LI$, $\mathcal{F}$) $\cup$ GetIntersection($LI$, $\mathcal{F}$)
 4: $I \leftarrow$ GetIntervals($\mathcal{P}$)
 5: $REG \leftarrow$ GetRegions ($I$)
 6: $L \leftarrow$ GetLevels ($REG$)
 7: Assign tuples in $R$ to intervals
 8: while |reported groups| $\leq k$ do
 9:     while $Q$ is empty do
10:         $l \leftarrow L$.getNextGroup()
11:         for all region $reg$ in $l$.getRegions() do
12:             $\mathcal{R} \leftarrow reg$.GetRelations()
13:             $reg$.assign($RJ(\mathcal{R}, \mathcal{F}, k, C)$)
14:             $Q$.add($reg$.getNextGroup())
15:         end for
16:     end while
17:     $g \leftarrow$ Q.dequeue()
18:     report $g$
19:     $Q$.add($g$.getRegion().getNextGroup())
20: end while
```

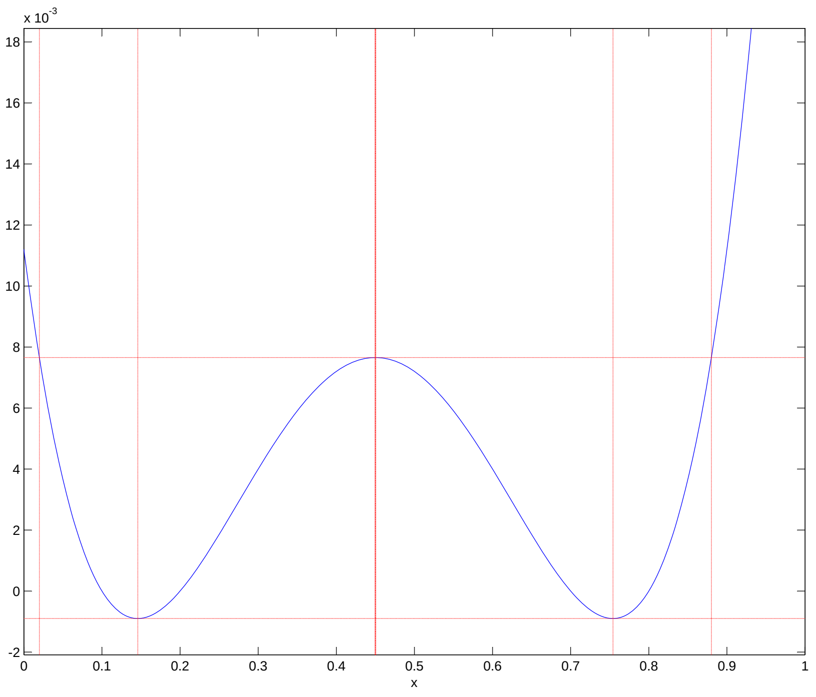


Figure 4.7: Levels and Regions

intervals, regions, and levels are computed and tuples are distributed over the corresponding regions. In the next step, the algorithm iterates over each level. For each region, tuples assigned to the intervals are returned as a set of relations. Then, an instance of rank join algorithm is assigned to each region and it is used to get the next group from each region. The algorithm iterates until returning the top k groups. In each iteration, it gets the group on the top of the priority queue and reports it. Then another group is added to the queue from the same region.

Figure 4.8 depicts the first four steps of the $HRAG-L$ algorithm on example 4.1. After regions construction, tuples are assigned to the corresponding region. $\{t_{10}\}$ to ($[0, 0.0199]$). $\{t_1, t_2\}$ to ($[0.8801, 1]$). $\{t_8, t_9\}$ to ($[0.0199, 0.1459]$). No tuple to ($[0.1459, 0.45]$). $\{t_5, t_6, t_7\}$ to ($[0.45, 0.7541]$). $\{t_3, t_4\}$ to ($[0.7541, 0.8801]$).

Then the algorithm starts by getting one group from each region in the first level and enqueuing them to the priority queue. The top group t_1 is reported and a new group is constructed from the same region. If the queue is empty and no additional group in the current level, a new level is introduced. The process continues until the required number of groups is reported.

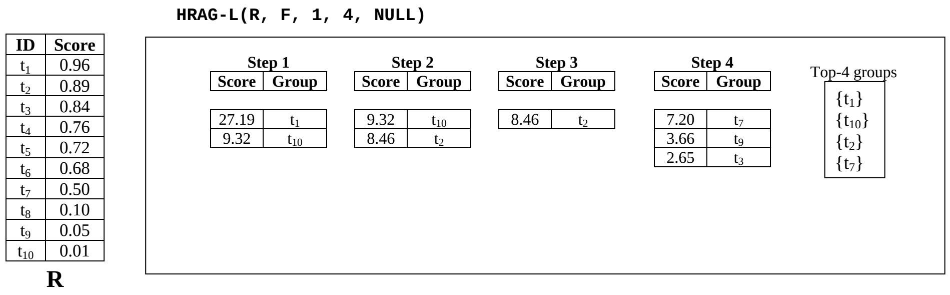


Figure 4.8: Applying $HRAG - L$ on example 4.1

4.3 Experiments

All experiments were conducted on a SunFire X4100 server with Dual Core 2.2GHz processor, and 8GB of RAM. The parameters of the experiments are the relation size R and the number of the groups required k. Two metrics are reported, the overall running time and the scan depth.

In the first experiment, the relation size is fixed to 1000 and the number of required groups changed from 1 to 50. Figure 4.9 compares the running time of the three algorithms. The new algorithms shows better running time over the algorithm based on Brute Force. Figure 4.10 compares the total scan depth of the three algorithms which are comparable.

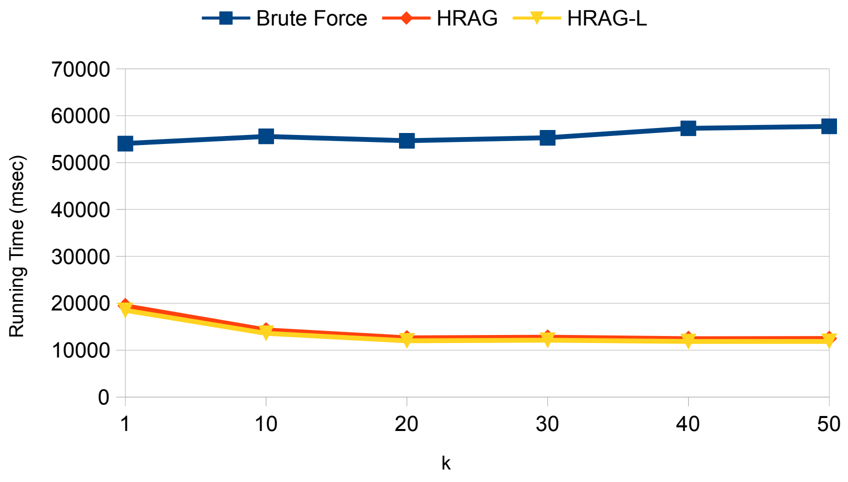


Figure 4.9: Change k - Running Time

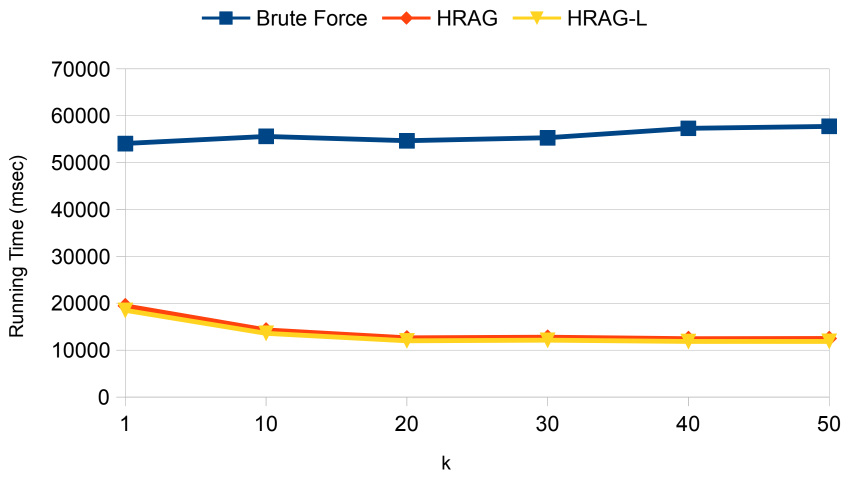


Figure 4.10: Change k - Scan Depth

In the second experiment, the number of required groups is fixed to 50 and the relation size changed from 100 to 1000. Figure 4.11 compares the running time of the three algorithms. The new algorithms shows better running time over the algorithm based on Brute Force.

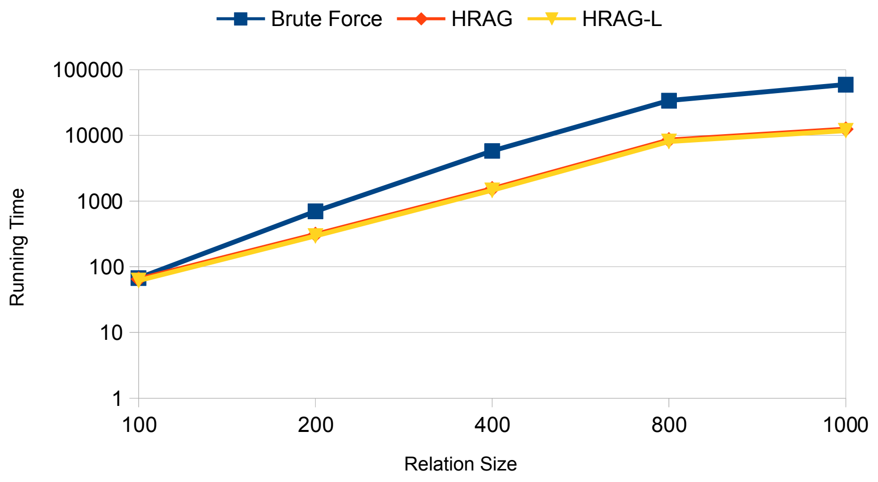


Figure 4.11: Change Relation Size - Running Time

As a conclusion, the new algorithms show better performance over the Brute Force approach.

4.4 Summary

In this chapter, the Holistic Ranking Aggregation Query (HRA) with group-based generic scoring function was studied. First, a new algorithm was introduced based on splitting the problem into multiple problems of holistic ranking aggregation with monotone function. Then, experiments comparing the different algorithms were presented.

Chapter 5

Conclusion and Future Work

In this chapter, the dissertation is concluded and directions for future work are presented.

5.1 Conclusion

In this dissertation, the Holistic Ranking Aggregation problem was formalized. A comprehensive survey about the interaction between ranking and aggregation was introduced. Algorithms were proposed to efficiently process ad-hoc holistic ranking aggregation for both monotone and generic scoring functions.

In the survey, multiple classifications of ranking aggregation processing techniques were introduced. One classification dealed with different interactions between ranking and grouping operations. Another one dealed with the different types of aggregation. Ranking Aggregation processing techniques were also classified according to the level of integration with the database systems. The last classification dealed with the type of the ranking function.

The Holistic Ranking Aggregation Query with a group-based monotone scoring function was studied. An algorithm based on the Rank-Join problem was introduced and its shortcomings was discussed. Then, two algorithms based on A^* were introduced. One was Bottom-up algorithm and the other was Top-down algorithm. Experiments comparing the different algorithms were presented.

Finally, the Holistic Ranking Aggregation Query with group-based generic scoring function was studied. Multiple algorithms were proposed based on splitting the problem into multiple problems of holistic ranking aggregation with monotone function. Experiments were held to show the efficiency of the new algorithms.

5.2 Future Work

The main direction of the future work is enriching the literature of the Holistic Ranking Aggregation problem.

• Ranking Aggregation Processing Techniques

As discussed in table 2.1, this dissertation focuses on the Group_then_Rank query model, holistic aggregation type, and application level implementation. Various interesting problems can be defined with other types. One of these problem is Rank_within_Group problem with holistic aggregation. In this problem, the user is interested in the top candidates of each group where the scoring function is holistic.

Another interesting problem is adding the holistic ranking aggregation to the database engine. This includes implementing it as operator, formalizing it in relational algebra, and defining various optimization techniques. This allows the engine to apply the appropriate optimization techniques to the new operators.

• Relax Problem Constraints

Constraints on the holistic ranking aggregation problem can be relaxed. In this dissertation, a special type of scoring functions was defined. A scoring function is a group-based scoring function if it is evaluated over all the group members and does not depend on other tuples. This restriction can be relaxed and the scoring function can be evaluated over any tuple.

Another interesting problem is relaxing the constraint that each tuple can belong only to one group. Thus, tuples can belong to different groups. Also, membership may be probabilistic where a tuple can belong to one group with certain probability and belong to another group with another probability.

APPENDICES

Appendix A

Using Dominanace in Rank-Join Problem

Rank-join problem is defined in definition 3.1.1. Given multiple relations and a scoring function, it computes the k join results over the given relations with the highest scores.

The idea of dominance studied in section 3.2.2 can be applied to the rank-join problem. In this appendix a new rank join algorithm based on the dominance is introduced. Then a comprehensive comparison between the new algorithm and the two famous rank-join algorithms J^* and $HRJN^*$ is presented.

A.1 DRJ Algorithm

A state, representing a candidate result, is defined as a decimal string with fixed size equal to the number of the given relations. The values at position p corresponds to the number of the tuples coming from the relation R_p. Figure A.1 shows the Hasse diagram of the state dominance where the number of relations to join is three.

The details of the DRJ algorithm are illustrated in algorithm 12. First, the algorithm creates a priority queue to hold the states based on the scoring function provided and initializesinitializes it by adding a state of length equal to the number of the input relations. Also, it creates a hash table to hash the tuples retrieved using the order of retrieval as a key. Then, the algorithm iterates until returning the top k groups. In each iteration, it gets the state on the top of the priority queue. If the state is a valid join, it constructs the corresponding result and reports it. Then it iterates over each relation in the state. For each relation, it retrieves the next tuple either from the hash or by calling the $getNextTuple$ function. Then it create the new state. If it is a new state it adjusts their scores using the new tuple score and adds it to the priority queue.

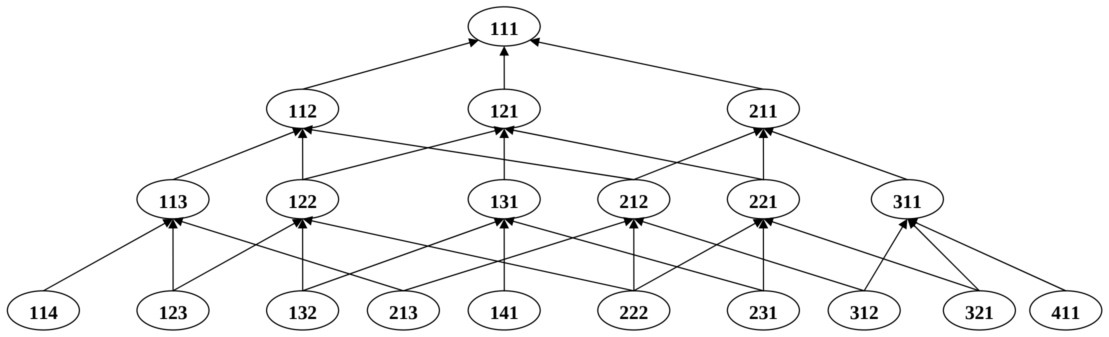


Figure A.1: The Hasse diagram of the state dominance

A.2 J^* Algorithm

J^* is a rank-join algorithm introduced by Natsev *et al* [13]. There are two types of states; complete and partial. A complete state represents a complete result similar to the state defined in DRJ . A partial state represents an incomplete result. The missing members are identified using placeholders pointing to the next tuple to be scanned. J^* algorithm is based on A^* search. The main idea is to maintain a priority queue of partial and complete states ordered on the upper bound of their scores. At each step, the algorithm examine the top state. The complete state is reported but the partial state is extended to two states and added to the priority queue. Figure A.2 shows the state space expansion for joining three relations.

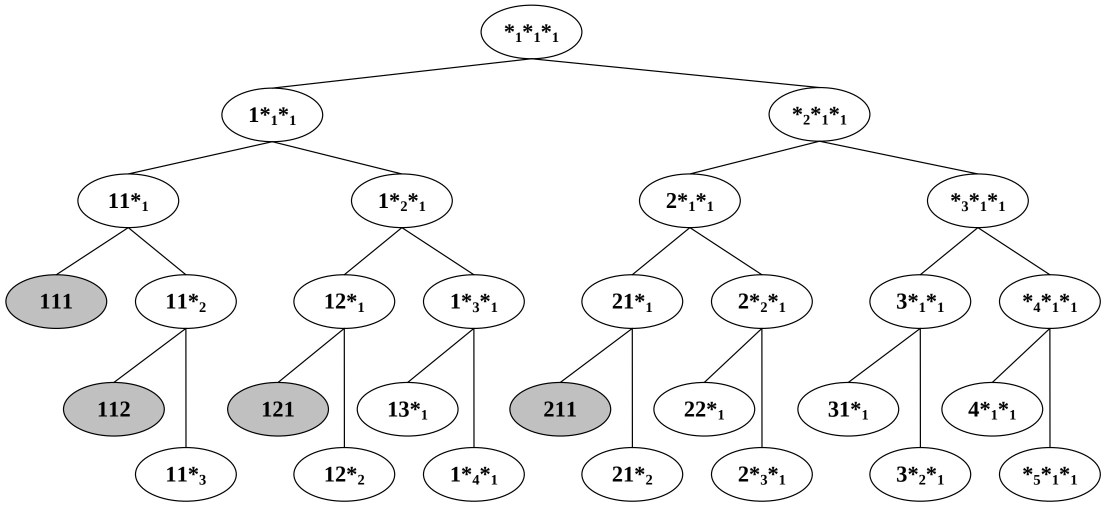


Figure A.2: J^* state space expansion for three relations

Algorithm 12 *DRJ* ($\mathcal{R}$, $\mathcal{F}$, k)

```
Require: $\mathcal{R}$: set of ranked relation
1: Create a priority queue $Q$ to order the states based on $\mathcal{F}$
2: Create a hash table $H$ to hash the tuples retrieved, the key is the order of the tuple retrieval
3: $s \leftarrow 11...1$ where $|s| = |\mathcal{R}|$
4: $Q$.enqueue($s$)
5: while |reported tuples| $\leq k$ do
6:     $s \leftarrow$ Q.dequeue()
7:     if $s$ is valid join then
8:         $j \leftarrow s$.constructResult()
9:         report j
10:    end if
11:    for $i = 0$ to $|s|$ do
12:        if $(s[i] + 1)$ exists in H then
13:            $t \leftarrow H$.get($s[i] + 1$)
14:        else
15:            $t \leftarrow R$.getNextTuple()
16:            $H$.add($s[i] + 1$, $t$)
17:        end if
18:        $s^{'} \leftarrow s$
19:        $s^{'}[i] \leftarrow s[i] + 1$
20:        if $s^{'}$ is a new state then
21:            $s^{'}$.adjustScore(i, t)
22:            Q.enqueue($s^{'}$)
23:        end if
24:    end for
25: end while
```

A.3 $HRJN^*$ Algorithm

$HRJN^*$ is a rank-join algorithm introduced by Ilyas *et al* [6]. Figure A.3 shows the state space expansion for joining three relations.

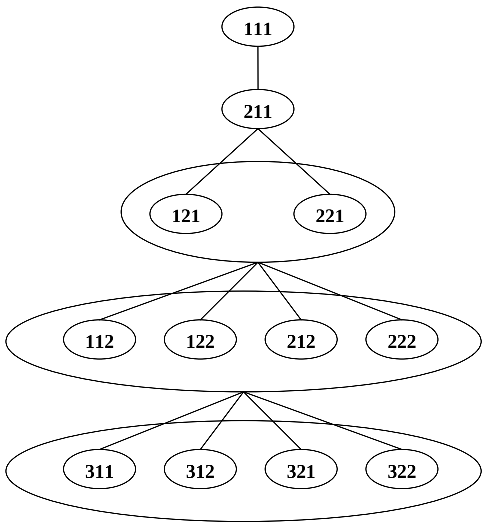


Figure A.3: $HRJN^*$ state space expansion for three relations

A.4 Analysis

DRJ algorithm is compared to J^* and $HRJN^*$ algorithms.

States
J^* deals with both partial and complete states, while $HRJN^*$ and DRJ deals with complete states only.

State Generation
In J^* , the generator is the partial states. Given a partial state, it introduces two states; one considers the corresponding tuple and the other ignores it. In $HRJN^*$, the generation process is independent of the states. Given a new tuple, it generates all the valid complete results that contain t and the tuples seen. In DRJ , the generator is the complete state. Given a complete state, it generates only states that are candidates to be on the top of the priority queue.

Joining
J^* check the state validity after generation and before inserting in the priority queue. $HRJN^*$ constructs valid joins using a hash structure. DRJ check the state validity after removed from the top of the priority queue not on generation. J^* and DRJ can work on general joining condition while $HRJN^*$ works only on equality joins.

Pruning
In J^* , if a state has a property then all its descendants will have the same property.

This enables J^* to prune the state if the property is invalid and as a consequent all its descendants will not be generated.

In $HRJN^*$, the generation technique prevent the generation of invalid states, only valid states are materialized.

In DRJ , since the generation is dependent on the states and not all the descendants of a state have the same properties, thus no pruning is allowed. This means that the subsequent invalid states can be generated.

Reporting
In J^* , a complete state on the top of the priority queue is reported. In $HRJN^*$, a complete state on the top of the priority queue and satisfying the threshold is reported. In DRJ , a complete state on the top of the priority queue and is satisfying the joining condition is reported.

Search Technique
J^* is based on A^* Search, $HRJN^*$ is based on Breadth First Search, and DRJ is based on Uniform Cost Search.

A.5 Experiments

To evaluate the performance of the new rank-join algorithm DRJ , it is compared to the two famous rank-join algorithms $HRJN^*$ and J^* .

Three performance metrics are presented; Maximum Queue Size, Total Scan Depth, and Overall Running Time.

Experiments are repeated for two different implementations; Two-way (Operator) and Multi-way. In the Two-way implementation, DRJ makes use of expanding minimum number of states but in the Multi-way it does not make use of global information from different relations and does not perform early pruning.

Change K
In this experiment, the number of required answers changed from 1 to 100 and join selectivity is fixed to 0.2 and the number of relations is fixed to 4. Figure A.4a and A.4b compare the maximum queue size. In the Two-way implementation, DRJ uses the smallest queue size. However in the Multi-way implementation DRJ has the worst maximum queue size.

Figure A.4c and A.4d compare the total scan depth. The three algorithms are comparable in the total scan depth in both implementations.

Figure A.4e and A.4f compare the overall running time. In the two-way implementation, DRJ shows better overall running time than J^* . However it shows worse overall running time than $HRJN^*$. In the Multi-way implementation, DRJ has the worst overall running time.

Change Number of Relations
In this experiment, the number of input relations changed from 3 to 6, the join selectivity is fixed to 0.2 and number of required answers is fixed to 50. Figure A.5a and A.5b compare the maximum queue size. In the Two-way implementation, DRJ uses the smallest queue size. However in the Multi-way implementation DRJ has the worst maximum queue size.

Figure A.5c and A.5d compare the total scan depth. The three algorithms are comparable in the total scan depth in both implementations.

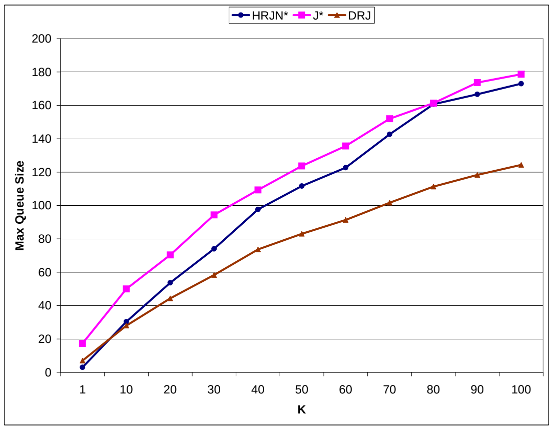
Figure A.5e and A.5f compare the overall running time. In the two-way implementation, DRJ shows better overall running time than J^* . However it shows worse overall running time than $HRJN^*$ except when the number of relations is 5. In the Multi-way implementation, DRJ has the worst overall running time.

Change Join Selectivity
In this experiment, the join selectivity changed from 0.2 to 1.0, the number of required answers is fixed to 50 and number of relations is fixed to 4. Figure A.6a and A.6b compare the maximum queue size. In the Two-way implementation, DRJ uses the smallest queue size. However in the Multi-way implementation DRJ has the worst maximum queue size.

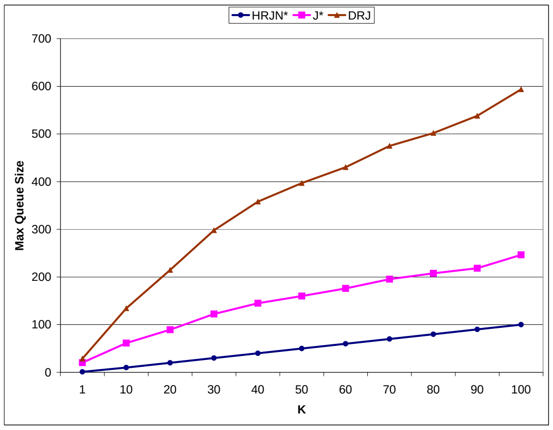
Figure A.6c and A.6d compare the total scan depth. The three algorithms are comparable in the total scan depth in both implementations.

Figure A.6e and A.6f compare the overall running time. In the two-way implementation, DRJ shows better overall running time than J^* . However it shows worse overall running time than $HRJN^*$. In the Multi-way implementation, DRJ has the worst overall running time.

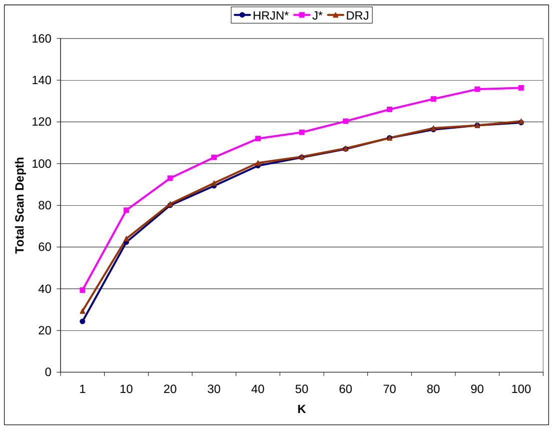
Both $HRJN^*$ and J^* make advantages of early pruning. Moreover $HRJN^*$ makes advantages of fixed size priority queue.



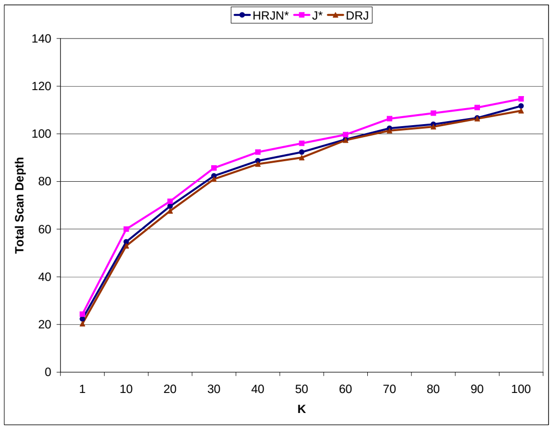
(a) Max Queue Size (Two-way)



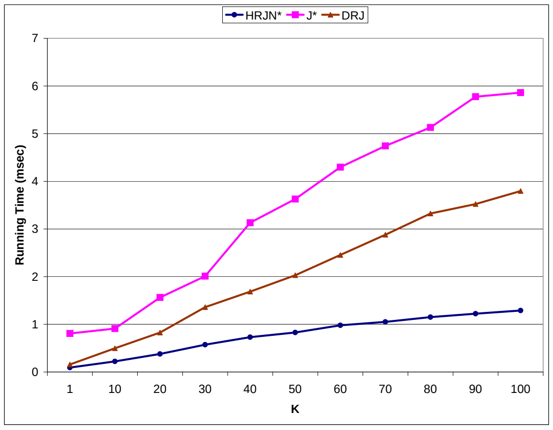
(b) Max Queue Size (Multi-way)



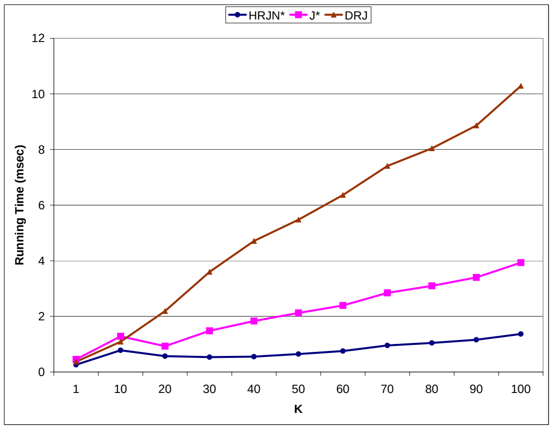
(c) Total Scan Depth (Two-way)



(d) Total Scan Depth (Multi-way)

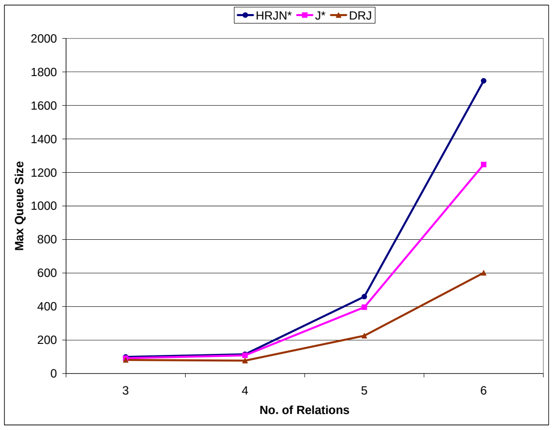


(e) Overall Running Time (Two-way)

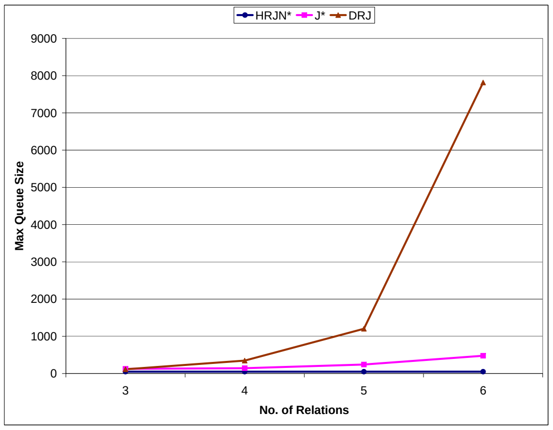


(f) Overall Running Time (Multi-way)

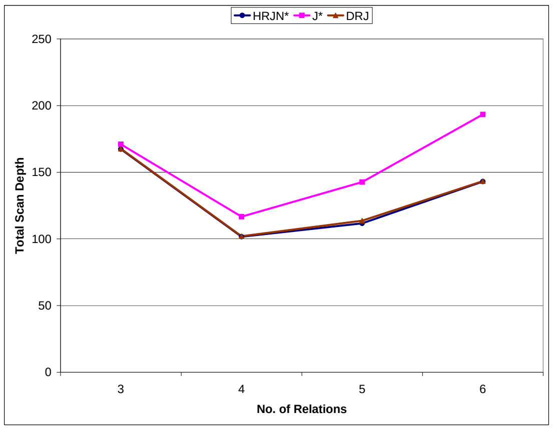
Figure A.4: Change K



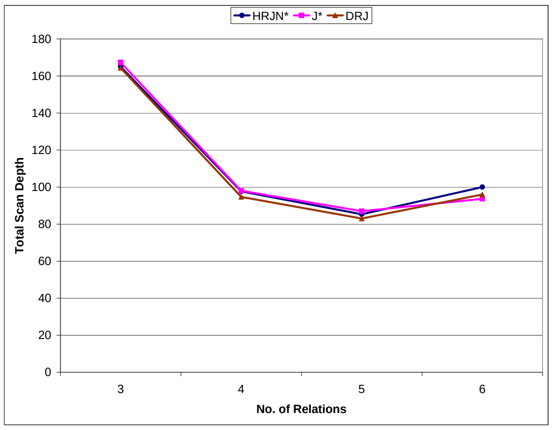
(a) Max Queue Size (Two-way)



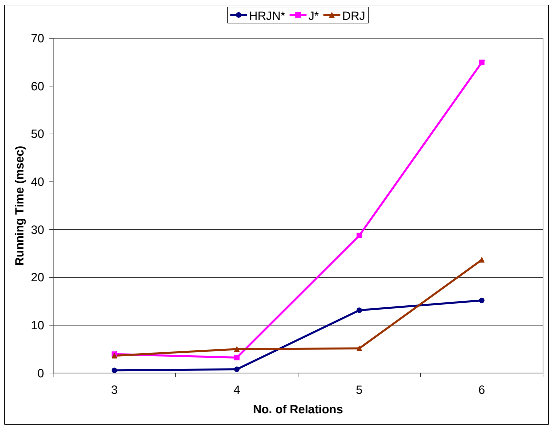
(b) Max Queue Size (Multi-way)



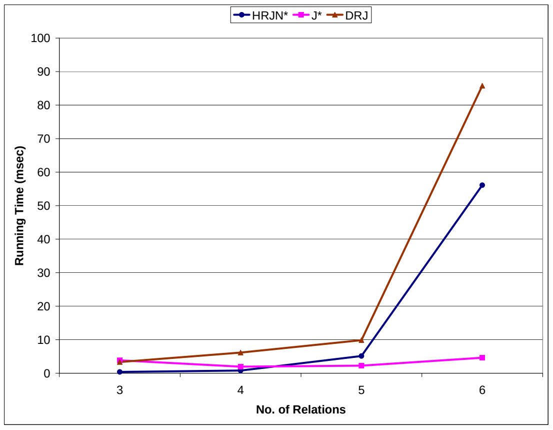
(c) Total Scan Depth (Two-way)



(d) Total Scan Depth (Multi-way)

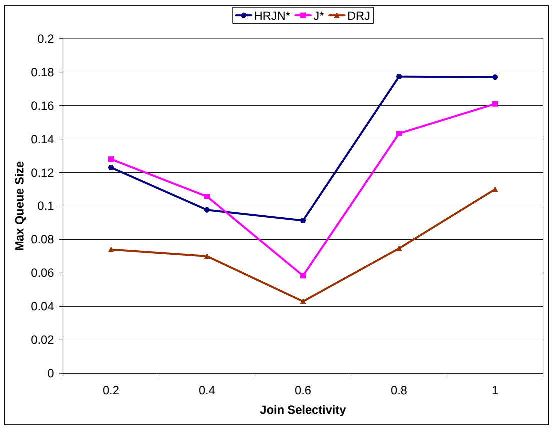


(e) Overall Running Time (Two-way)

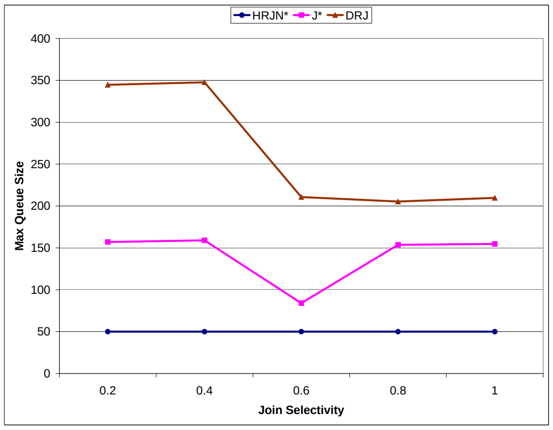


(f) Overall Running Time (Multi-way)

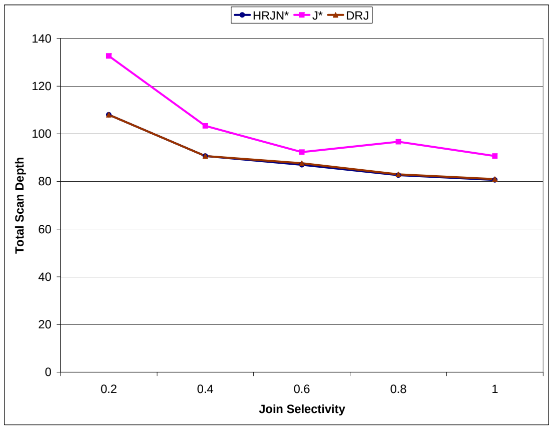
Figure A.5: Change Number of Relations



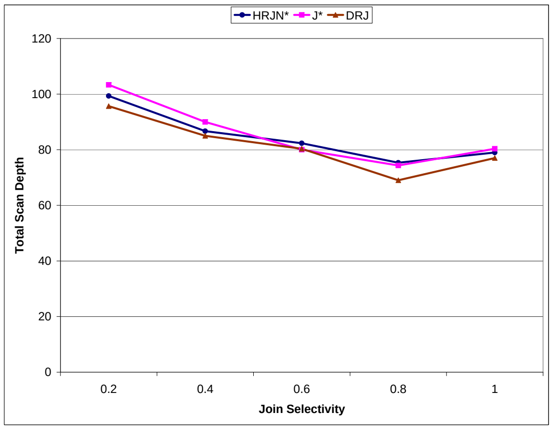
(a) Max Queue Size (Two-way)



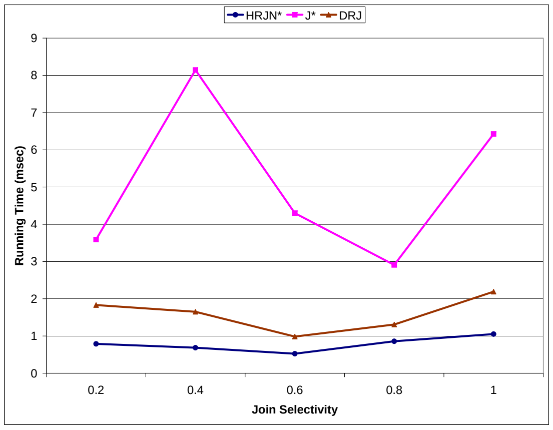
(b) Max Queue Size (Multi-way)



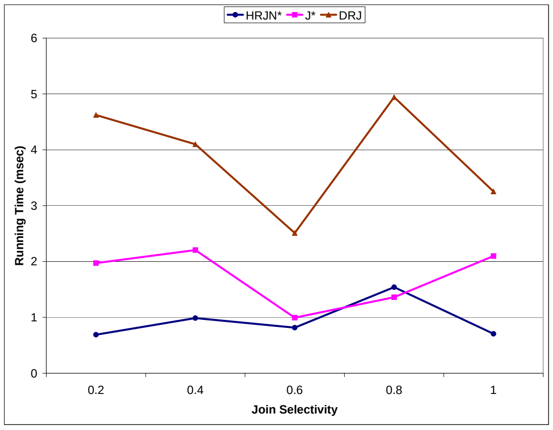
(c) Total Scan Depth (Two-way)



(d) Total Scan Depth (Multi-way)



(e) Overall Running Time (Two-way)



(f) Overall Running Time (Multi-way)

Figure A.6: Change Join Selectivity

Bibliography

[1] Shyam Antony, Ping Wu, Divyakant Agrawal, and Amr El Abbadi. Aggregate skyline: Analysis for online users. *Applications and the Internet, IEEE/IPSJ International Symposium on*, 0:50–56, 2009.

[2] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*, pages 25–71. Springer Berlin Heidelberg, 2006.

[3] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 102–113. ACM, 2001.

[4] Hua gang Li, Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Ranking aggregates. Technical report, 2004.

[5] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[7] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):1–58, 2008.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[9] Richard E. Korf. Artificial intelligence search algorithms. In *In Algorithms and Theory of Computation Handbook*. CRC Press, 1996.

[10] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 61–72, 2006.

[11] Chengkai Li, Min Wang, Lipyeow Lim, Haixun Wang, and Kevin Chen-Chuan Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 127–138, 2007.

[12] Jim Melton. Database language sql. In *Handbook on Architectures of Information Systems*, pages 105–132. 2006.

[13] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 281–290, 2001.

[14] Yasin N. Silva, Ahmed M. Aly, Walid G. Aref, and Per-Ake Larson. Simdb: a similarity-aware database system. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 1243–1246, 2010.

[15] Yasin N. Silva, Walid G. Aref, and Mohamed H. Ali. Similarity group-by. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 904–915, 2009.

[16] Yasin N. Silva, Muhammad U. Arshad, and Walid G. Aref. Exploiting similarity-aware grouping in decision support systems. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 1144–1147, 2009.

[17] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin. Top-k query processing in uncertain databases. In *In ICDE*, pages 896–905, 2007.

[18] Xiaoxia Wang and Max Bramer. Exploring web search results clustering. In *Research and Development in Intelligent Systems XXIII*, pages 393–397. Springer, 2007.

[19] Bin Wu and Ajay D. Kshemkalyani. Objective-greedy algorithms for long-term web prefetching. In *NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium*, pages 61–68, 2004.

[20] Dong Xin, Jiawei Han, and Kevin C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 103–114, 2007.

[21] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.

[22] Weipeng Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.

[23] Man Lung Yiu, Nikos Mamoulis, and Vagelis Hristidis. Extracting k most important groups from data efficiently. *Data & Knowledge Engineering*, 66(2):289–310, August 2008.

[24] Chengyang Zhang and Yan Huang. Cluster by: a new sql extension for spatial data aggregation. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–4, 2007.