Technische Universität München

Fakultät für Informatik

Masterarbeit

Evaluation of Optimization Techniques for Aggregation

Chenxi Li

Technische Universität München

Fakultät für Informatik

Masterarbeit

Evaluation of Optimization Techniques for Aggregation

Evaluation von Optimierungstechniken fr Aggregation

Author:	Chenxi Li
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Viktor Leis
Submission Date:	15.01.2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

January 16, 2017 Chenxi Li

Acknowledgments

I would like to thank Dr. Viktor Leis for his ideas and instructions while I am conducting the work. I also received much help from him and Prof. Dr. Thomas Neumann when I was in the lecture course *"Database Systems on Modern CPU Architectures"* and practical course *"Database Implementation"*. I really appreciate chair for database systems of TUM that they lead me to the world of database systems.

I also would like to thank Bernhard Radke for his program which implements DPccp and index-based join sampling. It saves me so much time that I can focus on eager aggregation techniques. And estimates for group by become much better with his estimation method for join.

I really appreciate my parents for their financial support without which I cannot finish the master.

Abstract

Aggregations are almost always done at the top of operator tree after all selections and joins in a SQL query. But actually they can be done before joins and make later joins much cheaper when used properly. Although some enumeration algorithms considering eager aggregation are proposed, no sufficient evaluations are available to guide the adoption of this technique in practice. And no evaluations are done for real data sets and real queries with estimated cardinalities. That means it is not known how eager aggregation performs in the real world.

In this thesis, a new estimation method for group by and join combining traditional estimation method and index-based join sampling is proposed and evaluated. Two enumeration algorithms considering eager aggregation are implemented and compared in the context of estimated cardinality. We find that the new estimation method works well with little overhead and that under certain conditions, eager aggregation can dramatically accelerate queries.

Contents

1 Introduction

1.1 Relational Database Management System

Relational model [3] was proposed in 1970. It has two very important ideas that influence future database products called relational database management system(RDMBS). The representation of data and access path to data should be independent and not known to end users. With structured query language(SQL) [1], users can specify the kind of data they want and don't need to consider how the data is retrieved and processed. This greatly simplifies the use of RDBMS, while choosing a query plan is not an easy problem and performance with different plans can be orders of magnitudes different. A cost based query optimizer [9] compares the cost of each plan in a enumeration space and take the plan with minimum cost.

1.2 Query Optimizer

A classic query optimizer consists of enumeration space, cost model and cardinality estimation. All components are evaluated in [5] and it finds that cardinality estimation has the biggest influence on the quality of the produced plan.

1.2.1 Enumeration Space

Dynamic programming is usually used to explore enumeration space when the number of tables are not too large. It can memorize best plans for all intermediate results until obtaining the best plan when all tables are joined. Three dynamic programming based enumeration algorithms which can freely reorder inner joins including DPsize, DPsub and DPccp are described and compared in detail in [8]. DPsize enumerates join order bottom up in the order of number of tables. It obtains cost of all intermediate results with *n-1* tables before obtaining intermediate results with *n* tables. DPsub iterates over all possible non-empty subsets. Both DPsize and DPsub can encounter intermediate results that can only be joined through cartesian join which should be avoided as it has bigger possibility to largely increase intermediate cardinalities. Then such intermediate results are abandoned. DPcpp builds a query graph before enumerating and will only enumerate connected pairs and thus can avoid cartesian join. [4] pushes it further and can reorder all kinds of joins and group by.

Although dynamic programming is a wonderful technique, it can be quite expensive when joining a large number of tables. Heuristic algorithms such as [10] which can produce sound plans efficiently are good choices in this situation.

Another consideration in enumeration space is the shape of join trees. Left-deep tree, right-deep tree, zig-zag tree and bushy tree are evaluated in [5].

1.2.2 Cost Model

When evaluating whether a plan is good or not, we need to have a cost for each plan. Plan with a minimum cost is considered by the optimizer to be the best plan, though it may be not the best in reality. To model the cost of plan, we need to model the cost of each physical operator such as table scan, selection and join. In traditional RDBMS, accessing to disk is the bottleneck, which means it should have a bigger proportion in the cost function. Except for disk access, cpu performance, memory bandwidth should also be considered. In main memory RDBMS, the cost model can be much simpler. [5] shows that there is no much difference on the produced plan when using a sophisticated or a simple cost model. But it doesn't mean a sophisticated cost model is not useful. [7] shows a very sophisticated cost model considering even TLB and cache misses.

1.2.3 Cardinality Estimation

Cardinalities are very important inputs for cost model, especially in main memory database. Although decades of research have been done and some sophisticated approach exist, estimation method used in production is quite simple. For example, PostgreSQL has a statistic and formula based approach which assumes inclusive, uniformity and independence of data. This can be accurate when all assumptions meet, but quite bad when there are join crossing correlations. Different from PostgreSQL, some systems such as HyPer have a sampling based approach. In query optimization phase, selections on base tables and joins are done on samples of tables to estimate cardinalities. Currently, HyPer only use sampling approach to estimate base tables. [11] obtains a plan with traditional approach, executes the plan on samples and revises estimated cardinalities. It then obtains a new plan with revised estimates and then repeat the process until the plan doesn't change anymore. [6] takes advantage of available index structure and can obtain all estimates before executing the plan. It is independent of runtime and can be injected in any query optimizer easily.

1.3 Aggregation and Group By

Aggregation is used to calculate summary of attributes from multiple rows on a number of groups, including max, min, sum, avg and count. Usually aggregation and group by are performed after all selections and joins. This is simple to implement but not always optimal. Doing aggregation and group by before joins are introduced in [2] [12]. It may largely increase query performance when used properly.

1.4 Outline

The main contributions of the thesis are listed as following:

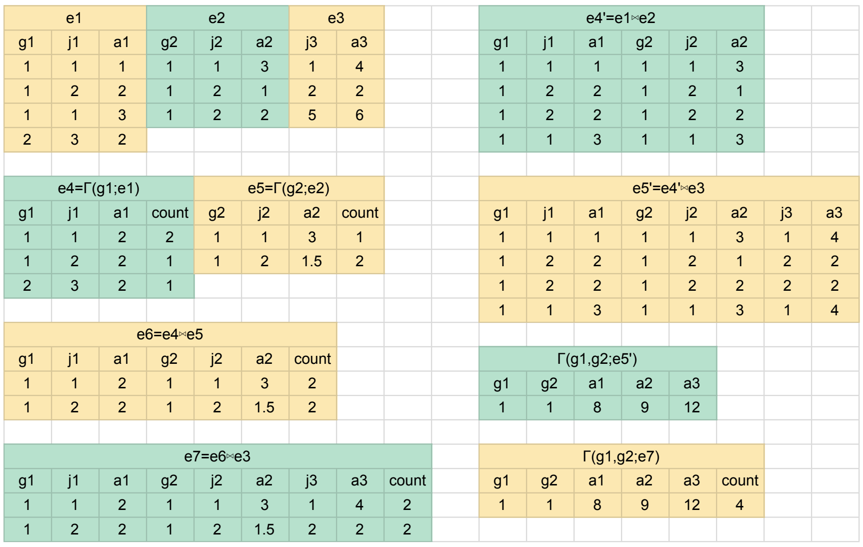
- A new estimation method for group by and join combining traditional approach and index-based join sampling is proposed and evaluated. Both estimates for group by and join are better than any technique alone.
- Two enumeration algorithms with eager aggregation are implemented and evaluated ,namely complete enumeration and heuristic with real data set and real queries. We find that three conditions need be be met to enable eager aggregation dramatically accelerate queries. And even when the conditions are incorrect, it is not dangerous to enable eager aggregation, which means it won't largely decrease query performance.

The rest of the thesis is organized as follows: section 2 first explains what is eager aggregation and how is done with examples. Then estimation method for group by and join is introduced. Enumeration algorithms for eager aggregation conclude the section. Section 3 evaluates different estimation techniques for group by and join as well as effects and overhead of enumeration algorithms. Finally, queries that are suitable for eager aggregation are described, which can give you a guidance when and when not to use the technique.

2 Eager Aggregation

2.1 Introduction to Eager Aggregation

Almost all RDBMS perform Aggregation in the final step of a query after all joins. As we know, only Sybase applies eager aggregation, which is really a pity as it can dramatically increase performance when used properly. To use eager aggregation, join and aggregation operator need to be changed slightly to produce correct results since some attributes are eliminated in aggregation. Count for each row needs to be maintained in both aggregation and join operators. In this work, we only consider inner join, because we aim to explore the performance gain and loss with eager aggregation but not enumeration algorithms which are explained clearly in [4].



e1			e2			e3	
g1	j1	a1	g2	j2	a2	j3	a3
1	1	1	1	1	3	1	4
1	2	2	1	2	1	2	2
1	1	3	1	2	2	5	6
2	3	2					

e4'=e1⋈e2					
g1	j1	a1	g2	j2	a2
1	1	1	1	1	3
1	2	2	1	2	1
1	2	2	1	2	2
1	1	3	1	1	3

e4=Γ(g1;e1)				e5=Γ(g2;e2)			
g1	j1	a1	count	g2	j2	a2	count
1	1	2	2	1	1	3	1
1	2	2	1	1	2	1.5	2
2	3	2	1				

e5'=e4'⋈e3							
g1	j1	a1	g2	j2	a2	j3	a3
1	1	1	1	1	3	1	4
1	2	2	1	2	1	2	2
1	2	2	1	2	2	2	2
1	1	3	1	1	3	1	4

e6=e4⋈e5						
g1	j1	a1	g2	j2	a2	count
1	1	2	1	1	3	2
1	2	2	1	2	1.5	2

Γ(g1,g2;e5')				
g1	g2	a1	a2	a3
1	1	8	9	12

e7=e6⋈e3								
g1	j1	a1	g2	j2	a2	j3	a3	count
1	1	2	1	1	3	1	4	2
1	2	2	1	2	1.5	2	2	2

Γ(g1,g2;e7)					
g1	g2	a1	a2	a3	count
1	1	8	9	12	4

Figure 2.1: An example with e1, e2 and e3

Usually aggregation is done grouping by attributes which are specified in group by clause of SQL language. It will eliminate all attributes except those that are in the group by clause. Therefore, in eager aggregation, in order that tables still have attributes to join with other tables after eager aggregation, join attributes are also needed to be included in the group by attributes except for those attributes which are already in group by clause in SQL language. For example, there are two tables s with attributes (g_1, j_1) and t with attribute j_2 and join condition is $j_1 = j_2$. If we want to do an eager aggregation on s before joining s and t, we need to group by

(g_1, j_1) instead of (g_1).

Aggregation consists of count, sum, avg, min and max. An example in figure 2.1 is shown on how to get correct results for count, sum and avg. Count and sum is shown in the figure, then avg can be calculated as sum/count. When doing aggregation on intermediate results, including base tables, the values for attributes which are required to be aggregated need to be divided by count. That is how 1.5 in e_5 comes from. Then in the final aggregation, all values will be multiplied by their corresponding counts. Min and max are much simpler, we can simply do normal join on intermediate results obtained with eager aggregation and don't need to maintain count information but only need to memorize the minimum or maximum value.

2.2 Construction of Eager Aggregation Operator

When construing eager aggregation operators, a local group by clause should be constructed, at least including join attributes and attributes that are in group by clause specified in SQL query. It is also possible to include some other attributes, but this will increase the number of groups, making eager aggregation less useful. Although sometimes it can increase chances to do eager aggregation, we argue that it is not worth doing it. Therefore, we simply do eager aggregation grouping by attributes included in join expressions and group by clause.

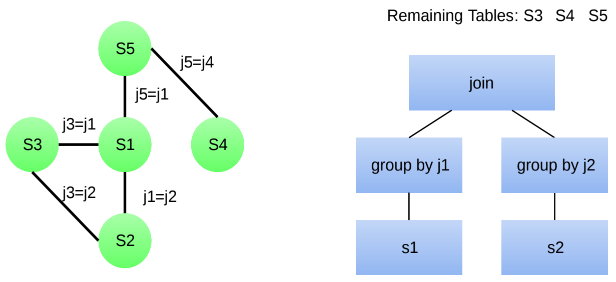


Figure 2.2: Join graph and intermediate operator tree

It is obvious that the attributes in group by clause should be included in eager aggregation. The reason for including join attributes is that eager aggregation may eliminate some columns which will be needed later, which leads to an invalid eager aggregation. Therefore, when constructing eager aggregation operators, we need to check whether it is valid or not. Figure 2.2 shows the query graph of the query below which doesn't have an explicit group by and will produce one group. But we can do eager aggregation grouping by attributes involved in joins, which are j1, j2, j3, j4 and j5 in the figure. The intermediate operator tree contains s1 and s2. s3, s4 and s5 are still needed to be joined. In order to decide whether we can do eager aggregation when building join tree consists of s1 and s2, we need to check for each remaining table whether they have join condition with the potentially eliminated attributes from s1 and s2. We can see from the query graph that s3 and s5 are both joined with j1 or j2, and s4 doesn't have a join condition with s1 or s2. Therefore, it is safe to do eager aggregation on j1 or j2. If s1 or s2 has other attributes which will be used to join later, then eager aggregation on top of that table cannot be done.

```
SELECT sum(s1.a1) FROM s1, s2, s3, s4, s5 WHERE s1.j1=s2.j2 AND
s2.j2=s3.j3 AND s1.j1=s3.j3 AND s1.j1=s5.j5 AND s4.j4=s5.j5;
```

2.3 Considerations in Eager Aggregation

Avoid Unnecessary Grouping

It is possible to have unnecessary grouping operators when we group by a key. But the good news is that we can know the key for base tables from schema and compute the keys for all intermediate results. There can be multiple keys, let *K(e)* be a set of keys which are sets containing one or some attributes. Keys for base tables are known and we can compute the keys for inner joins in the following step:

- In case j_1 is a key of e_1 and j_2 is a key of e_2

  $$K(e_1 \bowtie_{j_1=j_2} e_2) = K(e_1) \cup K(e_1)$$

  Each key of e_1 and e_2 is also a key in the join result

- In case j_1 is a key and j_2 is not

  $$K(e_1 \bowtie_{j_1=j_2} e_2) = K(e_1)$$

  Each key of e_2 is also a key in the join result

- In case j_1 and j_2 are both not keys

  $$K(e_1 \bowtie_{j_1=j_2} e_2) = \bigcup_{k_1 \in K(e_1), k_2 \in K(e_2)} k_1 \cup k_2$$

  Each pair of keys from e_1 and e_2 is also a key in the join result

In JOB queries, we don't have the first case. When doing eager aggregation, our local group by attributes consist of join attributes and overall group by attributes which will definitely not keys. The only possible situation of grouping by keys is on base table, which will definitely increase the cost. Therefore, our query optimizers will never consider grouping by keys in eager aggregation. But to reduce the number of produced plans, we will check whether it is a key in base table.

2.4 Cardinality Estimation for Group By

To enable query optimizer consider eager aggregation, we need a way to estimate cardinality of group by and a function to calculate the cost of it. In our implementation, cardinality multiplied by an adjustable parameter is used to calculate the cost. To estimate cardinality of gorup by, we need some statistics which are number of tuples(*ntuples*) for each table involved in group by, number of selected rows(*nrows*) for those tables, and number of distinct values(*n_distinct*) for each relevant attribute. *nrows* is equal to *ntuples* if there is no selection on that table. Then the cardinality estimation for group by is calculated in the following way:

1. Eliminate the attribute with bigger n_distinct if two equivalence attributes from different tables exist. For example, group by (j_1, j_2) and $j_1 = j_2$ is a join condition.

2. For all attributes from a single table, multiply all *n_distinct*, and clamp it to $\frac{ntuples}{10}$. If there exists some attributes whose *n_distinct* are bigger than $\frac{ntuples}{10}$, then set the value to the largest *n_distinct*. Then multiply the value with selectivity which is calculated from formulas shown below:

3. Repeat step 2 for all tables, multiply all the values and then clamp it to the number of tuples before group by.

The selectivity mentioned above can be calculated in the three following ways according to PostgreSQL.

1. $\dfrac{n\_distinct * nrows}{ntuples}$
   The idea of this formula is that n_distinct should be proportional to selectivity of base table.

2. $n\_distinct * (1 - (\dfrac{n\_distinct - 1}{n\_distinct})^{nrows})$
   The probability that one particular value doesn't exist in nrows is $(\frac{n\_distinct-1}{n\_distinct})^{nrows})$, so the probability that it exists is 1 minus this value. Then the expected number of groups is *n_distinct* times it.

3. $n\_distinct * (1 - (\dfrac{ntuples - nrows}{n_distinct})^{\dfrac{ntuples}{ndistinct}})$

2.5 Cardinality Estimation for Join and Group By

Since cardinality for group by must be smaller than its input which is mostly a join, estimates for group by rely on estimates for join. Traditional approach tend to underestimate cardinality of joins. In this case, estimates for group by can be bounded by estimates for joins and then also be underestimated. This is a bad news for query optimizer, because it will consider this eager aggregation a waste since it cannot reduce intermediate results but only increase the cost of plans.

Therefore, good estimates for joins are badly needed. A sampling approach with little overhead based on available index structure is proposed in [6]. We will consider both traditional approach and this novel sampling approach in this work.

When using traditional approach, estimates for group by are obtained as described in section 2.4. estimates for joins are calculated through the formula:

$$|T_1 \bowtie_{x=y} T_2| = \frac{|T_1||T_2|}{max(dom(x), dom(y))}$$

Where T_1 and T_2 are inputs which can be table scans, selections or joins. $dom(x)$ is the number of distinct values of attribute x.

When using the index based join sampling, estimates for joins without considering eager aggregation are obtained before enumeration of plans with eager aggregation. There are at most 2^n(n is the number of tables) intermediate results considering only joins. We can also use sampling approach to estimate group by, but this will lead to too many intermediate results. Therefore, A combination of traditional approach and index based join sampling is used to avoid sampling group bys.

As is shown in figure 2.3, estimates in red are obtained with index-based join sampling, while estimates in black are calculated in enumeration phase and will be inputs for cost function. When estimating the join, estimate for group by which is 20 is compared with original estimate 1000. eager aggregation makes right table 50 times smaller, the estimate for join should also be 50 times smaller. Then, $\frac{3000}{3000} * \frac{1000}{20} * 8000 = 1600$ will be the final estimate for the join.

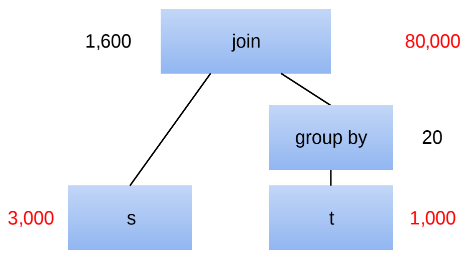


Figure 2.3: An example for estimating group by and join

2.6 Cost Function

The cost function is similar to the one used in [5] with a few differences. We don't have index nested loop join in query engine and we have aggregation operator. Cost for aggregation is also very simple, multiplying its cardinality with a parameter μ which is for balancing the cost with table scan and join. We use the cardinality of itself instead of its input for cache and rehashing issues. When the number of groups is small, even a big input doesn't cost too much. Of course, a more sophisticated model should consider both. τ is another parameter for balancing. Specifically, we set μ = 0.6 and τ = 0.2 for all experiments if not specified explicitly.

$$C_{mm}(T) = \begin{cases} \tau \cdot |R| & if \quad T = R \vee T = \sigma(R) \\ |T| + C_{mm}(T_1) + C_{mm}(T_2) & if \quad T = T_1 \bowtie T_2 \\ \mu \cdot |T| + C_{mm}(T_c) & if \quad T = \Gamma(T_c) \end{cases}$$

2.7 Implementation for Aggregation and Join

Section 2.1 shows how aggregation and join are done to enable eager aggregation by maintaining count information. We implement a standard hash based aggregation algorithm with unordered_map in C++ standard library which uses linked list to solve collisions. When the load factor is going to surpass its max_load_factor, it performs a rehash. After aggregation, count for each group is stored.

Hashjoin is implemented with unordered_map which has a vector as value, making it more efficient than unordered_multimap for locality reason. To maintain count information, each match needs to multiply count from two sides and make it the count for the new tuple. If aggregation is not done before join, there is no count column, in which case count is 1 by default.

2.8 Enumeration Algorithm for Eager Aggregation

The enumeration algorithm is built on top of DPccp [8] which constructs a query graph and enumerates connected pairs and described in detail in [4]. CPccp simply joins the pair, while the new enumeration algorithm will check both left and right side whether eager aggregation can be done on any side. At most four join trees shown in figure 2.4 will be produced instead of only 1 in DPccp.

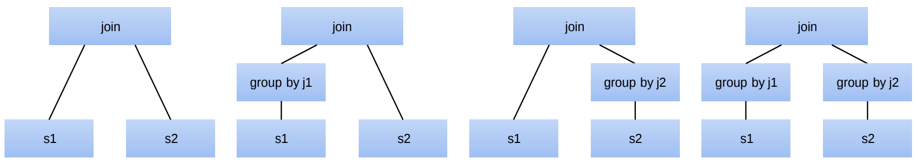


Figure 2.4: 4 operator trees

Algorithm 1 describes the framework of the algorithm. Line *1* and *2* initializes the table, setting the best plan consists of only one table to the table itself. Line 3 enumerates all connected pairs S_1 and S_2 of the query graph. It guarantees that when S_1 and S_2 are enumerated, their plans have already been known. There might be multiple versions of BUILDPLANS with different performance and quality which will be described shortly.

Algorithm 1 General framework

```
Input: a set of relations $R = R_0, R_1, ..., R_{n-1}$
       a query graph H
Output: An optimal bushy operator tree
1 for all $R_i$ ∈ R
2     DPTable[$R_i$] = $R_i$
3 for all csg-cmp-pairs($S_1$, $S_2$)
4     BUILDPLANS($S_1$, $S_2$)
5 return DPTable[R]
```

Complete Enumeration

Without eager aggregation, only the best plan for each intermediate result needs to be maintained since it meets principle of optimality. Eager aggregation will increase the cost but decrease the cardinality of a join tree compared to the join tree without eager aggregation. The reduced cardinality may compensate the cost of bigger join trees. Therefore, We cannot determine which tree will be the best when it is enumerated. In order to have the optimal plan in the end, all four trees need to be stored. Every time construing an intermediate result, at most 4 trees need to be stored, making the number of intermediate results increase exponentially. This approach is not realistic when the number of tables in a query is big.

In algorithm 2, line 1 and 2 iterate all plans of S_1 and S_2. Line 3 builds at most 4 operator trees as is shown in figure 2.4.

Algorithm 2 BUILDPLANS(S_1, S_2) with Complete Enumeration

```
Input: a set of relations $S_1 and S_2$
1 for each $T_1$ ∈ DPTable[S1]
2     for each $T_2$ ∈ DPTable[S2]
3         for each $T$ ∈ optrees($T_1$, $T_2$)
4             DPTable[$S_1 \cup S_2$].APPEND(T)
```

Heuristic

When the number of tables in a query is too big, to reduce optimization overhead, a heuristic which will only keep the local optimal operator tree can be used. This may throw some potentially useful plans with eager aggregation. But the produced optimal plan will definitely be better than or same as not considering eager aggregation at all.

In algorithm 3, line 2 compares the new operator tree with the current best operator tree and keep it if it is better.

Algorithm 3 BUILDPLANS(S_1, S_2) with Heuristic

```
input: a set of relations $S_1 and S_2$
1 for each $T \in$ optrees($T_1$, $T_2$)
2     if $COST(T) < COST(DPTable(S_1 \cup S_2))$
3         DPTable[$S_1 \cup S_2$] = T
```

3 Evaluation

3.1 Experimental Setup

All experiments run on a server with two X5570 2.93GHz 4 core CPUs and 48 GB RAM. The whole IMDB dataset and statistics such as domains are mapped into memory before queries are executed.

3.2 IMDB data set and JOB queries

Internet Movie Data Base(IMDB) contains information about movies and related facts such as actors and production companies. 33 query structures with different selectivity from very low to high on base tables are constructed, which is called Join Order Benchmark(JOB). Queries with same structure but different selectivity will have different optimal plans and runtimes. There are 21 tables in IMDB and 113 queries in JOB queries. One example is shown below:

```
SELECT sum(t.production_year) AS movie_year FROM
company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t WHERE
ct.kind = 'production companies' AND
it.info = 'top 250 rank' AND
mc.note not like '%(as Metro-Goldwyn-Mayer Pictures)%' AND
(mc.note like '%(co-production)%' or mc.note like
'%(presents)%') AND join conditions;
```

In addition, another 33 queries with same structure but no selectivity are constructed, which means all tuples in base tables are selected. These queries will be extremely slow without eager aggregation when the number of joins is big, because no selectivity predicate will lead to quite large results. Such query is like:

```
SELECT sum(t.production_year) AS movie_year FROM
company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t Where
join conditions;
```

3.3 Loss and Gain of Eager Aggregation

In order to get an idea of how eager aggregation works in different settings, before running JOB queries which consist of many complex joins, four simple joins are executed and compared in non eager aggregation, partial aggregation and double eager aggregation case. Experiments show that the technique can not only dramatically increase the performance of query, but can also make the query slower.

We use table scan of following tables shown in figure 3.1 directly as input of join operators. Hash join is applied, small table is used to build hash table and big table is used to probe the hash table. Domains for *movie_id* are very large, while very small for *info_type_id*. As shown in previous section that the aggregation is hash based, it can be very efficient when domains are small, since the whole hash table can be in the CPU cache. Another benefit of small domains is that there will be only a few tuples left after aggregation, making the join much cheaper. To the contrary, large domains will not only make aggregation expensive, but also have no positive effect on join performance. An extreme case is when the domain is equal to number of tuples, which makes eager aggregation totally a waste.

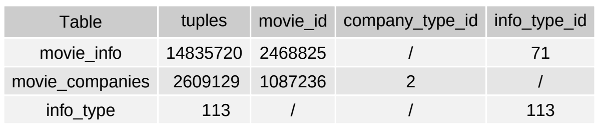


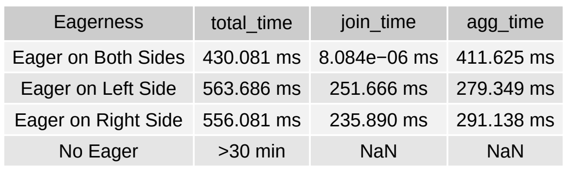
Table	tuples	movie_id	company_type_id	info_type_id
movie_info	14835720	2468825	/	71
movie_companies	2609129	1087236	2	/
info_type	113	/	/	113

Figure 3.1: Tables with statistics

Both are Big Tables with Small Domains

Big table with small domains will degrade hash join to be like a nested loop join. Each tuple in the probe table will have to iterate a large number of build keys. In our experiment, we are not able to get a result within 30 minutes. With eager aggregation, the join becomes quite efficient since there are only 2 tuples left after aggregation. Eager aggregation on both sides will lead to a join between 2 tables with 2 tuples. Eager aggregation on one side will make a join similar to a table scan. More importantly, the eager aggregation itself is also quite efficient in this case, because it is very cache friendly when domain is small.

```
SELECT sum(mc1.company_type_id) FROM movie_companies AS mc1,
movie_companies AS mc2 WHERE mc1.company_type_id= mc2.company_
type_id;
```



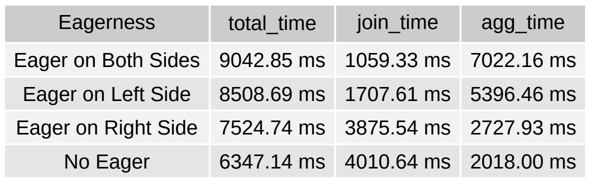
Eagerness	total_time	join_time	agg_time
Eager on Both Sides	430.081 ms	8.084e−06 ms	411.625 ms
Eager on Left Side	563.686 ms	251.666 ms	279.349 ms
Eager on Right Side	556.081 ms	235.890 ms	291.138 ms
No Eager	>30 min	NaN	NaN

Figure 3.2: Both are Big Tables with Small Domains

Both are Big Tables with Large Domains

Large domains in a big table will make aggregation slow since the eager aggregation is cache unfriendly in this case. Also it won't dramatically increase the performance of later joins since the number of tuples is not reduced much. But the good news is that even eager aggregation on both sides is slightly slower than no eager aggregation version, which means it is not too dangerous to have the plan with eager aggregation even in the situation which is not suitable for doing eager aggregation.

```
SELECT sum(mc.movie_id) FROM movie_info AS mi, movie_companies
AS mc WHERE mi.movie_id = mc.movie_id;
```



Eagerness	total_time	join_time	agg_time
Eager on Both Sides	9042.85 ms	1059.33 ms	7022.16 ms
Eager on Left Side	8508.69 ms	1707.61 ms	5396.46 ms
Eager on Right Side	7524.74 ms	3875.54 ms	2727.93 ms
No Eager	6347.14 ms	4010.64 ms	2018.00 ms

Figure 3.3: Both are Big Tables with Large Domains

One Big Table with Small Domains and One Small Table

The performance is not very different in this case with or without eager aggregation. Join can be done in the way that the small table as build table and big table as probe table, which is not expensive. Also eager aggregation on big table is efficient since it is cache friendly. But eager aggregation version is still faster than non eager aggregation one, which is positive.

```
SELECT sum(mi.info_type_id) FROM movie_info AS mi, info_type
AS it where mi.info_type_id = it.id;
```

Eagerness	total_time	join_time	agg_time
Eager on Both Sides	1460.56 ms	2.5773e−05 ms	1299.81 ms
Eager on Left Side	1431.08 ms	3.2437e−05 ms	1268.92 ms
Eager on Right Side	2545.81 ms	1893.65 ms	479.94 ms
No Eager	2231.64 ms	1523.91 ms	441.30 ms

Figure 3.4: One Big Table with Small Domains and One Small Table

One Big Table with Large Domains and One Small Table

This case is a nightmare for eager aggregation. On the one hand, eager aggregation on big table with big domain is quite expensive, on the other hand, although there is a big table in the join, small table can be used as probe table, making the join still cheap. The experiment shows that with eager aggregation, performance can be almost 20 times slower. Fortunately, this plan will not be selected by query optimizer if the estimation for group by is not too bad.

```
SELECT sum(mi.movie_id) FROM movie_info AS mi, info_type
AS it WHERE mi.movie_id = it.id;
```

Eagerness	total_time	join_time	agg_time
Eager on Both Sides	7958.55 ms	184.33 ms	7215.21 ms
Eager on Left Side	7127.21 ms	421.23 ms	6223.71 ms
Eager on Right Side	371.27 ms	226.25 ms	1.1017e−04 ms
No Eager	424.59 min	226.95 ms	3.1807e−05 ms

Figure 3.5: One Big Table with Large Domains and One Small Table

3.4 Estimation Accuracy for Group By

Estimates with Traditional Approach

We store all plans with an aggregation operator on the top which are produced during enumeration and execute those plans to obtain real cardinalities.

The method described in section 2.4 is used to estimate cardinalities for group by. All three formulas are applied and analyzed and the results are shown in figure 3.6 and 3.7. In general, estimates for group by tend to be underestimated. But estimates for group by with one attribute and two attributes are quite different. Estimates for one attribute is stable and similar when the number of tables is different, while quite different for two attributes. Estimates for two attributes are quite similar to estimates for joins shown in the fourth figure of figure 3.7. It is not surprising after knowing how it is estimated. Group by multiple attributes need to multiply several values, making the estimate very big. But the estimate must be clamped to its input which is the estimate for joins. When the number of tables is 1, there is no join and therefore it is different from remaining estimates.

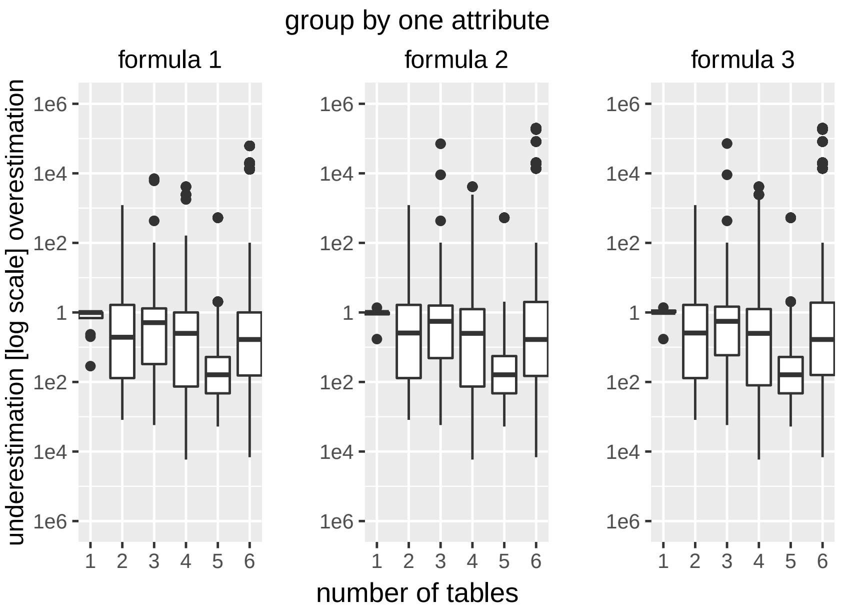


Figure 3.6: Quality of cardinality estimates for group by with one attribute in comparison with the true cardinalities

The estimate is quite close to real cardinality only when grouping by one attribute on a single table with formula 2 and 3. Formula 1 can be orders of magnitude different, because it simply multiples domain of attribute by selectivity. This can be accurate when the attribute is a key, but can be quite inaccurate when it is in a big table with a small domain. Grouping by two attributes when the number of tables is two is also good, which is a little surprising when compared with grouping by one attribute. The reason is that most estimates are bounded by estimates for joins and there are many join crossing correlations when grouping by one attribute, but not many when grouping by two attributes. This result may be coincidence and not be robust since there are only 93 and 31 results for one and two attributes, respectively. The distribution of results is shown in figure 3.8. Figure 3.8a shows the number of results for different number of tables and 3.21b shows the percentage of results that are bounded or unbounded by joins with different number of tables. We can see that except when the number of tables is 1, most results are bounded by joins. Therefore, it is possible to improve the estimates for group by by having a better technique for estimating joins.

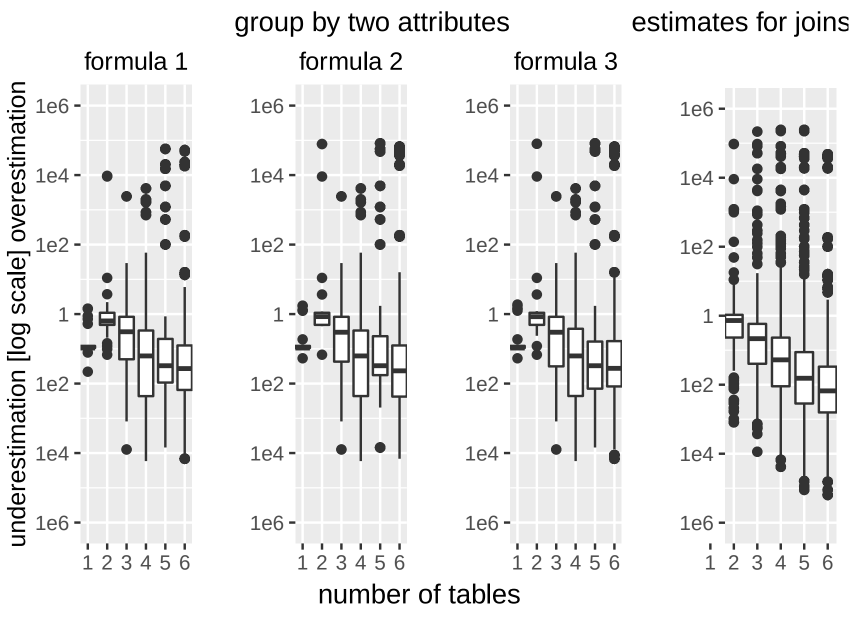
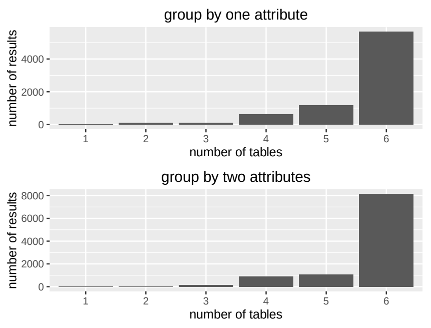
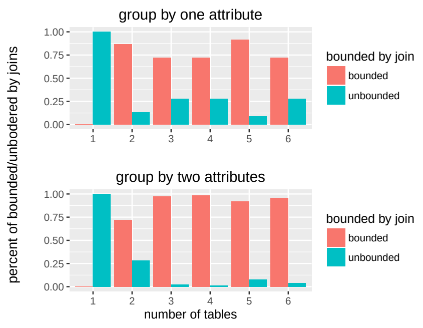


Figure 3.7: Quality of cardinality estimates for group by with two attribute in comparison with the true cardinalities



(a) Estimates with different number of attributes



(b) Estimates related to estimates for joins

Figure 3.8: Number of estimates for group by

Estimates with Real Cardinalities for Joins

To see whether this assumption is correct, we try to estimate cardinalities for group by with true cardinalities for joins, unlike the cardinalities for joins used above which are also estimated. We can see that the estimates are good with a few number of joins but become worse with the increase number of joins in figure 3.9. This is caused by the wrong assumption we made. The estimation approach assumes that the number of groups will not be changed during joins. But actually join crossing correlations may keep all tuples in one group and rule out all other groups

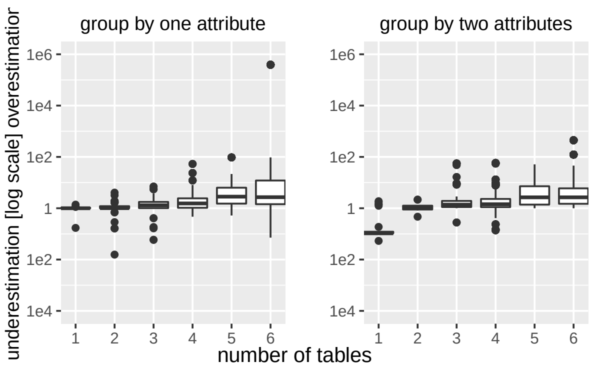


Figure 3.9: Quality of estimates for group by with real estimates for joins in comparison with true cardinalities

Estimates with Index-Based Join Sampling

Index-based join sampling was proposed in [6]. Figure 3.10 reproduces the result to give you an idea about the quality of estimates for joins. Two things different from the original work is that we use first 80 queries instead of all 113 queries and set the sample size to be 100 instead of 1000. Estimates with 100 and 1000 are quite similar when the number of joins is fewer than 10 and this is mostly true in the first 80 queries. We will see what the quality of estimates is like after combining the index-based join sampling with estimates for group by. The overhead of the new approach is still same as index-based join sampling.

Estimates for Joins

Figure 3.10 shows estimates for joins produced by index-based join sampling approach with different budgets. More budget there is, better estimates and also more overhead there is. These estimates are for joins which doesn't consider eager aggregation and will be injected in query optimizer to calculate estimates for group by and new estimates for joins with eager aggregation. 100k will be used by default for all experiments if not specified.

Estimates with Unlimited Budget

We can see from figure 3.11 that the quality of estimates for group by with index-based join sampling is quite similar to estimates with real cardinalities for joins, overestimating a little bit. But the bad news is that the quality of estimates for joins become worse, especially when grouping by two attributes. Estimates for joins are

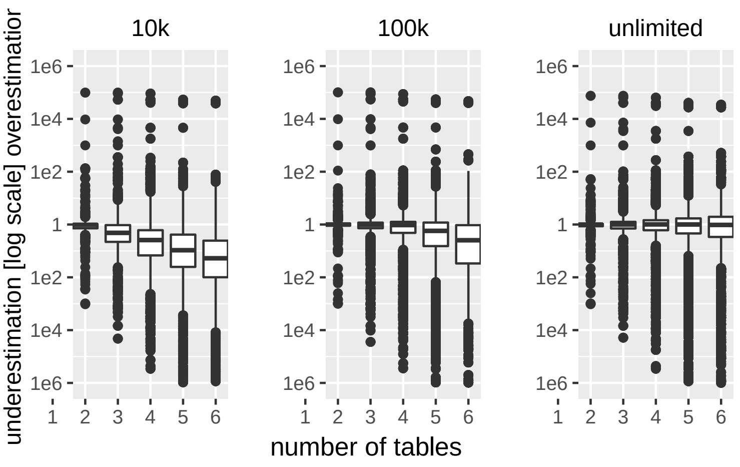
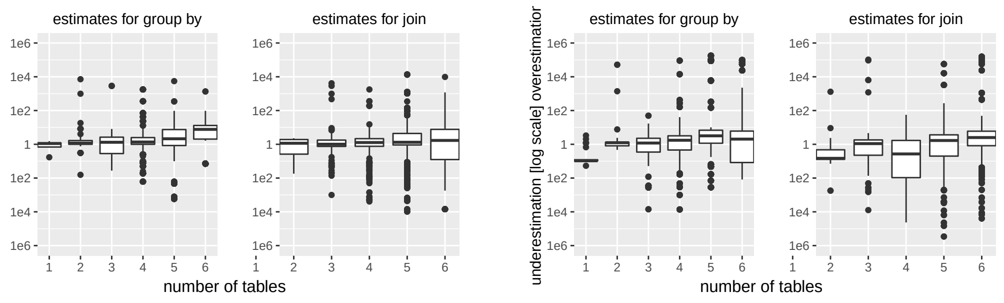


Figure 3.10: Quality of Cardinality Estimates for Joins by index-based join sampling in Comparison with True Cardinalities

overestimated instead of being underestimated as before. This is caused by the overestimation of group by.



(a) group by one attribute

(b) group by two attributes

Figure 3.11: Quality of Cardinality Estimates for Group By and Join with unlimited budget in Comparison with True Cardinalities

Estimates with Budget = 10k and Budget = 100k

Figure 3.12 shows a very interesting result. Estimates for group by with 100k is even better than when using real cardinalities for joins. Estimates for joins also becomes better compared with the estimates directly from index-based join sampling. This is because estimates for group by tend to be overestimated, while estimates for join tend to be underestimated. The combination compensate each other, making both of them better, which is ”two wrongs make a right”. There should be a budget between

10k and 100k making the estimates even better.

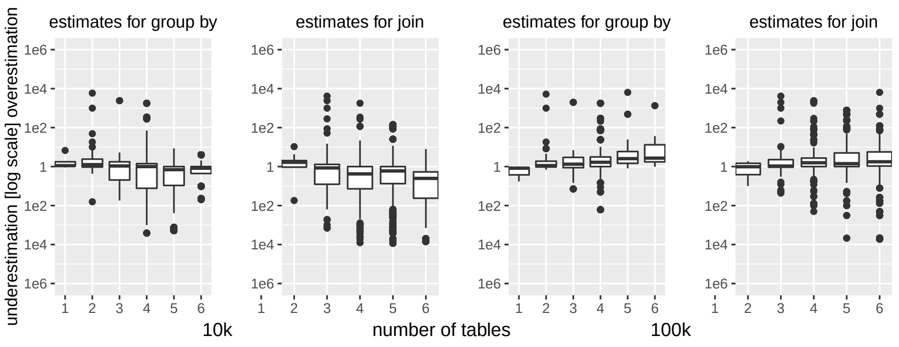


Figure 3.12: Quality of Cardinality Estimates with Different Budget in Comparison with True Cardinalities

3.5 Plan Quality with Traditional Estimated Cardinality

Plans with eager aggregation obtained by complete enumeration and heuristic are compared to plans without eager aggregation. They behave very different when there is selection or no selection on base tables.

Plan Generated with Complete Enumeration

Complete enumeration stores all intermediate results produced during optimization phase, making it very expensive. When the number of tables in a query is big, optimization cannot be finished in a reasonable time. Therefore, we cannot get results for all 113 JOB queries. Instead, first 80 queries are used.

Selection with Selectivity

Selectivity means some tuples in base tables will be selected, but not all tuples. It has queries ranging from very low selectivity to high selectivity.

As is shown in figure 3.15, few queries benefit from eager aggregation and some queries even get worse. But the good news is that the performance decrease is not too much. Eager aggregation can be considered to be safe.

Selection with no Selectivity

No selectivity means all tuples in base tables will be selected, making intermediates results too big to fit in main memory. Although operating system can handle this through virtual memory subsystem, the query becomes so slow that it cannot finish

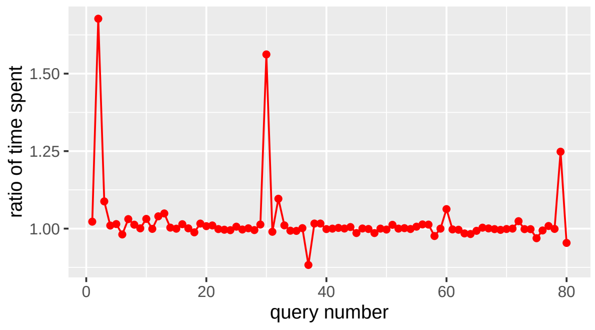


Figure 3.13: Performance gain with eager aggregation by complete enumeration for queries with selectivity

in a reasonable time. Therefore, we don't have results for all 33 queries but only a few of them.

Different from results obtained above, eager aggregation can greatly accelerate such queries. More importantly, with eager aggregation, queries that cannot finish in a reasonable time previously can also finish. This is because eager aggregation will largely reduce intermediate results, making joins faster and intermediate results fit in main memory. We can see that eager aggregation can not only accelerate queries but also save main memory resource.

We set different tgus, namely 0 and 0.6. Eager aggregation will always be done when tgu = 0 since eager aggregation has no cost. When tgu = 0.6, some eager aggregation cannot be recognized by the query optimizer and chance of improving performance is lost, which is shown in figure 3.14. This is caused by the inaccuracy of cardinality estimates. Estimates for group by are bounded by estimates for joins. When bounded, eager aggregation will not decrease cardinalities of intermediate results and is considered to be a waste.

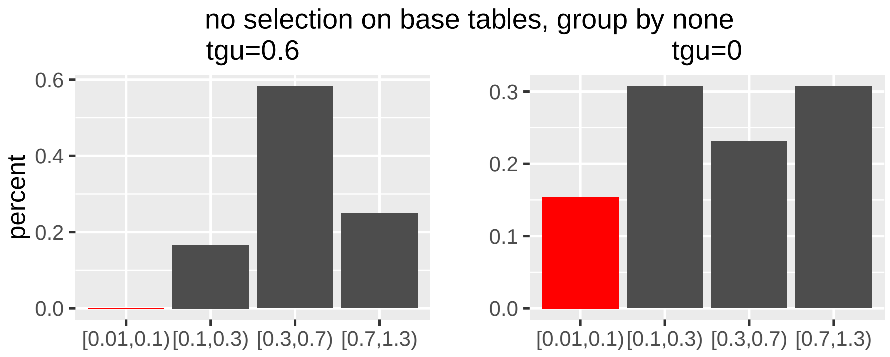


Figure 3.14: Slowdown of queries with eager aggregation by complete enumeration

3.6 Plan Quality with Index-Based Join Sampling Estimated Cardinality

Plan Generated with Complete Enumeration

Selection with Selectivity

Figure 3.15 shows that the performance is not that different with or without eager aggregation. The situation is same as when traditional cardinality estimation method is used. It is better not to use eager aggregation at all when selectivity on base tables is high.

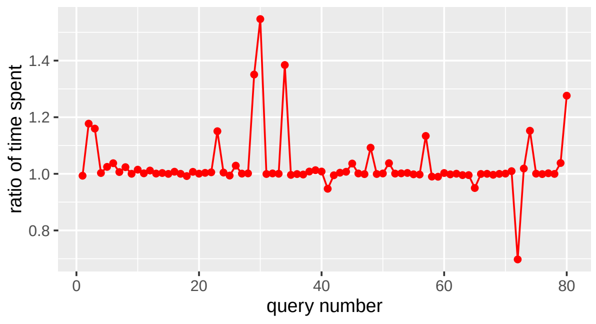


Figure 3.15: Performance gain with eager aggregation by complete enumeration for queries with selectivity

Selection with no Selectivity

There are 3 kinds of queries, grouping by no attributes, grouping by *it, kind_id* whose domain is 7 and grouping by *ci|mk|mc, movie_id* whose domain is 2528312. They will produce only one group, a small number of groups, a large number of groups, respectively. In Figure 3.16, the query is becoming more and more complex with the increase of query number. It shows that more complex the query is, better performance it gains with eager aggregation. And queries which produce fewer groups will gain more performance than queries that produce more groups. When the number of groups is big enough, eager aggregation may decrease performance.

Then different tgu is also tested to see whether the query optimizer misses some eager aggregation opportunities. Figure 3.19 shows a proper tgu is better than 0, which is different from using traditional estimation method that cannot recognize good plans with eager aggregation due to inaccuracy of cardinality estimates. The improved cardinality estimates indeed help query optimizer.

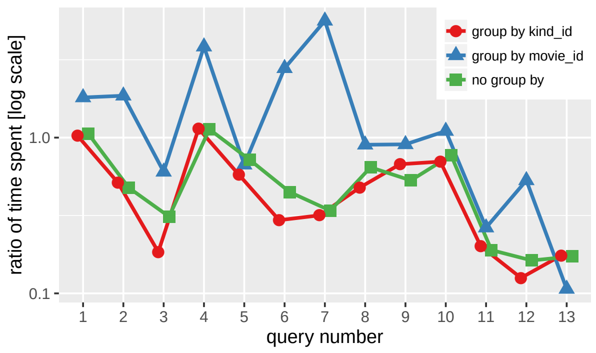


Figure 3.16: Performance gain with eager aggregation by complete enumeration for queries without selectivity

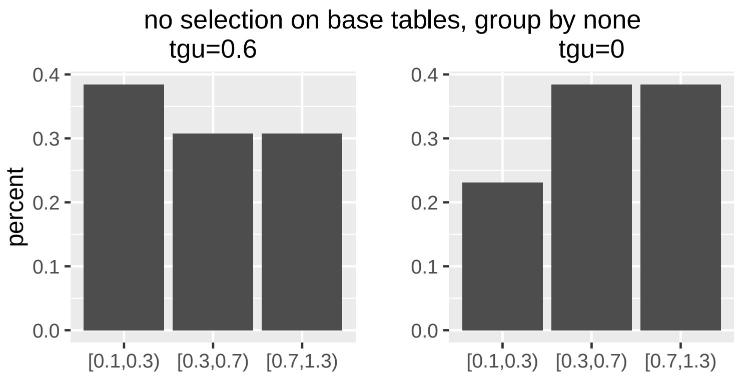


Figure 3.17: Slowdown of queries with eager aggregation by complete enumeration

Plan Generated with Heuristic

Selection with Selectivity

As is shown in figure 3.18, all queries have similar performance, which makes eager aggregation a waste. But the good thing is that no queries become slower because of eager aggregation.

Selection with no Selectivity

The result shown in figure 3.19 is quite similar to the result in figure 3.16. Although heuristic can improve less performance than complete enumeration, the overhead is also much less.

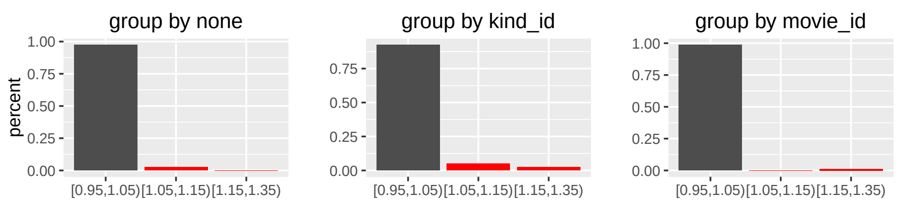


Figure 3.18: Slowdown of queries with selectivity with eager aggregation by heuristic

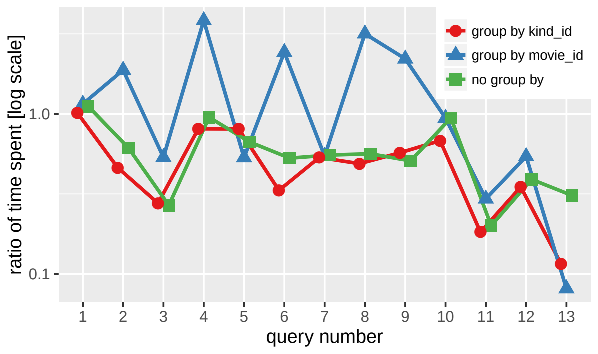


Figure 3.19: Performance gain with eager aggregation by heuristic for queries without selectivity

3.7 Estimation and Enumeration Overhead

Estimation overhead of index-based join sampling with sample size = 1000 is shown in [6].

Enumeration overhead varies greatly with different algorithms. As expected, the algorithm which doesn't consider eager aggregation has the lowest overhead, while complete enumeration has the highest overhead. When the number of tables in a query is big, the overhead can be more than 10 seconds. Heuristic has similar overhead with the algorithm which doesn't have eager aggregation. Therefore, Complete enumeration can be used when the number of tables is small and heuristic can be used as an alternative when the number of tables is big.

3.8 Queries that are Suitable for Eager Aggregation

The results have shown that JOB queries with selectivity don't benefit much from eager aggregation, while JOB queries with no selectivity benefit a lot. Following are the reasons that eager aggregation cannot increase performance for some queries:

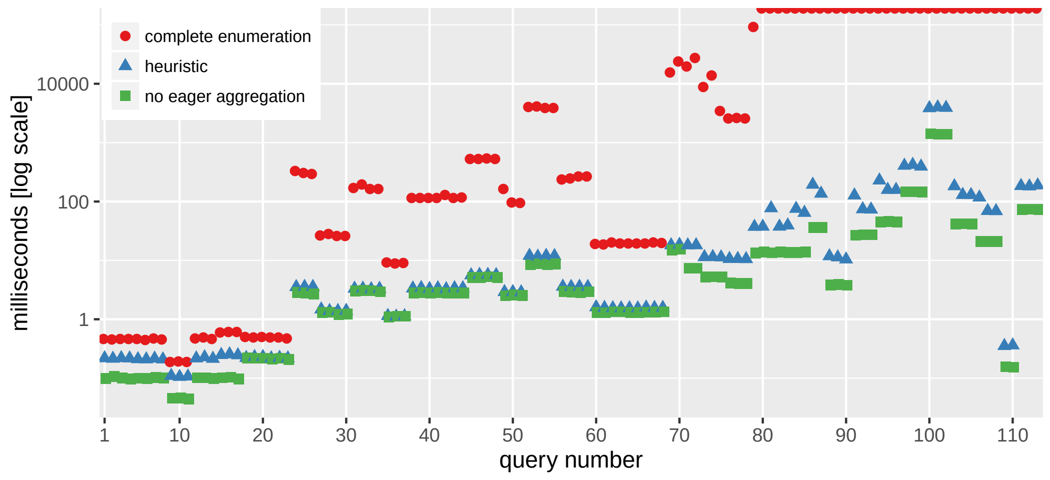


Figure 3.20: Overhead of different optimization algorithms

too many groups are produced

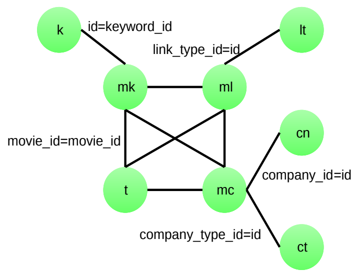
Grouping by *it*, *kind_id*, *ci|mk|mc*, *movie_id* or no attributes perform very different. When there are too many groups, eager aggregation won't largely reduce intermediate results, making it a waste. AS aggregation in this case is quite expensive, performance of eager aggregation can become much worse. But this won't happen with good estimates for group by and join.

Domain of Movie_id is Large

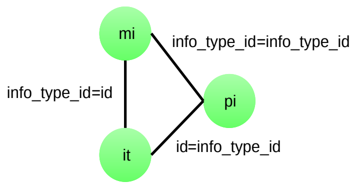
One reason is that domain of movie_id is very large. In JOB queries, many tables are joined between movie_ids. Doing eager aggregation grouping by by movie_id will not largely reduce the number of tuples. And doing eager aggregation grouping by other attributes with small domains not including movie_id will eliminate movie_ids, making it an invalid eager aggregation. In most queries, eager aggregation will be done almost at the top of an operator tree after joining almost all tables. In this case, operators below the eager aggregation cannot benefit from the reduced number of tuples and not many operators left above it. Then, only a few operators benefit from eager aggregation.

Figure 3.21a shows a typical example from JOB query. Tables *mk*, *ml*, *t* and *mc* are joined with *movie_id* and some of them also join other tables through foreign key and primary key. Some simple eager aggregation ideas are described below to help you understand. There are some possibilities to do eager aggregation at the beginning:

- on *t* by *movie_id*
- on *mk* by *movie_id*, *keyword_id*
- on *mc* by *movie_id*, *company_id*, *company_type_id*



(a) imdb_cluster3a_1_avg



(b) constructed

Figure 3.21: Query graph of a typical query and a constructed query

- on *ml* by *movie_id, link_type_id*

We can see that all the possibilities have *movie_id* whose domain is large in group by attributes. Grouping by such attributes with such a large domain will not greatly reduced intermediate results.

After joining all *mk, m, t* and *mc* first through *movie_id* without doing eager aggregation, we can then group by *keyword_id*, *link_type_id*, *company_id* and *company_type_id*. But grouping by so many attributes won't largely reduce intermediate results, either.

We can also do eager aggregation before the last join. For example, we can first join all tables except for *ct* and do eager aggregation on *company_type_id* which has 2 distinct values. Although this will largely reduce intermediate results and make the join with *ct* much faster, only one join can benefit from eager aggregation. But as we know from sction 3.3, it doesn't make much difference on performance since *ct* is a small table. The only case that we can benefit much is when *ct* is a big table with a small domain on join predicate. But we don't have this case in JOB queries since all joins are done between *movie_id* or foreign key and primary key whose domain is same as number of rows in the table.

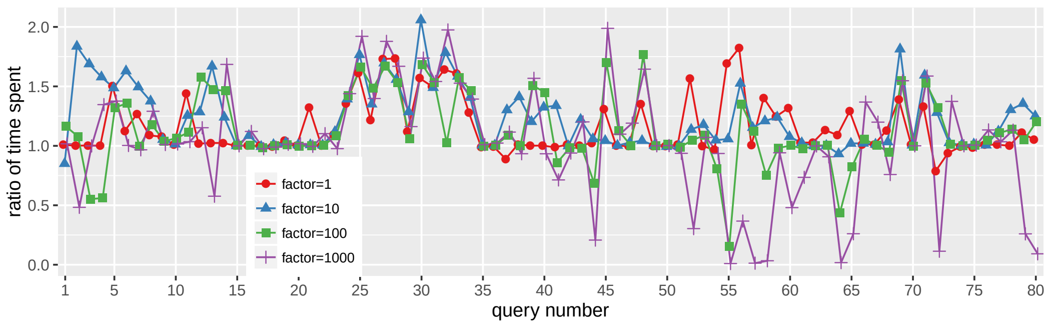


Figure 3.22: Performance gain with eager aggregation for queries with selectivity when domain of movie_id is different

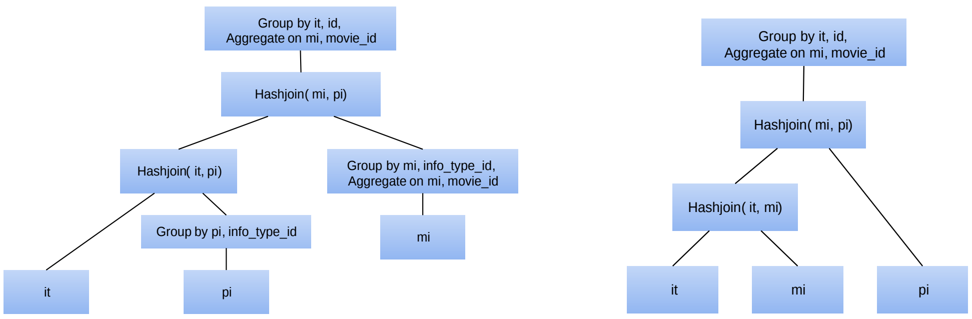
To explore how domain affects performance of eager aggregation, we reduce domain of *movie_id* by *10, 100, 1000*, respectively. *movie_id* is set to be *movie_id % (count /*

factor) + *1* where *count* is the initial domain of *movie_id* and factor is *10, 100, 1000*. This will make domain of *movie_id* smaller, aggregation cheaper and join more expensive. 80 queries are in the experiments and the bigger the query number is, more complex the query is. As is shown in figure 3.22, the reduced movie_id will accelerate complex queries, while doesn't affect simple queries that much. And performance gain for complex queries is more important than simple queries, as complex queries will take much longer time. For example, reduction from 10 mins to 1 mins is more important than reduction from 100ms to 10ms.

Selectivity on Base Tables is high

High selectivity on base tables will rule out most tuples, making later operators much cheaper. For example, in figure 3.21a, if all tables *k, lt, cn* have high selectivities, joins between *mk* and *k*, *ml* and *lt*, *mc* and *cn* will be cheap and will not produce too many tuples, making later operators also cheap. In this case, almost all joins are done between big table and small table or between small table and small table. As we know from section 3.3, eager aggregation will not work well in this case.

An example which meets the three requirements


Group by it, id,
Aggregate on mi, movie_id
Hashjoin(mi, pi)
Hashjoin(it, pi)
Group by mi, info_type_id,
Aggregate on mi, movie_id
Group by pi, info_type_id
mi
it
pi
Group by it, id,
Aggregate on mi, movie_id
Hashjoin(mi, pi)
Hashjoin(it, mi)
it
mi
pi

(a) query plan with eager aggregation

(b) query plan without eager aggregation

Figure 3.23: query plans for constructed query

To demonstrate the three requirements above, a simple query with 3 tables shown in figure 3.23b outside of JOB queries is constructed. This query meets both of the three requirements mentioned above, small domain and no selectivity on base tables. All *mi, mi_idx* and *pi* have *info_type_id* as foreign key that references *it*'s primary key *id* whose domain is 113. Both *mi* and *pi* are big tables with small domains in join attribute and *it* is a small table with primary key. Plans with eager aggregation and without eager aggregation are shown in figure 3.23.

Plan (a) takes 1769.8 ms while plan (b) cannot finish within 1 hour and throws std::bac_alloc exception which means there is no enough memory. We can see that

Eager aggregation can not only make query more efficient but also save resources when used properly. Even such a simple query that meets the three requirements above can gain much performance with eager aggregation.

But if there is a high selection on table *it*, the join between *it* and *pi* will be much cheaper. And they will also produce smaller intermediate results, making the join between *mi* and *mi* and *pi* cheaper again in which case eager aggregation is not useful at all.

4 Conclusion and Future Work

In this thesis, a new estimation method for group by and join combining traditional approach and index-based join sampling is proposed and evaluated. As group by tends to be overestimated and join tends to be underestimated, the new estimation method works even better than only estimating group by or join alone. Although 3 formulas for estimating group by looks quite different, they are similar in practice. Two enumeration algorithms which are complete enumeration with high overhead and heuristic with low overhead are implemented and evaluated on IMDB data set and JOB queries with estimates from the new estimation method. They work best when data set and queries meet three conditions: Tables are big while selectivity on base tables is low, join attributes have small domains and the query will produce a small number of groups. In this case, they can not only dramatically accelerate queries but also greatly save memory resources. When the number of tables in a query is big, the overhead of complete enumeration is so big that it is not realistic to be used in which case heuristic is a safe alternative.

Currently, most evaluations are performed on grouping by one attribute. Grouping by multiple attributes can be done as future work. But we think that it will not be much different as long as the number of produced groups is similar. As the effects of eager aggregation technique relies heavily on data set and queries, to answer whether it is really useful in practice, we need to evaluate it on real workloads from database or data warehousing customers.

Bibliography

[1] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[2] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB*, volume 94, pages 354–366, 1994.

[3] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[4] Marius Eich and Guido Moerkotte. Dynamic programming: The next step. *Technical report*, 2014.

[5] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[6] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling.

[7] Stefan Manegold et al. *Understanding, modeling, and improving main-memory database performance*. 2002.

[8] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*, pages 930–941. VLDB Endowment, 2006.

[9] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[10] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB JournalâThe International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[11] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. Sampling-based query re-optimization. *arXiv preprint arXiv:1601.05748*, 2016.

[12] Weipeng P Yan and P-A Larson. Performing group-by before join [query processing]. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 89–100. IEEE, 1994.