Dynamic query optimization for data integration systems

Maurin Gilles
ISIMA, Aubière, France

Maxime Buron
LIMOS, Aubière, France

March 6, 2025

Abstract

Query optimization in Database Management Systems relies on statistics about data distribution in the sources to estimate an optimal plan to execute a query. In data integration, such statistics are often not available, and alternative techniques should be considered. The most efficient solutions in this field tend to have ”dynamic” approaches, in which the missing statistics are computed during the query execution, in order to find a better query execution plan to switch to during runtime. In this work, we have an overview of the issues and solutions existing for dynamic query optimization in data integration systems. We focus more precisely on the join ordering problem, and how to estimate joins without initial knowledge on the sources. We propose a solution involving dynamic histograms, which provide decent estimates in this very constrained context.

Keywords: data integration, query optimization, dynamic histogram

---

Contents

1 Introduction

There is an ever-increasing amount of data available, which is used in many key applications from decision-making systems to training AI models. Two major goals arise that one would expect when working with such data. Firstly, the quality and the semantic of the data have to be guarantied, especially in scenarios where data can be heterogeneous. It is a key characteristic of the data to be used by others systems and understood by human users. Secondly, storage and access to data have to be efficient, to make data querying and manipulation quick and as economical as possible regarding resources. Database management systems (DBMS) have been developed since the 1960's and improved towards these goals, with important works on optimizing each step of the whole process. [GUW08a] However, they rely on the principle that the data is stored on a centralized system. This tends to become a limit, in the common situations where many data are managed by different organizations that are interested in sharing and linking these data together to bring new knowledge to light. [IH02]

Example 1: *A travel agency TA plans travels thanks to its partners, an airline AL and a hotel chain HC. TA stores information about its customers and their preferences in an internal DBMS. To plan a travel, this data has to be linked with information from AL and HC's own databases, and an external API that provides real-time exchange rates. In addition, TA, AL and HC might use DBMS with different models (relational, XML, RDF...)*

Data integration systems (DIS) aim to provide solutions for problems such as the one exposed in the previous example, by providing a uniform and semantic access to data from sources that evolve independently, on different formats, and whose access to might be unstable. To do so, they need to face a lack of preliminary information about the data.

These issues are specific to DIS: the representation of the data available through these systems and the way queries are evaluated on them require new techniques that aren't developed in traditional DBMS. The common approach in data integration to deal with the heterogeneity of the source data models is to use *mappings* from each of these models to a global and unique one, in order to process queries over this unique model. The mappings define the translation of the source schemata to one global schema used for the data available through the DIS. There are multiple approaches to manage efficiently the correspondences between the global and local (or source) schemata using views, depending on whether the global schema is expressed in terms of the local ones (Global as View), or the local schemata are expressed as functions of the global one (Local as View). These approaches, and the mixed one Global and Local as View, are thoroughly explained in [KP09]. Several mapping languages have been developed, referenced in the introduction of [Bur+20].

Orthogonally to the representation of the data in DIS, different query evaluation strategies have been developed. There are two main strategies: data materialization and data mediation. In the data materialization strategy, the data available through the mappings at the global schema level is first stored to a DBMS and this data is then queried using traditional techniques. The data mediation strategy aims at leaving the data in the source by rewriting the query on the global schema to a query plan on the different sources. The module of a DIS that evaluates this query plan by querying each source and combining the results to form the answers of the query on the global schema is called a *mediator.*

This paper assumes a mapping over a relational model and focuses on a crucial mission of the mediator which is query plan optimization. Query evalution in relational DBMS has been widely discussed, but always laying on hypotheses that cannot be applied in a DIS mediation environment. Our aim is to propose a solution to avoid these assumptions, and more particularly those that stipulate any preliminary knowledge of the data distribution in the sources.

2 Static versus Dynamic query optimization

2.1 Principle of query optimization

A *query* is a question from a user about the data managed by the DBMS or DIS. *Query processing* defines the establishment of a process to answer queries, and specifies the concrete steps of this process. The first step is to *parse* the query in a sequence of operations applied on the sources. The result is a *query plan*, which is equivalent to a logical expression in relational algebra in our case.

Example 2: *Information about dogs, cats and rabbits are stored in three different sources. For each pet, the information contains its name and an id for its master.*

User query: *"What dogs, cats and rabbits have the same master?"*

Query expressed in an uniform language (SQL here):

```
SELECT Dogs.name, Cats.name,
    Rabbit.name
FROM Dogs
    NATURAL JOIN Cats
        ON Dogs.master=Cats.master
    NATURAL JOIN Rabbits
        ON Dogs.master=Rabbits.master
```

Query plan in relational algebra:
$\pi_{Dogs.name,Cats.name,Rabbits.name}((Dogs \bowtie_{master} Cats) \bowtie_{master} Rabbits)$

The query plan defines the sequence of operators that will be applied to the data in order to produce an output answering the query. Figure 1 is an example of a common tree representation for such plans.

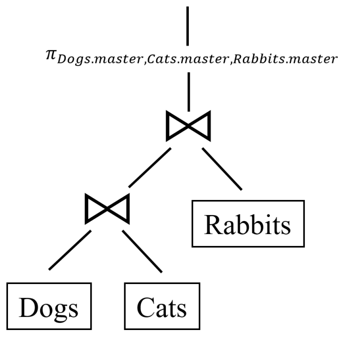

Figure 1: Query plan example

A query can have several equivalent plans, since a different sequence of operators could produce the same output. Two plans are called *equivalent* when they always produce similar outputs on identical sources. In our previous example, equivalent plans include those where where dogs are joined with rabbits first, and those where cats and rabbits are joined first.

So every query can be answered by a number of equivalent plans that grows very fast when queries become more complex. Even when considering only the problem of ordering joins for N sources, there are $N!$ ways to order the sources, and C_{N-1} ways to parenthesize these joins, C_n being the n^{th} Catalan number, which already gives $N!C_{N-1} = \frac{(2N)!}{(N+1)!}$ plans. Although these plans are algebraically similar, the difference between their computation times can be arbitrarily large. This phenomenon will be illustrated for the join ordering problem in the example of section 2.3. The goal of query optimization is to search for the best, or at least a good enough query plan to answer the query.

Defining what makes a plan better than another is a whole question, usually dealt with defining a *cost metric*, even if the idea of multi-objective query optimization considering multiple cost metrics simultaneously has been proposed for DBMS [TK17].

In the relational model, some optimization strategies are proven to always improve the query plan [RG02]. However, some issues don't have a fixed solution and require to be adapted for each query. Estimating the size of a join is a particularly important problem, presented in 2.2. Those issues are traditionally solved thanks to information about the data in the sources that makes an estimation of the cost of each plan possible. Making those estimations without such information is a major issue in DIS to which this paper proposes a partial solution.

2.2 Estimating the size of a join

Estimating the size of joins is a key problem in query optimization, since join operators are very common, can appear in large number in a single query, and their disposition can have a gigantic influence of the query plan efficiency. To illustrate this, some concepts must be introduced:

Relation: A source in a relational model.

Equi-join: $A \bowtie_x B$ is an equi-join when the condition for a tuple t_A from A to be joined with a tuple t_B from B is an equality of their values for the attribute x : $t_A(x) = t_B(x)$

Cardinality: $|A|$ corresponds to the number of tuples in the relation A. This definition can be extended, so $|A \bowtie_x B|$ corresponds to the number of tuples resulting from A joined to B over the attribute x.

Bounds for $|A \bowtie_x B|$: We have the following inequality:

$$0 \leq |A \bowtie_x B| \leq |A||B|, \quad \forall x$$

0 is reached when the set of values for the attribute x in the relation A is disjoint from the set of values for x in B. On the other hand, $|A||B|$ is reached when the set of

values for x in A and B are equals.

Selectivity: Another important metric for joins is their selectivity μ:

$$\mu_{AB}(x) = \frac{|A \bowtie_x B|}{|A||B|}$$

The bounds for the selectivity are:

$$0 \leq \mu_{AB}(x) \leq 1, \quad \forall x$$

Intuitively, the selectivity measures whether the cardinality of a join is close to its bounds, independently from the size of its sources. The point of selectivity is that it can be estimated more easily than the real cardinality of the join that would require information about the cardinality of the sources. Also, since the formula is identical for any attribute x, and we are only interested in equi-joins here, the x can be forgotten for a lighter notation.

Empirical selectivity: When computing a join iteratively from sources read randomly, one would expect to be able to estimate the cardinality at some point. We pose the empirical selectivity, which converges to the real selectivity when the proportion of sources read increases.

$$\tilde{\mu}_{AB}(x) = \frac{n(A \bowtie_x B)}{n(A)n(B)}$$

With $n(A)$ the number of tuples read from A so far, and $n(A \bowtie B)$ the number of tuples resulting from the join so far.

This naive estimate based on empirical selectivity is limited in practice: since the tuples resulting from $|A \bowtie B|$ are generally a small proportion of all the possible pairs from $A \times B$, a large amount of data must be read for the estimate to be trustworthy. Better solutions to estimate selectivity and thereby cardinalities have been proposed. The main families of algorithms are well presented in section 3 of [LBP21].

2.3 Estimating the size of multiple joins

Complex queries usually have more than two sources to join, and a fundamental question in query optimization is how to order these joins. The following example motivates the need to define an optimal order.

Example 3: *Consider three sources A,B and C, with cardinalities of 1000, 100 and 10, that we want to join them over a certain attribute. The values that A, B and C take for this attribute are distributed as follows:*

Value	Occurence		
	A	B	C
1	2	50	1
2	998	50	0
3	0	0	9

The result of the final join $A \bowtie B \bowtie C$ will be 100 triples of (1,1,1). Now let's compare the two plans from 2:

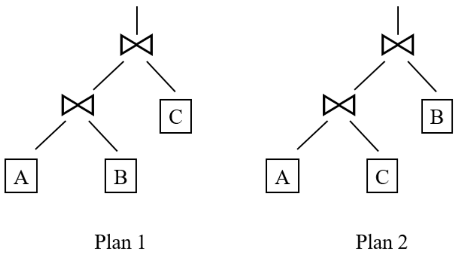

Figure 2: Example: two equivalent query plans

The selectivities and cardinalities of every join involved are summarized here:

Plan 1		
Join	(A,B)	((A,B),C)
Selectivity	0,5	0,0002
Cardinality	50.000	100
Plan 2		
Join	(A,C)	((A,C),B)
Selectivity	0,0002	0,5
Cardinality	2	100

We can use the number of tuples compared as a metric for the complexity of a plan. Let's assume that joins are computed with a nested loops algorithms, which has a complexity of $|A||B|$ *for an equi-join* $A \bowtie B$*. Then plan 1 has a complexity of* 600.000 *comparisons, against* 10.200 *for plan 2.*
This simple example gives an idea of the tremendous discrepancies that can exist between plans for queries with more joins and larger sources.

Let's call *tree* the representation of the order of the joins used in figure 2, *lower joins* those closer to the sources, i.e. those that will be computed first, and *upper joins* those closer to the "root", i.e. those that will be computed last.

Estimating selectivity for upper joins is not a problem. It can be done recursively, since

$$\mu_{(A \bowtie B),C} = \frac{|A \bowtie B \bowtie C|}{|A||B||C|\mu_{AB}}$$

However, this estimates for the selectivities of the joins in plan 1 from our previous example cannot be deconstructed to provide any estimate for selectivities in plan 2.

These heuristics remain extremely useful for static cost-based optimization, as estimates of join cardinalaties are necessary to compute the cost of a given plan.

2.4 Dynamic query optimization

Dynamic query optimization had initially been thought for DBMS handling data that evolves a lot. In those cases, estimations made during the query compilation might not concur with the real data distribution at execution time. This problem led to the idea of dynamic query optimization, as opposed to a static query optimization, in which a plan is chosen at compilation and executed uninterruptedly then. Two main approaches can be considered for dynamic query optimization:

- Computing several plans or sub-plans, that would be optimal for different values of some unknown parameters such as cardinalities or selectivities. [Gan98]
- Computing an only initial plan, and find alternatives during execution. [KD98; IHW04].

This second approach seems more appropriate in our case. Firstly, computing several plans during the compilation phase can lead to improve the initialization time, and thereby the time to get a first output. This is more of a problem in some common data integration cases when getting the first results quickly is equally or more important than getting the totality of the results quickly. For example, when the DIS is part of a WEB application, data and output are streamed anyway, so the best result will be the one where the stream starts the soonest. Secondly, the number of alternative plans is expected to grow exponentially with the number of unknown parameters, which we assume to be high in data integration. Finally, the second solution can take a more obvious advantage of pipelined queries, which will be discussed in section 5.

Our method for dynamic query optimization can be summarized as processing the query through these steps:

1. Compute an initial plan to start with
2. Collect new information during execution
3. At some point, find a better plan thanks to the new information
4. Switch to this new plan and continue execution

The main contribution of this paper is presented in section 3 and focuses on step 2 of the process above. Some perspectives for steps 1, 3 and 4 are presented in section 5.

In dynamic query optimization, one would like to be able to estimate selectivities and thereby costs of alternative plans thanks to the information gained while a first plan is already being executed. Therefore, counting the tuples read from the sources and resulting from the joins isn't enough, and more detailed information must be computed. This is the topic of the following section.

3 Dynamic histograms

3.1 Serial histograms to estimate joins

Histograms are widely used in DBMS to easily maintain a general representation of data distribution in a source. A histogram is a set of *buckets*, in such a way that every value possible taken by the data is associated to a single bucket. A histogram is *serial* when its buckets are contiguous, non-overlapping intervals.

Most of equi-joins are performed over integer attributes, for which serial histograms are a natural way to maintain an overall idea of a distribution with minimal information. For each bucket in the histogram, the only values needed are the lower-bound, the upper-bound, and the number of tuples in the source whose value for x is between these bounds.

Estimating the result of an equi-join of two sources A and B over an attribute x can be done in a linear amount of time if we know histograms for x in A and B. The incredible added value of histograms is that non only they can be used to estimate the selectivity of a join with any other source over the same attribute, but they also allow an estimate of the distribution of these joins. It is thereby possible to pipeline estimates, even though their quality are diminished when the number of joins increases, because of an unrealistic uniformity assumption within the buckets.

Algorithm 1 Creating a join estimate histogram

Input: H_A, H_B Histograms on sources A and B
Output: H_{AB} An estimate join histogram for $A \bowtie B$.

```
1: for $b_i \in H_A,\ b_j \in H_B$ do
2:     if $b_i \cap b_j \neq \varnothing$ then
3:         $b_{ij} \leftarrow New\ Bucket$
4:         $b_{ij}.upb \leftarrow min(b_i.upb, b_j.upb)$
5:         $b_{ij}.lwb \leftarrow max(b_i.lwb, b_j.lwb)$
6:         $b_{ij}.len \leftarrow b_{ij}.upb - b_{ij}.lwb + 1$
7:         $k_i \leftarrow b_i.nelt/b_i.len$
8:         $k_j \leftarrow b_j.nelt/b_j.len$
9:         $b_{ij}.nelt \leftarrow b_{ij}.len \cdot k_i \cdot k_j$
10:        $H_{AB}.append(b_{ij})$
11:    end if
12: end for
```

Note that even if the algorithm seems to have a complexity of $O(N \cdot M)$ with N, M the number of buckets in H_A and H_B, for serial histograms, the estimate histogram can be generated in a single linear run, and is ensured to have no more than $N + M - 1$ buckets. Figure 3 is a simple example of an estimate join histogram generated with this algorithm.

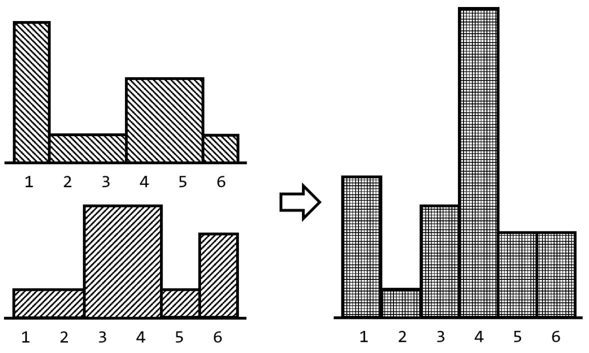

Figure 3: Histogram estimating a join

3.2 Types of histograms

To construct a histogram over a data set, one needs to determine a rule to associate each value of the data set A to a bucket in the histogram. The two most intuitive rules are *equi-width* and *equi-depth* histograms.

- **Equi-width** histograms have N buckets associated with intervals of equal length. Thus, buckets associated with intervals with more frequent values will contain more elements than buckets with associated to intervals with rare values.
- **Equi-depth** (or equi-height or equi-sum) histograms have N buckets, each containing the same number of values. Thus, most frequent values will fall in buckets associated with narrower intervals, and rare values will fall in buckets associated with wider intervals.

Figure 4 illustrates the difference between these two types of histogram with their rough shape for a Gaussian dataset.

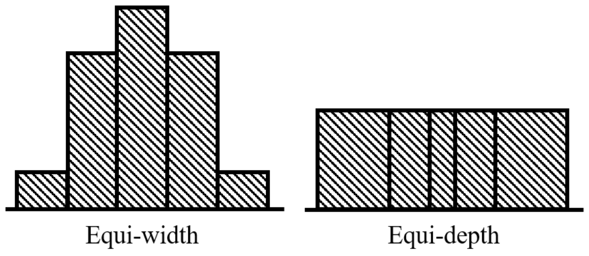

Figure 4: Equi-width vs. equi-depth histogram of a Gaussian data set.

These two types of histograms still find many concrete applications although they are based on very trivial rules. When it comes to use histogram to estimate selectivity of joins, better strategies have been experimented:

- **V-optimal** histograms [Poo+96] aim to minimize the weighted variance $\sum_{i=1}^{N} n_i v_i$ where n_i is the number of elements in the i^{th} bucket, and v_i the inner variance of the bucket.
- **Maxdiff** histograms [Poo+96] put the $N-1$ boundaries of the buckets between the values with highest frequency difference.
- **Wavelet-based** histograms [MVW98] perform a wavelet transform on the cumulative data distribution of the source, and keep the most significant wavelet coefficients to build histograms that describe the best the overall data distribution.
- **End-biased** histograms [IP95] can be applied to all the previously mentioned types of histograms. The idea of an end-biased strategy is to keep the most frequent values in singleton buckets. It tends to improve the accuracy of join estimations since punctual values are frequently overrepresented.

3.3 Motivation for dynamic histograms

Histograms provide precious information about sources that allow valuables estimations of joins, so they are often used in DBMS for query optimization. All the types of histograms presented in the previous section can be generated in a constant amount of passes over the data set. However, for sources whose data is meant to change, one would expect a solution to keep histograms up-to-date by applying small modifications to the existing one, rather than generating a new one from scratch every time the data is modified.

Equi-width is the only type of histograms where this operation is trivial, since buckets are meant to change only when data is added over the existing bounds. Dynamic maintenance strategies have been developed for equi-depth and V-optimal histograms in [DIR99], and for wavelet-based histograms in [MVW00].

Histograms that adapt dynamically to evolving sources have been thought for DBMS, but the idea of using similar techniques in dynamic query optimization makes sense. Assuming that a source is read randomly, a shape of the overall distribution can be expected to be determined relatively quickly. If histograms are dynamically constructed while reading multiple sources randomly, they could be used to estimate the cost of different plans joining these sources, without preliminary information on them. So a dynamic query optimization strategy could be to start executing an initial plan joining sources without any preliminary information available, make dynamic histograms on these sources during execution, use them to compute alternative plans, and switch to an optimal one for the end of the execution.

3.4 Algorithm

Our algorithm lays on a choice to never change the bounds of an existing bucket since no information is kept about the data distribution within the buckets. Instead, when a value is read that doesn't fall into an existing bucket, it is added to the histogram as a singleton. Then, for the number of buckets to be stable, two buckets are merged together.

Algorithm 2 Dynamic histogram construction

Input: A, a data source
Output: $H_A(x)$, a histogram for A on the attribute x

```
 1: $H_A \leftarrow New\ Histogram$
 2: for $t \in A$ do
 3:     $x \leftarrow t(x)$
 4:     if $\exists\ b_i \in H_A,\ b_i.lwb \leq x \leq b_i.upb$ then
 5:         $update(b_i, x)$
 6:     else
 7:         $H_A.append(New\ Bucket(\{x\}))$
 8:         $b_i, b_j \leftarrow findBucketsToMerge(H_A)$
 9:         $merge(b_i, b_j)$
10:     end if
11: end for
```

Note that for the dynamic histogram to remain serial, the two buckets to merge must be neighbors. In our algorithms, it implies $j = i + 1$.

Multiple approaches could be considered for the choice of the two buckets to merge. We have chosen to compare two strategies. The first one is an attempt to imitate histograms that are proven to give good estimates when they are made over static data. The second one is a greedy strategy, that happens to be relevant for an iterative construction, and even better than more complex histograms in practice, as shown in 4

Our strategy imitating histograms efficient for static data is based on the V-optimal objective function. V-optimal histograms produce good estimations, and they rely only on the weighted variance which can be easily computed iteratively without preliminary knowledge, and can also be easily calculated for the merger of two buckets.

For each bucket, the information that needs to be kept is the number n of elements it contains, the mean $\overline{x}$, sum of squared deviations s, and variance v of the elements it contains. Then, when adding a new element x to the bucket, all the information can be updated iteratively with Welford's algorithm:

$$\begin{cases} n_{new} \leftarrow n_{old} + 1 \\ \overline{x}_{new} \leftarrow \overline{x}_{old} + (x - \overline{x}_{old})/(n_{new}) \\ s_{new} \leftarrow s_{old} + (x - \overline{x}_{old})(x - \overline{x}_{new}) \\ v_{new} \leftarrow (s_{new})(n_{new} - 1) \end{cases}$$

When merging two buckets b_i and b_j, information about the merged bucket $b_{i,j}$ can be calculated as follows:

$$\begin{cases} n_{i,j} \leftarrow n_i + n_j \\ \overline{x}_{i,j} \leftarrow (n_i\overline{x}_i + n_j\overline{x}_j)/(n_i + n_j) \\ s_{i,j} \leftarrow s_i + s_j + \frac{n_i n_j}{n_i + n_j}(\overline{x}_i - \overline{x}_j)^2 \\ v_{i,j} \leftarrow (s_{ij})/(n_{ij} - 1) \end{cases}$$

The objective a V-optimal histogram is to minimize the total weighted variance. Therefore, the optimal buckets regarding this objective are b_i and b_{i+1} found with:

$$\min_i n_{i,i+1}v_{i,i+1} - n_i v_i - n_{i+1}v_{i+1}$$

Our choice for the greedy algorithm is to merge the buckets with the fewest elements. This is somehow similar to the equi-depth idea of focusing precisely on the important values and "sacrifice" the intervals with rare values. This algorithm needs need only to remember the bounds and the number of elements in each bucket, so updating and merging are trivial operations.

For this choice of implementation, we have the following complexities over N the number of buckets in the histogram

- Finding in which existing bucket the value falls in if does can be done with a dichotomy search, so in $O(log_2(N+1))$ operations. It is fortunately reasonable, since this search will have to be done at every iteration while constructing the histogram.

- Updating the histogram can be updated in $O(1)$ operations, as detailed above.

- Finding the two buckets to merge is a linear search since we keep the serial property, so $O(N)$. Hopefully, this is expected to happen rather rarely: merging two buckets is only triggered when a value falls in the space between the boundaries of the existing buckets. But merging two buckets reduces this space, and therefore the risk of having to merge buckets again.

- Merging two identified buckets is done in $O(1)$, as it has been seen above.

4 Experimentation

We have tested the quality of our dynamic histograms through an implementation in the Julia programming language.

4.1 Quality of the estimations

We have measured the quality of our estimations with two measures: the normalized root mean square deviation (NRMSD) to capture the overall magnitude of errors while accounting for large deviations, and the symmetric mean absolute percentage error (SMAPE) to evaluate the relative accuracy of predictions across different scales:

$$NRMSD : \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

$$SMAPE : \frac{100}{n}\sum_{i=1}^{n}\frac{|\hat{y}_i - y_i|}{(|\hat{y}_i| + |y_i|)/2}$$

Where $\hat{y}$ is a vector of estimations for the actual values in y.

Our tests use randomly generated sources with four common types of distributions: uniform, uniform with some overrepresented values, Gaussian, and Zipf. All these distributions are joined with each other, on overlapping intervals of 100%, 90%, 50% and

10%. Eventually, all the combinations are performed 10 times, Eventually, all the combinations are performed 10 times, so 640 binary equi-joins are performed in a total set of tests.

The following table shows results of such tests with 30 buckets histograms constructed over the whole sources, for sources of 100,000 elements in intervals of size 1000, with actual results from 6.406 to 549.170.206 and a median at 493.441:

	Greedy	V-optimal like
NRMSD	0,0036	0,0419
SMAPE	20,84%	31,77%

This first test is already encouraging regarding the possibility to make good estimations with dynamic histograms. The following sections will study more precisely the influence of some important parameters.

4.2 Convergence of the estimations

Figures 5 and 6 show the evolution of the error depending on the proportion of the source read. Clearly, the error converges fast, and estimates don't improve much when approximately 10% of the data has been read.

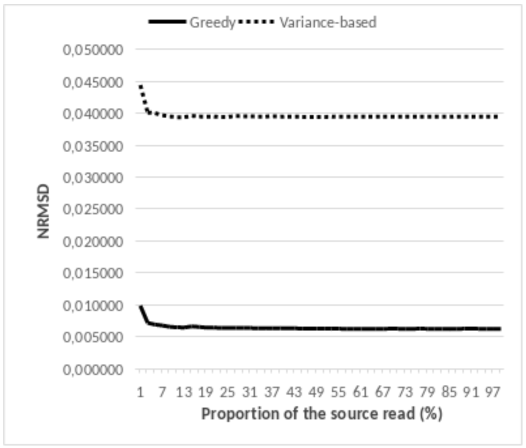

Figure 5: NRMSD convergence

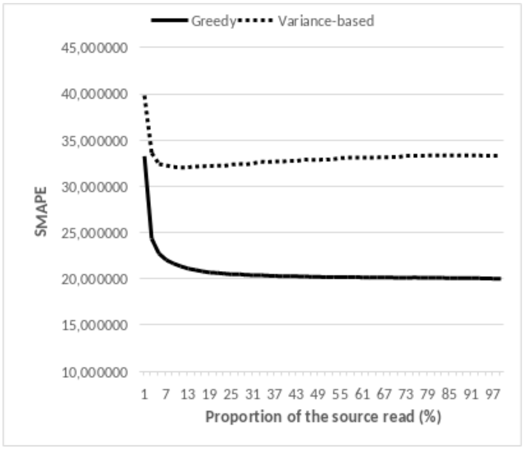

Figure 6: SMAPE convergence

These curves have been made for sources similar to those use previously. Tests with sources of a different size or a different number of buckets have similar shapes.

This is an engaging result in our dynamic query optimization context: since the estimations converge fast, histograms won't have to be computed for long, and the search of an alternative plan can be triggered at a an early stage of the execution.

These figures also confirm the better efficiency of the greedy algorithm over the one inspired by V-optimal histograms.

4.3 Choosing a number of buckets

Figures 7 and 8 show how the number of buckets influences the error. Tests have been made for intervals of size 1000 with histograms with 4, 8, 16, 32, 64, 128 and 256 buckets.

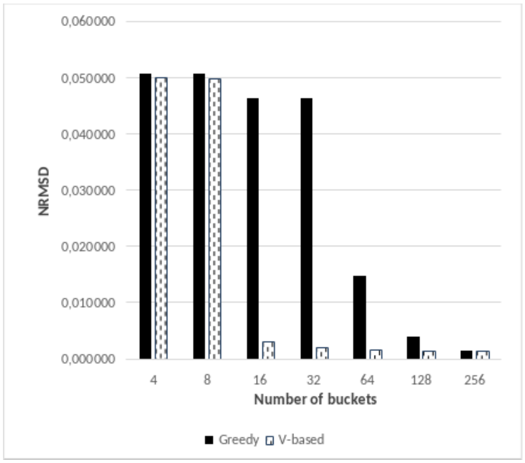

Figure 7: NRMSD sensitivity to the number of buckets

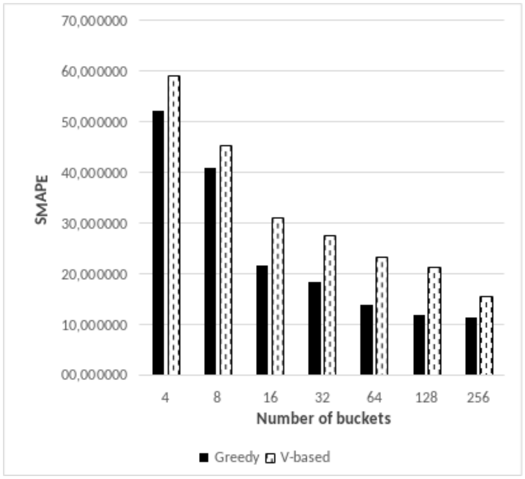

Figure 8: SMAPE sensitivity to the number of buckets

The SMAPE decreases an expectable way, and the greedy dynamic histograms still perform better than the V-based ones. The results for NRMSD seem more surprising at first sight, but aren't that much when thinking about the meaning of the measure. NRMSD is particularly sensible to huge errors, and estimating the result of joins involving hundreds of thousands of elements with a small number of buckets can't avoid such errors. This chart also shows that V-based dynamic histograms are more resistant to huge errors with a restricted number of buckets than greedy ones. However, this remain few advantage when greedy dynamic histograms do better on so many other plans.

5 Query plan transformation

The goal of this section is to see how dynamic histograms can be used in a global dynamic query optimization process. Our experiments didn't go further than join estimation with histograms, so this section is more about ideas and perspectives that haven't been tested yet. In 5.1, we present the concept of pipelined query processing, which is essential for a dynamic query optimization strategy to be possible. In 5.2, we present an intersting algorithm to perform joins in a pipelined context. In 5.3, we justify the choice of a particular join tree structure for the initial plan. In 5.4, we see how to chose an alternative plan thanks to the information provided by the dynamic histograms.

5.1 Pipelined query processing

We have seen in 2.1 that a query plan can be represented as a binary tree where nodes are operators of relational algebra, and leaves are relations (sources). An intuitive way to execute a plan is to process the operators iteratively in a postfix order. A first limit of this method is that it returns all the results as one, at the end of the whole query execution. For a query whose execution takes a lot of time, this can be "blocking". One may prefer to get the results iteratively, thus reducing the time to wait before the first ones are returned.

This iterative way to find results is possible by *pipelining* operators. [LR05] Let's illustrate this principle by comparing a sequential and pipelined processing of the very simple example 2 of 2.1:

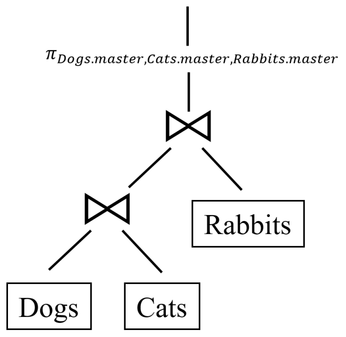

Figure 9: Query plan example

This query is composed of two equi-joins, and a projection over the joined relations. The sequential processing would do as follows:

1. Join $Dogs \bowtie Cats$. Let's call DC the result
2. Join $DC \bowtie Rabbits$. Let's call DCR the result
3. Projection $\pi_{\dots}(DCR)$

In pipelined query processing, every operator can be thought as if they provide an interface with a $next()$ function that returns the next tuple resulting from the sub-query of which it is the root. Then, the processing of the global query is performed by calling $next()$ from the root until no more results are found. In our example, this would work as follows:

1. $next()$ from the root calls $next()$ from the Projection ($\pi_{\dots}$).
2. $next()$ from the Projection ($\pi_{\dots}$) calls $next()$ from the Join $((Dogs \bowtie Cats) \bowtie Rabbits)$ and performs the projection on the resulting tuple.
3. $next()$ from the Join $((Dogs \bowtie Cats) \bowtie Rabbits)$ calls $next()$ from the Join $(Dogs \bowtie Cats)$ and $next()$ from the Relation *Rabbits* to find a matching tuple to return.
4. $next()$ from the Join $(Dogs \bowtie Cats)$ calls $next()$ from the Relations *Dogs* and *Cats* to find a matching tuple to return.
5. $next()$ from a Relation iterates over the Relation.

Dynamic query processing takes another advantage of pipelined query processing. In sequential processing, operators, including joins, are meant to be executed one after the other. So not to lose the job done so far, only alternative plans that begin with the same sub-query should be considered, which loses all the point of dynamic processing if the first join in this initial plan is a poor choice. In addition, dynamic histograms are made over a source only when the first operator involving this relation is reached. So, alternative plans involving this source can only be properly approximated at this point of the global execution. This forces to perform local transformations on sub-queries that can be estimated, rather than a global transformation that could be way better. With pipelined query processing, dynamic histograms can be constructed over all the sources in parallel, providing estimates that allow the search of a global optimal plan. Not only this transformation would be better, but it would also have to be done only once during execution, and the cost of building histograms could be saved for the end of the execution.

5.2 Double Pipelined Hash Join

Different algorithms exist to perform equi-joins. In our pipelined-dynamic case, an interesting one is the *Double Pipelined Hash Join* [Ive+99], that we will abbreviate as *2PHJ*.

The 2PHJ is based on the use of hash tables over the sources. The sketchy idea for an equi-join $A \bowtie_x B$ is that when a tuple t_A is read from a source A, its value for x is inserted in a hash table HTA. Then, all the matches for t_A in the joined source B read so far can be found by probing $t_A(x)$ in HTB, the hash table of values for x in B.

Algorithm 3 Double Pipelined Hash Join

```
Input: $A, B, x$
Output: The join of $A$ and $B$ on attribute $x$.
1: $HTA \leftarrow NewHashtable$
2: $HTB \leftarrow NewHashtable$
3: $ReadFromA \leftarrow false$
4: while A or B is not empty do
5:     if A is empty then
6:         $ReadFromA \leftarrow false$
7:     end if
8:     if B is empty then
9:         $ReadFromA \leftarrow true$
10:    end if
11:    if $ReadFromA$ then
12:        $t \leftarrow A.next()$
13:        $x \leftarrow t(x)$
14:        $HTA.insert(key : x, value : t)$
15:        if $HTB.contains(x)$ then
16:            $Results.append(t \cup HTB.get(x))$
17:        end if
18:    else
19:        $t \leftarrow B.next()$
20:        $x \leftarrow t(x)$
21:        $HTB.insert(key : x, value : t)$
22:        if $HTA.contains(x)$ then
23:            $Results.append(t \cup HTA.get(x))$
24:        end if
25:    end if
26:    $ReadFromA \leftarrow not(ReadFromA)$
27: end while
```

We can easily be convinced that all the pairs will be found: for every $t_A \in A$, matching tuples $t_B \in B$ such as $t_A(x) = t_B(x)$ will be found when t_A will be read if t_B has already been read. All the remaining matches are with tuples t_B that have not been read yet. But since t_A is added to the hash table HTA, these matches will be found when t_B will be read. Similarly, we can be sure that every pair (t_A, t_B) will be computed only once: either when t_A is read or when t_B is read.

This algorithm has some advantages, and one of them is its great efficiency. Inserting and searching in a hash table can both be done in an average time complexity of $\Theta(1)$. The 2PHJ algorithm does each of these operations exactly once for each tuple it reads, and each tuple from each source is read exactly once. So the whole join is performed in an average complexity time $\Theta(|A| + |B|)$. Since every tuple from each source must be read at least once to be sure that the join has been fully performed, we have $O(|A| + |B|)$ a naive algorithmic lower bound for the time complexity of a join, making 2PHJ at least average-optimal.

Another advantage of 2PHJ is that it reads tuples from the sources one by one, making it naturally suitable for pipelined query processing. 2PHJ is even more efficient for queries involving multiple relations joined over a same attribute (*ex:* $(A \bowtie_x B) \bowtie_x C$), called *star queries*. In this case, it is not necessary to create hash tables for the intermediary joins, since those over the relations are enough. In the simplest $(A \bowtie_x B) \bowtie_x C$ situation, only the hash table over C must be created, since finding the matches when reading from C can be done by probing $t_C(x)$ in HTA first and HTB then.

Eventually, 2PHJ is symmetric and reads from its left and right sources iteratively. In our dynamic application, this gives a chance for all the histograms to be constructed from the beginning of the execution, and thereby reduces the time before global estimations are possible.

5.3 Choosing an initial plan

Since our work is meant to face an inability to determine an optimal plan in advance, we can't chose an initial plan based on its estimated cost. However, some plans can be expected to be more relevant than others in a dynamic query processing, if they allow information gain more efficiently.

Ordering joins is equivalent to chose an order for the sources and a binary tree structure. Each couple (sources order, tree structure) corresponds to an unique way of ordering joins. Without information on the sources, the order doesn't matter, but a choice remains regarding the tree structure.

We usually consider only the three structures of the figure 10, since they correspond to the "extreme" cases. Note that a right-deep tree is equivalent to a left-deep tree with a reversed order of the sources for a symmetrical join implementation such as 2PHJ.

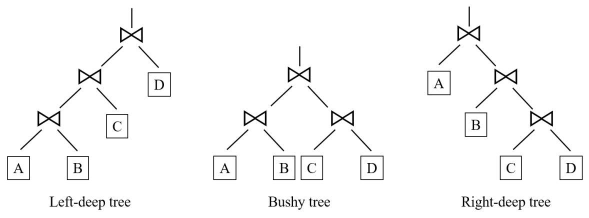

Figure 10: Join tree structures.

Traditional query optimization in DBMS usually focuses on left-deep trees. The main motivation is to reduce the number of possible plans by the significant factor of a Catalan number. Even if the optimal left-deep plan may not be a global optimum, it will be good enough in practice, and the space pruning allowed by this constraint is worth the risk. [GUW08b].

In our case, bushy tree structures can be intuited as a better choice. We aim to have estimate histograms for every source, so we would prefer to balance the time spent reading each source. However, with the pipeling system presented in 5.1 and the 2PHJ implementation of 3, we see that for a left-deep tree structure, the relation in the right child of the top join will be read into at a half of the iterations. The second uppermost will be read into at a quarter of the iterations, and so on. So, for N joins, the two lowest relations can be expected to be read into only approximately once every 2^N iterations. On the other hand, in a balanced bushy tree, every source will be read into at an approximately equal frequency, so about once every N iterations.

5.4 Switching plan

In the previous sections, we have seen how to chose an initial query plan, how to pipeline the processing of this plan to be able to collect information during execution, and how to use this information to create dynamic histograms that allow good estimates for the joins. All these steps are intended to make possible the computation of an alternative near-optimal plan to switch to.

At this point, our problem converges with the traditional DBMS problematic of static join ordering with estimates, for which good solutions have already been proposed in the literature [MN06; DT07; HD23]. Recent work, particularly T. Neumann's, deals with the question of queries involving a huge number of joins, and may be particularly relevent in our case. [Neu09; NR18].

6 Conclusions

In this paper, we have seen that the need in data integration systems to deal with autonomous sources leads to a lack of knowledge about these sources at compilation that prevents the use of traditional query optimization strategies. We have had a deeper look on the crucial problem of ordering joins. We have seen that this problem can be faced with dynamic query optimization, on condition to be provided with a way of estimating the results of the joins. We have shown that histograms can be good estimates, and briefly presented the different strategies to build optimal ones. We have proposed a way to build histograms in dynamic query processing, and implemented it to confirm their quality as estimates. Finally, we have seen how these estimates can actually be integrated to a dynamic query optimization context thanks to pipelined query processing.

References

[IP95] Yannis E. Ioannidis and Viswanath Poosala. "Histogram-based solutions to diverse database estimation problems". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 10–18.

[Poo+96] Viswanath Poosala et al. "Improved histograms for selectivity estimation of range predicates". In: *SIGMOD Rec.* 25.2 (June 1996), pp. 294–305. ISSN: 0163-5808. DOI: 10.1145/235968.233342. URL: https://doi.org/10.1145/235968.233342.

[Gan98] Sumit Ganguly. "Design and Analysis of Parametric Query Optimization Algorithms". In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 228–238. ISBN: 1558605665.

[KD98] Navin Kabra and David J. DeWitt. "Efficient mid-query re-optimization of suboptimal query execution plans". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 106–117. ISSN: 0163-5808. DOI: 10.1145/276305.276315. URL: https://doi.org/10.1145/276305.276315.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. "Wavelet-based histograms for selectivity estimation". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 448–459. ISSN: 0163-5808. DOI: 10.1145/276305.276344. URL: https://doi.org/10.1145/276305.276344.

[DIR99] Donko Donjerkovic, Yannis Ioannidis, and Raghu Ramakrishnan. *Dynamic histograms: Capturing evolving data sets.* Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1999.

[Ive+99] Zachary G. Ives et al. "An adaptive query execution system for data integration". In: *SIGMOD Rec.* 28.2 (June 1999), pp. 299–310. ISSN: 0163-5808. DOI: 10.1145/304181.304209. URL: https://doi.org/10.1145/304181.304209.

[MVW00] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. "Dynamic Maintenance of Wavelet-Based Histograms". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 101–110. ISBN: 1558607153.

[IH02] Zachary George Ives and Alon Halevy. "Efficient query processing for data integration". AAI3062954. PhD thesis. USA, 2002. Chap. 1. ISBN: 0493814914.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* 3rd ed. New York, NY: McGraw-Hill Professional, Aug. 2002, pp. 409–414.

[IHW04] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. "Adapting to source properties in processing data integration queries". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data.* SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 395–406. ISBN: 1581138598. DOI: 10.1145/1007568.1007613. URL: https://doi.org/10.1145/1007568.1007613.

[LR05] Bin Liu and Elke A. Rundensteiner. "Revisiting pipelined parallelism in multi-join query processing". In: *Proceedings of the 31st International Conference on Very Large Data Bases.* VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 829–840. ISBN: 1595931546.

[MN06] 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.* VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 930–941.

[DT07] David DeHaan and Frank Wm. Tompa. "Optimal top-down join enumeration". In: SIGMOD '07 (2007), pp. 785–796. DOI: 10.1145/1247480.1247567. URL: https://doi.org/10.1145/1247480.1247567.

[GUW08a] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database systems.* 2nd ed. Upper Saddle River, NJ: Pearson, June 2008, pp. 1–5.

[GUW08b] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database systems.* 2nd ed. Upper Saddle River, NJ: Pearson, June 2008, pp. 815–818.

[KP09] Yannis Katsis and Yannis Papakonstantinou. "View-based Data Integration". In: *Encyclopedia of Database Systems.* Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 3332–3339. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1072. URL: https://doi.org/10.1007/978-0-387-39940-9_1072.

[Neu09] Thomas Neumann. "Query simplification: graceful degradation for join-order optimization". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data.* SIGMOD '09. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 403–414. ISBN: 9781605585512. DOI: 10.1145/1559845.1559889. URL: https://doi.org/10.1145/1559845.1559889.

[TK17] Immanuel Trummer and Christoph Koch. "Multi-objective parametric query optimization". In: *Commun. ACM* 60.10 (Sept. 2017), pp. 81–89. ISSN: 0001-0782. DOI: 10.1145/3068612. URL: https://doi.org/10.1145/3068612.

[NR18] Thomas Neumann and Bernhard Radke. “Adaptive Optimization of Very Large Join Queries”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 677–692. ISBN: 9781450347037. DOI: 10.1145/3183713.3183733. URL: https://doi.org/10.1145/3183713.3183733.

[Bur+20] Maxime Buron et al. “Obi-Wan: ontology-based RDF integration of heterogeneous data”. In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 2933–2936. ISSN: 2150-8097. DOI: 10.14778/3415478.3415512. URL: https://doi.org/10.14778/3415478.3415512.

[LBP21] Hai Lan, Zhifeng Bao, and Yuwei Peng. “A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration”. en. In: *Data Sci. Eng.* 6.1 (Mar. 2021), pp. 86–101.

[HD23] Immanuel Haffner and Jens Dittrich. “Efficiently Computing Join Orders with Heuristic Search”. In: *Proc. ACM Manag. Data* 1.1 (May 2023). DOI: 10.1145/3588927. URL: https://doi.org/10.1145/3588927.